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

Douglas Gregor, Jeremiah Willcock, and Andrew Lumsdaine
Open Systems Laboratory
Indiana University
Bloomington, IN 47405
{dgregor, jewillco, lums}@osl.iu.edu

Document number: N2083=06-0153
Revises document number: N2039=06-0109
Date: 2006-09-08
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes changes to Chapter 24 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 Iterators library [lib.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	`<iterator>`
24.3 Iterator primitives	`<iterator>`
?? Predefined iterators	
?? Stream iterators	

24.1 Iterator requirements [lib.iterator.requirements]

1 Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner. To be able to construct template algorithms that work correctly and efficiently on different types of data structures, the library formalizes not just the interfaces but also the semantics and complexity assumptions of iterators. All input iterators `i` support the expression `*i`, resulting in a value of some class, enumeration, or built-in type `T`, called the *value type* of the iterator. All output iterators support the expression `*i = o` where `o` is a value of some type that is in the set of types that are *writable* to the particular iterator type of `i`. All iterators `i` for which the expression `(*i).m` is well-defined, support the expression `i->m` with the same semantics as `(*i).m`. For every iterator type `X` for which equality is defined, there is a corresponding signed integral type called the *difference type* of the iterator.

```
concept IteratorAssociatedTypes<typename X> {
  typename value_type = X::value_type;
  typename difference_type = X::difference_type;
  typename reference = X::reference;
  typename pointer = X::pointer;
};
```

2 Since iterators are an abstraction of pointers, their semantics is a generalization of most of the semantics of pointers in C++. This ensures that every function template that takes iterators works as well with regular pointers. This International Standard defines ~~five categories of iterators~~eight iterator concepts, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, mutable forward iterators, *bidirectional iterators*, mutable bidirectional iterators, *random access iterators*, and mutable random access iterators as shown in Table 2.

3 Forward iterators satisfy all the requirements of the input ~~and output~~ iterators and can be used whenever ~~either kind~~an input iterator is specified. Mutable forward iterators satisfy all the requirements of forward and output iterators, an can

Table 2: Relations among iterator ~~categories~~concepts

Random Access	→ **Bidirectional**	→ **Forward**	→ **Input**
↑	↑	↑	
Mutable Random Access	→ **Mutable Bidirectional**	→ **Mutable Forward**	→ **Output**

be used whenever either kind is specified. Bidirectional iterators also satisfy all the requirements of the forward iterators and can be used whenever a forward iterator is specified. Random access iterators also satisfy all the requirements of bidirectional iterators and can be used whenever a bidirectional iterator is specified.

4 ~~Besides its category, a forward, bidirectional, or random access iterator can also be mutable or constant depending on whether the result of the expression *i behaves as a reference or as a reference to a constant. Constant iterators do not satisfy the requirements for output iterators, and the result of the expression *i (for constant iterator i) cannot be used in an expression where an lvalue is required.~~ The mutable variants of the forward, bidirectional, and random access iterator concepts satisfy the requirements for output iterators, and can be used wherever an output iterator is required. Non-mutable iterators are referred to as constant iterators.

5 Just as a regular pointer to an array guarantees that there is a pointer value pointing past the last element of the array, so for any iterator type there is an iterator value that points past the last element of a corresponding container. These values are called *past-the-end* values. Values of an iterator `i` for which the expression `*i` is defined are called *dereferenceable*. The library never assumes that past-the-end values are dereferenceable. Iterators can also have singular values that are not associated with any container. [*Example:* After the declaration of an uninitialized pointer `x` (as with `int* x;`), `x` must always be assumed to have a singular value of a pointer. — *end example*] Results of most expressions are undefined for singular values; the only exceptions are destroying an iterator that holds a singular value and the assignment of a non-singular value to an iterator that holds a singular value. In this case the singular value is overwritten the same way as any other value. Dereferenceable values are always non-singular.

6 An iterator `j` is called *reachable* from an iterator `i` if and only if there is a finite sequence of applications of the expression `++i` that makes `i == j`. If `j` is reachable from `i`, they refer to the same container.

7 Most of the library's algorithmic templates that operate on data structures have interfaces that use ranges. A *range* is a pair of iterators that designate the beginning and end of the computation. A range `[i,i)` is an empty range; in general, a range `[i,j)` refers to the elements in the data structure starting with the one pointed to by `i` and up to but not including the one pointed to by `j`. Range `[i,j)` is valid if and only if `j` is reachable from `i`. The result of the application of functions in the library to invalid ranges is undefined.

