Iterator Facade and Adaptor
abstract: We propose a set of class templates that help programmers build standard-conforming iterators, both from scratch and by adapting other iterators.

Table of Contents

Motivation
Impact on the Standard
Design
  Iterator Concepts
  Interoperability
  Iterator Facade
    Usage
    Iterator Core Access
      operator[]
      operator->
  Iterator Adaptor
  Specialized Adaptors

Proposed Text

Header <iterator_helper> synopsis [lib.iterator.helper.synopsis]

Iterator facade [lib.iterator.facade]
  Class template iterator_facade
  iterator_facade Requirements
  iterator_facade operations

Iterator adaptor [lib.iterator.adaptor]
  Class template iterator_adaptor
  iterator_adaptor requirements
  iterator_adaptor base class parameters
  iterator_adaptor public operations
  iterator_adaptor protected member functions
  iterator_adaptor private member functions

Specialized adaptors [lib.iterator.special.adaptors]
  Indirect iterator
Motivation

Iterators play an important role in modern C++ programming. The iterator is the central abstraction of the algorithms of the Standard Library, allowing algorithms to be re-used in a wide variety of contexts. The C++ Standard Library contains a wide variety of useful iterators. Every one of the standard containers comes with constant and mutable iterators [2], and also reverse versions of those same iterators which traverse the container in the opposite direction. The Standard also supplies `istream_iterator` and `ostream_iterator` for reading from and writing to streams, `insert_iterator`, `front_insert_iterator` and `back_insert_iterator` for inserting elements into containers, and `raw_storage_iterator` for initializing raw memory [7].

Despite the many iterators supplied by the Standard Library, obvious and useful iterators are missing, and creating new iterator types is still a common task for C++ programmers. The literature documents several of these, for example `line_iterator` [3] and `Constant_iterator` [9]. The iterator abstraction is so powerful that we expect programmers will always need to invent new iterator types.
Although it is easy to create iterators that *almost* conform to the standard, the iterator requirements contain subtleties which can make creating an iterator which *actually* conforms quite difficult. Further, the iterator interface is rich, containing many operators that are technically redundant and tedious to implement. To automate the repetitive work of constructing iterators, we propose *iterator_facade*, an iterator base class template which provides the rich interface of standard iterators and delegates its implementation to member functions of the derived class. In addition to reducing the amount of code necessary to create an iterator, the *iterator_facade* also provides compile-time error detection. Iterator implementation mistakes that often go unnoticed are turned into compile-time errors because the derived class implementation must match the expectations of the *iterator_facade*.

A common pattern of iterator construction is the adaptation of one iterator to form a new one. The functionality of an iterator is composed of four orthogonal aspects: traversal, indirection, equality comparison and distance measurement. Adapting an old iterator to create a new one often saves work because one can reuse one aspect of functionality while redefining the other. For example, the Standard provides *reverse_iterator*, which adapts any Bidirectional Iterator by inverting its direction of traversal. As with plain iterators, iterator adaptors defined outside the Standard have become commonplace in the literature:

- The iterators of the View Template Library[14], which adapts containers, are themselves adaptors over the underlying iterators.
- Smart iterators [5] adapt an iterator’s dereferencing behavior by applying a function object to the object being referenced and returning the result.
- Custom iterators [4], in which a variety of adaptor types are enumerated.
- Compound iterators [1], which access a slice out of a container of containers.
- Several iterator adaptors from the MTL [12]. The MTL contains a strided iterator, where each call to *operator++()* moves the iterator ahead by some constant factor, and a scaled iterator, which multiplies the dereferenced value by some constant.

To fulfill the need for constructing adaptors, we propose the *iterator_adaptor* class template. Instantiations of *iterator_adaptor* serve as a base classes for new iterators, providing the default behavior of forwarding all operations to the underlying iterator. The user can selectively replace these features in the derived iterator class. This proposal also includes a number of more specialized adaptors, such as the *transform_iterator* that applies some user-specified function during the dereference of the iterator.

**Impact on the Standard**

This proposal is purely an addition to the C++ standard library. However, note that this proposal relies on the proposal for New Iterator Concepts.

---

[1] We use the term concept to mean a set of requirements that a type must satisfy to be used with a particular template parameter.

[2] The term mutable iterator refers to iterators over objects that can be changed by assigning to the dereferenced iterator, while constant iterator refers to iterators over objects that cannot be modified.
Design

Iterator Concepts

This proposal is formulated in terms of the new iterator concepts as proposed in n1550, since user-defined and especially adapted iterators suffer from the well known categorization problems that are inherent to the current iterator categories.

This proposal does not strictly depend on proposal n1550, as there is a direct mapping between new and old categories. This proposal could be reformulated using this mapping if n1550 was not accepted.

Interoperability

The question of iterator interoperability is poorly addressed in the current standard. There are currently two defect reports that are concerned with interoperability issues.

Issue 179 concerns the fact that mutable container iterator types are only required to be convertible to the corresponding constant iterator types, but objects of these types are not required to interoperate in comparison or subtraction expressions. This situation is tedious in practice and out of line with the way built in types work. This proposal implements the proposed resolution to issue 179, as most standard library implementations do nowadays. In other words, if an iterator type A has an implicit or user defined conversion to an iterator type B, the iterator types are interoperable and the usual set of operators are available.

Issue 280 concerns the current lack of interoperability between reverse iterator types. The proposed new reverse iterator template fixes the issues raised in 280. It provides the desired interoperability without introducing unwanted overloads.

Iterator Facade

While the iterator interface is rich, there is a core subset of the interface that is necessary for all the functionality. We have identified the following core behaviors for iterators:

- dereferencing
- incrementing
- decrementing
- equality comparison
- random-access motion
- distance measurement

In addition to the behaviors listed above, the core interface elements include the associated types exposed through iterator traits: value_type, reference, difference_type, and iterator_category.

Iterator facade uses the Curiously Recurring Template Pattern (CRTP) [Cop95] so that the user can specify the behavior of iterator_facade in a derived class. Former designs used policy objects to specify the behavior, but that approach was discarded for several reasons:

1. the creation and eventual copying of the policy object may create overhead that can be avoided with the current approach.
2. The policy object approach does not allow for custom constructors on the created iterator types, an essential feature if iterator_facade should be used in other library implementations.
3. Without the use of CRTP, the standard requirement that an iterator’s `operator++` returns the iterator type itself would mean that all iterators built with the library would have to be specializations of `iterator_facade<...>`, rather than something more descriptive like `indirect_iterator<T*>`. Cumbersome type generator metafunctions would be needed to build new parameterized iterators, and a separate `iterator_adaptor` layer would be impossible.

Usage

The user of `iterator_facade` derives his iterator class from a specialization of `iterator_facade` and passes the derived iterator class as `iterator_facade`'s first template parameter. The order of the other template parameters have been carefully chosen to take advantage of useful defaults. For example, when defining a constant lvalue iterator, the user can pass a const-qualified version of the iterator’s `value_type` as `iterator_facade`’s `Value` parameter and omit the `Reference` parameter which follows.

