Boost.Iterator

David Abrahams

Jeremy Siek

Thomas Witt

Table of Contents

Introduction

The Boost Iterator Library contains two parts. The first is a system of concepts which extend the C++ standard iterator requirements. The second is a framework of components for building iterators based on these extended concepts and includes several useful iterator adaptors. The extended iterator concepts have been carefully designed so that old-style iterators can fit in the new concepts and so that new-style iterators will be compatible with old-style algorithms, though algorithms may need to be updated if they want to take full advantage of the new-style iterator capabilities. Several components of this library have been accepted into the C++ standard technical report. The components of the Boost Iterator Library replace the older Boost Iterator Adaptor Library.

New-Style Iterators

The iterator categories defined in C++98 are extremely limiting because they bind together two orthogonal concepts: traversal and element access. For example, because a random access iterator is required to return a reference (and not a proxy) when dereferenced, it is impossible to capture the capabilities of `vector<bool>::iterator` using the C++98 categories. This is the infamous "`vector<bool>` is not a container, and its iterators aren't random access iterators", debacle about which Herb Sutter wrote two papers for the standards comittee (N1185 and N1211), and a Guru of the Week. New-style iterators go well beyond patching up `vector<bool>`, though: there are lots of other iterators already in use which can't be adequately represented by the existing concepts. For details about the new iterator concepts, see our Standard Proposal for New-Style Iterators.

Iterator Facade and Adaptor

Writing standard-conforming iterators is tricky, but the need comes up often. In order to ease the implementation of new iterators, the Boost.Iterator library provides the facade class template, which implements many useful defaults and compile-time checks designed to help the iterator author ensure that his iterator is correct.

It is also common to define a new iterator that is similar to some underlying iterator or iterator-like type, but that modifies some aspect of the underlying type's behavior. For that purpose, the library supplies the adaptor class template, which is specially designed to take advantage of as much of the underlying type's behavior as possible.

Both facade and adaptor as well as many of the `specialized adaptors_` mentioned below have been proposed for standardization (Standard Proposal For Iterator Facade and Adaptor).

Specialized Adaptors

The iterator library supplies a useful suite of standard-conforming iterator templates based on the Boost iterator facade and adaptor templates.

- `counting_iterator`: an iterator over a sequence of consecutive values. Implements a "lazy sequence"
- `filter_iterator`: an iterator over the subset of elements of some sequence which satisfy a given predicate
- `function_output_iterator`: an output iterator wrapping a unary function object; each time an element is written into the dereferenced iterator, it is passed as a parameter to the function object.
- `indirect_iterator`: an iterator over the objects **pointed-to** by the elements of some sequence.
- `permutation_iterator`: an iterator over the elements of some random-access sequence, rearranged according to some sequence of integer indices.
- `reverse_iterator`: an iterator which traverses the elements of some bidirectional sequence in reverse. Corrects many of the shortcomings of C++98's

  ```
  std::reverse_iterator
  ```

  .

- `shared_container_iterator`: an iterator over elements of a container whose lifetime is maintained by a `shared_ptr` stored in the iterator.
- `transform_iterator`: an iterator over elements which are the result of applying some functional transformation to the elements of an underlying sequence. This component also replaces the old

  ```
  projection_iterator_adaptor
  ```

  .
- `zip_iterator`: an iterator over tuples of the elements at corresponding positions of heterogeneous underlying iterators.

Iterator Utilities

Traits

- `pointee.hpp`: Provides the capability to deduce the referent types of pointers, smart pointers and iterators in generic code. Used in `indirect_iterator`.
- `iterator_traits.hpp`: Provides **MPL** compatible metafunctions which retrieve an iterator's traits. Also corrects for the deficiencies of broken implementations of `std::iterator_traits`.

[* |interoperable|_ (PDF__): Provides an **MPL** compatible metafunction for testing iterator interoperability]

Testing and Concept Checking

- `iterator_concepts.hpp`: Concept checking classes for the new iterator concepts.
- `iterator_archetypes.hpp`: Concept archetype classes for the new iterators concepts.

Iterator Concepts

Access

Readable Iterator Concept

A class or built-in type `X` models the **Readable Iterator** concept for value type `T` if, in addition to `X` being Assignable and Copy Constructible, the following expressions are valid and respect the stated semantics. `U` is the type of any specified member of type `T`.

Table 1. Readable Iterator Requirements (in addition to Assignable and Copy Constructible)

Expression	Return Type	Note/Precondition
`iterator_traits<X>::value_type`	`T`	Any non-reference, non cv-qualified type
`*a`	Convertible to `T`	pre: `a` is dereferenceable. If `a == b` then `*a` is equivalent to `*b`.
`a->m`	`U&`	pre: `(*a).m` is well-defined. Equivalent to `(*a).m`.

Writable Iterator Concept

A class or built-in type `X` models the **Writable Iterator** concept if, in addition to `X` being Copy Constructible, the following expressions are valid and respect the stated semantics. Writable Iterators have an associated **set of value types**.

Table 2. Writable Iterator Requirements (in addition to Copy Constructible)

Expression	Return Type	Precondition
`*a = o`		pre: The type of `o` is in the set of value types of `X`

Swappable Iterator Concept

A class or built-in type `X` models the **Swappable Iterator** concept if, in addition to `X` being Copy Constructible, the following expressions are valid and respect the stated semantics.

Table 3. Swappable Iterator Requirements (in addition to Copy Constructible)

Expression	Return Type	Postcondition
`iter_swap(a, b)`	`void`	the pointed to values are exchanged

Note: An iterator that is a model of the **Readable** and **Writable Iterator** concepts is also a model of **Swappable Iterator**. **--end note**

Lvalue Iterator Concept

The **Lvalue Iterator** concept adds the requirement that the return type of `operator*` type be a reference to the value type of the iterator.

Table 4. Lvalue Iterator Requirements

Expression	Return Type	Note/Assertion
`*a`	`T&`	`T` is **cv** `iterator_traits<X>::value_type` where **cv** is an optional cv-qualification. pre: `a` is dereferenceable. If `a == b` then `*a` is equivalent to `*b`.

Traversal

Incrementable Iterator Concept

A class or built-in type `X` models the **Incrementable Iterator** concept if, in addition to `X` being Assignable and Copy Constructible, the following expressions are valid and respect the stated semantics.

Table 5. Incrementable Iterator Requirements (in addition to Assignable, Copy Constructible)

Expression	Return Type	Assertion/Semantics
`++r`	`X&`	`&r == &++r`
`r++`	`X`	`{ X tmp = r; ++r; return tmp; }`
`iterator_traversal<X>::type`	Convertible to `incrementable_traversal_tag`	

Single Pass Iterator Concept

A class or built-in type `X` models the **Single Pass Iterator** concept if the following expressions are valid and respect the stated semantics.

Table 6. Single Pass Iterator Requirements (in addition to Incrementable Iterator and Equality Comparable)

Expression	Return Type	Assertion/Semantics / Pre-/Post-condition
`++r`	`X&`	pre: `r` is dereferenceable; post: `r` is dereferenceable or `r` is past-the-end
`a == b`	convertible to `bool`	`==` is an equivalence relation over its domain
`a != b`	convertible to `bool`	`!(a == b)`
`iterator_traversal<X>::type`	Convertible to`single_pass_traversal_tag`	

Forward Traversal Concept

A class or built-in type `X` models the **Forward Traversal** concept if, in addition to `X` meeting the requirements of Default Constructible and Single Pass Iterator, the following expressions are valid and respect the stated semantics.

Table 7. Forward Traversal Iterator Requirements (in addition to Default Constructible and Single Pass Iterator)

Expression	Return Type	Assertion/Note
`X u;`	`X&`	note: `u` may have a singular value.
`++r`	`X&`	`r == s` and `r` is dereferenceable implies `++r == ++s`.
`iterator_traits<X>::difference_type`	A signed integral type representing the distance between iterators	
`iterator_traversal<X>::type`	Convertible to `forward_traversal_tag`	

Bidirectional Traversal Concept

A class or built-in type `X` models the **Bidirectional Traversal** concept if, in addition to `X` meeting the requirements of Forward Traversal Iterator, the following expressions are valid and respect the stated semantics.

Table 8. Bidirectional Traversal Iterator Requirements (in addition to Forward Traversal Iterator)

Expression	Return Type	Assertion/Semantics/Pre-/Post-condition
`--r`	`X&`	pre: there exists `s` such that `r == ++s`. post: `s` is dereferenceable. `--(++r) == r`. `--r == --s` implies `r == s`. `&r == &--r`.
`r--`	convertible to `const X&`	`{ X tmp = r; --r; return tmp; }`
`iterator_traversal<X>::type`	Convertible to `bidirectional_traversal_tag`	

Random Access Traversal Concept

A class or built-in type `X` models the **Random Access Traversal** concept if the following expressions are valid and respect the stated semantics. In the table below, `Distance` is `iterator_traits<X>::difference_type` and `n` represents a constant object of type `Distance`.

Table 9. Random Access Traversal Iterator Requirements (in addition to Bidirectional Traversal)

Expression	Return Type	Operational Semantics	Assertion/Precondition
`r += n`	`X&`	`{ Distance m = n; if (m >= 0) while (m--) ++r; else while (m++) --r; return r; }`	
`a + n`, `n + a`	`X`	`{ X tmp = a; return tmp+= n; }`	
`r -= n`	`X&`	`return r += -n`	
`a - n`	`X`	`{ X tmp = a; return tmp-= n; }`	
`b - a`	`Distance`	`a < b ? distance(a,b) : -distance(b,a)`	pre: there exists a value `n` of `Distance` such that `a + n == b`. `b == a + (b - a)`.
`a\[n\]`	convertible to T	`*(a + n)`	pre: a is a **Readable Iterator**
`a\[n\] = v`	convertible to T	`*(a + n) = v`	pre: a is a **Writable iterator**
`a < b`	convertible to `bool`	`b - a > 0`	< is a total ordering relation
`a > b`	convertible to `bool`	`b < a`	> is a total ordering relation
`a >= b`	convertible to `bool`	`!(a < b)`	
`a <= b`	convertible to `bool`	`!(a > b)`	
`iterator_traversal<X>::type`	convertible to `random_access_traversal_tag`		

Generic Iterators

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 10. Core Interface

Expression	Effects		
`i.dereference()`	Access the value referred to	`[i.equal(j)`	Compare for equality with `j`
`i.increment()`	Advance by one position		
`i.decrement()`	Retreat by one position		
`i.advance(n)`	Advance by `n` positions	`[i.distance_to(j)`	Measure the distance to `j`

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 interoperate 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 `iterator_facade` and each of the seven operators. In order to reduce the burden of limiting access, `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 `iterator_core_access` to make his core member functions available to the library.

