Filter Iterator

abstract: 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.

Table of Contents

filter_iterator synopsis
filter_iterator requirements
filter_iterator models
filter_iterator operations
Example

filter_iterator 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 /* see below */ iterator_category;

definition

filter_iterator();
filter_iterator(Predicate p, 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.

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

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

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

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

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

<table>
<thead>
<tr>
<th>If Iterator models</th>
<th>then filter_iterator models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readable Iterator, Single Pass Iterator</td>
<td>Input Iterator</td>
</tr>
<tr>
<td>Readable Lvalue Iterator, Forward Traversal Iterator</td>
<td>Forward Iterator</td>
</tr>
<tr>
<td>Writable Lvalue Iterator, Forward Traversal Iterator</td>
<td>Mutable Forward Iterator</td>
</tr>
<tr>
<td>Writable Lvalue Iterator, Bidirectional Iterator</td>
<td>Mutable Bidirectional Iterator</td>
</tr>
</tbody>
</table>
filter_iterator<P1, X> is interoperable with filter_iterator<P2, Y> if and only if X is interoperable with Y.

**filter_iterator operations**

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

`filter_iterator();`

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

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

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

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

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

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

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

`template <class OtherIterator>
filter_iterator(
    filter_iterator<Predicate, OtherIterator> const& t,
    typename enable_if_convertible<OtherIterator, Iterator>::type* = 0 // exposition
);``'

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

**Effects:** Constructs a filter iterator whose members are copied from t.

Predicate predicate() const;

Returns: m_pred

Iterator end() const;

Returns: m_end

Iterator const& base() const;

Returns: m_iterator

reference operator*() const;

Returns: *m_iter

filter_iterator& operator++();

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

template <class Predicate, class Iterator>
filter_iterator<Predicate,Iterator>
make_filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());

Returns: filter_iterator<Predicate,Iterator>(f, x, end)

template <class Predicate, class Iterator>
filter_iterator<Predicate,Iterator>
make_filter_iterator(Iterator x, Iterator end = Iterator());

Returns: filter_iterator<Predicate,Iterator>(x, end)

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.

```cpp
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.