Iterator Concepts for the C++0x Standard Library (Revision 5)

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

Document number: N2758=08-0268
Revises document number: N2739=08-0249
Date: 2008-09-19
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~~.

Purely editorial comments will be written in a separate, shaded box. These comments are not intended to be included in the working paper.

About the new iterator concept taxonomy

At the Library Working Group's request, we sought to determine whether we could eliminate the mutable iterator concepts from the iterator taxonomy. The observation made in Sophia-Antipolis was that the mutable iterator concepts were used very rarely, and in those places where they were used, we were able to discern simpler requirements. As a result of this investigation, we have eliminated the mutable iterator concepts and designed an improved iterator taxonomy that better describes the various kinds of iterators usable with the C++0x standard library.

The fundamental problem with the mutable iterator concepts is that they were initially ill-defined within C++98/03, mentioned only casually as iterators for which one could write a value to the result of dereferencing an iterator. This requirement was taken to mean a `CopyAssignable` requirement (which works reasonably well for C++98/03 forward iterators and above), but that fails in two important ways for C++0x:

— We can now construct sequences with types that are move-assignable but not copy-assignable. If we merely change the mutable iterator requirement to `MoveAssignable`, our definition of mutable iterator has changed from C++98/03. Besides, even this is incorrect: one can mutate values that aren't even move-assignable, e.g., by swapping values or acting on lvalues.

— As part of improving the iterator concepts, we have intended to better support proxy iterators (like the infamous `vector<bool>` iterator, although there exist many more important examples of such iterators) throughout the C++0x standard library. The simple notion of a copy-assignable value type does not match with a proxy reference that supports writing.

Thus, the most important realization is that there are multiple forms of mutability used within the C++0x standard library, and that these mutations involve both the value type of the iterator (e.g., the type actually stored in the container the iterator references) and the reference type of the iterator (which may be an lvalue reference, rvalue reference, or a proxy class). The new iterator taxonomy captures these forms of mutability through two iterator concepts: `OutputIterator` and `ShuffleIterator`.

The `OutputIterator` concept (24.1.2) is a faithful representation of a C++98/03 output iterator. Output iterators are an odd kind of iterator, because it does not make sense to say that a type `X` is an output iterator. Rather, one must say that `X` is an output iterator *for a value type T*. Moreover, a given type `X` can be an output iterator for a whole family of types, e.g., all types that can be printed, and can permit specific parameter-passing conventions. For example, `X` could support writing only rvalues of type `T`, or both lvalues and rvalues of type `T`. Thus, the `OutputIterator` concept is a two-parameter concept, one parameter for `X` and another for `T` (called `Value`). This is not new; however, the updated iterator taxonomy encodes the parameter-passing convention into the `Value` template parameter, so that the user of the output iterator can distinguish between writing lvalues and writing rvalues. For example, the `copy` algorithm provides the `reference` type of the input iterator as the output type of the output iterator:

```
template<InputIterator InIter,
         OutputIterator<auto, InIter::reference> OutIter>
  inline OutIter
  copy(InIter first, InIter last, OutIter result)
  {
    for (; first != last; ++result, ++first)
      *result = *first;
    return result;
  }
```

With this scheme, a typical input iterator that returns an lvalue reference will pass that lvalue reference on to the output iterator (as in C++98/03). More interesting, however, is when the input iterator is actually a `move_iterator`, whose reference type is an rvalue reference. In this case, we're writing rvalue-references to the output iterator, and therefore moving values from the input to the output sequence. The use of the `reference` type as the output type for the output iterator also copes with the transfer of values via proxies.

We have already noted that a single type `X` can be an output iterator for multiple, different value types. However, further study of the standard library algorithms illustrates that a type parameter might meet the `OutputIterator` requirements in multiple ways *within a single algorithm*. For example, in the `replace_copy` algorithm the `OutIter` parameter acts as an output iterator for both the input iterator's reference type (allowing moves rather than copies) and the type of the replacement value:

```
template<InputIterator InIter, typename OutIter, typename _Tp>
  requires OutputIterator<OutIter, InIter::reference>
```

```
      && OutputIterator<OutIter, const _Tp&>
      && HasEqualTo<InIter::value_type, _Tp>
OutIter
replace_copy(InIter first, InIter last,
             OutIter result,
             const _Tp& old_value, const _Tp& new_value)
{
  for ( ; first != last; ++first, ++result)
    if (*first == old_value)
      *result = new_value;
    else
      *result = *first;
  return result;
}
```