`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 `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.

`operator\[\]`

The indexing operator for a generalized iterator presents special challenges. A random access iterator's `operator[]` is only required to return something convertible to its `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. |counting|_), because `*(p+n)` is a reference into the temporary iterator `p+n`, which is destroyed when `operator[]` returns.

.. |counting| replace:: `counting_iterator`

Writable iterators built with `iterator_facade` implement the semantics required by the preferred resolution to `issue 299`_ and adopted by proposal n1550_: the result of `p[n]` is an object convertible to the iterator's `value_type`, and `p[n] = x` is equivalent to `*(p + n) = x` (Note: This result object may be implemented as a proxy containing a copy of `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 an `operator[]` that returns an lvalue in the derived iterator class; it will hide the one supplied by `iterator_facade` from clients of her iterator.

.. _n1550: http://anubis.dkuug.dk/JTC1/SC22/WG21/docs/papers/2003/n1550.html

.. _issue 299: http://anubis.dkuug.dk/jtc1/sc22/wg21/docs/lwg-active.html#299

.. _operator arrow:

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 `value_type`. When the `value_type` is a class, however, it must still be possible to access members through `operator->`. Therefore, an iterator whose `reference` type is not in fact a reference must return a proxy containing a copy of the referenced value from its `operator->`.

The return types for `iterator_facade`'s `operator->` and `operator[]` are not explicitly specified. Instead, those types are described in terms of a set of requirements, which must be satisfied by the `iterator_facade` implementation.

.. [Cop95] [Coplien, 1995] Coplien, J., Curiously Recurring Template Patterns, C++ Report, February 1995, pp. 24-27.

Reference

```
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;
 protected:
    typedef iterator_facade iterator_facade\_;
};

// Comparison operators
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 // exposition
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);

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&);
```

__ iterator category_

__ operator arrow_

__ brackets_

__ minus_

.. _iterator category:

The `iterator_category` member of `iterator_facade` is

.. parsed-literal::

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

where **iterator-category** is defined as follows:

.. include:: facade_iterator_category.rst

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:

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

template <typename T> enable_if_interoperable_impl<true,T>
{ typedef T type; };

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
    >
{};
```

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 `iterator_facade<X,V,C,R,D>`, `a` is an object of type `X`, `b` and `c` are objects of type `const X`, `n` is an object of `F::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`.

.. _core operations:

.. topic:: `iterator_facade` Core Operations

Table 11. Core Operations

Expression	Return Type	Assertion/Note	Used to implement Iterator Concept(s)
`c.dereference()`	`F::reference`		Readable Iterator, Writable Iterator
`c.equal(y)`	convertible to bool	true iff `c` and `y` refer to the same position	Single Pass Iterator
`a.increment()`	unused		Incrementable Iterator
`a.decrement()`	unused		Bidirectional Traversal Iterator
`a.advance(n)`	unused		Random Access Traversal Iterator
`c.distance_to(z)`	convertible to `F::difference_type`	equivalent to `distance(c, X(z))`.	Random Access Traversal Iterator

Operations

The operations in this section are described in terms of operations on the core interface of `Derived` which may be inaccessible (i.e. private). The implementation should access these operations through member functions of class `iterator_core_access`.

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

Returns: `static_cast<Derived const*>(this)->dereference()`

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

__ operator arrow_

Returns: If `reference` is a reference type, an object of type `pointer` equal to: `&static_cast<Derived const*>(this)->dereference()` Otherwise returns an object of unspecified type such that, `(*static_cast<Derived const*>(this))->m` is equivalent to `(w = **static_cast<Derived const*>(this), w.m)` for some temporary object `w` of type `value_type`.

.. _brackets:

```
*unspecified* operator[](difference_type n) const;
```

Returns: an object convertible to `value_type`. For constant objects `v` of type `value_type`, and `n` of type `difference_type`, `(*this)[n] = v` is equivalent to `*(*this + n) = v`, and `static_cast<value_type const&>((*this)[n])` is equivalent to `static_cast<value_type const&>(*(*this + n))`

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

Effects:

```
  static_cast<Derived*>(this)->increment();
  return *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,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`.
```

.. _minus:

```
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)`.
```

Tutorial

In this section we'll walk through the implementation of a few iterators using `iterator_facade`, based around the simple example of a linked list of polymorphic objects. This example was inspired by a `posting` by Keith Macdonald on the `Boost-Users` mailing list.

The Problem

Say we've written a polymorphic linked list node base class:

```
# include <iostream>

struct node_base
{
    node_base() : m_next(0) {}

    // Each node manages all of its tail nodes
    virtual ~node_base() { delete m_next; }

    // Access the rest of the list
    node_base* next() const { return m_next; }

    // print to the stream
    virtual void print(std::ostream& s) const = 0;

    // double the value
    virtual void double_me() = 0;

    void append(node_base* p)
    {
        if (m_next)
            m_next->append(p);
        else
            m_next = p;
    }

 private:
    node_base* m_next;
};
```

Lists can hold objects of different types by linking together specializations of the following template:

```
template <class T>
struct node : node_base
{
    node(T x)
      : m_value(x)
    {}

    void print(std::ostream& s) const { s << this->m_value; }
    void double_me() { m_value += m_value; }

 private:
    T m_value;
};
```

And we can print any node using the following streaming operator:

```
inline std::ostream& operator<<(std::ostream& s, node_base const& n)
{
    n.print(s);
    return s;
}
```

Our first challenge is to build an appropriate iterator over these lists.

A Basic Iterator Using `iterator_facade`

We will construct a `node_iterator` class using inheritance from `iterator_facade` to implement most of the iterator's operations.

```
# include "node.hpp"
# include <boost/iterator/iterator_facade.hpp>

class node_iterator
  : public boost::iterator_facade<...>
{
   ...
};
```

Template Arguments for `iterator_facade`

`iterator_facade` has several template parameters, so we must decide what types to use for the arguments. The parameters are `Derived`, `Value`, `CategoryOrTraversal`, `Reference`, and `Difference`.

`Derived`

Because `iterator_facade` is meant to be used with the CRTP [Cop95]_ the first parameter is the iterator class name itself, `node_iterator`.

`Value`

The `Value` parameter determines the `node_iterator`'s `value_type`. In this case, we are iterating over `node_base` objects, so `Value` will be `node_base`.

`CategoryOrTraversal`

Now we have to determine which `iterator traversal concept`_ our `node_iterator` is going to model. Singly-linked lists only have forward links, so our iterator can't can't be a `bidirectional traversal iterator`_. Our iterator should be able to make multiple passes over the same linked list (unlike, say, an `istream_iterator` which consumes the stream it traverses), so it must be a `forward traversal iterator`_. Therefore, we'll pass `boost::forward_traversal_tag` in this position _.

.. `iterator_facade` also supports old-style category tags, so we could have passed `std::forward_iterator_tag` here; either way, the resulting iterator's `iterator_category` will end up being `std::forward_iterator_tag`.

`Reference`

The `Reference` argument becomes the type returned by `node_iterator`'s dereference operation, and will also be the same as `std::iterator_traits<node_iterator>::reference`. The library's default for this parameter is `Value&`; since `node_base&` is a good choice for the iterator's `reference` type, we can omit this argument, or pass `use_default`.

`Difference`

The `Difference` argument determines how the distance between two `node_iterators` will be measured and will also be the same as `std::iterator_traits<node_iterator>::difference_type`. The library's default for `Difference` is `std::ptrdiff_t`,

an appropriate type for measuring the distance between any two addresses in memory, and one that works for almost any iterator, so we can omit this argument, too.

The declaration of `node_iterator` will therefore look something like:

```
# include "node.hpp"
# include <boost/iterator/iterator_facade.hpp>

class node_iterator
  : public boost::iterator_facade<
        node_iterator
      , node_base
      , boost::forward_traversal_tag
    >
{
   ...
};
```

Constructors and Data Members

Next we need to decide how to represent the iterator's position. This representation will take the form of data members, so we'll also need to write constructors to initialize them. The `node_iterator`'s position is quite naturally represented using a pointer to a `node_base`. We'll need a constructor to build an iterator from a `node_base*`, and a default constructor to satisfy the `forward traversal iterator_` requirements _. Our `node_iterator` then becomes:

```
# include "node.hpp"
# include <boost/iterator/iterator_facade.hpp>

class node_iterator
  : public boost::iterator_facade<
        node_iterator
      , node_base
      , boost::forward_traversal_tag
    >
{
 public:
    node_iterator()
      : m_node(0)
    {}

    explicit node_iterator(node_base* p)
      : m_node(p)
    {}

 private:
    ...
    node_base* m_node;
};
```

.. Technically, the C++ standard places almost no requirements on a default-constructed iterator, so if we were really concerned with efficiency, we could've written the default constructor to leave `m_node` uninitialized.

Implementing the Core Operations

The last step is to implement the `core operations_` required by the concepts we want our iterator to model. Referring to the table__, we can see that the first three rows are applicable because `node_iterator` needs to satisfy the requirements for `readable iterator_`, `single pass iterator_`, and `incrementable iterator_`.

__ `core operations_`

We therefore need to supply `dereference`, `equal`, and `increment` members. We don't want these members to become part of `node_iterator`'s public interface, so we can make them private and grant friendship to `boost::iterator_core_access`, a "back-door" that `iterator_facade` uses to get access to the core operations:

