Counting 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: 2004-11-01

abstract: How would you fill up a vector with the numbers zero through one hundred using `std::copy()`? The only iterator operation missing from builtin integer types is an `operator*()` that returns the current value of the integer. The counting iterator adaptor adds this crucial piece of functionality to whatever type it wraps. One can use the counting iterator adaptor not only with integer types, but with any incrementable type.

`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.

Table of Contents

`counting_iterator` 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.]

counting_iterator 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;
```

counting_iterator models

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.

counting_iterator 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

```
template <class Incrementable>
counting_iterator<Incrementable> make_counting_iterator(Incrementable x);
```

Returns: An instance of counting_iterator<Incrementable> with current constructed from x.

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.