This formulation of `replace_copy` uncovered an interesting issue. Since both of the `OutputIterator` requirements have different argument types, each contains dereference (`operator*`) and increment (`operator+`) operators. Through concept maps, it is conceivable (however unlikely) that these operators could be different for one requirement than the other, and therefore each use of `*` or `++` that applies to an instance of the `OutIter` type within `replace_copy` returns an ambiguity. The ambiguity is a result of incomplete concept analysis, and was solved by refactoring the requirements of `OutputIterator` into two concepts: `Iterator<X>`, which provides the syntax of `operator*` and `operator++` (and is also refined by the `InputIterator<X>` concept), and the `OutputIterator<X, Value>` concept, which adds assignability requirements from `Value` to the reference type of the output iterator. Thus, the requirements on `replace_copy` now say that there is only one `operator*`, but the reference type that it returns can be used in multiple, different ways. Moreover, we now have an actual root to our iterator hierarchy, the `Iterator<X>` concept, which provides only increment and dereference—the basics of moving through a sequence of values—but does not provide any read or write capabilities.

The `ShuffleIterator` concept (24.1.6) is a new kind of iterator that captures the requirements needed to shuffle values within a sequence using moves and swaps. `ShuffleIterator` allows one to move-construct or move-assign from an element in the sequence into a variable of the iterator's value type. This permits, for example, the pivot element in a quicksort to be extracted from the sequence and placed into a variable. Additionally, values can be move-assigned from a variable of the iterator's value type (e.g., moving the pivot back from the temporary into the sorted sequence at the right time). This concept is the proxy-aware conceptualization of the Swappable+MoveConstructible+MoveAssignable set of concepts from the reflector discussion starting with c++std-lib-21212. The `ShuffleIterator` concept is used sparingly, in those cases where the sequence cannot be efficiently reordered within simple swap operations.

Changes to the iterator taxonomy should not be taken lightly; even the apparently simple iterators in C++98/03 turned out to be surprisingly complicated, and have resulted in numerous defect reports and several attempts at revisions. C++0x iterators are decidedly more complex, due to the introduction of rvalue references and the desire to provide support for proxy iterators throughout the standard library. To verify the iterator concepts presented in this document, we have fully implemented these concepts in ConceptGCC, applied them to nearly every algorithm in the standard library, and tested the result against the full libstdc++ test suite to ensure backward compatibility with existing iterators. Indeed, many of the observations that drove this refactoring came from implementation experience: the `operator*` ambiguity, for example, was initially detected by ConceptGCC.

Changes from N2739

— Moved the dereferenceability requirement on `operator*` from the `InputIterator` concept to the `Iterator` concept, since it applies to all iterators.

— Translated the `BackwardTraversal` axiom of the `BidirectionalIterator` concept back into normative text, since we can't easily express the preconditions of the operations used.

— Fixed the associated function default implementations in `RandomAccessIterator`.

— Defined the term "iterator category" within the (deprecated) iterator traits section, so that others of these deprecated constructs can refer to iterator categories. Thanks to Alisdair Meredith for noting this omission.

— Made `iterator_traits` SFINAE-safe (D.10.1), which is important for backward compatibility. In particular, the concept maps that map C++03 iterators into C++0x iterators (by querying `iterator_traits`) need to be robust against looking at types that don't have all of the nested types, e.g., the instantiation of `iterator_traits<int>` should produce an empty class, not an error.

— Made a `ShuffleIterator` a `ForwardIterator`, and its value type both `MoveAssignable` and `Swappable`.

— Fixed the precondition for `RandomAccessIterator`'s `operator-`, from `a + n == b` to `a == b + n`.

Proposed Wording

Issues resolved by concepts

The following LWG are resolved by concepts. These issues should be resolved as NAD following the application of this proposal to the wording paper:

Issue 299. Incorrect return types for iterator dereference. Concepts specify precise return types for the iterator operations, including `operator[]`.

Issue 258. 24.1.5 contains unintented limitation for operator-. Concepts now specify that the difference type of an iterator is a signed integral type.

Issue 484. Convertible to T. With concepts, the iterator requirements also specify "convertible to T", and this conversion will automatically be used within constrained templates as necessary, so that the overload that will be selected becomes clear from the requirements of the template.