```
# include "node.hpp"
# include <boost/iterator/iterator_facade.hpp>

class node_iterator
  : public boost::iterator_facade<
        node_iterator
      , node_base
      , boost::forward_traversal_tag
    >
{
 public:
    node_iterator()
      : m_node(0) {}

    explicit node_iterator(node_base* p)
      : m_node(p) {}

 private:
    friend class boost::iterator_core_access;

    void increment() { m_node = m_node->next(); }

    bool equal(node_iterator const& other) const
    {
        return this->m_node == other.m_node;
    }

    node_base& dereference() const { return *m_node; }

    node_base* m_node;
};
```

Voila; a complete and conforming readable, forward-traversal iterator! For a working example of its use, see `this program`.

__ ../example/node_iterator1.cpp

A constant `node_iterator`

> **Constant and Mutable iterators**
> The term **mutable iterator** means an iterator through which the object it references (its "referent") can be modified. A **constant iterator** is one which doesn't allow modification of its referent.
> The words **constant** and **mutable** don't refer to the ability to modify the iterator itself. For example, an `int const*` is a non-`const` **constant iterator**, which can be incremented but doesn't allow modification of its referent, and `int* const` is a `const` **mutable iterator**, which cannot be modified but which allows modification of its referent.
> Confusing? We agree, but those are the standard terms. It probably doesn't help much that a container's constant iterator is called `const_iterator`.

Now, our `node_iterator` gives clients access to both `node`\ 's `print(std::ostream&) const` member function, but also its mutating `double_me()` member. If we wanted to build a **constant** `node_iterator`, we'd only have to make three changes:

```
class const_node_iterator
  : public boost::iterator_facade<
        node_iterator
      , node_base **const**
      , boost::forward_traversal_tag
    >
{
 public:
    const_node_iterator()
      : m_node(0) {}

    explicit const_node_iterator(node_base* p)
      : m_node(p) {}

 private:
    friend class boost::iterator_core_access;

    void increment() { m_node = m_node->next(); }

    bool equal(const_node_iterator const& other) const
    {
        return this->m_node == other.m_node;
    }

    node_base **const**\ & dereference() const { return \*m_node; }

    node_base **const**\ * m_node;
};
```

> `const` and an iterator's `value_type`
> The C++ standard requires an iterator's `value_type` **not** be `const`-qualified, so `iterator_facade` strips the `const` from its `Value` parameter in order to produce the iterator's `value_type`. Making the `Value` argument `const` provides a useful hint to `iterator_facade` that the iterator is a **constant iterator**, and the default `Reference` argument will be correct for all lvalue iterators.

As a matter of fact, `node_iterator` and `const_node_iterator` are so similar that it makes sense to factor the common code out into a template as follows:

```
template <class Value>
class node_iter
  : public boost::iterator_facade<
        node_iter<Value>
      , Value
      , boost::forward_traversal_tag
    >
{
 public:
    node_iter()
      : m_node(0) {}

    explicit node_iter(Value* p)
      : m_node(p) {}

 private:
    friend class boost::iterator_core_access;

    bool equal(node_iter<Value> const& other) const
    {
        return this->m_node == other.m_node;
    }

    void increment()
    { m_node = m_node->next(); }

    Value& dereference() const
    { return *m_node; }

    Value* m_node;
};
typedef node_iter<node_base> node_iterator;
typedef node_iter<node_base const> node_const_iterator;
```

Interoperability

Our `const_node_iterator` works perfectly well on its own, but taken together with `node_iterator` it doesn't quite meet expectations. For example, we'd like to be able to pass a `node_iterator` where a `node_const_iterator` was expected, just as you can with `std::list<int>`'s `iterator` and `const_iterator`. Furthermore, given a `node_iterator` and a `node_const_iterator` into the same list, we should be able to compare them for equality.

This expected ability to use two different iterator types together is known as |interoperability|_. Achieving interoperability in our case is as simple as templatizing the `equal` function and adding a templatized converting constructor _ _:

```
template <class Value>
class node_iter
  : public boost::iterator_facade<
        node_iter<Value>
      , Value
      , boost::forward_traversal_tag
    >
{
 public:
    node_iter()
      : m_node(0) {}

    explicit node_iter(Value* p)
      : m_node(p) {}

    template <class OtherValue>
    node_iter(node_iter<OtherValue> const& other)
      : m_node(other.m_node) {}

 private:
    friend class boost::iterator_core_access;
    template <class> friend class node_iter;

    template <class OtherValue>
    bool equal(node_iter<OtherValue> const& other) const
    {
        return this->m_node == other.m_node;
    }

    void increment()
    { m_node = m_node->next(); }

    Value& dereference() const
    { return *m_node; }

    Value* m_node;
};
typedef impl::node_iterator<node_base> node_iterator;
typedef impl::node_iterator<node_base const> node_const_iterator;
```

.. |interoperability| replace:: **interoperability** .. _interoperability: new-iter-concepts.html#interoperable-iterators-lib-interoperable-iterators

.. If you're using an older compiler and it can't handle this example, see the `example code`__ for workarounds.

.. If `node_iterator` had been a `random access traversal iterator`_, we'd have had to templatize its `distance_to` function as well.

__ ../example/node_iterator2.hpp

You can see an example program which exercises our interoperable iterators here.

Telling the Truth

Now `node_iterator` and `node_const_iterator` behave exactly as you'd expect... almost. We can compare them and we can convert in one direction: from `node_iterator` to `node_const_iterator`. If we try to convert from `node_const_iterator` to `node_iterator`, we'll get an error when the converting constructor tries to initialize `node_iterator`'s `m_node`, a `node*` with a `node const*`. So what's the problem?

The problem is that `boost::`|is_convertible|`_<node_const_iterator,node_iterator>::value` will be `true`, but it should be `false`. |is_convertible|_ lies because it can only see as far as the **declaration** of `node_iter`'s converting constructor, but can't

look inside at the **definition** to make sure it will compile. A perfect solution would make `node_iter`'s converting constructor disappear when the `m_node` conversion would fail.

.. |is_convertible| replace:: `is_convertible` .. _is_convertible: ../../type_traits/index.html#relationships

In fact, that sort of magic is possible using |enable_if|__. By rewriting the converting constructor as follows, we can remove it from the overload set when it's not appropriate:

```
#include <boost/type_traits/is_convertible.hpp>
#include <boost/utility/enable_if.hpp>

  ...

private:
  struct enabler {};

public:
  template <class OtherValue>
  node_iter(
      node_iter<OtherValue> const& other
    , typename boost::enable_if<
          boost::is_convertible<OtherValue*,Value*>
        , enabler
      >::type = enabler()
  )
    : m_node(other.m_node) {}
```

.. |enable_if| replace:: `boost::enable_if` __ ../../utility/enable_if.html

Wrap Up

This concludes our `iterator_facade` tutorial, but before you stop reading we urge you to take a look at |iterator_adaptor|__. There's another way to approach writing these iterators which might even be superior.

.. |iterator_adaptor| replace:: `iterator_adaptor` __ iterator_adaptor.html

.. `_iterator traversal concept`: new-iter-concepts.html#iterator-traversal-concepts-lib-iterator-traversal .. `_readable iterator`: new-iter-concepts.html#readable-iterators-lib-readable-iterators .. `_lvalue iterator`: new-iter-concepts.html#lvalue-iterators-lib-lvalue-iterators .. `_single pass iterator`: new-iter-concepts.html#single-pass-iterators-lib-single-pass-iterators .. `_incrementable iterator`: new-iter-concepts.html#incrementable-iterators-lib-incrementable-iterators .. `_forward traversal iterator`: new-iter-concepts.html#forward-traversal-iterators-lib-forward-traversal-iterators .. `_bidirectional traversal iterator`: new-iter-concepts.html#bidirectional-traversal-iterators-lib-bidirectional-traversal-iterators .. `_random access traversal iterator`: new-iter-concepts.html#random-access-traversal-iterators-lib-random-access-traversal-iterators

Iterator Adaptor

The `iterator_adaptor` class template adapts some `Base` _ type to create a new iterator. Instantiations of `iterator_adaptor` are derived from a corresponding instantiation of `iterator_facade` and implement the core behaviors in terms of the `Base` type. In essence, `iterator_adaptor` merely forwards all operations to an instance of the `Base` type, which it stores as a member.

.. 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 `reverse_iterator` adaptor.

The user of `iterator_adaptor` creates a class derived from an instantiation of `iterator_adaptor` and then selectively redefines some of the core member functions described in the `iterator_facade` core requirements table. The `Base` type need 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.

Reference

Synopsis

```
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 const& iter);
    typedef Base base_type;
    Base const& base() const;
 protected:
    typedef iterator_adaptor iterator_adaptor\_;
    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();

    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;

 private:
    Base m_iterator; // exposition only
};
```

__ base_parameters_

.. _requirements:

Requirements

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

.. _base_parameters:

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' = if (Value is use_default)
             return iterator_traits<Base>::value_type
         else
             return Value

   C' = if (CategoryOrTraversal is use_default)
             return iterator_traversal<Base>::type
         else
             return CategoryOrTraversal

   R' = if (Reference is use_default)
             if (Value is use_default)
                 return iterator_traits<Base>::reference
             else
                 return Value&
         else
             return Reference

   D' = if (Difference is use_default)
             return iterator_traits<Base>::difference_type
         else
             return Difference
```

Operations

Public

```
iterator_adaptor();
```

Requires: The `Base` type must be Default Constructible.
Returns: An instance of `iterator_adaptor` with `m_iterator` default constructed.

```
explicit iterator_adaptor(Base const& iter);
```

Returns: An instance of `iterator_adaptor` with `m_iterator` copy constructed from `iter`.

```
Base const& base() const;
```

Returns: `m_iterator`

Protected

```
Base const& base_reference() const;
```

Returns: A const reference to `m_iterator`.

```
Base& base_reference();
```

Returns: A non-const reference to `m_iterator`.