8 All the ~~categories of iterators~~iterator concepts require only those functions that are realizable ~~for a given category~~ in constant time (amortized). ~~Therefore, requirement tables for the iterators do not have a complexity column.~~

9 Destruction of an iterator may invalidate pointers and references previously obtained from that iterator.

10 An *invalid* iterator is an iterator that may be singular.[1)]

11 ~~In the following sections, a and b denote values of type const X, n denotes a value of the difference type Distance, u, tmp, and m denote identifiers, r denotes a value of X&, t denotes a value of value type T, o denotes a value of some type~~

[1)] This definition applies to pointers, since pointers are iterators. The effect of dereferencing an iterator that has been invalidated is undefined.

~~that is writable to the output iterator.~~

24.1.1 Input iterators [lib.input.iterators]

1 A class or a built-in type `X` satisfies the requirements of an input iterator for the value type `T` if ~~the following expressions are valid, where U is the type of any specified member of type T, as shown in Table 89.~~it meets the syntactic and semantic requirements of the `InputIterator` concept.

```
concept InputIterator<typename X>
  : IteratorAssociatedTypes<X>, CopyConstructible<X>, EqualityComparable<X> {

  where Assignable<X> && SameType<Assignable<X>::result_type, X&>;

  where SignedIntegral<difference_type> &&
        Convertible<reference, value_type> &&
        Convertible<pointer, const value_type*>;

  typename postincrement_result;
  where Dereferenceable<postincrement_result> &&
        Convertible<Dereferenceable<postincrement_result>::reference, value_type>;

  pointer operator->(X);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
  reference operator*(X);
};
```

2 ~~In Table 89~~In the `InputIterator` concept, the term *the domain of* `==` is used in the ordinary mathematical sense to denote the set of values over which `==` is (required to be) defined. This set can change over time. Each algorithm places additional requirements on the domain of `==` for the iterator values it uses. These requirements can be inferred from the uses that algorithm makes of `==` and `!=`. [*Example:*the call `find(a,b,x)` is defined only if the value of `a` has the property *p* defined as follows: `b` has property *p* and a value `i` has property *p* if `(*i==x)` or if `(*i!=x` and `++i` has property `p`). — *end example*]

[[**Remove Table 89: Input iterator requirements**]]

3 [*Note:* For input iterators, `a == b` does not imply `++a == ++b`. (Equality does not guarantee the substitution property or referential transparency.) Algorithms on input iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. ~~Value type T is not required to be an Assignable type (23.1).~~ These algorithms can be used with istreams as the source of the input data through the `istream_iterator` class. — *end note*]

```
reference operator*(X a);
```

4 *Requires:* `a` is dereferenceable.

5 *Remarks:* If `b` is a value of type `X`, `a == b` and `(a, b)` is in the domain of `==` then `*a` is equivalent to `*b`.

```
pointer operator->(X a);
```

6 *Requires:* `(*a).m` is well-defined

Effects: Equivalent to `(*a).m`.

24.1.2 Output iterators [lib.output.iterators]

1 A class or a built-in type `X` satisfies the requirements of an output iterator if ~~X is a CopyConstructible (20.1.3) and Assignable type (23.1) and also the following expressions are valid, as shown in Table 90~~meets the syntactic and semantic requirements of the OutputIterator or BasicOutputIterator concepts.

[[**Remove Table 90: Output iterator requirements**]]

2 [*Note:* The only valid use of an `operator*` is on the left side of the assignment statement. *Assignment through the same value of the iterator happens only once.* Algorithms on output iterators should never attempt to pass through the same iterator twice. They should be *single pass* algorithms. Equality and inequality might not be defined. Algorithms that take output iterators can be used with ostreams as the destination for placing data through the `ostream_iterator` class as well as with insert iterators and insert pointers. — *end note*]