The derived iterator class must define member functions implementing the iterator’s core behaviors. The following table describes expressions which are required to be valid depending on the category of the derived iterator type. These member functions are described briefly below and in more detail in the iterator facade requirements.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Effects</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i.dereference()</code></td>
<td>Access the value referred to</td>
</tr>
<tr>
<td><code>i.equal(j)</code></td>
<td>Compare for equality with j</td>
</tr>
<tr>
<td><code>i.increment()</code></td>
<td>Advance by one position</td>
</tr>
<tr>
<td><code>i.decrement()</code></td>
<td>Retreat by one position</td>
</tr>
<tr>
<td><code>i.advance(n)</code></td>
<td>Advance by n positions</td>
</tr>
<tr>
<td><code>i.distance_to(j)</code></td>
<td>Measure the distance to j</td>
</tr>
</tbody>
</table>

In addition to implementing the core interface functions, an iterator derived from `iterator_facade` typically defines several constructors. To model any of the standard iterator concepts, the iterator must at least have a copy constructor. Also, if the iterator type `X` is meant to be automatically interoperable with another iterator type `Y` (as with constant and mutable iterators) then there must be an implicit conversion from `X` to `Y` or from `Y` to `X` (but not both), typically implemented as a conversion constructor. Finally, if the iterator is to model Forward Traversal Iterator or a more-refined iterator concept, a default constructor is required.

Iterator Core Access

`iterator_facade` and the operator implementations need to be able to access the core member functions in the derived class. Making the core member functions public would expose an implementation detail to the user. The design used here ensures that implementation details do not appear in the public interface of the derived iterator type.

Preventing direct access to the core member functions has two advantages. First, there is no possibility for the user to accidently use a member function of the iterator when a member of the value type was intended. This has been an issue with smart pointer implementations in the past. The second and main advantage is that library implementers can freely exchange a hand-rolled iterator implementation for one based on `iterator_facade` without fear of breaking code that was accessing the public core member functions directly.
In a naive implementation, keeping the derived class’ core member functions private would require it to grant friendship to \texttt{iterator\_f
cade} and each of the seven operators. In order to reduce the burden of limiting access, \texttt{iterator\_core\_access} is provided, a class that acts as a gateway to the core member functions in the derived iterator class. The author of the derived class only needs to grant friendship to \texttt{iterator\_core\_access} to make his core member functions available to the library.

\texttt{iterator\_core\_access} will be typically implemented as an empty class containing only private static member functions which invoke the iterator core member functions. There is, however, no need to standardize the gateway protocol. Note that even if \texttt{iterator\_core\_access} used public member functions it would not open a safety loophole, as every core member function preserves the invariants of the iterator.

\texttt{operator[]} 

The indexing operator for a generalized iterator presents special challenges. A random access iterator’s \texttt{operator[]} is only required to return something convertible to its \texttt{value\_type}. Requiring that it return an lvalue would rule out currently-legal random-access iterators which hold the referenced value in a data member (e.g. \texttt{counting\_iterator}), because \texttt{*(p+n)} is a reference into the temporary iterator \texttt{p+n}, which is destroyed when \texttt{operator[]} returns.

Writable iterators built with \texttt{iterator\_facade} implement the semantics required by the preferred resolution to \texttt{issue 299} and adopted by proposal \texttt{n1550}: the result of \texttt{p[n]} is an object convertible to the iterator’s \texttt{value\_type}, and \texttt{p[n] = x} is equivalent to \texttt{*(p + n) = x} (Note: This result object may be implemented as a proxy containing a copy of \texttt{p+n}). This approach will work properly for any random-access iterator regardless of the other details of its implementation. A user who knows more about the implementation of her iterator is free to implement \texttt{operator[]} that returns an lvalue in the derived iterator class; it will hide the one supplied by \texttt{iterator\_facade} from clients of her iterator.

\texttt{operator->} 

The reference type of a readable iterator (and today’s input iterator) need not in fact be a reference, so long as it is convertible to the iterator’s \texttt{value\_type}. When the \texttt{value\_type} is a class, however, it must still be possible to access members through \texttt{operator->}. Therefore, an iterator whose \texttt{reference\_type} is not in fact a reference must return a proxy containing a copy of the referenced value from its \texttt{operator->}.

The return types for \texttt{iterator\_facade\_s operator->} and \texttt{operator[]} are not explicitly specified. Instead, those types are described in terms of a set of requirements, which must be satisfied by the \texttt{iterator\_facade} implementation.

\textbf{Iterator Adaptor} 

The \texttt{iterator\_adaptor} class template adapts some \texttt{Base} \cite{Coplien95} type to create a new iterator. Instantiations of \texttt{iterator\_adaptor} are derived from a corresponding instantiation of \texttt{iterator\_facade} and implement the core behaviors in terms of the \texttt{Base} type. In essence, \texttt{iterator\_adaptor} merely forwards all operations to an instance of the \texttt{Base} type, which it stores as a member.

The user of \texttt{iterator\_adaptor} creates a class derived from an instantiation of \texttt{iterator\_adaptor} and then selectively redefines some of the core member functions described in the table above. The \texttt{Base} type need

\cite{Coplien95} Coplien, J., Curiously Recurring Template Patterns, C++ Report, February 1995, pp. 24-27.

\texttt{[3]} The term “Base” here does not refer to a base class and is not meant to imply the use of derivation. We have followed the lead of the standard library, which provides a base() function to access the underlying iterator object of a \texttt{reverse\_iterator\_adaptor}. 


not meet the full requirements for an iterator. It need only support the operations used by the core interface functions of iterator_adaptor that have not been redefined in the user's derived class.

Several of the template parameters of iterator_adaptor default to use_default. This allows the user to make use of a default parameter even when she wants to specify a parameter later in the parameter list. Also, the defaults for the corresponding associated types are somewhat complicated, so metaprogramming is required to compute them, and use_default can help to simplify the implementation. Finally, the identity of the use_default type is not left unspecified because specification helps to highlight that the Reference template parameter may not always be identical to the iterator’s reference type, and will keep users from making mistakes based on that assumption.

**Specialized Adaptors**

This proposal also contains several examples of specialized adaptors which were easily implemented using iterator_adaptor:

- **indirect_iterator**, which iterates over iterators, pointers, or smart pointers and applies an extra level of dereferencing.

- A new **reverse_iterator**, which inverts the direction of a Base iterator’s motion, while allowing adapted constant and mutable iterators to interact in the expected ways (unlike those in most implementations of C++98).

- **transform_iterator**, which applies a user-defined function object to the underlying values when dereferenced.

- **filter_iterator**, which provides a view of an iterator range in which some elements of the underlying range are skipped.

- **counting_iterator**, which adapts any incrementable type (e.g. integers, iterators) so that incrementing/decrementing the adapted iterator and dereferencing it produces successive values of the Base type.

- **function_output_iterator**, which makes it easier to create custom output iterators.

Based on examples in the Boost library, users have generated many new adaptors, among them a permutation adaptor which applies some permutation to a random access iterator, and a stride adaptor, which adapts a random access iterator by multiplying its unit of motion by a constant factor. In addition, the Boost Graph Library (BGL) uses iterator adaptors to adapt other graph libraries, such as LEDA [10] and Stanford GraphBase [8], to the BGL interface (which requires C++ Standard compliant iterators).

