Concepts for the C++0x Standard Library: Iterators (Revision 3)

Douglas Gregor and Andrew Lumsdaine
dgregor@osl.iu.edu, lums@osl.iu.edu

Document number: N2734=08-0244
Revises document number: N2323=07-0183
Date: 2008-08-25
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to 24 of the C++ Standard Library in order to make full use of concepts [**?**]. 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 (N2691). 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 Iterators library [iterators]

2 The following subclauses describe iterator ~~requirements~~concepts, and components for iterator primitives, predefined iterators, and stream iterators, as summarized in Table 1.

Table 1: Iterators library summary

Subclause	Header(s)
24.1 ~~Requirements~~Concepts	`<iterator_concepts>`
[depr.lib.iterator.primitives] Iterator primitives 24.4 Predefined iterators [stream.iterators] Stream iterators	`<iterator>`

24.1 Iterator concepts [iterator.concepts]

The proposed wording for this section is in the separate proposal, "Iterator Concepts for the C++0x Standard Library".

24.2 Header `<iterator>` synopsis [iterator.synopsis]

```
namespace std {
  // [depr.lib.iterator.primitives], primitives:
  template<class Iterator> struct iterator_traits;
  template<class T> struct iterator_traits<T*>;

  template<class Category, class T, class Distance = ptrdiff_t,
       class Pointer = T*, class Reference = T&> struct iterator;

  struct input_iterator_tag { };
  struct output_iterator_tag { };
  struct forward_iterator_tag: public input_iterator_tag { };
  struct bidirectional_iterator_tag: public forward_iterator_tag { };
  struct random_access_iterator_tag: public bidirectional_iterator_tag { };

  // 24.3.4, iterator operations:
  template <~~class InputIterator~~InputIterator Iter~~, class Distance~~>
    void advance(~~InputIterator~~Iter& i, ~~Distance~~Iter::difference_type n);
  template <BidirectionalIterator Iter>
    void advance(Iter& i, Iter::difference_type n);
  template <RandomAccessIterator Iter>
```

```
  void advance(Iter& i, Iter::difference_type n);
template <class InputIteratorInputIterator Iter>
  typename iterator_traits<InputIterator>::difference_typeIter::difference_type
  distance(InputIteratorIter first, InputIteratorIter last);
template <RandomAccessIterator Iter>
  Iter::difference_type
  distance(Iter first, Iter last);
template <class InputIteratorInputIterator Iter>
  InputIteratorIter next(InputIteratorIter x,
    typename std::iterator_traits<InputIterator>::difference_typeIter::difference_type n = 1);
template <class BidirectionalIteratorBidirectionalIterator Iter>
  BidirectionalIteratorIter prev(BidirectionalIteratorIter x,
    typename std::iterator_traits<BidirectionalIterator>::difference_typeIter::difference_type n = 1);

// 24.4, predefined iterators:
template <classBidirectionalIterator Iterator> class reverse_iterator;

template <classBidirectionalIterator Iterator1, classBidirectionalIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasGreater<Iter1, Iter2>
  bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <classBidirectionalIterator Iterator1, classBidirectionalIterator Iterator2>
  requires HasNotEqualTo<Iter1, Iter2>
  bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter1, Iter2>
  bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLessEqual<Iter1, Iter2>
  bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasGreaterEqual<Iter1, Iter2>
  bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasMinus<Iter2::difference_type, Iter1::difference_type>
  auto operator-(
```

```
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.current - x.current);
template <class RandomAccessIterator Iterator>
  reverse_iterator<Iterator> operator+(
    typename reverse_iterator<Iterator>::difference_type Iter::difference_type n,
    const reverse_iterator<Iterator>& x);

template<BidirectionalIterator Iter>
concept_map BidirectionalIterator<reverse_iterator<Iter> > { }

template<RandomAccessIterator Iter>
concept_map RandomAccessIterator<reverse_iterator<Iter> > { }

template <class BackInsertionContainer Container> class back_insert_iterator;
template <class BackInsertionContainer Container>
  back_insert_iterator<Container> back_inserter(Container& x);
template<BackInsertionContainer Container>
  concept_map Iterator<back_insert_iterator<Container> > { }

template <class FrontInsertionContainer Container> class front_insert_iterator;
template <class FrontInsertionContainer Container>
  front_insert_iterator<Container> front_inserter(Container& x);
template<FrontInsertionContainer Container>
  concept_map Iterator<front_insert_iterator<Container> > { }

template <class InsertionContainer Container> class insert_iterator;
template <class InsertionContainer Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
template<InsertionContainer Container>
  concept_map Iterator<insert_iterator<Container> > { }

template <class InputIterator Iterator> class move_iterator;
template <class InputIterator Iterator1, class InputIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class InputIterator Iterator1, class InputIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator!=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
  requires HasLess<Iter1, Iter2>
  bool operator<(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
  requires HasLess<Iter2, Iter1>
  bool operator<=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
  requires HasLess<Iter2, Iter1>
```

```
  bool operator>(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
  requires HasLess<Iter1, Iter2>
  bool operator>=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
  requires HasMinus<Iter1, Iter2>
  auto operator-(
    const move_iterator<Iterator1>& x,
    const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
template <class RandomAccessIterator Iterator>
  move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>Iter::difference_type n, const move_iterator<Iterator>& x);
template <class InputIterator Iterator>
  move_iterator<Iterator> make_move_iterator(const Iterator& i);
template<InputIterator Iter>
  concept_map InputIterator<move_iterator<Iter> > { }
template<ForwardIterator Iter>
  concept_map ForwardIterator<move_iterator<Iter> > { }
template<BidirectionalIterator Iter>
  concept_map BidirectionalIterator<move_iterator<Iter> > { }
template<RandomAccessIterator Iter>
  concept_map RandomAccessIterator<move_iterator<Iter> > { }

// [stream.iterators], stream iterators:
template <class T, class charT = char, class traits = char_traits<charT>,
    class Distance = ptrdiff_t>
class istream_iterator;
template <class T, class charT, class traits, class Distance>
  bool operator==(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);
template <class T, class charT, class traits, class Distance>
  bool operator!=(const istream_iterator<T,charT,traits,Distance>& x,
          const istream_iterator<T,charT,traits,Distance>& y);

template <class T, class charT = char, class traits = char_traits<charT> >
    class ostream_iterator;

template<class charT, class traits = char_traits<charT> >
  class istreambuf_iterator;
template <class charT, class traits>
  bool operator==(const istreambuf_iterator<charT,traits>& a,
          const istreambuf_iterator<charT,traits>& b);
template <class charT, class traits>
  bool operator!=(const istreambuf_iterator<charT,traits>& a,
          const istreambuf_iterator<charT,traits>& b);

template <class charT, class traits = char_traits<charT> >
```

```
    class ostreambuf_iterator;
}
```