3 The `OutputIterator` concept describes an output iterator that may permit output of many different value types.

```
concept OutputIterator<typename X, typename Value> : CopyConstructible<X> {
  typename value_type = Value;
  typename reference = X::reference;

  where Assignable<X> && SameType<Assignable<X>::result_type, X&>;

  where SameType<value_type, Value> &&
        Assignable<reference, value_type>;

  typename postincrement_result;
  where Dereferenceable<postincrement_result&> &&
        Convertible<postincrement_result, const X&> &&
        Assignable<Dereferenceable<postincrement_result&>::reference,
                   value_type>;

  reference operator*(X&);
  X& operator++(X& r);
  postincrement_result operator++(X& r, int);
};

X& operator++(X& r);
```

4 *Postcondition:* `&r == &++r`

```
postincrement_result operator++(X& r, int);
```

5 *Effects:* equivalent to

```
  { X tmp = r;
  ++r;
  return tmp; }
```

6 The `BasicOutputIterator` concept describes an output iterator that has one, fixed value type. Unlike `OutputIterator`, `BasicOutputIterator` is a part of the iterator refinement hierarchy.

```
concept BasicOutputIterator<typename X>
  : IteratorAssociatedTypes<X>, CopyConstructible<X>
{
  where Assignable<X> && SameType<Assignable<X>::result_type, X&>;

  where Assignable<reference, value_type>;

  typename postincrement_result = X;
  where Dereferenceable<postincrement_result&> &&
        Assignable<Dereferenceable<postincrement_result&>::reference,
                   value_type> &&
        Convertible<postincrement_result, const X&>;

  reference operator*(X&);
  X& operator++(X&);
  postincrement_result operator++(X&, int);
};

X& operator++(X& r);
```

7 *Postcondition:* `&r == &++r`

```
postincrement_result operator++(X& r, int);
```

8 *Effects:* equivalent to

```
  { X tmp = r;
  ++r;
  return tmp; }
```

9 Every `BasicOutputIterator` is an `OutputIterator` for value types `Assignable` to its `value_type`. [2)]

```
template<BasicOutputIterator X, typename Value>
where Assignable<value_type, Value>
concept_map OutputIterator<X, Value> {
  typedef Value                       value_type;
  typedef X::reference                reference;
  typedef X::postincrement_result     postincrement_result;
};
```

24.1.3 Forward iterators [lib.forward.iterators]

1 A class or a built-in type `X` satisfies the requirements of a forward iterator if ~~the following expressions are valid, as shown in Table 91.~~it meets the syntactic and semantic requirements of the ForwardIterator or MutableForwardIterator concepts.

[[**Remove Table 91: Forward iterator requirements.**]]

```
concept ForwardIterator<typename X> : InputIterator<X>, DefaultConstructible<X> {
  where Convertible<reference, const value_type&> &&
```

[2)] This allows algorithms specified with `OutputIterator` (the less restrictive concept) to work with iterators that have concept maps for the more common `BasicOutputIterator` concept.

```
        Convertible<postincrement_result, const X&>;
};

concept MutableForwardIterator<typename X> : ForwardIterator<X>, BasicOutputIterator<X> {
  where SameType<reference, value_type&> && SameType<pointer, value_type*>;
};
```

The `ForwardIterator` concept here provides slightly weaker requirements on the `reference` and `pointer` types than the associated requirements table in C++98, because we only require convertibility to `const value_type&` and `const value_type*` instead of exact type matches. This convertibility is crucial, because the `MutableForwardIterator` concept must be a refinement of the `ForwardIterator` concept. This refinement **does not work** if `ForwardIterator` says that `reference` is `const value_type&` and then `MutableForwardIterator` changes that to say that `reference` is `value_type&`.

2 [*Note:* The condition that `a == b` implies `++a == ++b` (which is not true for input and output iterators) and the removal of the restrictions on the number of the assignments through the iterator (which applies to output iterators) allows the use of multi-pass one-directional algorithms with forward iterators. — *end note*]

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

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

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

24.1.4 Bidirectional iterators [lib.bidirectional.iterators]

1 A class or a built-in type `X` satisfies the requirements of a bidirectional iterator if ~~, in addition to satisfying the requirements for forward iterators, the following expressions are valid as shown in Table 92.~~it meets the syntactic and semantic requirements of the BidirectionalIterator or MutableBidirectionalIterator concept.