Private

```
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`

Tutorial

In this section we'll further refine the `node_iter` class template we developed in the |fac_tut|_. If you haven't already read that material, you should go back now and check it out because we're going to pick up right where it left off.

.. |fac_tut| replace:: `iterator_facade` tutorial .. _fac_tut: iterator_facade.html#tutorial-example

> **`node_base*` really is an iterator**
> It's not really a very interesting iterator, since `node_base` is an abstract class: a pointer to a `node_base` just points at some base subobject of an instance of some other class, and incrementing a `node_base*` moves it past this base subobject to who-knows-where? The most we can do with that incremented position is to compare another `node_base*` to it. In other words, the original iterator traverses a one-element array.

You probably didn't think of it this way, but the `node_base*` object that underlies `node_iterator` is itself an iterator, just like all other pointers. If we examine that pointer closely from an iterator perspective, we can see that it has much in common with the

`node_iterator` we're building. First, they share most of the same associated types (`value_type`, `reference`, `pointer`, and `difference_type`). Second, even some of the core functionality is the same: `operator*` and `operator==` on the `node_iterator` return the result of invoking the same operations on the underlying pointer, via the `node_iterator`'s |dereference_and_equal|_). The only real behavioral difference between `node_base*` and `node_iterator` can be observed when they are incremented: `node_iterator` follows the `m_next` pointer, while `node_base*` just applies an address offset.

.. |dereference_and_equal| replace:: `dereference` and `equal` member functions .. _dereference_and_equal: iterator_facade.html#implementing-the-core-operations

It turns out that the pattern of building an iterator on another iterator-like type (the `Base` _ type) while modifying just a few aspects of the underlying type's behavior is an extremely common one, and it's the pattern addressed by `iterator_adaptor`. Using `iterator_adaptor` is very much like using `iterator_facade`, but because iterator_adaptor tries to mimic as much of the `Base` type's behavior as possible, we neither have to supply a `Value` argument, nor implement any core behaviors other than `increment`. The implementation of `node_iter` is thus reduced to:

```
template <class Value>
class node_iter
  : public boost::iterator_adaptor<
        node_iter<Value>                // Derived
      , Value*                          // Base
      , boost::use_default              // Value
      , boost::forward_traversal_tag    // CategoryOrTraversal
    >
{
 private:
    struct enabler {};  // a private type avoids misuse

 public:
    node_iter()
      : node_iter::iterator_adaptor_(0) {}

    explicit node_iter(Value* p)
      : node_iter::iterator_adaptor_(p) {}

    template <class OtherValue>
    node_iter(
        node_iter<OtherValue> const& other
      , typename boost::enable_if<
            boost::is_convertible<OtherValue*,Value*>
          , enabler
        >::type = enabler()
    )
      : node_iter::iterator_adaptor_(other.base()) {}

 private:
    friend class boost::iterator_core_access;
    void increment() { this->base_reference() = this->base()->next(); }
};
```

Note the use of `node_iter::iterator_adaptor_` here: because `iterator_adaptor` defines a nested `iterator_adaptor_` type that refers to itself, that gives us a convenient way to refer to the complicated base class type of `node_iter<Value>`. [Note: this technique is known not to work with Borland C++ 5.6.4 and Metrowerks CodeWarrior versions prior to 9.0]

You can see an example program that exercises this version of the node iterators `here`.

In the case of `node_iter`, it's not very compelling to pass `boost::use_default` as `iterator_adaptor` 's `Value` argument; we could have just passed `node_iter` 's `Value` along to `iterator_adaptor`, and that'd even be shorter! Most iterator class templates built with `iterator_adaptor` are parameterized on another iterator type, rather than on its `value_type`. For example, `boost::reverse_iterator` takes an iterator type argument and reverses its direction of traversal, since the original iterator and the reversed one have all the same associated types, `iterator_adaptor` 's delegation of default types to its `Base` saves the implementor of `boost::reverse_iterator` from writing:

```
std::iterator_traits<Iterator>::*some-associated-type*
```

at least four times.

We urge you to review the documentation and implementations of |reverse_iterator|_ and the other Boost `specialized iterator adaptors`__ to get an idea of the sorts of things you can do with `iterator_adaptor`. In particular, have a look at |transform_iterator|, which is perhaps the most straightforward adaptor, and also |counting_iterator|, which demonstrates that `iterator_adaptor`'s `Base` type needn't be an iterator.

.. |reverse_iterator| replace:: `reverse_iterator` .. _reverse_iterator: reverse_iterator.html

.. |counting_iterator| replace:: `counting_iterator` .. _counting_iterator: counting_iterator.html

.. |transform_iterator| replace:: `transform_iterator` .. _transform_iterator: transform_iterator.html

__ index.html#specialized-adaptors

Specialized Adaptors

Counting Iterator

A `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.

Example

This example fills an array with numbers and a second array with pointers into the first array, using `counting_iterator` for both tasks. Finally `indirect_iterator` is used to print out the numbers into the first array via indirection through the second array.

```
int N = 7;
std::vector<int> numbers;
typedef std::vector<int>::iterator n_iter;
std::copy(boost::counting_iterator<int>(0),
          boost::counting_iterator<int>(N),
          std::back_inserter(numbers));

std::vector<std::vector<int>::iterator> pointers;
std::copy(boost::make_counting_iterator(numbers.begin()),
      boost::make_counting_iterator(numbers.end()),
      std::back_inserter(pointers));

std::cout << "indirectly printing out the numbers from 0 to "
      << N << std::endl;
std::copy(boost::make_indirect_iterator(pointers.begin()),
      boost::make_indirect_iterator(pointers.end()),
      std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
```

The output is:

```
indirectly printing out the numbers from 0 to 7
0 1 2 3 4 5 6
```

The source code for this example can be found here.

Reference

Synopsis

```
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.

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 must also be valid:

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

Concepts

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`.

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(counting_iterator const& 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`

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.

Example

This example uses `filter_iterator` and then `make_filter_iterator` to output only the positive integers from an array of integers. Then `make_filter_iterator` is is used to output the integers greater than `-2`.

```
struct is_positive_number {
  bool operator()(int x) { return 0 < x; }
};

int main()
{
  int numbers_[] = { 0, -1, 4, -3, 5, 8, -2 };
  const int N = sizeof(numbers_)/sizeof(int);

  typedef int* base_iterator;
  base_iterator numbers(numbers_);

  // Example using filter_iterator
  typedef boost::filter_iterator<is_positive_number, base_iterator>
    FilterIter;

  is_positive_number predicate;
  FilterIter filter_iter_first(predicate, numbers, numbers + N);
  FilterIter filter_iter_last(predicate, numbers + N, numbers + N);

  std::copy(filter_iter_first, filter_iter_last, std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;

  // Example using make_filter_iterator()
  std::copy(boost::make_filter_iterator<is_positive_number>(numbers, numbers + N),
            boost::make_filter_iterator<is_positive_number>(numbers + N, numbers + N),
            std::ostream_iterator<int>(std::cout, " "));
  std::cout << std::endl;

  // Another example using make_filter_iterator()
  std::copy(
      boost::make_filter_iterator(
          std::bind2nd(std::greater<int>(), -2)
        , numbers, numbers + N)

    , boost::make_filter_iterator(
          std::bind2nd(std::greater<int>(), -2)
        , numbers + N, numbers + N)

    , std::ostream_iterator<int>(std::cout, " ")
```

```
  );

  std::cout << std::endl;

  return boost::exit_success;
}
```

The output is:

```
4 5 8
4 5 8
0 -1 4 5 8
```

The source code for this example can be found here.

Reference

Synopsis

```
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
        );
    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 Bidirectional Traversal Iterator then `iterator_category` is convertible to `std::bidirectional_iterator_tag`. Otherwise, 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`.

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`.

Concepts

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

Table 12. Traversal

If `Iterator` models	**then `filter_iterator` models**
Single Pass Iterator	Single Pass Iterator
Forward Traversal Iterator	Forward Traversal Iterator
Bidirectional Traversal Iterator	Bidirectional Traversal Iterator

Table 13. Access

If `Iterator` models	**then `filter_iterator` models**
Readable Iterator	Readable Iterator
Writable Iterator	Writable Iterator
Lvalue Iterator	Lvalue Iterator

Table 14. C++03

If `Iterator` models	**then `filter_iterator` models**
Readable Iterator, Single Pass Iterator	Input Iterator
Readable Lvalue Iterator, Forward Traversal Iterator	Forward Iterator
Writable Lvalue Iterator, Forward Traversal Iterator	Mutable Forward Iterator
Writable Lvalue Iterator, Bidirectional Iterator	Mutable Bidirectional Iterator

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

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());
```

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`

```
Ierator 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`

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.

Example

```
struct string_appender
{
    string_appender(std::string& s)
        : m_str(&s)
    {}

    void operator()(const std::string& x) const
    {
        *m_str += x;
    }

    std::string* m_str;
};

int main(int, char*[])
{
  std::vector<std::string> x;
  x.push_back("hello");
  x.push_back(" ");
  x.push_back("world");
  x.push_back("!");

  std::string s = "";
  std::copy(x.begin(), x.end(),
    boost::make_function_output_iterator(string_appender(s)));

  std::cout << s << std::endl;

  return 0;
}
```

Reference

Synopsis

```
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
};
```

Requirements

`UnaryFunction` must be Assignable and Copy Constructible.

Concepts

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

Operations

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

Effects: Constructs an instance of `function_output_iterator` with `m_f` constructed from `f`.

```
unspecified_type operator*();
```

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

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

Returns: `*this`.

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

Returns: `*this`.

Indirect Iterator

`indirect_iterator` adapts an iterator by applying an **extra** dereference inside of `operator*()`. For example, this iterator adaptor makes it possible to view a container of pointers (e.g. `list<foo*>`) as if it were a container of the pointed-to type (e.g. `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.

Example

This example prints an array of characters, using `indirect_iterator` to access the array of characters through an array of pointers. Next `indirect_iterator` is used with the `transform` algorithm to copy the characters (incremented by one) to another array. A constant indirect iterator is used for the source and a mutable indirect iterator is used for the destination. The last part of the example prints the original array of characters, but this time using the `make_indirect_iterator` helper function.