24.3 Iterator primitives [iterator.primitives]

24.3.4 Iterator operations [iterator.operations]

1 Since only random access iterators provide + and - operators, the library provides two function templates `advance` and `distance`. These function templates use + and - for random access iterators (and are, therefore, constant time for them); for input, forward and bidirectional iterators they use ++ to provide linear time implementations.

```
template <class InputIterator~~InputIterator~~ Iter, class Distance~~Iter~~>
  void advance(InputIterator~~Iter~~& i, Distance~~Iter~~::difference_type n);
template <BidirectionalIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
template <RandomAccessIterator Iter>
  void advance(Iter& i, Iter::difference_type n);
```

Note that we have eliminated the `Distance` parameter in favor of the `difference_type` of the iterator, which more accurately reflects how the iterator can move.

2 *Requires:* `n` shall be negative only for bidirectional and random access iterators.

3 *Effects:* Increments (or decrements for negative `n`) iterator reference `i` by `n`.

```
template <class InputIterator~~InputIterator~~ Iter>
  typename iterator_traits<InputIterator>::difference_type~~Iter::difference_type~~
  distance(InputIterator~~Iter~~ first, InputIterator~~Iter~~ last);
template <RandomAccessIterator Iter>
  Iter::difference_type
  distance(Iter first, Iter last);
```

4 *Effects:* Returns the number of increments or decrements needed to get from `first` to `last`.

5 *Requires:* `last` shall be reachable from `first`.

```
template <class InputIterator~~InputIterator~~ Iter>
  InputIterator~~Iter~~ next(InputIterator~~Iter~~ x,
    typename std::iterator_traits<InputIterator>::difference_type~~Iter::difference_type~~ n = 1);
```

6 *Effects:*Equivalent to `advance(x, n); return x;`

```
template <class BidirectionalIterator~~BidirectionalIterator~~ Iter>
  BidirectionalIterator~~Iter~~ prev(BidirectionalIterator~~Iter~~ x,
    typename std::iterator_traits<BidirectionalIterator>::difference_type~~Iter::difference_type~~ n = 1);
```

7 *Effects:*Equivalent to `advance(x, -n); return x;`

24.4 Predefined iterators [predef.iterators]

24.4.1 Reverse iterators [reverse.iterators]

1 Bidirectional and random access iterators have corresponding reverse iterator adaptors that iterate through the data struc-

ture in the opposite direction. They have the same signatures as the corresponding iterators. The fundamental relation between a reverse iterator and its corresponding iterator `i` is established by the identity: `&*(reverse_iterator(i)) == &*(i - 1)`.