[[**Remove Table 92: Bidirectional iterator requirements.**]]

```
concept BidirectionalIterator<typename X> :  ForwardIterator<X> {
  typename postdecrement_result;
  where Dereferenceable<postdecrement_result> &&
        Convertible<Dereferenceable<postdecrement_result>::reference,
                    value_type> &&
        Convertible<postdecrement_result, const X&>;

  X& operator--(X&);
  postdecrement_result operator--(X&, int);
};

concept MutableBidirectionalIterator<typename X>
  : BidirectionalIterator<X>, MutableForwardIterator<X> { };
```

2 [*Note:* Bidirectional iterators allow algorithms to move iterators backward as well as forward. — *end note*]

```
X& operator--(X& r);
```

3 *Precondition:* there exists `s` such that `r == ++s`.

4 *Postcondition:* `r` is dereferenceable.

5 *Effects:* `--(++r) == r`.
`--r == --s` implies `r == s`.
`&r == &--r`.

```
postdecrement_result operator--(X& r, int);
```

6 *Effects:* equivalent to

```
{ X tmp = r;
--r;
return tmp; }
```

24.1.5 Random access iterators [lib.random.access.iterators]

1 A class or a built-in type `X` satisfies the requirements of a random access iterator if ~~, in addition to satisfying the requirements for bidirectional iterators, the following expressions are valid as shown in Table 93.~~it meets the syntactic and semantic requirements of the RandomAccessIterator or MutableRandomAccessIterator concept.

```
concept RandomAccessIterator<typename X> : BidirectionalIterator<X>, LessThanComparable<X> {
  X& operator+=(X&, difference_type);
  X  operator+ (X, difference_type);
  X  operator+ (difference_type, X);
  X& operator-=(X&, difference_type);
  X  operator- (X, difference_type);

  difference_type operator-(X, X);
  reference operator[](X, difference_type);
};

concept MutableRandomAccessIterator<typename X>
  : RandomAccessIterator<X>, MutableBidirectionalIterator<X> { };
```

[[**Remove Table 93: Random access iterator requirements.**]]

```
X& operator+=(X&, difference_type);
```

2 *Effects:* equivalent to

```
{ difference_type m = n;
  if (m >= 0) while (m--) ++r;
  else while (m++) --r;
  return r; }
```

```
X operator+(X a, difference_type n);
X operator+(difference_type n, X a);
```

3 *Effects:* equivalent to

```
{ X tmp = a;
return tmp += n; }
```

4 *Postcondition:* `a + n == n + a`

```
X& operator-=(X& r, difference_type n);
```

Returns: `r += -n`

```
X operator-(X, difference_type);
```

Effects: equivalent to

```
{ X tmp = a;
  return tmp -= n; }
```

```
difference_type operator-(X a, X b);
```

Precondition: there exists a value `n` of `difference_type` such that `a + n == b`.

Effects: `b == a + (b - a)`

Returns: `(a < b) ?  distance(a,b) :  -distance(b,a)`

Pointers are mutable random access iterators with the following concept map

```
namespace std {
  template<typename T> concept_map MutableRandomAccessIterator<T*> {
    typedef T value_type;
    typedef std::ptrdiff_t difference_type;
    typedef T& reference;
    typedef T* pointer;
  };
}
```

and pointers to const are random access iterators

```
namespace std {
  template<typename T> concept_map RandomAccessIterator<const T*> {
    typedef T value_type;
    typedef std::ptrdiff_t difference_type;
    typedef const T& reference;
    typedef const T* pointer;
  };
}
```

[*Note:* If there is an additional pointer type `__far` such that the difference of two `__far` is of type `long`, an implementation may define

```
  template<class T> concept_map RandomAccessIterator<T __far*> {
    typedef long difference_type;
    typedef T value_type;
    typedef T __far* pointer;
    typedef T __far& reference;
  };
```

— end note]

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

Note: Update synopsis

24.3 Iterator primitives [lib.iterator.primitives]

1 To simplify the ~~task of defining iterators~~use of iterators and provide backward compatibility with previous C++ Standard Libraries, the library provides several classes and functions:

