Iterator Concepts for the C++0x Standard Library (Revision 2)

Douglas Gregor, Jeremy Siek and Andrew Lumsdaine
dgregor@osl.iu.edu, jeremy.siek@colorado.edu, lums@osl.iu.edu

Document number: N2624=08-0134
Revises document number: N2570=08-0080
Date: 2008-05-19
Project: Programming Language C++, Library Working Group
Reply-to: Douglas Gregor <dgregor@osl.iu.edu>

Introduction

This document proposes new iterator concepts in the C++0x Standard Library. It describes a new header `<iterator_concepts>` that contains these concepts, along with concept maps and `iterator_traits` specializations that provide backward compatibility for existing iterators and generic algorithms.

Within the proposed wording, text that has been added will be presented in blue and underlined when possible. Text that has been removed will be presented in red,~~with strike-through when possible~~. Non-editorial changes from the previous wording are highlighted in green.

Purely editorial comments will be written in a separate, shaded box.

Changes from N2570

— The `value_type` of an iterator is an `ObjectType`.

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>`
D.10 Iterator primitives	`<iterator>`
[predef.iterators] Predefined iterators	
[stream.iterators] Stream iterators	

The following section has been renamed from “Iterator requirements” to “Iterator concepts”.

24.1 Iterator concepts [iterator.concepts]

1 The `<iterator_concepts>` header describes requirements on iterators.

Header `<iterator_concepts>` synopsis

```
namespace std {
  // 24.1.1, input iterators:
  concept InputIterator<typename X> see below;

  // 24.1.2, output iterators:
  concept OutputIterator<typename X, typename Value> see below;
  concept BasicOutputIterator<typename X> see below;

  template<BasicOutputIterator X, typenameCopyAssignable Value>
    requires HasCopyAssign<X::reference, Value>
    concept_map OutputIterator<X, Value> see below;

  // 24.1.3, forward iterators:
  concept ForwardIterator<typename X> see below;
  concept MutableForwardIterator<typename X> see below;

  // 24.1.4, bidirectional iterators:
```

```
    concept BidirectionalIterator<typename X> see below;
    concept MutableBidirectionalIterator<typename X> see below;

    // 24.1.5, random access iterators:
    concept RandomAccessIterator<typename X> see below;
    concept MutableRandomAccessIterator<typename X> see below;
    template<ObjectType T> concept_map MutableRandomAccessIterator<T*> see below;
    template<ObjectType T> concept_map RandomAccessIterator<const T*> see below;

    // 24.1.6, swappable iterators:
    concept SwappableIterator<typename X> see below;
  }
```

2 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.

3 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~~nine 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, and swappable iterators, as shown in Table 2.

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

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

4 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 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.

5 ~~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.

6 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.

7 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.

8 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.

9 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.~~

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

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

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

24.1.1 Input iterators [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 95.~~it meets the syntactic and semantic requirements of the `InputIterator` concept.

```
concept InputIterator<typename X> : Semiregular<X>, EqualityComparable<X> {
  ~~typename~~ObjectType value_type = typename X::value_type;
  MoveConstructible reference = typename X::reference;
  MoveConstructible pointer = typename X::pointer;

  SignedIntegralLike difference_type = typename X::difference_type;

  requires IntegralType<difference_type>
        && Convertible<reference, value_type const &>;
        && Convertible<pointer, const value_type*>;

  MoveConstructible postincrement_result;
  requires Dereferenceable<postincrement_result> &&
```

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

```
            Convertible<Dereferenceable<postincrement_result>::reference, value_type>;

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

2 ~~In Table 95~~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 96: 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 const& a);
```

4 *Requires:* `a` is dereferenceable.

5 *Returns:* the value referenced by the iterator

6 *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 const& a);
```

7 *Returns:* a pointer to the value referenced by the iterator

```
bool operator==(X const& a, X const& b); // inherited from EqualityComparable<X>
```

8 If two iterators `a` and `b` of the same type are equal, then either `a` and `b` are both dereferenceable or else neither is dereferenceable.

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

9 *Precondition:* `r` is dereferenceable

10 *Postcondition:* `r` is dereferenceable or `r` is past-the-end. Any copies of the previous value of `r` are no longer required either to be dereferenceable or in the domain of `==`.

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

11 *Effects:* equivalent to `{ T tmp = *r; ++r; return tmp; }`.

24.1.2 Output iterators [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 96~~meets the syntactic and

semantic requirements of the `OutputIterator` or `BasicOutputIterator` concepts.

[[Remove Table 97: 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 reference;
  requires HasCopyAssign<reference, Value> \addedCC&& CopyAssignable<Value>;

  typename postincrement_result;
  requires Dereferenceable<postincrement_result> &&
           Convertible<postincrement_result, const X&> &&
           HasCopyAssign<Dereferenceable<postincrement_result>::reference, Value>;

  reference operator*(X&);
  X& operator++(X&);
  postincrement_result operator++(X&, 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> : CopyConstructible<X> {
  typenameObjectType value_type = typename X::value_type;
  MoveConstructible reference = typename X::reference;

  requires HasCopyAssign<reference, value_type> && CopyAssignable<value_type>;

  typename postincrement_result;
  requires Dereferenceable<postincrement_result> &&
           HasCopyAssign<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 copy-assignable to its `reference` type. [2)]

```
template<BasicOutputIterator X, typenameCopyAssignable Value>
requires CopyAssignable<X::reference, Value>
concept_map OutputIterator<X, Value> {
  typedef X::reference              reference;
  typedef X::postincrement_result   postincrement_result;
}
```

24.1.3 Forward iterators [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 97.~~it meets the syntactic and semantic requirements of the ForwardIterator or MutableForwardIterator concepts.

[[Remove Table 98: Forward iterator requirements.]]