2 This mapping is dictated by the fact that while there is always a pointer past the end of an array, there might not be a valid pointer before the beginning of an array.

24.4.1.1 Class template `reverse_iterator` [reverse.iterator]

```
namespace std {
  template <class BidirectionalIterator Iterator>
  class reverse_iterator : public
        iterator<typename iterator_traits<Iterator>::iterator_category,
        typename iterator_traits<Iterator>::value_type,
        typename iterator_traits<Iterator>::difference_type,
        typename iterator_traits<Iterator>::pointer,
        typename iterator_traits<Iterator>::reference> {
  protected:
    Iterator current;
  public:
    typedef Iterator iterator_type;
    typedef Iter::value_type value_type;
    typedef typename iterator_traits<Iterator>::difference_typeIter::difference_type difference_type;
    typedef typename iterator_traits<Iterator>::referenceIter::reference reference;
    typedef typename iterator_traits<Iterator>::pointerIter::pointer pointer;

    reverse_iterator();
    explicit reverse_iterator(Iterator x);
    template <class U>
      requires HasConstructor<Iter, const U&>
      reverse_iterator(const reverse_iterator<U>& u);
    template <class U>
      requires HasAssign<Iter, const U&>
      reverse_iterator operator=(const reverse_iterator<U>& u);

    Iterator base() const;      // explicit
    reference operator*() const;
    pointer   operator->() const;

    reverse_iterator& operator++();
    reverse_iterator  operator++(int);
    reverse_iterator& operator--();
    reverse_iterator  operator--(int);

    requires RandomAccessIterator<Iter> reverse_iterator  operator+ (difference_type n) const;
    requires RandomAccessIterator<Iter> reverse_iterator& operator+=(difference_type n);
    requires RandomAccessIterator<Iter> reverse_iterator  operator- (difference_type n) const;
    requires RandomAccessIterator<Iter> reverse_iterator& operator-=(difference_type n);
    requires RandomAccessIterator<Iter> unspecified operator[](difference_type n) const;
  };
```

```
  template <classBidirectionalIterator Iterator1, classBidirectionalIterator Iterator2>
    requires HasEqualTo<Iter1, Iter2>
    bool operator==(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
    requires HasGreater<Iter1, Iter2>
    bool operator<(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <classBidirectionalIterator Iterator1, classBidirectionalIterator Iterator2>
    requires HasNotEqualTo<Iter1, Iter2>
    bool operator!=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
    requires HasLess<Iter1, Iter2>
    bool operator>(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
    requires HasLessEqual<Iter1, Iter2>
    bool operator>=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
    requires HasGreaterEqual<Iter1, Iter2>
    bool operator<=(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y);
  template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
    requires HasMinus<Iter2::difference_type, Iter1::difference_type>
    auto operator-(
      const reverse_iterator<Iterator1>& x,
      const reverse_iterator<Iterator2>& y) -> decltype(y.current - x.current);
  template <classRandomAccessIterator Iterator>
    reverse_iterator<Iterator> operator+(
      typename reverse_iterator<Iterator>::difference_typeIter::difference_type n,
      const reverse_iterator<Iterator>& x);

  template<BidirectionalIterator Iter>
  concept_map BidirectionalIterator<reverse_iterator<Iter> > { }

  template<RandomAccessIterator Iter>
  concept_map RandomAccessIterator<reverse_iterator<Iter> > { }
}
```

24.4.1.2 `reverse_iterator` requirements [reverse.iter.requirements]

Remove [reverse.iter.requirements]

1 ~~The template parameter `Iterator` shall meet all the requirements of a Bidirectional Iterator ([bidirectional.iterators]).~~

2 ~~Additionally, `Iterator` shall meet the requirements of a Random Access Iterator ([random.access.iterators]) if any of the members `operator+` (24.4.1.3.8), `operator-` (24.4.1.3.10), `operator+=` (24.4.1.3.9), `operator-=` (24.4.1.3.11), `operator []` (24.4.1.3.12), or the global operators `operator<` (24.4.1.3.14), `operator>` (24.4.1.3.16), `operator <=` (24.4.1.3.18), `operator>=` (24.4.1.3.17), `operator-` (24.4.1.3.19) or `operator+` (24.4.1.3.20). is referenced in a way that requires instantiation ([temp.inst]).~~

24.4.1.3 `reverse_iterator` operations [reverse.iter.ops]

24.4.1.3.1 `reverse_iterator` constructor [reverse.iter.cons]

```
reverse_iterator();
```

1 *Effects:* Default initializes `current`. Iterator operations applied to the resulting iterator have defined behavior if and only if the corresponding operations are defined on a default constructed iterator of type `Iterator`.

```
explicit reverse_iterator(Iter~~ator~~ x);
```

2 *Effects:* Initializes `current` with x.

```
template <class U>
  requires HasConstructor<Iter, const U&>
  reverse_iterator(const reverse_iterator<U> &u);
```

3 *Effects:* Initializes `current` with u`.current`.