Issue 742. Enabling swap for proxy iterators. The concepts proposal provides a two-parameter swap that is available when `swap(w, v)` is valid or when the types of `w` and `v` are the same and that type is `MoveAssignable` and `MoveConstructible`, per the `std::swap` algorithm. The use of this `HasSwap` concept in the iterator concepts and in algorithms makes proxy iterators viable throughout the standard library.

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>`
?? Predefined 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 {
  concept Iterator<typename X> see below;

  // 24.1.1, input iterators:
  concept InputIterator<typename X> see below;

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

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

  // 24.1.4, bidirectional iterators:
  concept BidirectionalIterator<typename X> see below;

  // 24.1.5, random access iterators:
  concept RandomAccessIterator<typename X> see below;
```

```
  template<ObjectType T> concept_map RandomAccessIterator<T*> see below;
  template<ObjectType T> concept_map RandomAccessIterator<const T*> see below;

  // 24.1.6, shuffle iterators:
  auto concept ShuffleIterator<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 iterators meet the requirements of the `Iterator` concept. 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~~several iterator concepts, according to the operations defined on them: *input iterators*, *output iterators*, *forward iterators*, *bidirectional iterators*, ~~and~~ *random access iterators*, and shuffle iterators, as shown in Table 2.

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

Random Access	→ **Bidirectional**	→ **Forward**	→ **Input**	→ **Iterator**
		↑		↑
		Shuffle	→	**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. 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.~~ Iterators that meet the requirements of the `OutputIterator` concept are called mutable iterators. 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.~~

```
concept Iterator<typename X> : Semiregular<X> {
  MoveConstructible reference = typename X::reference;
  MoveConstructible postincrement_result;

  requires HasDereference<postincrement_result>;

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

13 The `Iterator` concept forms the basis of the iterator concept taxonomy, and every iterator meets the requirements of the `Iterator` concept. This concept specifies operations for dereferencing and incrementing the iterator, but provides no way to manipulate values. Most algorithms will require addition operations to read (24.1.1) or write (24.1.2) values, or to provide a richer set of iterator movements (24.1.3, 24.1.4, 24.1.5).

Of particular interest in this concept is the dereference operator, which accepts an rvalue reference to an iterator. This permits non-`const` lvalues and rvalues of iterators to be dereferenced, but it represents a minor break from C++98/03 where one could dereference a const iterator (not an iterator-to-const; those are unaffected). We expect the impact to be minimal, given that one cannot increment const iterators, and if there were an algorithm that dereferences a const iterator, it could just make a non-const copy of the iterator to dereference. We have verified this change with ConceptGCC and found no ill effects.

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

14 *Requires:* `a` is dereferenceable.

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

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

15 *Effects:* equivalent to `{ X tmp = r; ++r; return tmp; }`.

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> : Iterator<X>, EqualityComparable<X> {
  ObjectType value_type = typename X::value_type;
  MoveConstructible pointer = typename X::pointer;

  SignedIntegralLike difference_type = typename X::difference_type;

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

  requires Convertible<HasDereference<postincrement_result>::result_type, const value_type&>;

  pointer operator->(const X&);
}
```

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&& a); // inherited from Iterator<X>
```

4 *Returns:* the value referenced by the iterator

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->(const X& a);
```

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

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

7 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);
```

8 *Precondition:* `r` is dereferenceable

9 *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 `==`.

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` concept.

[[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.

```
auto concept OutputIterator<typename X, typename Value> {
  requires Iterator<X>;

  typename reference = Iterator<X>::reference;
  typename postincrement_result = Iterator<X>::postincrement_result;
  requires SameType<reference, Iterator<X>::reference>
        && SameType<postincrement_result, Iterator<X>::postincrement_result>
        && Convertible<postincrement_result, const X&>
        && HasAssign<reference, Value>
        && HasAssign<HasDereference<postincrement_result>::result_type, Value>;
}
```

4 [*Note:* Any iterator that meets the additional requirements specified by `OutputIterator` for a given `Value` type is considered an output iterator. — *end note*]

```
X& operator++(X& r); // from Iterator<X>
```

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

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 concept.

[[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;
  }
}
```

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 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.

3 [*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*]

`X& operator++(X& r);` *// inherited from InputIterator<X>*

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

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 concept.

