Reverse Iterator

**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:** 2004-01-13

**Copyright:** Copyright David Abrahams, Jeremy Siek, and Thomas Witt 2003. All rights reserved

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

## Table of Contents

- reverse_iterator synopsis
- reverse_iterator requirements
- reverse_iterator models
- reverse_iterator operations

### reverse_iterator 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`.

**reverse_iterator requirements**

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

**reverse_iterator models**

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

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

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

**reverse_iterator operations**

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

```cpp
reverse_iterator();
```

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

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

```cpp
explicit reverse_iterator(Iterator x);
```

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

```cpp
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()`.

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

*Returns:* `m_iterator`

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

*Effects:*

```cpp
Iterator tmp = m_iterator;
return *--tmp;
```

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

```cpp
```
Effects: \texttt{--m\_iterator}

Returns: \texttt{*this}

\begin{verbatim}
reverse\_iterator& \texttt{operator--();}
\end{verbatim}

Effects: \texttt{++m\_iterator}

Returns: \texttt{*this}

\begin{verbatim}
template \texttt{<class BidirectionalIterator>}
reverse\_iterator\texttt{<BidirectionalIterator>n}
make\_reverse\_iterator\texttt{(BidirectionalIterator x);} \texttt{\n}
\end{verbatim}

Returns: An instance of \texttt{reverse\_iterator\texttt{<BidirectionalIterator>}} with a \texttt{current} constructed from \texttt{x}.

\section*{Example}

The following example prints an array of characters in reverse order using \texttt{reverse\_iterator}.

\begin{verbatim}
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\t" << letters << std::endl;

boost::reverse\_iterator\texttt{<base\_iterator>}
    reverse\_letters\_first(letters + N),
    reverse\_letters\_last(letters);

std::cout << "sequence in reverse order:\t\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:\texttt{::make\texttt{\_reverse\_iterator\texttt{(reverse\_letters\_last),
    boost:\texttt{::make\texttt{\_reverse\_iterator\texttt{(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.