```
char characters[] = "abcdefg";
const int N = sizeof(characters)/sizeof(char) - 1; // -1 since characters has a null char
char* pointers_to_chars[N];                        // at the end.
for (int i = 0; i < N; ++i)
  pointers_to_chars[i] = &characters[i];

// Example of using indirect_iterator

boost::indirect_iterator<char**, char>
  indirect_first(pointers_to_chars), indirect_last(pointers_to_chars + N);

std::copy(indirect_first, indirect_last, std::ostream_iterator<char>(std::cout, ","));
std::cout << std::endl;


// Example of making mutable and constant indirect iterators

char mutable_characters[N];
char* pointers_to_mutable_chars[N];
for (int j = 0; j < N; ++j)
  pointers_to_mutable_chars[j] = &mutable_characters[j];

boost::indirect_iterator<char* const*> mutable_indirect_first(pointers_to_mutable_chars),
  mutable_indirect_last(pointers_to_mutable_chars + N);
boost::indirect_iterator<char* const*, char const> const_indirect_first(pointers_to_chars),
  const_indirect_last(pointers_to_chars + N);

std::transform(const_indirect_first, const_indirect_last,
       mutable_indirect_first, std::bind1st(std::plus<char>(), 1));

std::copy(mutable_indirect_first, mutable_indirect_last,
      std::ostream_iterator<char>(std::cout, ","));
std::cout << std::endl;


// Example of using make_indirect_iterator()

std::copy(boost::make_indirect_iterator(pointers_to_chars),
      boost::make_indirect_iterator(pointers_to_chars + N),
      std::ostream_iterator<char>(std::cout, ","));
std::cout << std::endl;
```

The output is:

```
a,b,c,d,e,f,g,
b,c,d,e,f,g,h,
a,b,c,d,e,f,g,
```

The source code for this example can be found here.

Reference

Synopsis

```
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(
        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;
```

Requirements

The expression `*v`, where `v` is an object of `iterator_traits<Iterator>::value_type`, shall be valid expression and convertible to `reference`. `Iterator` shall model the traversal concept indicated by `iterator_category`. `Value`, `Reference`, and `Difference` shall be chosen so that `value_type`, `reference`, and `difference_type` meet the requirements indicated by `iterator_category`.

> Note: there are further requirements on the `iterator_traits<Iterator>::value_type` if the `Value` parameter is not `use_default`, as implied by the algorithm for deducing the default for the `value_type` member.

Concepts

In addition to the concepts indicated by `iterator_category` and by `iterator_traversal<indirect_iterator>::type`, a specialization of `indirect_iterator` models the following concepts, Where `v` is an object of `iterator_traits<Iterator>::value_type`:

Readable Iterator if `reference(*v)` is convertible to `value_type`.

Writable Iterator if `reference(*v) = t` is a valid expression (where `t` is an object of type `indirect_iterator::value_type`)

Lvalue Iterator if `reference` is a reference type.

`indirect_iterator<X,V1,C1,R1,D1>` is interoperable with `indirect_iterator<Y,V2,C2,R2,D2>` if and only if `X` is interoperable with `Y`.

Operations

In addition to the operations required by the concepts described above, specializations of `indirect_iterator` provide the following operations:

```
indirect_iterator();
```

Requires: `Iterator` must be Default Constructible.
Effects: Constructs an instance of `indirect_iterator` with a default-constructed `m_iterator`.

```
indirect_iterator(Iterator x);
```

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

```
template <
    class Iterator2, class Value2, unsigned Access, class Traversal
  , class Reference2, class Difference2
>
indirect_iterator(
    indirect_iterator<
         Iterator2, Value2, Access, Traversal, Reference2, Difference2
    > const& y
  , typename enable_if_convertible<Iterator2, Iterator>::type* = 0 // exposition
);
```

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

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

Returns: `m_iterator`

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

Returns: `**m_iterator`

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

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

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

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

Permutation Iterator

The permutation iterator adaptor provides a permuted view of a given range. That is, the view includes every element of the given range but in a potentially different order. The adaptor takes two arguments:

- an iterator to the range V on which the permutation will be applied
- the reindexing scheme that defines how the elements of V will be permuted.

Note that the permutation iterator is not limited to strict permutations of the given range V. The distance between begin and end of the reindexing iterators is allowed to be smaller compared to the size of the range V, in which case the permutation iterator only provides a permutation of a subrange of V. The indexes neither need to be unique. In this same context, it must be noted that the past the end permutation iterator is completely defined by means of the past-the-end iterator to the indices.

Example

```
using namespace boost;
int i = 0;

typedef std::vector< int > element_range_type;
typedef std::list< int > index_type;

static const int element_range_size = 10;
static const int index_size = 4;

element_range_type elements( element_range_size );
for(element_range_type::iterator el_it = elements.begin() ; el_it != elements.end() ; ++el_it)
  *el_it = std::distance(elements.begin(), el_it);

index_type indices( index_size );
for(index_type::iterator i_it = indices.begin() ; i_it != indices.end() ; ++i_it )
  *i_it = element_range_size - index_size + std::distance(indices.begin(), i_it);
std::reverse( indices.begin(), indices.end() );

typedef permutation_iterator< element_range_type::iterator, index_type::iterator > permuta↵
tion_type;
permutation_type begin = make_permutation_iterator( elements.begin(), indices.begin() );
permutation_type it = begin;
permutation_type end = make_permutation_iterator( elements.begin(), indices.end() );

std::cout << "The original range is : ";
std::copy( elements.begin(), elements.end(), std::ostream_iterator< int >( std::cout, " " ) );
std::cout << "\n";

std::cout << "The reindexing scheme is : ";
std::copy( indices.begin(), indices.end(), std::ostream_iterator< int >( std::cout, " " ) );
std::cout << "\n";

std::cout << "The permutated range is : ";
std::copy( begin, end, std::ostream_iterator< int >( std::cout, " " ) );
std::cout << "\n";

std::cout << "Elements at even indices in the permutation : ";
it = begin;
for(i = 0; i < index_size / 2 ; ++i, it+=2 ) std::cout << *it << " ";
std::cout << "\n";

std::cout << "Permutation backwards : ";
it = begin + (index_size);
assert( it != begin );
for( ; it-- != begin ; ) std::cout << *it << " ";
std::cout << "\n";

std::cout << "Iterate backward with stride 2 : ";
it = begin + (index_size - 1);
for(i = 0 ; i < index_size / 2 ; ++i, it-=2 ) std::cout << *it << " ";
std::cout << "\n";
```

The output is:

```
The original range is : 0 1 2 3 4 5 6 7 8 9
The reindexing scheme is : 9 8 7 6
The permutated range is : 9 8 7 6
Elements at even indices in the permutation : 9 7
Permutation backwards : 6 7 8 9
Iterate backward with stride 2 : 6 8
```

The source code for this example can be found here.

Reference

Synopsis

```
template< class ElementIterator
    , class IndexIterator
    , class ValueT        = use_default
    , class CategoryT     = use_default
    , class ReferenceT    = use_default
    , class DifferenceT   = use_default >
class permutation_iterator
{
public:
  permutation_iterator();
  explicit permutation_iterator(ElementIterator x, IndexIterator y);

  template< class OEIter, class OIIter, class V, class C, class R, class D >
  permutation_iterator(
  permutation_iterator<OEIter, OIIter, V, C, R, D> const& r
  , typename enable_if_convertible<OEIter, ElementIterator>::type* = 0
  , typename enable_if_convertible<OIIter, IndexIterator>::type* = 0
  );
  reference operator*() const;
  permutation_iterator& operator++();
  ElementIterator const& base() const;
private:
  ElementIterator m_elt;      // exposition only
  IndexIterator m_order;      // exposition only
};

template <class ElementIterator, class IndexIterator>
permutation_iterator<ElementIterator, IndexIterator>
make_permutation_iterator( ElementIterator e, IndexIterator i);
```

Requirements

`ElementIterator` shall model Random Access Traversal Iterator. `IndexIterator` shall model Readable Iterator. The value type of the `IndexIterator` must be convertible to the difference type of `ElementIterator`.

Concepts

`permutation_iterator` models the same iterator traversal concepts as `IndexIterator` and the same iterator access concepts as `ElementIterator`.

If `IndexIterator` models Single Pass Iterator and `ElementIterator` models Readable Iterator then `permutation_iterator` models Input Iterator.

If `IndexIterator` models Forward Traversal Iterator and `ElementIterator` models Readable Lvalue Iterator then `permutation_iterator` models Forward Iterator.

If `IndexIterator` models Bidirectional Traversal Iterator and `ElementIterator` models Readable Lvalue Iterator then `permutation_iterator` models Bidirectional Iterator.

If `IndexIterator` models Random Access Traversal Iterator and `ElementIterator` models Readable Lvalue Iterator then `permutation_iterator` models Random Access Iterator.

`permutation_iterator<E1, X, V1, C2, R1, D1>` is interoperable with `permutation_iterator<E2, Y, V2, C2, R2, D2>` if and only if `X` is interoperable with `Y` and `E1` is convertible to `E2`.

Operations

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

```
permutation_iterator();
```

Effects: Default constructs `m_elt` and `m_order`.

```
explicit permutation_iterator(ElementIterator x, IndexIterator y);
```

Effects: Constructs `m_elt` from `x` and `m_order` from `y`.

```
template< class OEIter, class OIIter, class V, class C, class R, class D >
permutation_iterator(
permutation_iterator<OEIter, OIIter, V, C, R, D> const& r
, typename enable_if_convertible<OEIter, ElementIterator>::type* = 0
, typename enable_if_convertible<OIIter, IndexIterator>::type* = 0
);
```

Effects: Constructs `m_elt` from `r.m_elt` and `m_order` from `y.m_order`.

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

Returns: `*(m_elt + *m_order)`

```
permutation_iterator& operator++();
```

Effects: `++m_order`
Returns: `*this`

```
ElementIterator const& base() const;
```

Returns: `m_order`

```
template <class ElementIterator, class IndexIterator>
permutation_iterator<ElementIterator, IndexIterator>
make_permutation_iterator(ElementIterator e, IndexIterator i);
```

Returns: `permutation_iterator<ElementIterator, IndexIterator>(e, i)`

Reverse Iterator

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

Example

The following example prints an array of characters in reverse order using `reverse_iterator`.