**Proposed Text**

**Header <iterator_helper> synopsis [lib.iterator.helper.synopsis]**

```cpp
struct use_default;

struct iterator_core_access { /* implementation detail */ };  

template <>
    class Derived
    , class Value
    , class CategoryOrTraversal
```
class iterator_facade;

template <
    class Derived
    , class Base
    , class Value = use_default
    , class CategoryOrTraversal = use_default
    , class Reference = use_default
    , class Difference = use_default
>
    class iterator_adaptor;

template <
    class Iterator
    , class Value = use_default
    , class CategoryOrTraversal = use_default
    , class Reference = use_default
    , class Difference = use_default
>
    class indirect_iterator;

template <class Dereferenceable>
    struct pointee;

template <class Dereferenceable>
    struct indirect_reference;

template <class Iterator>
    class reverse_iterator;

template <
    class UnaryFunction
    , class Iterator
    , class Reference = use_default
    , class Value = use_default
>
    class transform_iterator;

template <class Predicate, class Iterator>
    class filter_iterator;

template <
    class Incrementable
    , class CategoryOrTraversal = use_default
    , class Difference = use_default
>
    class counting_iterator;

template <class UnaryFunction>
    class function_output_iterator;
**Iterator facade [lib.iterator.facade]**

*Iterator facade* is a base class template that implements the interface of standard iterators in terms of a few core functions and associated types, to be supplied by a derived iterator class.

**Class template iterator_facade**

```
template <
    class Derived,
    class Value,
    class CategoryOrTraversal,
    class Reference = Value&,
    class Difference = ptrdiff_t
>
class iterator_facade {
public:
    typedef remove_const<Value>::type value_type;
    typedef Reference reference;
    typedef Value* pointer;
    typedef Difference difference_type;
    typedef /* see below */ iterator_category;

    reference operator*() const;
    /* see below */ operator->() const;
    /* see below */ operator[](difference_type n) const;
    Derived& operator++();
    Derived& operator++(int);
    Derived& operator--();
    Derived& operator--(int);
    Derived& operator+=(difference_type n);
    Derived& operator-=(difference_type n);
    Derived operator-(difference_type n) const;
};
```

// Comparison operators
```
typename enable_if interoperable<Dr1,Dr2,bool>::type // exposition
operator ==<iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs, 
iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs>;
```

```
typename enable_if interoperable<Dr1,Dr2,bool>::type 
operator !=<iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs, 
iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs>;
```

```
typename enable_if interoperable<Dr1,Dr2,bool>::type 
operator <<iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs, 
iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs>;
```
```cpp
template <class Dr1, class V1, class TC1, class R1, class D1,  
class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type  
operator <=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,  
           iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

template <class Dr1, class V1, class TC1, class R1, class D1,  
class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type  
operator >(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,  
           iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

template <class Dr1, class V1, class TC1, class R1, class D1,  
class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type  
operator >=(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,  
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

// Iterator difference
template <class Dr1, class V1, class TC1, class R1, class D1,  
class Dr2, class V2, class TC2, class R2, class D2>
/* see below */
operator-(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,  
          iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

// Iterator addition
template <class Dr, class V, class TC, class R, class D>
Derived operator+ (iterator_facade<Dr,V,TC,R,D> const&,
                    typename Derived::difference_type n);

template <class Dr, class V, class TC, class R, class D>
Derived operator+ (typename Derived::difference_type n,  
                    iterator_facade<Dr,V,TC,R,D> const&);

The iterator_category member of iterator_facade is

``iterator-category``(CategoryOrTraversal, value_type, reference)

where iterator-category is defined as follows:

``iterator-category``(C,R,V) :
  if (C is convertible to std::input_iterator_tag  
      || C is convertible to std::output_iterator_tag  
  )
    return C
  else if (C is not convertible to incrementable_traversal_tag)
    the program is ill-formed
  else return a type X satisfying the following two constraints:
    1. X is convertible to X1, and not to any more-derived
       type, where X1 is defined by:
```
if (R is a reference type
    && C is convertible to forward_traversal_tag)
    {
        if (C is convertible to random_access_traversal_tag)
            X1 = random_access_iterator_tag
        else if (C is convertible to bidirectional_traversal_tag)
            X1 = bidirectional_iterator_tag
        else
            X1 = forward_iterator_tag
    }
else
    {
        if (C is convertible to single_pass_traversal_tag
            && R is convertible to V)
            X1 = input_iterator_tag
        else
            X1 = C
    }

2. **category-to-traversal**(X) is convertible to the most
derived traversal tag type to which X is also
convertible, and not to any more-derived traversal tag
type.

[Note: the intention is to allow iterator_category to be one of the five original category tags when convert-
ibility to one of the traversal tags would add no information]

The **enable_if_interoperable** template used above is for exposition purposes. The member operators should
only be in an overload set provided the derived types Dr1 and Dr2 are interoperable, meaning that at least one
of the types is convertible to the other. The **enable_if_interoperable** approach uses SFINAE to take the
operators out of the overload set when the types are not interoperable. The operators should behave as-if
**enable_if_interoperable** were defined to be:

```cpp
template <bool, typename> enable_if_interoperable_impl
{};

template <typename T> enable_if_interoperable_impl<true,T>
{ typedef T type; }; // if T is convertible

template<typename Dr1, typename Dr2, typename T>
struct enable_if_interoperable
    : enable_if_interoperable_impl<
        is_convertible<Dr1,Dr2>::value || is_convertible<Dr2,Dr1>::value
    , T
> {}
```

**iterator_facade** Requirements

The following table describes the typical valid expressions on **iterator_facade**’s Derived parameter, depending on the iterator concept(s) it will model. The operations in the first column must be made accessible to member functions of class **iterator_core.access**. In addition, **static_cast<Derived*>(iterator_facade*)** shall be well-formed.
In the table below, $F$ is $\text{iterator\_facade}\langle X, V, C, R, D \rangle$, $a$ is an object of type $X$, $b$ and $c$ are objects of type $\text{const } X$, $n$ is an object of $F::\text{difference\_type}$, $y$ is a constant object of a single pass iterator type interoperable with $X$, and $z$ is a constant object of a random access traversal iterator type interoperable with $X$.

**iterator\_facade Core Operations**

<table>
<thead>
<tr>
<th>Expression</th>
<th>Return Type</th>
<th>Assertion/Note</th>
<th>Used to implement Iterator Concept(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$c.\text{dereference}()$</td>
<td>$F::\text{reference}$</td>
<td></td>
<td>Readable Iterator, Writable Iterator</td>
</tr>
<tr>
<td>$c.\text{equal}(y)$</td>
<td>convertible to bool</td>
<td>true iff $c$ and $y$ refer to the same position.</td>
<td>Single Pass Iterator</td>
</tr>
<tr>
<td>$a.\text{increment}()$</td>
<td>unused</td>
<td></td>
<td>Incrementable Iterator</td>
</tr>
<tr>
<td>$a.\text{decrement}()$</td>
<td>unused</td>
<td></td>
<td>Bidirectional Traversal Iterator</td>
</tr>
<tr>
<td>$a.\text{advance}(n)$</td>
<td>unused</td>
<td></td>
<td>Random Access Traversal Iterator</td>
</tr>
<tr>
<td>$c.\text{distance_to}(z)$</td>
<td>convertible to $F::\text{difference_type}$</td>
<td>equivalent to $\text{distance}(c, X(z))$</td>
<td>Random Access Traversal Iterator</td>
</tr>
</tbody>
</table>