24.3.1 Deprecated Iterator traits [lib.iterator.traits]

1 ~~To implement algorithms only in terms of iterators, it is often necessary to determine the value and difference types that correspond to a particular iterator type. Accordingly, it is required that if~~Iterator traits provide an auxiliary mechanism for accessing the associated types of an iterator. If `Iterator` is the type of an iterator, the types

```
iterator_traits<Iterator>::difference_type
iterator_traits<Iterator>::value_type
iterator_traits<Iterator>::iterator_category
```

shall be defined as the iterator's difference type, value type and iterator category (24.3.3), respectively. In addition, the types

```
iterator_traits<Iterator>::reference
iterator_traits<Iterator>::pointer
```

shall be defined as the iterator's reference and pointer types, that is, for an iterator object `a`, the same type as the type of `*a` and `a->`, respectively. ~~In the case of an output iterator, the types~~

~~`iterator_traits<Iterator>::difference_type`~~
~~`iterator_traits<Iterator>::value_type`~~
~~`iterator_traits<Iterator>::reference`~~
~~`iterator_traits<Iterator>::pointer`~~

~~may be defined as void.~~

2 The template `iterator_traits<Iterator>` is defined as

```
namespace std {
  template<class Iterator> struct iterator_traits {
    typedef typename Iterator::difference_type difference_type;
    typedef typename Iterator::value_type value_type;
    typedef typename Iterator::pointer pointer;
    typedef typename Iterator::reference reference;
    typedef typename Iterator::iterator_category iterator_category;
  };
}
```

3 It ~~is~~may be specialized for pointers [3] as

[3] These specializations are redundant, because concept maps for pointers are provided for the corresponding iterator concepts. These concept maps will be used with the concept-based partial specializations of `iterator_traits`.

```
namespace std {
  template<class T> struct iterator_traits<T*> {
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef T* pointer;
    typedef T& reference;
    typedef random_access_iterator_tag iterator_category;
  };
}
```

and for pointers to const as

```
namespace std {
  template<class T> struct iterator_traits<const T*> {
    typedef ptrdiff_t difference_type;
    typedef T value_type;
    typedef const T* pointer;
    typedef const T& reference;
    typedef random_access_iterator_tag iterator_category;
  };
}
```

4 [*Note:* If there is an additional pointer type `__far` such that the difference of two `__far` is of type `long`, an implementation may define

```
  template<class T> struct iterator_traits<T __far*> {
    typedef long difference_type;
    typedef T value_type;
    typedef T __far* pointer;
    typedef T __far& reference;
    typedef random_access_iterator_tag iterator_category;
  };
```

— end note]

5 [*Example:* To implement a generic `reverse` function, a C++ program can do the following:

```
template <class BidirectionalIterator>
void reverse(BidirectionalIterator first, BidirectionalIterator last) {
   typename iterator_traits<BidirectionalIterator>::difference_type n =
         distance(first, last);
   --n;
   while(n > 0) {
       typename iterator_traits<BidirectionalIterator>::value_type
                tmp = *first;
       *first++ = *--last;
       *last = tmp;
       n -= 2;
   }
}
```

— *end example*]

6 `iterator_traits` is specialized for any type `Iterator` for which there is a concept map for any of the iterator concepts (24.1).[4)]

```
template<InputIterator Iterator> struct iterator_traits<Iterator> {
  typedef Iterator::difference_type           difference_type;
  typedef Iterator::value_type                value_type;
  typedef Iterator::pointer                   pointer;
  typedef Iterator::reference                 reference;
  typedef input_iterator_tag                  iterator_category;
};

template<BasicOutputIterator Iterator> struct iterator_traits<Iterator> {
  typedef Iterator::difference_type           difference_type;
  typedef Iterator::value_type                value_type;
  typedef Iterator::pointer                   pointer;
  typedef Iterator::reference                 reference;
  typedef output_iterator_tag                 iterator_category;
};

template<ForwardIterator Iterator> struct iterator_traits<Iterator> {
  typedef Iterator::difference_type           difference_type;
  typedef Iterator::value_type                value_type;
  typedef Iterator::pointer                   pointer;
  typedef Iterator::reference                 reference;
  typedef forward_iterator_tag                iterator_category;
};

template<BidirectionalIterator Iterator> struct iterator_traits<Iterator> {
  typedef Iterator::difference_type           difference_type;
  typedef Iterator::value_type                value_type;
  typedef Iterator::pointer                   pointer;
  typedef Iterator::reference                 reference;
  typedef bidirectional_iterator_tag          iterator_category;
};

template<RandomAccessIterator Iterator> struct iterator_traits<Iterator> {
  typedef Iterator::difference_type           difference_type;
  typedef Iterator::value_type                value_type;
  typedef Iterator::pointer                   pointer;
  typedef Iterator::reference                 reference;
  typedef random_access_iterator_iterator_tag iterator_category;
};
```

[4)]These specializations permit forward compatibility of iterators, allowing those iterators that provide only concept maps to be used through `iterator_traits`.