```
char letters_[] = "hello world!";
const int N = sizeof(letters_)/sizeof(char) - 1;
typedef char* base_iterator;
base_iterator letters(letters_);
std::cout << "original sequence of letters:\t\t\t" << letters_ << std::endl;

boost::reverse_iterator<base_iterator>
  reverse_letters_first(letters + N),
  reverse_letters_last(letters);

std::cout << "sequence in reverse order:\t\t\t";
std::copy(reverse_letters_first, reverse_letters_last,
          std::ostream_iterator<char>(std::cout));
std::cout << std::endl;

std::cout << "sequence in double-reversed (normal) order:\t";
std::copy(boost::make_reverse_iterator(reverse_letters_last),
          boost::make_reverse_iterator(reverse_letters_first),
          std::ostream_iterator<char>(std::cout));
std::cout << std::endl;
```

The output is:

```
original sequence of letters:                   hello world!
sequence in reverse order:                      !dlrow olleh
sequence in double-reversed (normal) order:     hello world!
```

The source code for this example can be found here.

Reference

Synopsis

```
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`.

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`.

Concepts

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 15. Categories

If `I` models	then `reverse_iterator<I>` models
Readable Lvalue Iterator, Bidirectional Traversal Iterator	Bidirectional Iterator
Writable Lvalue Iterator, Bidirectional Traversal Iterator	Mutable Bidirectional Iterator
Readable Lvalue Iterator, Random Access Traversal Iterator	Random Access Iterator
Writable Lvalue Iterator, Random Access Traversal Iterator	Mutable Random Access Iterator

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

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`

Shared Container Iterator

Defined in header `boost/shared_container_iterator.hpp`.

The purpose of the shared container iterator is to attach the lifetime of a container to the lifetime of its iterators. In other words, the container will not be deleted until after all its iterators are destroyed. The shared container iterator is typically used to implement functions that return iterators over a range of objects that only need to exist for the lifetime of the iterators. By returning a pair of shared iterators from a function, the callee can return a heap-allocated range of objects whose lifetime is automatically managed.

The shared container iterator augments an iterator over a shared container. It maintains a reference count on the shared container. If only shared container iterators hold references to the container, the container's lifetime will end when the last shared container iterator over it is destroyed. In any case, the shared container is guaranteed to persist beyond the lifetime of all the iterators. In all other ways, the shared container iterator behaves the same as its base iterator.

Synopsis

```
namespace boost {
  template <typename Container>
  class shared_container_iterator;

  template <typename Container>
  shared_container_iterator<Container>
  make_shared_container_iterator(typename Container::iterator base,
    boost::shared_ptr<Container> const& container);

  std::pair<
    typename shared_container_iterator<Container>,
    typename shared_container_iterator<Container>
  >
  make_shared_container_range(boost::shared_ptr<Container> const& container);
}
```

The Shared Container Iterator Type

```
template <typename Container> class shared_container_iterator;
```

The class template `shared_container_iterator` is the shared container iterator type. The `Container` template type argument must model the Container concept.

Example

The following example illustrates how to create an iterator that regulates the lifetime of a reference counted `std::vector`. Though the original shared pointer `ints` ceases to exist after `set_range()` returns, the `shared_counter_iterator` objects maintain references to the underlying vector and thereby extend the container's lifetime.

`shared_iterator_example1.cpp`:

```
#include "shared_container_iterator.hpp"
#include "boost/shared_ptr.hpp"
#include <algorithm>
#include <iostream>
#include <vector>

typedef boost::shared_container_iterator< std::vector<int> > iterator;


void set_range(iterator& i, iterator& end)  {

  boost::shared_ptr< std::vector<int> > ints(new std::vector<int>());

  ints->push_back(0);
  ints->push_back(1);
  ints->push_back(2);
  ints->push_back(3);
  ints->push_back(4);
  ints->push_back(5);

  i = iterator(ints->begin(),ints);
  end = iterator(ints->end(),ints);
}


int main() {

  iterator i,end;

  set_range(i,end);

  std::copy(i,end,std::ostream_iterator<int>(std::cout,","));
  std::cout.put('\n');

  return 0;
}
```

The output from this part is:

```
0,1,2,3,4,5,
```

Table 16. Template Parameters

Parameter	Description
Container	The type of the container that we wish to iterate over. It must be a model of the Container concept.

Concepts

The `shared_container_iterator` type models the same iterator concept as the base iterator (`Container::iterator`).

Operations

The `shared_container_iterator` type implements the member functions and operators required of the Random Access Iterator concept, though only operations defined for the base iterator will be valid. In addition it has the following constructor:

```
shared_container_iterator(Container::iterator const& it,
                          boost::shared_ptr<Container> const& container)
```

The Shared Container Iterator Object Generator

```
template <typename Container>
shared_container_iterator<Container>
make_shared_container_iterator(Container::iterator base,
                               boost::shared_ptr<Container> const& container)
```

This function provides an alternative to directly constructing a `shared_container_iterator`. Using the object generator, a `shared_container_iterator` can be created and passed to a function without explicitly specifying its type.

Example

This example, similar to the previous, uses `make_shared_container_iterator()` to create the iterators.

shared_iterator_example2.cpp:

```
#include "shared_container_iterator.hpp"
#include "boost/shared_ptr.hpp"
#include <algorithm>
#include <iterator>
#include <iostream>
#include <vector>


template <typename Iterator>
void print_range_nl (Iterator begin, Iterator end) {
  typedef typename std::iterator_traits<Iterator>::value_type val;
  std::copy(begin,end,std::ostream_iterator<val>(std::cout,","));
  std::cout.put('\n');
}


int main() {

  typedef boost::shared_ptr< std::vector<int> > ints_t;
  {
    ints_t ints(new std::vector<int>());

    ints->push_back(0);
    ints->push_back(1);
    ints->push_back(2);
    ints->push_back(3);
    ints->push_back(4);
    ints->push_back(5);

    print_range_nl(boost::make_shared_container_iterator(ints->begin(),ints),
                   boost::make_shared_container_iterator(ints->end(),ints));
  }



  return 0;
}
```

Observe that the `shared_container_iterator` type is never explicitly named. The output from this example is the same as the previous.

The Shared Container Iterator Range Generator

```
template <typename Container>
std::pair<
  shared_container_iterator<Container>,
  shared_container_iterator<Container>
>
make_shared_container_range(boost::shared_ptr<Container> const& container);
Class shared_container_iterator is meant primarily to return, using iterators, a range of val↵
ues that we can guarantee will be alive as long as the iterators are. This is a convenience func↵
tion to do just that. It is equivalent to
std::make_pair(make_shared_container_iterator(container->begin(),container),
               make_shared_container_iterator(container->end(),container));
```

Example

In the following example, a range of values is returned as a pair of shared_container_iterator objects.

shared_iterator_example3.cpp:

```
#include "shared_container_iterator.hpp"
#include "boost/shared_ptr.hpp"
#include "boost/tuple/tuple.hpp" // for boost::tie
#include <algorithm>              // for std::copy
#include <iostream>
#include <vector>


typedef boost::shared_container_iterator< std::vector<int> > iterator;

std::pair<iterator,iterator>
return_range() {
  boost::shared_ptr< std::vector<int> > range(new std::vector<int>());
  range->push_back(0);
  range->push_back(1);
  range->push_back(2);
  range->push_back(3);
  range->push_back(4);
  range->push_back(5);
  return boost::make_shared_container_range(range);
}


int main() {


  iterator i,end;

  boost::tie(i,end) = return_range();

  std::copy(i,end,std::ostream_iterator<int>(std::cout,","));
  std::cout.put('\n');

  return 0;
}
```

Though the range object only lives for the duration of the `return_range` call, the reference counted `std::vector` will live until `i` and `end` are both destroyed. The output from this example is the same as the previous two.

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.

Example

This is a simple example of using the transform_iterators class to generate iterators that multiply (or add to) the value returned by dereferencing the iterator. It would be cooler to use lambda library in this example.

```
int x[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
const int N = sizeof(x)/sizeof(int);

typedef boost::binder1st< std::multiplies<int> > Function;
typedef boost::transform_iterator<Function, int*> doubling_iterator;

doubling_iterator i(x, boost::bind1st(std::multiplies<int>(), 2)),
  i_end(x + N, boost::bind1st(std::multiplies<int>(), 2));

std::cout << "multiplying the array by 2:" << std::endl;
while (i != i_end)
  std::cout << *i++ << " ";
std::cout << std::endl;

std::cout << "adding 4 to each element in the array:" << std::endl;
std::copy(boost::make_transform_iterator(x, boost::bind1st(std::plus<int>(), 4)),
     boost::make_transform_iterator(x + N, boost::bind1st(std::plus<int>(), 4)),
     std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
```

The output is:

```
multiplying the array by 2:
2 4 6 8 10 12 14 16
adding 4 to each element in the array:
5 6 7 8 9 10 11 12
```

The source code for this example can be found here.

Reference

Synopsis

```
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
      , typename enable_if_convertible<I2, Iterator>::type* = 0      // exposition only
      , typename enable_if_convertible<F2, UnaryFunction>::type* = 0 // exposition only
  );
  UnaryFunction functor() const;
  Iterator const& base() const;
  reference operator*() const;
  transform_iterator& operator++();
  transform_iterator& operator--();
private:
  Iterator m_iterator; // exposition only
  UnaryFunction m_f;   // exposition only
};
```

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`.

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.

Concepts

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 17. Category

If `Iterator` models	**then `transform_iterator` models**
Single Pass Iterator	Input Iterator
Forward Traversal Iterator	Forward Iterator
Bidirectional Traversal Iterator	Bidirectional Iterator
Random Access Traversal Iterator	Random Access Iterator

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`.

Operations

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

```
transform_iterator();
```

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

```
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`.