**iterator\_facade operations**

The operations in this section are described in terms of operations on the core interface of $\text{Derived}$ which may be inaccessible (i.e. private). The implementation should access these operations through member functions of class $\text{iterator\_core\_access}$.

```cpp
reference operator*() const;

operator->() const; (see below)
```

**Returns:** If $\text{reference}$ is a reference type, an object of type $\text{pointer}$ equal to:

```cpp
&\text{static\_cast<Derived const*>(this)}->\text{dereference()}
```

Otherwise returns an object of unspecified type such that, $(\text{static\_cast<Derived const*>(this)})->m$ is equivalent to $(w = **\text{static\_cast<Derived const*>(this)}, w.m)$ for some temporary object $w$ of type $\text{value\_type}$.

```cpp
unspecified operator[](difference\_type n) const;
```

**Returns:** an object convertible to $\text{value\_type}$. For constant objects $v$ of type $\text{value\_type}$, and $n$ of type $\text{difference\_type}$, $(\text{*(this)\[n\]} = v)$ is equivalent to $(\text{*(this + n)} = v)$, and $\text{static\_cast<value\_type const&>>(\text{*(this)\[n\]})}$ is equivalent to $\text{static\_cast<value\_type const&>(*(this + n))}$.

```cpp
Derived& operator++();
```

**Effects:** $\text{static\_cast<Derived*>(this)}->\text{increment()}$; return $\text{static\_cast<Derived*>(this)}$;
Derived operator++(int);

Effects: Derived tmp(static_cast<Derived const*>(this));
++*this;
return tmp;

Derived& operator--();

Effects: static_cast<Derived*>(this)->decrement();
return *static_cast<Derived*>(this);

Derived operator--(int);

Effects: Derived tmp(static_cast<Derived const*>(this));
--*this;
return tmp;

Derived& operator+=(difference_type n);

Effects: static_cast<Derived*>(this)->advance(n);
return *static_cast<Derived*>(this);

Derived operator-=(difference_type n);

Effects: static_cast<Derived*>(this)->advance(-n);
return *static_cast<Derived*>(this);

Derived operator-(difference_type n) const;

Effects: Derived tmp(static_cast<Derived const*>(this));
return tmp -= n;

template <class Dr, class V, class TC, class R, class D>
Derived operator+ (iterator_facade<Dr,V,TC,R,D> const&,
typename Derived::difference_type n);

template <class Dr, class V, class TC, class R, class D>
Derived operator+ (typename Derived::difference_type n,
iterator_facade<Dr,V,TC,R,D> const&);

Effects: Derived tmp(static_cast<Derived const*>(this));
return tmp += n;

template <class Dr1, class V1, class TC1, class R1, class D1,
class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if interoperable<Dr1,Dr2,bool>::type
operator == (iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

Returns: if is_convertible<Dr2,Dr1>::value
then (Dr1 const&)lhs).equal((Dr2 const&)rhs).
Otherwise, ((Dr2 const&)rhs).equal((Dr1 const&)lhs).
template <class Dr1, class V1, class TC1, class R1, class D1, class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator !(iterator_facade<Dr1,V1,TC1,R1,D1> const & lhs,
       iterator_facade<Dr2,V2,TC2,R2,D2> const & rhs);

Returns: if is_convertible<Dr2,Dr1>::value
  then  !((Dr1 const&)lhs).equal((Dr2 const&)rhs).
Otherwise,  !((Dr2 const&)rhs).equal((Dr1 const&)lhs).

template <class Dr1, class V1, class TC1, class R1, class D1, class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator !(iterator_facade<Dr1,V1,TC1,R1,D1> const & lhs,
       iterator_facade<Dr2,V2,TC2,R2,D2> const & rhs);

Returns: if is_convertible<Dr2,Dr1>::value
  then  ((Dr1 const&)lhs).distance_to((Dr2 const&)rhs) < 0.
Otherwise,  ((Dr2 const&)rhs).distance_to((Dr1 const&)lhs) > 0.

template <class Dr1, class V1, class TC1, class R1, class D1, class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator <=(iterator_facade<Dr1,V1,TC1,R1,D1> const & lhs,
       iterator_facade<Dr2,V2,TC2,R2,D2> const & rhs);

Returns: if is_convertible<Dr2,Dr1>::value
  then  ((Dr1 const&)lhs).distance_to((Dr2 const&)rhs) <= 0.
Otherwise,  ((Dr2 const&)rhs).distance_to((Dr1 const&)lhs) >= 0.

template <class Dr1, class V1, class TC1, class R1, class D1, class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1,Dr2,bool>::type
operator >=(iterator_facade<Dr1,V1,TC1,R1,D1> const & lhs,
       iterator_facade<Dr2,V2,TC2,R2,D2> const & rhs);

Returns: if is_convertible<Dr2,Dr1>::value
  then  ((Dr1 const&)lhs).distance_to((Dr2 const&)rhs) >= 0.
Otherwise,  ((Dr2 const&)rhs).distance_to((Dr1 const&)lhs) <= 0.
template <class Dr1, class V1, class TC1, class R1, class D1,
         class Dr2, class V2, class TC2, class R2, class D2>
typename enable_if interoperable<Dr1,Dr2,difference>::type
operator -(iterator_facade<Dr1,V1,TC1,R1,D1> const& lhs,
            iterator_facade<Dr2,V2,TC2,R2,D2> const& rhs);

Return Type: if is_convertible<Dr2,Dr1>::value
    then difference shall be iterator_traits<Dr1>::difference_type.
Otherwise difference shall be iterator_traits<Dr2>::difference_type

Returns: if is_convertible<Dr2,Dr1>::value
    then -((Dr1 const&)lhs).distance_to((Dr2 const&)rhs).
Otherwise, ((Dr2 const&)rhs).distance_to((Dr1 const&)lhs).

Iterator adaptor [lib.iterator.adaptor]

Each specialization of the iterator_adaptor class template is derived from a specialization of iterator_facade. The core interface functions expected by iterator_facade are implemented in terms of the iterator_adaptor’s Base template parameter. A class derived from iterator_adaptor typically redefines some of the core interface functions to adapt the behavior of the Base type. Whether the derived class models any of the standard iterator concepts depends on the operations supported by the Base type and which core interface functions of iterator_facade are redefined in the Derived class.

Class template iterator_adaptor

template <
    class Derived
 , class Base
 , class Value = use_default
 , class CategoryOrTraversal = use_default
 , class Reference = use_default
 , class Difference = use_default>
>
class iterator_adaptor
 : public iterator_facade<Derived, V’, C’, R’, D’> // see details
{
    friend class iterator_core_access;
public:
    iterator_adaptor();
    explicit iterator_adaptor(Base iter);
    Base const& base() const;
protected:
    Base const& base_reference() const;
    Base& base_reference();
private: // Core iterator interface for iterator_facade.
    typename iterator_adaptor::reference dereference() const;

    template <
    class OtherDerived, class OtherIterator, class V, class C, class R, class D
    >
    bool equal(iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D> const& x) const;
void advance(typename iterator_adaptor::difference_type n);
void increment();
void decrement();

void advance(typename iterator_adaptor::difference_type n);
void increment();
void decrement();

private:
  Base m_iterator; // exposition only
};

**iterator_adaptor requirements**

static_cast<Derived*>(iterator_adaptor*) shall be well-formed. The Base argument shall be Assignable and Copy Constructible.