24.4.1.3.2 `reverse_iterator::operator=` [reverse.iter.op=]

```
template <class U>
  requires HasAssign<Iter, const U&>
  reverse_iterator&
    operator=(const reverse_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to current.

2 *Returns:* `*this`.

24.4.1.3.3 Conversion [reverse.iter.conv]

```
Iter~~ator~~ base() const;          // explicit
```

1 *Returns:* `current`.

24.4.1.3.4 `operator*` [reverse.iter.op.star]

```
reference operator*() const;
```

1 *Effects:*

```
this->tmp = current;
--this->tmp;
return *this->tmp;
```

2 [*Note:* This operation must use an auxiliary member variable, rather than a temporary variable, to avoid returning a reference that persists beyond the lifetime of its associated iterator. (See [iterator.requirements].) The name of this member variable is shown for exposition only. — *end note*]

24.4.1.3.5 operator-> [reverse.iter.opref]

```
pointer operator->() const;
```

1 *Returns:*

```
&(operator*());
```

24.4.1.3.6 operator++ [reverse.iter.op++]

```
reverse_iterator& operator++();
```

1 *Effects:* `--current;`

2 *Returns:* `*this`.

```
reverse_iterator operator++(int);
```

3 *Effects:*

```
reverse_iterator tmp = *this;
--current;
return tmp;
```

24.4.1.3.7 operator-- [reverse.iter.op--]

```
reverse_iterator& operator--();
```

1 *Effects:* `++current`

2 *Returns:* `*this`.

```
reverse_iterator operator--(int);
```

3 *Effects:*

```
reverse_iterator tmp = *this;
++current;
return tmp;
```

24.4.1.3.8 operator+ [reverse.iter.op+]

```
requires RandomAccessIterator<Iter>
reverse_iterator
operator+(typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `reverse_iterator(current-n)`.

24.4.1.3.9 operator+= [reverse.iter.op+=]

```
requires RandomAccessIterator<Iter>
reverse_iterator&
operator+=(typename reverse_iterator<Iterator>::difference_type n);
```

1 *Effects:* `current -= n;`

2 *Returns:* `*this`.

24.4.1.3.10 operator- [reverse.iter.op-]

```
requires RandomAccessIterator<Iter>
reverse_iterator
operator-(typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `reverse_iterator(current+n)`.

24.4.1.3.11 operator-= [reverse.iter.op-=]

```
requires RandomAccessIterator<Iter>
reverse_iterator&
operator-=(typename reverse_iterator<Iterator>::difference_type n);
```

1 *Effects:* `current += n;`

2 *Returns:* `*this`.

24.4.1.3.12 operator[] [reverse.iter.opindex]

```
requires RandomAccessIterator<Iter>
unspecified operator[](
    typename reverse_iterator<Iterator>::difference_type n) const;
```

1 *Returns:* `current[-n-1]`.

24.4.1.3.13 operator== [reverse.iter.op==]

```
template <classBidirectionalIterator Iterator1, classBidirectionalIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

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

24.4.1.3.14 operator< [reverse.iter.op<]

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasGreater<Iter1, Iter2>
  bool operator<(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current > y.current`.

24.4.1.3.15 operator!= [reverse.iter.op!=]

```
template <classBidirectionalIterator Iterator1, classBidirectionalIterator Iterator2>
  requires HasNotEqualTo<Iter1, Iter2>
  bool operator!=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current != y.current`.

24.4.1.3.16 operator> [reverse.iter.op>]

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter1, Iter2>
  bool operator>(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current < y.current`.

24.4.1.3.17 operator>= [reverse.iter.op>=]

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLessEqual<Iter1, Iter2>
  bool operator>=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current <= y.current`.

24.4.1.3.18 operator<= [reverse.iter.op<=]

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasGreaterEqual<Iter1, Iter2>
  bool operator<=(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y);
```

1 *Returns:* `x.current >= y.current`.