```
template<class F2, class I2, class R2, class V2>
transform_iterator(
      transform_iterator<F2, I2, R2, V2> const& t
    , typename enable_if_convertible<I2, Iterator>::type* = 0    // exposition only
    , typename enable_if_convertible<F2, UnaryFunction>::type* = 0 // exposition only
);
```

Returns: An instance of `transform_iterator` with `m_f` initialized to `t.functor()` and `m_iterator` initialized to `t.base()`.
Requires: `OtherIterator` is implicitly convertible to `Iterator`.

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

Returns: `m_f`

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

Returns: `m_iterator`

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

Returns: `m_f(*m_iterator)`

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

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

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

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

Zip Iterator

The zip iterator provides the ability to parallel-iterate over several controlled sequences simultaneously. A zip iterator is constructed from a tuple of iterators. Moving the zip iterator moves all the iterators in parallel. Dereferencing the zip iterator returns a tuple that contains the results of dereferencing the individual iterators.

Example

There are two main types of applications of the `zip_iterator`. The first one concerns runtime efficiency: If one has several controlled sequences of the same length that must be somehow processed, e.g., with the `for_each` algorithm, then it is more efficient to perform just one parallel-iteration rather than several individual iterations. For an example, assume that `vect_of_doubles` and `vect_of_ints` are two vectors of equal length containing doubles and ints, respectively, and consider the following two iterations:

```
std::vector<double>::const_iterator beg1 = vect_of_doubles.begin();
std::vector<double>::const_iterator end1 = vect_of_doubles.end();
std::vector<int>::const_iterator beg2 = vect_of_ints.begin();
std::vector<int>::const_iterator end2 = vect_of_ints.end();

std::for_each(beg1, end1, func_0());
std::for_each(beg2, end2, func_1());
```

These two iterations can now be replaced with a single one as follows:

```
std::for_each(
  boost::make_zip_iterator(
    boost::make_tuple(beg1, beg2)
    ),
  boost::make_zip_iterator(
    boost::make_tuple(end1, end2)
    ),
  zip_func()
  );
```

A non-generic implementation of `zip_func` could look as follows:

```
struct zip_func :
  public std::unary_function<const boost::tuple<const double&, const int&>&, void>
{
  void operator()(const boost::tuple<const double&, const int&>& t) const
  {
    m_f0(t.get<0>());
    m_f1(t.get<1>());
  }

private:
  func_0 m_f0;
  func_1 m_f1;
};
```

The second important application of the `zip_iterator` is as a building block to make combining iterators. A combining iterator is an iterator that parallel-iterates over several controlled sequences and, upon dereferencing, returns the result of applying a functor to the values of the sequences at the respective positions. This can now be achieved by using the `zip_iterator` in conjunction with the `transform_iterator`.

Suppose, for example, that you have two vectors of doubles, say `vect_1` and `vect_2`, and you need to expose to a client a controlled sequence containing the products of the elements of `vect_1` and `vect_2`. Rather than placing these products in a third vector, you can use a combining iterator that calculates the products on the fly. Let us assume that `tuple_multiplies` is a functor that works like `std::multiplies`, except that it takes its two arguments packaged in a tuple. Then the two iterators `it_begin` and `it_end` defined below delimit a controlled sequence containing the products of the elements of `vect_1` and `vect_2`:

```
typedef boost::tuple<
  std::vector<double>::const_iterator,
  std::vector<double>::const_iterator
  > the_iterator_tuple;

typedef boost::zip_iterator<
  the_iterator_tuple
  > the_zip_iterator;

typedef boost::transform_iterator<
  tuple_multiplies<double>,
  the_zip_iterator
  > the_transform_iterator;

the_transform_iterator it_begin(
  the_zip_iterator(
    the_iterator_tuple(
      vect_1.begin(),
      vect_2.begin()
      )
    ),
  tuple_multiplies<double>()
  );

the_transform_iterator it_end(
  the_zip_iterator(
    the_iterator_tuple(
      vect_1.end(),
      vect_2.end()
      )
    ),
  tuple_multiplies<double>()
  );
```

Reference

Synopsis

```
template<typename IteratorTuple>
class zip_iterator
{

public:
  typedef /* see below */ reference;
  typedef reference value_type;
  typedef value_type* pointer;
  typedef /* see below */ difference_type;
  typedef /* see below */ iterator_category;

  zip_iterator();
  zip_iterator(IteratorTuple iterator_tuple);

  template<typename OtherIteratorTuple>
  zip_iterator(
        const zip_iterator<OtherIteratorTuple>& other
      , typename enable_if_convertible<
            OtherIteratorTuple
          , IteratorTuple>::type* = 0     // exposition only
  );

  const IteratorTuple& get_iterator_tuple() const;

private:
  IteratorTuple m_iterator_tuple;     // exposition only
};

template<typename IteratorTuple>
zip_iterator<IteratorTuple>
make_zip_iterator(IteratorTuple t);
```

The `reference` member of `zip_iterator` is the type of the tuple made of the reference types of the iterator types in the `IteratorTuple` argument.

The `difference_type` member of `zip_iterator` is the `difference_type` of the first of the iterator types in the `IteratorTuple` argument.

The `iterator_category` member of `zip_iterator` is convertible to the minimum of the traversal categories of the iterator types in the `IteratorTuple` argument. For example, if the `zip_iterator` holds only vector iterators, then `iterator_category` is convertible to `boost::random_access_traversal_tag`. If you add a list iterator, then `iterator_category` will be convertible to `boost::bidirectional_traversal_tag`, but no longer to `boost::random_access_traversal_tag`.

Requirements

All iterator types in the argument `IteratorTuple` shall model Readable Iterator.

Concepts

The resulting `zip_iterator` models Readable Iterator.

The fact that the `zip_iterator` models only Readable Iterator does not prevent you from modifying the values that the individual iterators point to. The tuple returned by the `zip_iterator`'s `operator*` is a tuple constructed from the reference types of the individual iterators, not their value types. For example, if `zip_it` is a `zip_iterator` whose first member iterator is an `std::vec-`

tor<double>::iterator, then the following line will modify the value which the first member iterator of zip_it currently points to:

```
zip_it->get<0>() = 42.0;
```

Consider the set of standard traversal concepts obtained by taking the most refined standard traversal concept modeled by each individual iterator type in the IteratorTuple argument.The zip_iterator models the least refined standard traversal concept in this set.

zip_iterator<IteratorTuple1> is interoperable with zip_iterator<IteratorTuple2> if and only if IteratorTuple1 is interoperable with IteratorTuple2.

Operations

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

```
zip_iterator();
```

Returns: An instance of zip_iterator with m_iterator_tuple default constructed.

```
zip_iterator(IteratorTuple iterator_tuple);
```

Returns: An instance of zip_iterator with m_iterator_tuple initialized to iterator_tuple.

```
template<typename OtherIteratorTuple>
zip_iterator(
      const zip_iterator<OtherIteratorTuple>& other
    , typename enable_if_convertible<
          OtherIteratorTuple
        , IteratorTuple>::type* = 0     // exposition only
);
```

Returns: An instance of zip_iterator that is a copy of other.
Requires: OtherIteratorTuple is implicitly convertible to IteratorTuple.

```
const IteratorTuple& get_iterator_tuple() const;
```

Returns: m_iterator_tuple

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

Returns: A tuple consisting of the results of dereferencing all iterators in m_iterator_tuple.

```
zip_iterator& operator++();
```

Effects: Increments each iterator in m_iterator_tuple.
Returns: *this

```
zip_iterator& operator--();
```

Effects: Decrements each iterator in m_iterator_tuple.
Returns: *this

```
template<typename IteratorTuple>
zip_iterator<IteratorTuple>
make_zip_iterator(IteratorTuple t);
```

Returns: An instance of `zip_iterator<IteratorTuple>` with `m_iterator_tuple` initialized to `t`.

Utilities

Iterator Archetypes

The `iterator_archetype` class constructs a minimal implementation of one of the iterator access concepts and one of the iterator traversal concepts. This is used for doing a compile-time check to see if a the type requirements of a template are really enough to cover the implementation of the template. For further information see the documentation for the |concepts|_ library.

Synopsis

```
namespace iterator_archetypes
{
    // Access categories

    typedef /*implementation  defined*/ readable_iterator_t;
    typedef /*implementation  defined*/ writable_iterator_t;
    typedef /*implementation  defined*/ readable_writable_iterator_t;
    typedef /*implementation  defined*/ readable_lvalue_iterator_t;
    typedef /*implementation  defined*/ writable_lvalue_iterator_t;

}

template <
    class Value
  , class AccessCategory
  , class TraversalCategory
>
class iterator_archetype
{
    typedef /* see below */ value_type;
    typedef /* see below */ reference;
    typedef /* see below */ pointer;
    typedef /* see below */ difference_type;
    typedef /* see below */ iterator_category;
};
```

Access Category Tags

The access category types provided correspond to the following standard iterator access concept combinations:

```
readable_iterator_t :=
  Readable Iterator
writable_iterator_t :=
  Writeable Iterator
readable_writable_iterator_t :=
  Readable Iterator & Writeable Iterator & Swappable Iterator
readable_lvalue_iterator_t :=
  Readable Iterator & Lvalue Iterator
writeable_lvalue_iterator_t :=
  Readable Iterator & Writeable Iterator & Swappable Iterator & Lvalue Iterator
```

Traits

The nested trait types are defined as follows:

```
if (AccessCategory == readable_iterator_t)

  value_type = Value
  reference  = Value
  pointer    = Value*

else if (AccessCategory == writable_iterator_t)

  value_type = void
  reference  = void
  pointer    = void

else if (AccessCategory == readable_writable_iterator_t)

  value_type = Value

  reference :=

    A type X that is convertible to Value for which the following
    expression is valid. Given an object x of type X and v of type
    Value.

    x = v

  pointer    = Value*

else if (AccessCategory == readable_lvalue_iterator_t)

  value_type = Value
  reference  = Value const&
  pointer    = Value const*

else if (AccessCategory == writable_lvalue_iterator_t)

  value_type = Value
  reference  = Value&
  pointer    = Value*

if ( TraversalCategory is convertible to forward_traversal_tag )

  difference_type := ptrdiff_t

else

  difference_type := unspecified type


iterator_category :=

  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 (reference is a reference type
              && TraversalCategory is convertible to forward_traversal_tag)
          {
              if (TraversalCategory is convertible to random_access_traversal_tag)
                  X1 = random_access_iterator_tag
              else if (TraversalCategory is convertible to bidirectional_traversal_tag)
                  X1 = bidirectional_iterator_tag
              else
                  X1 = forward_iterator_tag
```