**iterator_adaptor base class parameters**

The $V'$, $C'$, $R'$, and $D'$ parameters of the iterator_facade used as a base class in the summary of iterator_adaptor above are defined as follows:

$$V' = \text{if (Value is use\_default)}$$
$$\quad \text{return iterator_traits<Base>::value\_type}$$
$$\text{else}$$
$$\quad \text{return Value}$$

$$C' = \text{if (CategoryOrTraversal is use\_default)}$$
$$\quad \text{return iterator_traversal<Base>::type}$$
$$\text{else}$$
$$\quad \text{return CategoryOrTraversal}$$

$$R' = \text{if (Reference is use\_default)}$$
$$\quad \text{if (Value is use\_default)}$$
$$\quad \quad \text{return iterator_traits<Base>::reference}$$
$$\quad \text{else}$$
$$\quad \quad \text{return Value\&}$$
$$\text{else}$$
$$\quad \text{return Reference}$$

$$D' = \text{if (Difference is use\_default)}$$
$$\quad \text{return iterator_traits<Base>::difference\_type}$$
$$\text{else}$$
$$\quad \text{return Difference}$$

**iterator_adaptor public operations**

iterator_adaptor();
explicit iterator_adaptor(Base iter);

Returns: An instance of iterator_adaptor with m_iterator default constructed.

Base const& base() const;

Returns: m_iterator

iterator_adaptor protected member functions

Base const& base_reference() const;

Returns: A const reference to m_iterator.

Base& base_reference();

Returns: A non-const reference to m_iterator.

iterator_adaptor private member functions

typename iterator_adaptor::reference dereference() const;

Returns: *m_iterator

template <
class OtherDerived, class OtherIterator, class V, class C, class R, class D
>
bool equal(iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D> const& x) const;

Returns: m_iterator == x.base()

void advance(typename iterator_adaptor::difference_type n);

Effects: m_iterator += n;

void increment();

Effects: ++m_iterator;

void decrement();

Effects: --m_iterator;

template <
class OtherDerived, class OtherIterator, class V, class C, class R, class D
>
typename iterator_adaptor::difference_type distance_to(
  iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D> const& y) const;

Returns: y.base() - m_iterator
Specialized adaptors [lib.iterator.special.adaptors]

The enable_if_convertible<\text{X,Y}>::type expression used in this section is for exposition purposes. The converting constructors for specialized adaptors should be only be in an overload set provided that an object of type \text{X} is implicitly convertible to an object of type \text{Y}. The signatures involving enable_if_convertible should behave as-if enable_if_convertible were defined to be:

\begin{verbatim}
template <bool> enable_if_convertible_impl

{ };

template <> enable_if_convertible_impl<true>
{ struct type; };
\end{verbatim}

\begin{verbatim}
template<typename From, typename To>
struct enable_if_convertible
  : enable_if_convertible_impl<is_convertible<From,To>::value>
{ };
\end{verbatim}

If an expression other than the default argument is used to supply the value of a function parameter whose type is written in terms of enable_if_convertible, the program is ill-formed, no diagnostic required.

\textbf{Note:} The enable_if_convertible approach uses SFINAE to take the constructor out of the overload set when the types are not implicitly convertible.

Indirect iterator

indirect_iterator adapts an iterator by applying an extra dereference inside of \texttt{operator\*{}}. For example, this iterator adaptor makes it possible to view a container of pointers (e.g. \texttt{list<foo*>}) as if it were a container of the pointed-to type (e.g. \texttt{list<foo>}). indirect_iterator depends on two auxiliary traits, pointee and indirect_reference, to provide support for underlying iterators whose value_type is not an iterator.

Class template pointee

\begin{verbatim}
template <class Dereferenceable>
struct pointee
{
  typedef /* see below */ type;
};
\end{verbatim}

\textbf{Requires:} For an object \texttt{x} of type \texttt{Dereferenceable}, \texttt{*x} is well-formed. If \texttt{++x} is ill-formed it shall neither be ambiguous nor shall it violate access control, and \texttt{Dereferenceable::element_type} shall be an accessible type. Otherwise \texttt{iterator_traits\texttt{<Dereferenceable>::value_type}} shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of pointee]

\begin{verbatim}
type is determined according to the following algorithm, where \texttt{x} is an object of type \texttt{Dereferenceable}:
if ( \texttt{++x} is ill-formed )
{
  return "Dereferenceable::element_type"
}
else if (\texttt{"*x"} is a mutable reference to
\end{verbatim}
std::iterator_traits<Dereferenceable>::value_type

else

    return std::iterator_traits<Dereferenceable>::value_type const

Class template indirect_reference

template <class Dereferenceable>
struct indirect_reference
 {
 typedef /* see below */ type;
 
 };  

Requires: For an object \( x \) of type Dereferenceable, \(*x\) is well-formed. If \( ++x \) is ill-formed it shall neither be ambiguous nor shall it violate access control, and \( \text{pointee} \text{<Dereferenceable>::type} \) shall be well-formed. Otherwise \( \text{iterator_traits} \text{<Dereferenceable>::reference} \) shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of \text{indirect_reference}]

type is determined according to the following algorithm, where \( x \) is an object of type \( \text{Dereferenceable} \):

if ( \( ++x \) is ill-formed )
   return "\text{pointee} \text{<Dereferenceable>::type}"
else
   std::iterator_traits<

Class template indirect_iterator

template <
   class Iterator,
   class Value = use_default,
   class CategoryOrTraversal = use_default,
   class Reference = use_default,
   class Difference = use_default
>
class indirect_iterator
 {
 public:
   typedef /* see below */ value_type;
   typedef /* see below */ reference;
   typedef /* see below */ pointer;
   typedef /* see below */ difference_type;
   typedef /* see below */ iterator_category;

   indirect_iterator();
   indirect_iterator(Iterator x);

   template <
class Iterator2, class Value2, class Category2
, class Reference2, class Difference2
>
indirect_iterator<
    Iterator2, Value2, Category2, Reference2, Difference2
  > const y
  , typename enable_if_convertible<Iterator2, Iterator>::type* = 0 // exposition
);

Iterator const& base() const;
reference operator*() const;
indirect_iterator& operator++();
indirect_iterator& operator--();

private:
  Iterator m_iterator; // exposition
};

The member types of indirect_iterator are defined according to the following pseudo-code, where V is iterator_traits<Iterator>::value_type

if (Value is use_default) then
    typedef remove_const<pointee<V>::type>::type value_type;
else
    typedef remove_const<Value>::type value_type;

if (Reference is use_default) then
    if (Value is use_default) then
        typedef indirect_reference<V>::type reference;
    else
        typedef Value& reference;
else
    typedef Reference reference;

if (Value is use_default) then
    typedef pointee<V>::type* pointer;
else
    typedef Value* pointer;

if (Difference is use_default)
    typedef iterator_traits<Iterator>::difference_type difference_type;
else
    typedef Difference difference_type;

if (CategoryOrTraversal is use_default)
    typedef iterator_category (iterator_traversal<Iterator>::type,"reference","value_type"
  ) iterator_category;
else
    typedef iterator_category (CategoryOrTraversal,"reference","value_type"
  ) iterator_category;
indirect_iterator requirements

The expression \(*v\), where \(v\) is an object of \(\text{iterator_traits<Iterator>::value\_type}\), shall be valid expression and convertible to reference. \(\text{Iterator}\) shall model the traversal concept indicated by \(\text{iterator\_category}\). \(\text{Value, Reference, and Difference}\) shall be chosen so that \(\text{value\_type, reference, and difference\_type}\) meet the requirements indicated by \(\text{iterator\_category}\).