24.4.1.3.19 operator- [reverse.iter.opdiff]

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasMinus<Iter2::difference_type, Iter1::difference_type>
  auto operator-(
    const reverse_iterator<Iterator1>& x,
    const reverse_iterator<Iterator2>& y) -> decltype(y.current - x.current);
```

1 *Returns:* `y.current - x.current`.

24.4.1.3.20 operator+ [reverse.iter.opsum]

```
template <classRandomAccessIterator Iterator>
  reverse_iterator<Iterator> operator+(
    typename reverse_iterator<Iterator>::difference_typeIter::difference_type n,
    const reverse_iterator<Iterator>& x);
```

1 *Returns:* `reverse_iterator<Iterator> (x.current - n)`.

24.4.1.4 Concept maps [reverse.iter.maps]

```
template<BidirectionalIterator Iter>
concept_map BidirectionalIterator<reverse_iterator<Iter> > { }
```

1 *Note:* This concept map template states that reverse iterators are themselves bidirectional iterators.

```
template<RandomAccessIterator Iter>
concept_map RandomAccessIterator<reverse_iterator<Iter> > { }
```

2 *Note:* This concept map template states that reverse iterators are themselves random access iterators when the underlying iterator is a random access iterator.

24.4.2 Insert iterators [insert.iterators]

1 To make it possible to deal with insertion in the same way as writing into an array, a special kind of iterator adaptors, called *insert iterators*, are provided in the library. With regular iterator classes,

```
while (first != last) *result++ = *first++;
```

causes a range [first,last) to be copied into a range starting with result. The same code with result being an insert iterator will insert corresponding elements into the container. This device allows all of the copying algorithms in the library to work in the *insert mode* instead of the *regular overwrite* mode.

2 An insert iterator is constructed from a container and possibly one of its iterators pointing to where insertion takes place if it is neither at the beginning nor at the end of the container. Insert iterators satisfy the requirements of output iterators. operator* returns the insert iterator itself. The assignment operator=(const T& x) is defined on insert iterators to allow writing into them, it inserts x right before where the insert iterator is pointing. In other words, an insert iterator is like a cursor pointing into the container where the insertion takes place. back_insert_iterator inserts elements at the end of a container, front_insert_iterator inserts elements at the beginning of a container, and insert_iterator inserts elements where the iterator points to in a container. back_inserter, front_inserter, and inserter are three functions making the insert iterators out of a container.

24.4.2.1 Class template back_insert_iterator [back.insert.iterator]

```
namespace std {
  template <class BackInsertionContainer Container>
  class back_insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {
  protected:
    Container* container;

  public:
    typedef Container container_type;
    typedef void value_type;
    typedef void difference_type;
    typedef back_insert_iterator<Container>& reference;
    typedef void pointer;

    explicit back_insert_iterator(Container& x);
    back_insert_iterator<Container>&
      operator=(typename const Container::const_reference value_type& value);
    back_insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

    back_insert_iterator<Container>& operator*();
    back_insert_iterator<Container>& operator++();
    back_insert_iterator<Container>  operator++(int);
  };

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

  template<BackInsertionContainer Container>
    concept_map Iterator<back_insert_iterator<Container> > { }
}
```

24.4.2.2 back_insert_iterator operations [back.insert.iter.ops]

24.4.2.2.1 back_insert_iterator constructor [back.insert.iter.cons]

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

1 *Effects:* Initializes container with &x.

24.4.2.2.2 back_insert_iterator::operator= [back.insert.iter.op=]

```
back_insert_iterator<Container>&
  operator=(typenameconst Container::const_referencevalue_type& value);
```

1 *Effects:* container->push_back(*container, value);

2 *Returns:* *this.

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

3 *Effects:*container->push_back(*container, std::move(value));

4 *Returns:**this.

24.4.2.2.3 back_insert_iterator::operator* [back.insert.iter.op*]

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

1 *Returns:* *this.

24.4.2.2.4 back_insert_iterator::operator++ [back.insert.iter.op++]

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

1 *Returns:* *this.

24.4.2.2.5 back_inserter [back.inserter]

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

1 *Returns:* back_insert_iterator<Container>(x).

24.4.2.2.6 Concept maps [back.insert.iter.maps]

```
template<BackInsertionContainer Container>
  concept_map Iterator<back_insert_iterator<Container> > { }
```

1 *Note:*Declares that back_insert_iterator is an iterator.

24.4.2.3 Class template `front_insert_iterator` [front.insert.iterator]

```
namespace std {
  template <class FrontInsertionContainer Container>
  class front_insert_iterator :
    public iterator<output_iterator_tag,void,void,void,void> {
  protected:
    Container* container;

  public:
    typedef Container container_type;
    typedef void value_type;
    typedef void difference_type;
    typedef front_insert_iterator<Container>& reference;
    typedef void pointer;

    explicit front_insert_iterator(Container& x);
    front_insert_iterator<Container>&
      operator=(typename const Container::const_reference value_type& value);
    front_insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

    front_insert_iterator<Container>& operator*();
    front_insert_iterator<Container>& operator++();
    front_insert_iterator<Container>  operator++(int);
  };