```
        }
        else
        {
            if (TraversalCategory is convertible to single_pass_traversal_tag
                && reference != void)
                X1 = input_iterator_tag
            else
                X1 = output_iterator_tag
        }
```

2. X is convertible to TraversalCategory

Requirements

The `AccessCategory` argument must be one of the predefined access category tags. The `TraversalCategory` must be one of the standard traversal tags. The `Value` type must satisfy the requirements of the iterator concept specified by `AccessCategory` and `TraversalCategory` as implied by the nested traits types.

Concepts

`iterator_archetype` models the iterator concepts specified by the `AccessCategory` and `TraversalCategory` arguments. `iterator_archetype` does not model any other access concepts or any more derived traversal concepts.

Concept Checking

The iterator concept checking classes provide a mechanism for a template to report better error messages when a user instantiates the template with a type that does not meet the requirements of the template. For an introduction to using concept checking classes, see the documentation for the boost::concept_check library.

`iterator_concepts.hpp` Synopsis

```
namespace boost_concepts {

    // Iterator Access Concepts

    template <typename Iterator>
    class ReadableIteratorConcept;

    template <
        typename Iterator
      , typename ValueType = std::iterator_traits<Iterator>::value_type
    >
    class WritableIteratorConcept;

    template <typename Iterator>
    class SwappableIteratorConcept;

    template <typename Iterator>
    class LvalueIteratorConcept;

    // Iterator Traversal Concepts

    template <typename Iterator>
    class IncrementableIteratorConcept;

    template <typename Iterator>
    class SinglePassIteratorConcept;

    template <typename Iterator>
    class ForwardTraversalConcept;

    template <typename Iterator>
    class BidirectionalTraversalConcept;

    template <typename Iterator>
    class RandomAccessTraversalConcept;

    // Interoperability

    template <typename Iterator, typename ConstIterator>
    class InteroperableIteratorConcept;

}
```

Iterator Traits

`std::iterator_traits` provides access to five associated types of any iterator: its `value_type`, `reference`, `pointer`, `iterator_category`, and `difference_type`. Unfortunately, such a "multi-valued" traits template can be difficult to use in a metaprogramming context. `<boost/iterator/iterator_traits.hpp>` provides access to these types using a standard metafunctions_.

Synopsis

Header `<boost/iterator/iterator_traits.hpp>`:

```
template <class Iterator>
struct iterator_value
{
    typedef typename
      std::iterator_traits<Iterator>::value_type
    type;
};

template <class Iterator>
struct iterator_reference
{
    typedef typename
      std::iterator_traits<Iterator>::reference
    type;
};

template <class Iterator>
struct iterator_pointer
{
    typedef typename
      std::iterator_traits<Iterator>::pointer
    type;
};

template <class Iterator>
struct iterator_difference
{
    typedef typename
      detail::iterator_traits<Iterator>::difference_type
    type;
};

template <class Iterator>
struct iterator_category
{
    typedef typename
      detail::iterator_traits<Iterator>::iterator_category
    type;
};
```

Broken Compiler Notes

Because of workarounds in Boost, you may find that these metafunctions actually work better than the facilities provided by your compiler's standard library.

On compilers that don't support partial specialization, such as Microsoft Visual C++ 6.0 or 7.0, you may need to manually invoke BOOST_BROKEN_COMPILER_TYPE_TRAITS_SPECIALIZATION on the `value_type` of pointers that are passed to these metafunctions.

Because of bugs in the implementation of GCC-2.9x, the name of `iterator_category` is changed to `iterator_category_` on that compiler. A macro, `BOOST_ITERATOR_CATEGORY`, that expands to either `iterator_category` or `iterator_category_`, as appropriate to the platform, is provided for portability.

Iterator Utilities

Traits

Overview

Have you ever wanted to write a generic function that can operate on any kind of dereferenceable object? If you have, you've probably run into the problem of how to determine the type that the object "points at":

```
template <class Dereferenceable>
void f(Dereferenceable p)
{
    *what-goes-here?* value = \*p;
    ...
}
```

`pointee`

It turns out to be impossible to come up with a fully-general algorithm to do determine **what-goes-here** directly, but it is possible to require that `pointee<Dereferenceable>::type` is correct. Naturally, `pointee` has the same difficulty: it can't determine the appropriate `::type` reliably for all `Dereferenceable`s, but it makes very good guesses (it works for all pointers, standard and boost smart pointers, and iterators), and when it guesses wrongly, it can be specialized as necessary:

```
namespace boost
{
  template <class T>
  struct pointee<third_party_lib::smart_pointer<T> >
  {
      typedef T type;
  };
}
```

`indirect_reference`

`indirect_reference<T>::type` is rather more specialized than `pointee`, and is meant to be used to forward the result of dereferencing an object of its argument type. Most dereferenceable types just return a reference to their pointee, but some return proxy references or return the pointee by value. When that information is needed, call on `indirect_reference`.

Both of these templates are essential to the correct functioning of `indirect_iterator`.

Reference

`pointeee`

```
template <class Dereferenceable>
struct pointee
{
    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 `Dereferenceable::element_type` shall be an accessible type. Otherwise `iterator_traits<Dereferenceable>::value_type` shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of `pointee`]

`type` is determined according to the following algorithm, where `x` is an object of type `Dereferenceable`:

```
if ( ++x is ill-formed )
{
    return `Dereferenceable::element_type`
}
else if (`*x` is a mutable reference to
         std::iterator_traits<Dereferenceable>::value_type)
{
    return iterator_traits<Dereferenceable>::value_type
}
else
{
    return iterator_traits<Dereferenceable>::value_type const
}
```

`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 `pointee<Dereferenceable>::type&` shall be well-formed. Otherwise `iterator_traits<Dereferenceable>::reference` shall be well formed. [Note: These requirements need not apply to explicit or partial specializations of `indirect_reference`]

`type` is determined according to the following algorithm, where `x` is an object of type `Dereferenceable`:

```
if ( ++x is ill-formed )
    return `pointee<Dereferenceable>::type&`
else
    std::iterator_traits<Dereferenceable>::reference
```

Testing and Concept Checking

The iterator concept checking classes provide a mechanism for a template to report better error messages when a user instantiates the template with a type that does not meet the requirements of the template.

For an introduction to using concept checking classes, see the documentation for the `boost::concept_check` library.

Reference

Iterator Access Concepts

- |Readable|_
- |Writable|_
- |Swappable|_
- |Lvalue|_

Iterator Traversal Concepts .....

- |Incrementable|_

- |SinglePass|_
- |Forward|_
- |Bidir|_
- |Random|_

`iterator_concepts.hpp` Synopsis

```
namespace boost_concepts {

    // Iterator Access Concepts

    template <typename Iterator>
    class ReadableIteratorConcept;

    template <
        typename Iterator
      , typename ValueType = std::iterator_traits<Iterator>::value_type
    >
    class WritableIteratorConcept;

    template <typename Iterator>
    class SwappableIteratorConcept;

    template <typename Iterator>
    class LvalueIteratorConcept;

    // Iterator Traversal Concepts

    template <typename Iterator>
    class IncrementableIteratorConcept;

    template <typename Iterator>
    class SinglePassIteratorConcept;

    template <typename Iterator>
    class ForwardTraversalConcept;

    template <typename Iterator>
    class BidirectionalTraversalConcept;

    template <typename Iterator>
    class RandomAccessTraversalConcept;

    // Interoperability

    template <typename Iterator, typename ConstIterator>
    class InteroperableIteratorConcept;

}
```

Upgrading from the old Boost Iterator Adaptor Library

If you have been using the old Boost Iterator Adaptor library to implement iterators, you probably wrote a `Policies` class which captures the core operations of your iterator. In the new library design, you'll move those same core operations into the body of the iterator class itself. If you were writing a family of iterators, you probably wrote a type generator to build the `iterator_adaptor` specialization you needed; in the new library design you don't need a type generator (though may want to keep it around as a compatibility aid for older code) because, due to the use of the Curiously Recurring Template Pattern (CRTP) [Cop95]_, you can now define the iterator class yourself and acquire functionality through inheritance from `iterator_facade` or `iterator_adaptor`. As a result, you also get much finer control over how your iterator works: you can add additional constructors, or even override the iterator functionality provided by the library.

If you're looking for the old `projection_iterator` component, its functionality has been merged into transform_iterator: as long as the function object's `result_type` (or the `Reference` template argument, if explicitly specified) is a true reference type, transform_iterator will behave like `projection_iterator` used to.

History

In 2000 Dave Abrahams was writing an iterator for a container of pointers, which would access the pointed-to elements when dereferenced. Naturally, being a library writer, he decided to generalize the idea and the Boost Iterator Adaptor library was born. Dave was inspired by some writings of Andrei Alexandrescu and chose a policy based design (though he probably didn't capture Andrei's idea very well - there was only one policy class for all the iterator's orthogonal properties). Soon Jeremy Siek realized he would need the library and they worked together to produce a "Boostified" version, which was reviewed and accepted into the library. They wrote a paper and made several important revisions of the code.

Eventually, several shortcomings of the older library began to make the need for a rewrite apparent. Dave and Jeremy started working at the Santa Cruz C++ committee meeting in 2002, and had quickly generated a working prototype. At the urging of Mat Marcus, they decided to use the GenVoca/CRTP pattern approach, and moved the policies into the iterator class itself. Thomas Witt expressed interest and became the voice of strict compile-time checking for the project, adding uses of the SFINAE technique to eliminate false converting constructors and operators from the overload set. He also recognized the need for a separate `iterator_facade`, and factored it out of `iterator_adaptor`. Finally, after a near-complete rewrite of the prototype, they came up with the library you see today.

[Coplien, 1995] Coplien, J., Curiously Recurring Template Patterns, C++ Report, February 1995, pp. 24-27.