[Note: there are further requirements on the \(\text{iterator\_traits<Iterator>::value\_type}\) if the \(\text{Value}\) parameter is not use\_default, as implied by the algorithm for deducing the default for the \(\text{value\_type}\) member.]

indirect_iterator models

In addition to the concepts indicated by \(\text{iterator\_category}\) and by \(\text{iterator\_traversal<indirect\_iterator>::type}\), a specialization of \(\text{indirect\_iterator}\) models the following concepts, Where \(v\) is an object of \(\text{iterator\_traits<Iterator>::value\_type}\):

- Readable Iterator if \(\text{reference(*v)}\) is convertible to \(\text{value\_type}\).
- Writable Iterator if \(\text{reference(*v) = t}\) is a valid expression (where \(t\) is an object of type \(\text{indirect\_iterator::value\_type}\))
- Lvalue Iterator if \(\text{reference}\) is a reference type.

\(\text{indirect\_iterator<X,V1,C1,R1,D1>}\) is interoperable with \(\text{indirect\_iterator<Y,V2,C2,R2,D2>}\) if and only if \(X\) is interoperable with \(Y\).

indirect_iterator operations

In addition to the operations required by the concepts described above, specializations of \(\text{indirect\_iterator}\) provide the following operations.

\(\text{indirect\_iterator<\text{Iterator}\>}\);

**Requires:** \(\text{Iterator}\) must be Default Constructible.

**Effects:** Constructs an instance of \(\text{indirect\_iterator}\) with a default-constructed \(\text{m\_iterator}\).

\(\text{indirect\_iterator<\text{Iterator}\ x>}\);

**Effects:** Constructs an instance of \(\text{indirect\_iterator}\) with \(\text{m\_iterator}\) copy constructed from \(x\).

\(\text{template<}\)
    \(\text{class Iterator2, class Value2, unsigned Access, class Traversal, class Reference2, class Difference2}\)
\(\text{> indirect\_iterator<}\)
    \(\text{\text{Iterator2, Value2, Access, Traversal, Reference2, Difference2}}\)
\(\text{const y, typename enable\_if\_convertible<Iterator2, Iterator>::type* = 0 // exposition}}\);

**Requires:** \(\text{Iterator2}\) is implicitly convertible to \(\text{Iterator}\).
**Effects:** Constructs an instance of `indirect_iterator` whose `m_iterator` subobject is constructed from `y.base()`.

**Returns:** `m_iterator`

```
Iterator const& base() const;
```

**Returns:** `**m_iterator`

```
indirect_iterator& operator++();
```

**Effects:** `++m_iterator`

**Returns:** `*this`

```
indirect_iterator& operator--();
```

**Effects:** `--m_iterator`

**Returns:** `*this`

---

**Reverse iterator**

The reverse iterator adaptor iterates through the adapted iterator range in the opposite direction.

**Class template** `reverse_iterator`

```cpp
template <class Iterator>
class reverse_iterator
{
public:
    typedef iterator_traits<Iterator>::value_type value_type;
    typedef iterator_traits<Iterator>::reference reference;
    typedef iterator_traits<Iterator>::pointer pointer;
    typedef iterator_traits<Iterator>::difference_type difference_type;
    typedef /* see below */ iterator category;

    reverse_iterator() {}
    explicit reverse_iterator(Iterator x);

    template<class OtherIterator>
    reverse_iterator(
        reverse_iterator<OtherIterator> const& r,
        typename enable_if<convertible<OtherIterator, Iterator>::type* = 0 // exposition
    );

    Iterator const& base() const;
    reference operator*() const;
    reverse_iterator& operator++();
    reverse_iterator& operator--();

private:
    Iterator m_iterator; // exposition
};
```
If `Iterator` models Random Access Traversal Iterator and Readable Lvalue Iterator, then `iterator_category` is convertible to `random_access_iterator_tag`. Otherwise, if `Iterator` models Bidirectional Traversal Iterator and Readable Lvalue Iterator, then `iterator_category` is convertible to `bidirectional_iterator_tag`. Otherwise, `iterator_category` is convertible to `input_iterator_tag`.

**reverse_iterator requirements**

`Iterator` must be a model of Bidirectional Traversal Iterator. The type `iterator_traits<Iterator>::reference` must be the type of `*i`, where `i` is an object of type `Iterator`.

**reverse_iterator models**

A specialization of `reverse_iterator` models the same iterator traversal and iterator access concepts modeled by its `Iterator` argument. In addition, it may model old iterator concepts specified in the following table:

<table>
<thead>
<tr>
<th>If <code>I</code> models</th>
<th>then <code>reverse_iterator&lt;I&gt;</code> models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readable Lvalue Iterator, Bidirectional Traversal Iterator</td>
<td>Bidirectional Iterator</td>
</tr>
<tr>
<td>Writable Lvalue Iterator, Bidirectional Traversal Iterator</td>
<td>Mutable Bidirectional Iterator</td>
</tr>
<tr>
<td>Readable Lvalue Iterator, Random Access Traversal Iterator</td>
<td>Random Access Iterator</td>
</tr>
<tr>
<td>Writable Lvalue Iterator, Random Access Traversal Iterator</td>
<td>Mutable Random Access Iterator</td>
</tr>
</tbody>
</table>

`reverse_iterator<X>` is interoperable with `reverse_iterator<Y>` if and only if `X` is interoperable with `Y`.

**reverse_iterator operations**

In addition to the operations required by the concepts modeled by `reverse_iterator`, `reverse_iterator` provides the following operations.

`reverse_iterator();`

**Requires:** `Iterator` must be Default Constructible.

**Effects:** Constructs an instance of `reverse_iterator` with `m_iterator` default constructed.

`explicit reverse_iterator(Iterator x);`

**Effects:** Constructs an instance of `reverse_iterator` with `m_iterator` copy constructed from `x`.

`template<class OtherIterator>`
`reverse_iterator(
    reverse_iterator<OtherIterator> const& r,
    typename enable_if_convertible<OtherIterator, Iterator>::type* = 0 // exposition
);`

**Requires:** `OtherIterator` is implicitly convertible to `Iterator`. 
Effects: Constructs instance of reverse_iterator whose m_iterator subobject is constructed from y.base().