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

  template<FrontInsertionContainer Container>
    concept_map Iterator<front_insert_iterator<Container> > { }
}
```

24.4.2.4 `front_insert_iterator` operations [front.insert.iter.ops]

24.4.2.4.1 `front_insert_iterator` constructor [front.insert.iter.cons]

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

1 *Effects:* Initializes `container` with `&x`.

24.4.2.4.2 `front_insert_iterator::operator=` [front.insert.iter.op=]

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

1 *Effects:* ~~`container->`~~`push_front(*container, value);`

2 *Returns:* `*this`.

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

3 *Effects:* container->push_front(*container, std::move(value));

4 *Returns:* *this.

24.4.2.4.3 front_insert_iterator::operator* [front.insert.iter.op*]

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

1 *Returns:* *this.

24.4.2.4.4 front_insert_iterator::operator++ [front.insert.iter.op++]

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

1 *Returns:* *this.

24.4.2.4.5 front_inserter [front.inserter]

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

1 *Returns:* front_insert_iterator<Container>(x).

24.4.2.4.6 Concept maps [front.insert.iter.maps]

```
template<FrontInsertionContainer Container>
  concept_map Iterator<front_insert_iterator<Container> > { }
```

1 *Note:* Declares that front_insert_iterator is an iterator.

24.4.2.5 Class template insert_iterator [insert.iterator]

```
namespace std {
  template <class InsertionContainer 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;
    typedef void value_type;
    typedef void difference_type;
```

```
    typedef insert_iterator<Container>& reference;
    typedef void pointer;

    insert_iterator(Container& x, typename Container::iterator i);
    insert_iterator<Container>&
      operator=(typenameconst Container::const_referencevalue_type& value);
    insert_iterator<Container>&
      operator=(typename Container::value_type&& value);

    insert_iterator<Container>& operator*();
    insert_iterator<Container>& operator++();
    insert_iterator<Container>& operator++(int);
  };

  template <classInsertionContainer Container>
    insert_iterator<Container> inserter(Container& x, typename Container::iterator i);

  template<InsertionContainer Container>
    concept_map Iterator<insert_iterator<Container> > { }
}
```

24.4.2.6 `insert_iterator` operations [insert.iter.ops]

24.4.2.6.1 `insert_iterator` constructor [insert.iter.cons]

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

1 *Effects:* Initializes `container` with *&x* and `iter` with *i*.

24.4.2.6.2 `insert_iterator::operator=` [insert.iter.op=]

```
insert_iterator<Container>&
  operator=(typenameconst Container::const_referencevalue_type& value);
```

1 *Effects:*

```
iter = container->insert(*container, iter, value);
++iter;
```

2 *Returns:* `*this`.

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

3 *Effects:*

```
iter = container->insert(*container, iter, std::move(value));
++iter;
```

4 *Returns:* `*this`.

24.4.2.6.3 `insert_iterator::operator*` [insert.iter.op*]

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

1 *Returns:* `*this`.

24.4.2.6.4 `insert_iterator::operator++` [insert.iter.op++]

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

1 *Returns:* `*this`.

24.4.2.6.5 `inserter` [inserter]

```
template <class InsertionContainer Container>
  insert_iterator<Container> inserter(Container& x, typename Container::iterator i);
```

1 *Returns:* `insert_iterator<Container>(x, i)`.

24.4.2.6.6 Concept maps [insert.iter.maps]

```
template<InsertionContainer Container>
  concept_map Iterator<insert_iterator<Container> > { }
```

1 *Note:* Declares that `insert_iterator` is an iterator.

24.4.3 Move iterators [move.iterators]

1 Class template `move_iterator` is an iterator adaptor with the same behavior as the underlying iterator except that its dereference operator implicitly converts the value returned by the underlying iterator's dereference operator to an rvalue reference. Some generic algorithms can be called with move iterators to replace copying with moving.

2 [*Example:*

```
set<string> s;
// populate the set s
vector<string> v1(s.begin(), s.end());           // copies strings into v1
vector<string> v2(make_move_iterator(s.begin()),
                  make_move_iterator(s.end())); // moves strings into v2
```

— *end example*]

24.4.3.1 Class template `move_iterator` [move.iterator]

```
namespace std {
  template <class InputIterator Iterator>
  class move_iterator {
  public:
```