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

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

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

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 *Requires:* `--(++r) == r` and, given lvalues `a` and `b` of type `X`, `--a == --b` implies `a == b`

5 *Postcondition:* `r` is dereferenceable. `&r == &--r`.

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

6 *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` concept.

```
concept RandomAccessIterator<typename X> : BidirectionalIterator<X>, LessThanComparable<X> {
  MoveConstructible subscript_reference;
  requires Convertible<subscript_reference, const value_type&>;

  X& operator+=(X&, difference_type);
  X  operator+ (const X& x, difference_type n) { X tmp(x); tmp += n; return tmp; }
  X  operator+ (difference_type n, const X& x) { X tmp(x); tmp += n; return tmp; }
  X& operator-=(X&, difference_type);
  X  operator- (const X& x, difference_type n) { X tmp(x); tmp -= n; return tmp; }

  difference_type operator-(const X&, const X&);
  subscript_reference operator[](const X& x, difference_type n);
}
```

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

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

2 *Effects:* equivalent to

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

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

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

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

6 *Effects:* equivalent to

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

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

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

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

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

```
subscript_reference operator[](const X& x, difference_type n);
```

10 *Requires:* `(const value_type&)x[n]` is equivalent to `*(x + n)`.

11 Pointers are random access iterators with the following concept map

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

12 [*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 RandomAccessIterator<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]

24.1.6 Shuffle iterators [shuffle.iterators]

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

```
auto concept ShuffleIterator<typename X> {
  requires ForwardIterator<X>
        && OutputIterator<X, RvalueOf<ForwardIterator<X>::value_type>::type>
        && OutputIterator<X, RvalueOf<ForwardIterator<X>::reference>::type>
        && Constructible<ForwardIterator<X>::value_type,
                         RvalueOf<ForwardIterator<X>::reference>::type>
        && MoveConstructible<ForwardIterator<X>::value_type>
        && MoveAssignable<ForwardIterator<X>::value_type>
        && Swappable<ForwardIterator<X>::value_type>
        && HasAssign<ForwardIterator<X>::value_type,
                     RvalueOf<ForwardIterator<X>::reference>::type>
        && HasSwap<ForwardIterator<X>::reference, ForwardIterator<X>::reference>;
}
```

2 A shuffle iterator is a form of forward and output iterator that allows values to be moved into or out of a sequence, along with permitting efficient swapping of values within the sequence. Shuffle iterators are typically used in algorithms that need to rearrange the elements within a sequence in a way that cannot be performed efficiently with swaps alone.

3 [*Note:* Any iterator that meets the additional requirements specified by `ShuffleIterator` is considered a shuffle iterator. *— end note*]

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 `Iter~~ator~~` is the type of an iterator, the types

```
iterator_traits<Iter~~ator~~>::difference_type
iterator_traits<Iter~~ator~~>::value_type
iterator_traits<Iter~~ator~~>::iterator_category
```

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

```
iterator_traits<Iter~~ator~~>::reference
iterator_traits<Iter~~ator~~>::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<Iter~~ator~~>::difference_type
iterator_traits<Iter~~ator~~>::value_type
iterator_traits<Iter~~ator~~>::reference
iterator_traits<Iter~~ator~~>::pointer
```

may be defined as `void`.

Add the following new paragraph

2 The *category* of an iterator roughly describes which of the iterator concepts (24.1) the iterator satisfies. Iterator categories refer to iterators as defined by ISO/IEC 14882:2003, and can be one of *input iterator*, *output iterator*, *forward iterator*, *bidirectional iterator*, or *random access iterator*.

2 ~~The~~If the type `Iter` has nested types `difference_type`, `value_type`, `pointer`, `reference`, and `iterator_category`, then the template `iterator_traits<Iter~~ator~~>` is defined as

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

otherwise, it is defined as

```
namespace std {
  template<class Iter~~ator~~> struct iterator_traits { };
}
```

The existing paragraphs 3–5 of this section are unchanged. Add a new paragraph at the end of this section:

7 For each iterator category, a partial specialization of the `iterator_traits` class template provide appropriate type definitions for programs that use the deprecated iterator traits mechanism. These partial specializations provide backward compatibility for unconstrained templates using iterators as specified by the corresponding requirements tables of ISO/IEC 14882:2003.

```
  concept IsReference<typename T> { } // exposition only
  template<typename T> concept_map IsReference<T&> { }

  concept IsPointer<typename T> { } // exposition only
  template<typename T> concept_map IsPointer<T*> { }

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

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