Iterator const & base() const;

Returns: m_iterator

reference operator*() const;

Effects:

Iterator tmp = m_iterator;
return *--tmp;

reverse_iterator& operator++();

Effects: --m_iterator

Returns: *this

reverse_iterator& operator--();

Effects: +=m_iterator

Returns: *this

Transform iterator

The transform iterator adapts an iterator by modifying the operator* to apply a function object to the result of dereferencing the iterator and returning the result.

Class template transform_iterator

template <class UnaryFunction,
    class Iterator,
    class Reference = use_default,
    class Value = use_default>
class transform_iterator
{
public:
    typedef /* see below */ value_type;
    typedef /* see below */ reference;
    typedef /* see below */ pointer;
    typedef iterator_traits<Iterator>::difference_type difference_type;
    typedef /* see below */ iterator_category;

    transform_iterator();
    transform_iterator(Iterator const & x, UnaryFunction f);

    template<class F2, class I2, class R2, class V2>
    transform_iterator(transform_iterator<F2, I2, R2, V2> const & t
If Reference is use_default then the reference member of transform_iterator is result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type. Otherwise, reference is Reference.

If Value is use_default then the value_type member is remove_cv<remove_reference<reference>::type. Otherwise, value_type is Value.

If Iterator models Readable Lvalue Iterator and if Iterator models Random Access Traversal Iterator, then iterator_category is convertible to random_access_iterator_tag. Otherwise, if Iterator models Bidirectional Traversal Iterator, then iterator_category is convertible to bidirectional_iterator_tag. Otherwise iterator_category is convertible to forward_iterator_tag. If Iterator does not model Readable Lvalue Iterator then iterator_category is convertible to input_iterator_tag.

**transform_iterator requirements**

The type UnaryFunction must be Assignable, Copy Constructible, and the expression f(*i) must be valid where f is an object of type UnaryFunction, i is an object of type Iterator, and where the type of f(*i) must be result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type.

The argument Iterator shall model Readable Iterator.

**transform_iterator models**

The resulting transform_iterator models the most refined of the following that is also modeled by Iterator.

- Writable Lvalue Iterator if transform_iterator::reference is a non-const reference.
- Readable Lvalue Iterator if transform_iterator::reference is a const reference.
- Readable Iterator otherwise.

The transform_iterator models the most refined standard traversal concept that is modeled by the Iterator argument.

If transform_iterator is a model of Readable Lvalue Iterator then it models the following original iterator concepts depending on what the Iterator argument models.

<table>
<thead>
<tr>
<th>If Iterator models</th>
<th>Then transform_iterator models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Pass Iterator</td>
<td>Input Iterator</td>
</tr>
<tr>
<td>Forward Traversal Iterator</td>
<td>Forward Iterator</td>
</tr>
<tr>
<td>Bidirectional Traversal Iterator</td>
<td>Bidirectional Iterator</td>
</tr>
<tr>
<td>Random Access Traversal Iterator</td>
<td>Random Access Iterator</td>
</tr>
</tbody>
</table>
If `transform_iterator` models Writable Lvalue Iterator then it is a mutable iterator (as defined in the old iterator requirements).

`transform_iterator<F1, X, R1, V1>` is interoperable with `transform_iterator<F2, Y, R2, V2>` if and only if `X` is interoperable with `Y`.

**transform_iterator operations**

In addition to the operations required by the concepts modeled by `transform_iterator`, `transform_iterator` provides the following operations.

```cpp
transform_iterator();
```

**Returns:** An instance of `transform_iterator` with `m_f` and `m_iterator` default constructed.

```cpp
transform_iterator(Iterator const& x, UnaryFunction f);
```

**Returns:** An instance of `transform_iterator` with `m_f` initialized to `f` and `m_iterator` initialized to `x`.

```cpp
template<class OtherIterator, class R2, class V2>
transform_iterator(
    transform_iterator<UnaryFunction, OtherIterator, R2, V2> const& t,
    typename enable_if<convertible<OtherIterator, Iterator>::type* = 0 // exposition
);```

**Returns:** An instance of `transform_iterator` that is a copy of `t`.

**Requires:** `OtherIterator` is implicitly convertible to `Iterator`.

```cpp
UnaryFunction functor() const;
```

**Returns:** `m_f`

```cpp
Iterator const& base() const;
```

**Returns:** `m_iterator`

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

**Returns:** `m_f(*m_iterator)`

```cpp
transform_iterator& operator++();
```

**Effects:** `++m_iterator`  
**Returns:** *this

```cpp
transform_iterator& operator--();
```

**Effects:** `--m_iterator`  
**Returns:** *this
Filter iterator

The filter iterator adaptor creates a view of an iterator range in which some elements of the range are skipped. A predicate function object controls which elements are skipped. When the predicate is applied to an element, if it returns true then the element is retained and if it returns false then the element is skipped over. When skipping over elements, it is necessary for the filter adaptor to know when to stop so as to avoid going past the end of the underlying range. A filter iterator is therefore constructed with pair of iterators indicating the range of elements in the unfiltered sequence to be traversed.

Class template filter_iterator

```cpp
template <class Predicate, class Iterator>
class filter_iterator
{
    public:
        typedef iterator_traits<Iterator>::value_type value_type;
        typedef iterator_traits<Iterator>::reference reference;
        typedef iterator_traits<Iterator>::pointer pointer;
        typedef iterator_traits<Iterator>::difference_type difference_type;
        typedef /* see below */ iterator_category;