```
  typedef Iterator                                                iterator_type;
  typedef typename iterator_traits<Iterator>Iter::difference_type   difference_type;
  typedef Iterator                                                pointer;
  typedef typename iterator_traits<Iterator>Iter::value_type        value_type;
  typedef typename iterator_traits<Iterator>::iterator_category iterator_category;
  typedef value_type&&                                            reference;

  move_iterator();
  explicit move_iterator(Iterator i);
  template <class U>
    requires HasConstructor<Iter, const U&>
    move_iterator(const move_iterator<U>& u);
  template <class U>
    requires HasAssign<Iter, const U&>
    move_iterator& operator=(const move_iterator<U>& u);

  iterator_type base() const;
  reference operator*() const;
  pointer operator->() const;

  move_iterator& operator++();
  move_iterator operator++(int);
  requires BidirectionalIterator<Iter> move_iterator& operator--();
  requires BidirectionalIterator<Iter> move_iterator operator--(int);

  requires RandomAccessIterator<Iter> move_iterator operator+(difference_type n) const;
  requires RandomAccessIterator<Iter> move_iterator& operator+=(difference_type n);
  requires RandomAccessIterator<Iter> move_iterator operator-(difference_type n) const;
  requires RandomAccessIterator<Iter> move_iterator& operator-=(difference_type n);
  requires RandomAccessIterator<Iter>
    unspecified operator[](difference_type n) const;

private:
  Iterator current;    // exposition only
};

template <classInputIterator Iterator1, classInputIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <classInputIterator Iterator1, classInputIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator!=(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter1, Iter2>
  bool operator<(
    const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter2, Iter1>
```

```
    bool operator<=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
    requires HasLess<Iter2, Iter1>
    bool operator>(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
  template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
    requires HasLess<Iter1, Iter2>
    bool operator>=(
      const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);

  template <class RandomAccessIterator Iterator1, class RandomAccessIterator Iterator2>
    requires HasMinus<Iter1, Iter2>
    auto operator-(
      const move_iterator<Iterator1>& x,
      const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
  template <class RandomAccessIterator Iterator>
    move_iterator<Iterator> operator+(
      typename move_iterator<Iterator>::Iter::difference_type n, const move_iterator<Iterator>& x);
  template <class InputIterator Iterator>
    move_iterator<Iterator> make_move_iterator(const Iterator& i);

  template<InputIterator Iter>
    concept_map InputIterator<move_iterator<Iter> > { }
  template<ForwardIterator Iter>
    concept_map ForwardIterator<move_iterator<Iter> > { }
  template<BidirectionalIterator Iter>
    concept_map BidirectionalIterator<move_iterator<Iter> > { }
  template<RandomAccessIterator Iter>
    concept_map RandomAccessIterator<move_iterator<Iter> > { }
}
```

24.4.3.2 `move_iterator` requirements [move.iter.requirements]

Remove [move.iter.requirements]

1 The template parameter `Iterator` shall meet the requirements for an Input Iterator ([input.iterators]). Additionally, if any of the bidirectional or random access traversal functions are instantiated, the template parameter shall meet the requirements for a Bidirectional Iterator ([bidirectional.iterators]) or a Random Access Iterator ([random.access.iterators]), respectively.

24.4.3.3 `move_iterator` operations [move.iter.ops]

24.4.3.3.1 `move_iterator` constructors [move.iter.op.const]

```
move_iterator();
```

1 *Effects:* Constructs a `move_iterator`, default initializing `current`.

```
explicit move_iterator(Iterator i};
```

2 *Effects:* Constructs a `move_iterator`, intializing `current` with `i`.

```
template <class U>
  requires HasConstructor<Iter, const U&>
  move_iterator(const move_iterator<U>& u);
```

3 *Effects:* Constructs a `move_iterator`, initializing `current` with `u.base()`.

4 ~~*Requires:* `U` shall be convertible to `Iterator`.~~

24.4.3.3.2 `move_iterator::operator=` [move.iter.op=]

```
template <class U>
  requires HasAssign<Iter, const U&>
  move_iterator& operator=(const move_iterator<U>& u);
```

1 *Effects:* Assigns `u.base()` to `current`.

2 ~~*Requires:* `U` shall be convertible to `Iterator`.~~

24.4.3.3.3 `move_iterator` conversion [move.iter.op.conv]

```
Iter~~ator~~ base() const;
```

1 *Returns:* `current`.

24.4.3.3.4 `move_iterator::operator*` [move.iter.op.star]

```
reference operator*() const;
```

1 *Returns:* `*current`, implicitly converted to an rvalue reference.

24.4.3.3.5 `move_iterator::operator->` [move.iter.op.ref]

```
pointer operator->() const;
```

1 *Returns:* `current`.

24.4.3.3.6 `move_iterator::operator++` [move.iter.op.incr]

```
move_iterator& operator++();
```

1 *Effects:* `++current`.

2 *Returns:* `*this`.