```
};

template<ForwardIterator Iter>
  requires IsReference<Iter::reference> && IsPointer<Iter::pointer>
  struct iterator_traits<Iter> {
    typedef Iter::difference_type           difference_type;
    typedef Iter::value_type                value_type;
    typedef Iter::pointer                   pointer;
    typedef Iter::reference                 reference;
    typedef forward_iterator_tag            iterator_category;
  };

template<BidirectionalIterator Iter>
  requires IsReference<Iter::reference> && IsPointer<Iter::pointer>
  struct iterator_traits<Iter> {
    typedef Iter::difference_type           difference_type;
    typedef Iter::value_type                value_type;
    typedef Iter::pointer                   pointer;
    typedef Iter::reference                 reference;
    typedef bidirectional_iterator_tag      iterator_category;
  };

template<RandomAccessIterator Iter>
  requires IsReference<Iter::reference> && IsPointer<Iter::pointer>
  struct iterator_traits<Iter> {
    typedef Iter::difference_type           difference_type;
    typedef Iter::value_type                value_type;
    typedef Iter::pointer                   pointer;
    typedef Iter::reference                 reference;
    typedef random_access_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 `Iter@~~ator~~@`, `iterator_traits<Iter@~~ator~~@>::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. [*Example:*

```
struct random_iterator
{
  typedef std::input_iterator_tag iterator_category;
  typedef int                     value_type;
  typedef int                     difference_type;
  typedef int*                    pointer;
  typedef int                     reference;

  random_iterator(int remaining = 0) : remaining(remaining) { }

  int operator*() const { return std::rand(); }
  int* operator->() const { return 0; }

  random_iterator& operator++() { --remaining; return *this; }

  random_iterator operator++(int) {
    random_iterator tmp(*this); ++(*this); return tmp;
  }

  int remaining;
```

```
  friend bool
  operator==(const random_iterator& i, const random_iterator& j)
  {
    return i.remaining == j.remaining;
  }

  friend bool
  operator!=(const random_iterator& i, const random_iterator& j)
  {
    return i.remaining != j.remaining;
  }
};

void f(random_iterator i, random_iterator j) {
  std::copy(i, j, std::ostream_iterator<int>(std::cout, " ")); // okay: standard library produces concept
                                                               // map InputIterator<random_iterator>
}
```

— end example]

2 For all iterator types except output iterators, the associated types `difference_type`, `value_type`, `pointer` and `reference` are given the same values as their counterparts in `iterator_traits`. For output iterators, the `reference` type is deduced from the type of the output iterator's dereference operator.

3 When the `iterator_traits` specialization contains the nested types `difference_type`, `value_type`, `pointer`, `reference` and `iterator_category`, the `iterator_traits` specialization is considered to be *valid*.

[*Example:* The following example is well-formed. The backward-compatbility concept map for `InputIterator` does not match because `iterator_traits<int>` is not valid.

```
template<IntegralLike 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 `Iterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iter>::iterator_category` convertible to `output_iterator_tag`, and that meets the syntactic requirements of the `Iterator` concept.

5 The library shall provide a concept map `InputIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iter>::iterator_category` convertible to `input_iterator_tag`, and that meets the syntactic requirements of the `InputIterator` concept.

6 The library shall provide a concept map `ForwardIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iterator>::iterator_category` convertible to `forward_iterator_tag`, and that meets the syntactic requirements of the `ForwardIterator` concept.

7 The library shall provide a concept map `BidirectionalIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iterator>::iterator_category` convertible to `bidirectional_iterator_tag`, and that meets the syntactic requirements of the `BidirectionalIterator` concept.

8 The library shall provide a concept map `RandomAccessIterator<Iter>` for any type `Iter` with a valid `iterator_traits<Iter>`, an `iterator_traits<Iterator>::iterator_category` convertible to `random_access_iterator_tag`, and that meets the syntactic requirements of the `RandomAccessIterator` concept.

Acknowledgments

Thanks to Beman Dawes for alerting us to omissions from the iterator concepts and Daniel Krügler for many helpful comments. Both Mat Marcus and Jaakko Järvi were particularly helpful in the design of the new iterator taxonomy.