```
concept ForwardIterator<typename X> : InputIterator<X>, Regular<X> {
  requires Convertible<postincrement_result, const X&>;

  axiom MultiPass(X a, X b) {
    if (a == b) *a == *b;
    if (a == b) ++a == ++b;
    &a == &++a;
  }
}
```

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

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.

The `ForwardIterator` concept here provides weaker requirements on the `reference` and `pointer` types than the associated requirements table in C++03, because these types do not need do not need to be true references or pointers to `value_type`. This change weakens the concept, meaning that C++03 iterators (which meet the stronger requirements) still meet these requirements, but algorithms that relied on these stricter requirements will no longer work just with the iterator requirements: they will need to specify true references or pointers as additional requirements. By weakening the requirements, however, we permit proxy iterators to model the forward, bidirectional, and random access iterator concepts.

```
X::X(); // inherited from Regular<X>
```

2 *Note:* the constructed object might have a singular value.

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

3 ~~*Requires:* If two iterators `a` and `b` of the same type are both dereferenceable, then `a == b` if and only if `*a` and `*b` are the same object.~~

4 [*Note:* The ~~condition~~axiom 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*]

24.1.4 Bidirectional iterators [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 98.~~it meets the syntactic and semantic requirements of the BidirectionalIterator or MutableBidirectionalIterator concept.

[[**Remove Table 99: Bidirectional iterator requirements.**]]

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

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

  axiom BackwardTraversal(X a, X b) {
    --(++a) == a;
    if (--a == --b) a == b;
    &a == &--a;
  }
}

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.

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

5 *Effects:* equivalent to

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

24.1.5 Random access iterators [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 99.~~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 const&, difference_type);
  X  operator+ (difference_type, Xconst&);
  X& operator-=(X&, difference_type);
  X  operator- (X const&, difference_type);

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

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

[[**Remove Table 100: Random access iterator requirements.**]]

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

2 *Effects:* equivalent to

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

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

3 *Effects:* equivalent to

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

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

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

5 *Returns:* `r += -n`

```
X operator-(X const& a, difference_type n);
```

6 *Effects:* equivalent to

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

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

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

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

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

10 Pointers are mutable random access iterators with the following concept map

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

11 [*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 <ObjectType T> concept_map MutableRandomAccessIterator<T __far*> {
  typedef long difference_type;
  typedef T value_type;
  typedef T __far* pointer;
  typedef T __far& reference;
}

template <ObjectType T> concept_map RandomAccessIterator<const T __far*> {
  typedef long difference_type;
```

```
    typedef T value_type;
    typedef const T __far* pointer;
    typedef const T __far& reference;
  }
```

— end note]

Add the following new section

24.1.6 Swappable iterators [swappable.iterators]

1 A class or built-in type `X` satisfies the requirements of a swappable iterator if it meets the syntactic and semantic requirements of the `SwappableIterator` concept.

```
auto concept SwappableIterator<typename X> {
  void iter_swap(X const&, X const&);
}
```

```
void iter_swap(X const& a, X  const& b);
```

2 Swaps the elements referenced by iterators `a` and `b`.

Appendix D (normative) Compatibility features [depr]

D.10 Iterator primitives [depr.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.

2 The `iterator_traits` and supporting facilities described in this section are deprecated. [*Note:* the iterator concepts (24.1) provide the equivalent functionality using the concept mechanism. *— end note*]

D.10.1 Iterator traits [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.~~

6 `iterator_traits` is specialized for any type `Iterator` for which there is a concept map for any of the iterator concepts (24.1) and `Iterator` meets the requirements stated in the corresponding requirements table of ISO/IEC 14882:2003. [*Note:* these specializations permit forward compatibility of iterators, allowing those iterators that provide only concept maps to be used through `iterator_traits`. They can be implemented via class template partial specializations such as the following.

```
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 void                                difference_type;
  typedef Iterator::value_type                value_type;
  typedef void                                pointer;
  typedef Iterator::reference                 reference;
  typedef output_iterator_tag                 iterator_category;
};
```

— end note]

D.10.2 Basic iterator [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;
  };
}
```

D.10.3 Standard iterator tags [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.**]]

D.10.4 Iterator backward compatibility [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 `OutputIterator` for any type `Iterator` with `iterator_traits<Iterator>::iterator_category` convertible to `output_iterator_tag`. [*Note:* the `reference` type of the `OutputIterator` must be deduced, because `iterator_traits` specifies that it will be `void`. *— end note*]

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

7 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 copy-assignable from the `value_type`.

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

9 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 copy-assignable from the `value_type`.

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

11 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 copy-assignable from the `value_type`.

Acknowledgments

Thanks to Beman Dawes for alerting us to omissions from the iterator concepts and Daniel Krügler for many helpful comments.