Iterator Facade

Author:	David Abrahams, Jeremy Siek, Thomas Witt
Contact:	dave@boost-consulting.com, jsiek@osl.iu.edu, witt@ive.uni-hannover.de
Organization:	Boost Consulting, Indiana University Open Systems Lab, University of Hanover Institute for Transport Railway Operation and Construction

Date:	2004-01-12

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

Table of Contents

- Overview
  - Usage
  - Iterator Core Access
  - `operator[]`
  - `operator->`
- Reference
  - `iterator_facade` Requirements
  - `iterator_facade` operations
- Tutorial Example
  - The Problem
  - A Basic Iterator Using `iterator_facade`
    - Template Arguments for `iterator_facade`
      - `Derived`
      - `Value`
      - `CategoryOrTraversal`
      - `Reference`
      - `Difference`
    - Constructors and Data Members
    - Implementing the Core Operations
  - A constant `node_iterator`
  - Interoperability
  - Telling the Truth
  - Wrap Up

Overview

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.

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_iterator`), because `*(p+n)` is a reference into the temporary iterator `p+n`, which is destroyed when `operator[]` returns.

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.

`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.

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;
};

// 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);
```

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

```
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&);
```

The `iterator_category` member of `iterator_facade` is

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

where *iterator-category* is defined as follows:

```
iterator-category(C,R,V) :=
   if (C is convertible to std::input_iterator_tag
       || C is convertible to std::output_iterator_tag
   )
       return C

   else if (C is not convertible to incrementable_traversal_tag)
```

```
        the program is ill-formed

    else return a type X satisfying the following two constraints:

       1. X is convertible to X1, and not to any more-derived
          type, where X1 is defined by:

            if (R is a reference type
                && C is convertible to forward_traversal_tag)
            {
                if (C is convertible to random_access_traversal_tag)
                    X1 = random_access_iterator_tag
                else if (C is convertible to bidirectional_traversal_tag)
                    X1 = bidirectional_iterator_tag
                else
                    X1 = forward_iterator_tag
            }
            else
            {
                if (C is convertible to single_pass_traversal_tag
                    && R is convertible to V)
                    X1 = input_iterator_tag
                else
                    X1 = C
            }

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

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

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

```
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
    >
{};
```

iterator_facade Requirements

The following table describes the typical valid expressions on iterator_facade's Derived parameter, depending on the iterator concept(s) it will model. The operations in the first column must be made accessible to member functions of class iterator_core_access. In addition, static_cast<Derived*>(iterator_facade* shall be well-formed.

In the table below, F is 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.

iterator_facade 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

iterator_facade 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)

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.

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`.

```
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 Example

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 [1].

[1] `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_iterator`s 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 [2]. 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;
};
```

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.

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;
};
```

[2] 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.

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

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 [3] [4]:

```
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;
```

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

[3] If you're using an older compiler and it can't handle this example, see the example code for workarounds.
[4] If `node_iterator` had been a random access traversal iterator, we'd have had to templatize its `distance_to` function as well.

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.

In fact, that sort of magic is possible using `boost::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) {}
```

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.