```
move_iterator& operator++(int);
```

3 *Effects:*

```
move_iterator tmp = *this;
++current;
return tmp;
```

24.4.3.3.7 move_iterator::operator-- [move.iter.op.decr]

```
requires BidirectionalIterator<Iter> move_iterator& operator--();
```

1 *Effects:* `--current`.

2 *Returns:* `*this`.

```
requires BidirectionalIterator<Iter> move_iterator& operator--(int);
```

3 *Effects:*

```
move_iterator tmp = *this;
--current;
return tmp;
```

24.4.3.3.8 move_iterator::operator+ [move.iter.op.+]

```
requires RandomAccessIterator<Iter> move_iterator operator+(difference_type n) const;
```

1 *Returns:* `move_iterator(current + n)`.

24.4.3.3.9 move_iterator::operator+= [move.iter.op.+=]

```
requires RandomAccessIterator<Iter> move_iterator& operator+=(difference_type n);
```

1 *Effects:* `current += n`.

2 *Returns:* `*this`.

24.4.3.3.10 move_iterator::operator- [move.iter.op.-]

```
requires RandomAccessIterator<Iter> move_iterator operator-(difference_type n) const;
```

1 *Returns:* `move_iterator(current - n)`.

24.4.3.3.11 move_iterator::operator-= [move.iter.op.-=]

```
requires RandomAccessIterator<Iter> move_iterator& operator-=(difference_type n);
```

1 *Effects:* `current -= n`.

2 *Returns:* `*this`.

24.4.3.3.12 `move_iterator::operator[]` [move.iter.op.index]

```
requires RandomAccessIterator<Iter>
  unspecified operator[](difference_type n) const;
```

1 *Returns:* `current[n]`, implicitly converted to an rvalue reference.

24.4.3.3.13 `move_iterator` comparisons [move.iter.op.comp]

```
template <classInputIterator Iterator1, classInputIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator==(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

1 *Returns:* `x.base() == y.base()`.

```
template <classInputIterator Iterator1, classInputIterator Iterator2>
  requires HasEqualTo<Iter1, Iter2>
  bool operator!=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

2 *Returns:* `!(x == y)`.

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter1, Iter2>
  bool operator<(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

3 *Returns:* `x.base() < y.base()`.

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter2, Iter1>
  bool operator<=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

4 *Returns:* `!(y < x)`.

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter2, Iter1>
  bool operator>(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

5 *Returns:* `y < x`.

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasLess<Iter1, Iter2>
  bool operator>=(const move_iterator<Iterator1>& x, const move_iterator<Iterator2>& y);
```

6 *Returns:* `!(x < y)`.

24.4.3.3.14 `move_iterator` non-member functions [move.iter.nonmember]

```
template <classRandomAccessIterator Iterator1, classRandomAccessIterator Iterator2>
  requires HasMinus<Iter1, Iter2>
  auto operator-(
    const move_iterator<Iterator1>& x,
```

```
    const move_iterator<Iterator2>& y) -> decltype(x.base() - y.base());
```

1 *Returns:* `x.base() - y.base()`.

```
template <class RandomAccessIterator Iterator>
  move_iterator<Iterator> operator+(
    typename move_iterator<Iterator>::difference_type n, const move_iterator<Iterator>& x);
```

2 *Returns:* `x + n`.

```
template <class InputIterator Iterator>
move_iterator<Iterator> make_move_iterator(const Iterator& i);
```

3 *Returns:* `move_iterator<Iterator>(i)`.

24.4.3.3.15 Concept maps [move.iter.maps]

```
template<InputIterator Iter>
  concept_map InputIterator<move_iterator<Iter> > { }
```

1 *Note:* Declares that a `move_iterator` is an input iterator.

```
template<ForwardIterator Iter>
  concept_map ForwardIterator<move_iterator<Iter> > { }
```

2 *Note:* Declares that a `move_iterator` is a forward iterator if its underlying iterator is a forward iterator.

```
template<BidirectionalIterator Iter>
  concept_map BidirectionalIterator<move_iterator<Iter> > { }
```

3 *Note:* Declares that a `move_iterator` is a bidirectional iterator if its underlying iterator is a bidirectional iterator.

```
template<RandomAccessIterator Iter>
  concept_map RandomAccessIterator<move_iterator<Iter> > { }
```

4 *Note:* Declares that a `move_iterator` is a random access iterator if its underlying iterator is a random access iterator.

Bibliography