        filter_iterator();
        filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());
        filter_iterator(Iterator x, Iterator end = Iterator());
        template<class OtherIterator>
        filter_iterator(filter_iterator<Predicate, OtherIterator> const& t
            , typename enable_if<convertible<OtherIterator, Iterator>::type* = 0 // exposition
            >::type* = 0 // exposition
            );
        Predicate predicate() const;
        Iterator end() const;
        Iterator const& base() const;
        reference operator*() const;
        filter_iterator& operator++();
    private:
        Predicate m_pred; // exposition only
        Iterator m_iter; // exposition only
        Iterator m_end;  // exposition only
};
```

If Iterator models Readable Lvalue Iterator and Forward Traversal Iterator then iterator_category is convertible to std::forward_iterator_tag. Otherwise iterator_category is convertible to std::input_iterator_tag.

filter_iterator requirements

The Iterator argument shall meet the requirements of Readable Iterator and Single Pass Iterator or it shall meet the requirements of Input Iterator.

The Predicate argument must be Assignable, Copy Constructible, and the expression p(x) must be valid where p is an object of type Predicate, x is an object of type iterator_traits<Iterator>::value_type, and where the type of p(x) must be convertible to bool.
**filter_iterator models**

The concepts that `filter_iterator` models are dependent on which concepts the `Iterator` argument models, as specified in the following tables.

<table>
<thead>
<tr>
<th>If <code>Iterator</code> models</th>
<th>then <code>filter_iterator</code> models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Pass Iterator</td>
<td>Single Pass Iterator</td>
</tr>
<tr>
<td>Forward Traversal Iterator</td>
<td>Forward Traversal Iterator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If <code>Iterator</code> models</th>
<th>then <code>filter_iterator</code> models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readable Iterator</td>
<td>Readable Iterator</td>
</tr>
<tr>
<td>Writable Iterator</td>
<td>Writable Iterator</td>
</tr>
<tr>
<td>Lvalue Iterator</td>
<td>Lvalue Iterator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>If <code>Iterator</code> models</th>
<th>then <code>filter_iterator</code> models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readable Iterator, Single Pass Iterator</td>
<td>Input Iterator</td>
</tr>
<tr>
<td>Readable Lvalue Iterator, Forward Traversal Iterator</td>
<td>Forward Iterator</td>
</tr>
<tr>
<td>Writable Lvalue Iterator, Forward Traversal Iterator</td>
<td>Mutable Forward Iterator</td>
</tr>
</tbody>
</table>

`filter_iterator<P1, X>` is interoperable with `filter_iterator<P2, Y>` if and only if `X` is interoperable with `Y`.

**filter_iterator operations**

In addition to those operations required by the concepts that `filter_iterator` models, `filter_iterator` provides the following operations.

`filter_iterator();`

**Requires:** `Predicate` and `Iterator` must be Default Constructible.

**Effects:** Constructs a `filter_iterator` whose `m_pred`, `m_iter`, and `m_end` members are a default constructed.

`filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());`

**Requires:** `Predicate` must be Default Constructible and `Predicate` is a class type (not a function pointer).

**Effects:** Constructs a `filter_iterator` where `m_iter` is either the first position in the range `[x, end)` such that `f(*m_iter) == true` or else `m_iter == end`. The member `m_pred` is constructed from `f` and `m_end` from `end`.

`filter_iterator(Iterator x, Iterator end = Iterator());`

**Requires:** `Predicate` must be Default Constructible and `Predicate` is a class type (not a function pointer).

**Effects:** Constructs a `filter_iterator` where `m_iter` is either the first position in the range `[x, end)` such that `m_pred(*m_iter) == true` or else `m_iter == end`. The member `m_pred` is default constructed.
template <class OtherIterator>
filter_iterator(
    filter_iterator<Predicate, OtherIterator> const& t,
    typename enable_if_convertible<OtherIterator, Iterator>::type* = 0 // exposition
);``

Requires: OtherIterator is implicitly convertible to Iterator.
Effects: Constructs a filter iterator whose members are copied from t.

Predicate predicate() const;

Returns: m_pred

Iterator end() const;

Returns: m_end

Iterator const& base() const;

Returns: m_iterator

reference operator*() const;

Returns: *m_iter

filter_iterator& operator++();

Effects: Increments m_iter and then continues to increment m_iter until either m_iter == m_end
or m_pred(*m_iter) == true.

Returns: *this

Counting iterator
counting_iterator adapts an object by adding an operator* that returns the current value of the object. All
other iterator operations are forwarded to the adapted object.

Class template counting_iterator
template <
class Incrementable
, class CategoryOrTraversal = use_default
, class Difference = use_default
>
class counting_iterator
{
public:
    typedef Incrementable value_type;
    typedef const Incrementable& reference;
    typedef const Incrementable* pointer;
    typedef /* see below */ difference_type;
typedef /* see below */ iterator
category;
counting_iterator();
counting_iterator(counting_iterator const& rhs);
explicit counting_iterator(Incrementable x);
Incrementable const& base() const;
reference operator*() const;
counting_iterator& operator++();
counting_iterator& operator--();
private:
    Incrementable m_inc; // exposition
};

If the Difference argument is use_default then difference_type is an unspecified signed integral type.
Otherwise difference_type is Difference.

iterator_category is determined according to the following algorithm:

if (CategoryOrTraversal is not use_default)
    return CategoryOrTraversal
else if (numeric_limits<Incrementable>::is_specialized)
    return iterator_category(
        random_access_traversal_tag, Incrementable, const Incrementable&
    )
else
    return iterator_category(
        iterator_traversal<Incrementable>::type,
        Incrementable, const Incrementable&
    )

[Note: implementers are encouraged to provide an implementation of operator- and a difference_type that avoids overflows in the cases where std::numeric_limits<Incrementable>::is_specialized is true.]

counting_iterator requirements

The Incrementable argument shall be Copy Constructible and Assignable.

If iterator_category is convertible to forward_iterator_tag or forward_traversal_tag, the following must be well-formed:

Incrementable i, j;
++i; // pre-increment
i == j; // operator equal

If iterator_category is convertible to bidirectional_iterator_tag or bidirectional_traversal_tag, the following expression must also be well-formed:

--i

If iterator_category is convertible to random_access_iterator_tag or random_access_traversal_tag, the following must also be valid:

counting_iterator::difference_type n;
i += n;
n = i - j;
i < j;
counting_iterator models

Specializations of counting_iterator model Readable Lvalue Iterator. In addition, they model the concepts corresponding to the iterator tags to which their iterator_category is convertible. Also, if CategoryOrTraversal is not use_default then counting_iterator models the concept corresponding to the iterator tag CategoryOrTraversal. Otherwise, if numeric_limits<Incrementable>::is_specialized, then counting_iterator models Random Access Traversal Iterator. Otherwise, counting_iterator models the same iterator traversal concepts modeled by Incrementable.

counting_iterator<X,C1,D1> is interoperable with counting_iterator<Y,C2,D2> if and only if X is interoperable with Y.

counting_iterator operations

In addition to the operations required by the concepts modeled by counting_iterator, counting_iterator provides the following operations.

counting_iterator();

Requires: Incrementable is Default Constructible.
Effects: Default construct the member m_inc.

counting_iterator(const counting_iterator& rhs);

Effects: Construct member m_inc from rhs.m_inc.

explicit counting_iterator(Incrementable x);

Effects: Construct member m_inc from x.

reference operator*() const;

Returns: m_inc

counting_iterator& operator++();

Effects: ++m_inc

Returns: *this

counting_iterator& operator--();

Effects: --m_inc

Returns: *this

Incrementable const& base() const;

Returns: m_inc
Function output iterator

The function output iterator adaptor makes it easier to create custom output iterators. The adaptor takes a unary function and creates a model of Output Iterator. Each item assigned to the output iterator is passed as an argument to the unary function. The motivation for this iterator is that creating a conforming output iterator is non-trivial, particularly because the proper implementation usually requires a proxy object.

Class template function_output_iterator

```cpp
template <class UnaryFunction>
class function_output_iterator {
public:
    typedef std::output_iterator_tag iterator_category;
    typedef void value_type;
    typedef void difference_type;
    typedef void pointer;
    typedef void reference;

    explicit function_output_iterator();
    explicit function_output_iterator(const UnaryFunction& f);

    /* see below */ operator*();
    function_output_iterator& operator++();
    function_output_iterator& operator++(int);
private:
    UnaryFunction m_f;       // exposition only
};
```

**function_output_iterator requirements**

UnaryFunction must be Assignable and Copy Constructible.

**function_output_iterator models**

function_output_iterator is a model of the Writable and Incrementable Iterator concepts.

**function_output_iterator operations**

```cpp
explicit function_output_iterator(const UnaryFunction& f = UnaryFunction());

    Effects: Constructs an instance of function_output_iterator with m_f constructed from f.
    operator*();

    Returns: An object r of unspecified type such that r = t is equivalent to m_f(t) for all t.
```

```cpp
function_output_iterator& operator++();

    Returns: *this
```

```cpp
function_output_iterator& operator++(int);

    Returns: *this
```