Note that we cannot support types that only have concept maps for `OutputIterator`, because we only have the iterator type (no value type). This is a (probably minor) problem for forward compatibility.

24.3.2 Deprecated Basic iterator [lib.iterator.basic]

We deprecated the basic `iterator` template because it isn't really the right way to specify iterators any more. Even when using this template, users should write concept maps so that (1) their iterators will work when `iterator_traits` and the backward-compatibility models go away, and (2) so that their iterators will be checked against the iterator concepts as early as possible.

1 The `iterator` template may be used as a base class to ease the definition of required types for new iterators.

```
namespace std {
  template<class Category, class T, class Distance = ptrdiff_t,
           class Pointer = T*, class Reference = T&>
  struct iterator {
        typedef T         value_type;
        typedef Distance  difference_type;
        typedef Pointer   pointer;
        typedef Reference reference;
        typedef Category  iterator_category;
  };
}
```

24.3.3 ~~Standard~~Deprecated iterator tags [lib.std.iterator.tags]

1 ~~It is often desirable for a function template specialization to find out what is the most specific category of its iterator argument, so that the function can select the most efficient algorithm at compile time. To facilitate this, the~~The library introduces *category tag* classes which are used as compile time tags ~~for algorithm selection.~~to distinguish the different iterator concepts when using the iterator_traits mechanism. They are: `input_iterator_tag`, `output_iterator_tag`, `forward_iterator_tag`, `bidirectional_iterator_tag` and `random_access_iterator_tag`. For every iterator of type `Iterator`, `iterator_traits<Iterator>::iterator_category` shall be defined to be the most specific category tag that describes the iterator's behavior.

```
namespace std {
  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 {};
}
```

2 **[[Remove this paragraph: It gives an example using `iterator_traits`, which we no longer encourage.]]**

24.3.4 Iterator operations [lib.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 <InputIterator Iter>
```

```
  void advance(Iter& i, 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);
```

We have made a minor change to the declaration of `advance`. In the existing standard, it has a second template parameter, `Distance`, which is the type of `n`. We have eliminated this parameter and instead used the `difference_type` iterator. This is clearly what was intended, but it may break backward compatibility in certain, very limited cases.

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

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

```
template<InputIterator Iter>
    Iter::difference_type
       distance(Iter first, 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`.

24.3.5 Iterator backward compatibility [lib.iterator.backward]

1 The library provides concept maps that allow iterators specified with `iterator_traits` to interoperate with algorithms that require iterator concepts.

2 The associated types `difference_type`, `value_type`, `pointer` and `reference` are given the same values as their counterparts in `iterator_traits`.

3 These concept maps shall only be defined when the `iterator_traits` specialization contains the nested types `difference_type`, `value_type`, `pointer`, `reference` and `iterator_category`.

[*Example:* The following example is well-formed. The backward-compatbility concept map for `InputIterator` does not match because `iterator_traits<int>` fails to provide the required nested types.

```
template<Integral T> void f(T);
template<InputIterator T> void f(T);

void g(int x) {
  f(x); // okay
}
```

— end example]

4 The library shall provide a concept map `InputIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `input_iterator_tag`.

5 The library shall provide a concept map `ForwardIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `forward_iterator_tag`.

6 The library shall provide a concept map `MutableForwardIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `forward_iterator_tag` for which the `reference` type is equal to `value_type&` and `value_type` is `Assignable`.

7 The library shall provide a concept map `BidirectionalIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `bidirectional_iterator_tag`.

8 The library shall provide a concept map `MutableBidirectionalIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `bidirectional_iterator_tag` for which the `reference` type is equal to `value_type&` and `value_type` is `Assignable`.

9 The library shall provide a concept map `RandomAccessIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `random_access_iterator_tag`.

10 The library shall provide a concept map `MutableRandomAccessIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `random_access_iterator_tag` for which the `reference` type is equal to `value_type&` and `value_type` is `Assignable`.

Note that we do not automatically provide concept maps that adapt output iterators, although we believe it is possible.

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.