Iterator Adaptor
abstract:

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

Table of Contents

Overview

Reference

iterator_adaptor requirements
iterator_adaptor base class parameters
iterator_adaptor public operations
iterator_adaptor protected member functions
iterator_adaptor private member functions

Tutorial Example

Overview

The `iterator_adaptor` class template adapts some Base [1] 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 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 table above. 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,

[1] 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 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**

```cpp
template <
    class Derived,
    , class Base
    , class Value = use_default
    , class CategoryOrTraversal = use_default
    , class Reference = use_default
    , class Difference = use_default
>
class iterator_adaptor
: public iterator_facade<Derived, V', C', R', D'> // see details
{
    friend class iterator_core_access;

public:
    iterator_adaptor();
    explicit iterator_adaptor(Base iter);
    Base const& base() const;

protected:
    Base const& base_reference() const;
    Base& base_reference();

private: // Core iterator interface for iterator_facade.
    typename iterator_adaptor::reference dereference() const;

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

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

    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
};
```
**iterator_adaptor requirements**

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

**iterator_adaptor base class parameters**

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

\[
V' = \begin{cases} 
    \text{if (Value is `use_default`) return } & \text{iterator_traits<Base>::value_type} \\
    \text{else return Value} & 
\end{cases}
\]

\[
C' = \begin{cases} 
    \text{if (CategoryOrTraversal is `use_default`) return } & \text{iterator_traversal<Base>::type} \\
    \text{else return CategoryOrTraversal} & 
\end{cases}
\]

\[
R' = \begin{cases} 
    \text{if (Reference is `use_default`) return } & \text{iterator_traits<Base>::reference} \\
    \text{if (Value is `use_default`) return } & \text{Value&} \\
    \text{else return Reference} & 
\end{cases}
\]

\[
D' = \begin{cases} 
    \text{if (Difference is `use_default`) return } & \text{iterator_traits<Base>::difference_type} \\
    \text{else return Difference} & 
\end{cases}
\]

**iterator_adaptor public operations**

`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 iter);`

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

`Base const& base() const;`

**Returns:** `m_iterator`
**iterator_adaptor protected member functions**

Base const& base_reference() const;

**Returns:** A const reference to m_iterator.

Base& base_reference();

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

**iterator_adaptor private member functions**

typename iterator_adaptor::reference dereference() const;

**Returns:** *m_iterator

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

**Returns:** m_iterator == x.base()

void advance(typename iterator_adaptor::difference_type n);

**Effects:** m_iterator += n;

void increment();

**Effects:** ++m_iterator;

void decrement();

**Effects:** --m_iterator;

template <
class OtherDerived, class OtherIterator, class V, class C, class R, class D
>

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

**Returns:** y.base() - m_iterator
Tutorial Example

In this section we’ll further refine the `node_iter` class template we developed in the `iterator_facade` tutorial. 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.

**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 which 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` member functions). However, the `operator++` for `node_iterator` behaves differently than for `node_base*` since it follows the m_next pointer.

It turns out that the pattern of building an iterator on another iterator-like type (the Base [1] 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:

```cpp
template <class Value>
class node_iter
{
public:
    node_iter() : super_t(0) {}

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

private:
    struct enabler {};

    typedef boost::iterator_adaptor<
        node_iter<Value>, Value*, boost::use_default,
        boost::forward_traversal_tag
    > super_t;

};
```

You may be wondering what the purpose of the `node_iter` is. It is intended to illustrate how to use the `iterator_adaptor` to build an iterator on another iterator-like type while modifying just a few aspects of its behavior. This pattern is extremely common and is well-addressed by `iterator_adaptor`. If you’re interested in learning more about iterators and iterators, I would recommend the book *Effective C++* by Scott Meyers.
```cpp
template <class OtherValue>
node_iter(
    node_iter<OtherValue> const& other,
    typename boost::enable_if<
        boost::is_convertible<OtherValue*,Value*> , enabler
    >::type = enabler() )
: super_t(other.base()) {}

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

You can see an example program which 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

```cpp
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.