Permutation Iterator

Author: Toon Knapen, David Abrahams, Roland Richter, Jeremy Siek
Contact: dave@boost-consulting.com, jsiek@osl.iu.edu
Organization: Boost Consulting, Indiana University Open Systems Lab
Date: 2004-11-01
Copyright: Copyright Toon Knapen, David Abrahams, Roland Richter, and Jeremy Siek 2003.

abstract: The permutation iterator adaptor provides a permuted view of a given range. That is, the view includes every element of the given range but in a potentially different order.

Table of Contents
Introduction
Reference
    permutation_iterator requirements
    permutation_iterator models
    permutation_iterator operations
Example

Introduction
The adaptor takes two arguments:

- an iterator to the range V on which the permutation will be applied
- the reindexing scheme that defines how the elements of V will be permuted.

Note that the permutation iterator is not limited to strict permutations of the given range V. The distance between begin and end of the reindexing iterators is allowed to be smaller compared to the size of the range V, in which case the permutation iterator only provides a permutation of a subrange of V. The indexes neither need to be unique. In this same context, it must be noted that the past the end permutation iterator is completely defined by means of the past-the-end iterator to the indices.

Reference

\[
\text{template< class ElementIterator, class IndexIterator, class ValueT = use_default, class CategoryT = use_default, class ReferenceT = use_default } \]

1
class permutation_iterator
{
public:
    permutation_iterator();
    explicit permutation_iterator(ElementIterator x, IndexIterator y);

    template< class OEIter, class OIIter, class V, class C, class R, class D >
    permutation_iterator(
        permutation_iterator<OEIter, OIIter, V, C, R, D> const& r,
        typename enable_if_convertible<OEIter, ElementIterator>::type* = 0,
        typename enable_if_convertible<OIIter, IndexIterator>::type* = 0
    );

    reference operator*() const;
    permutation_iterator& operator++();
    ElementIterator const& base() const;
private:
    ElementIterator m_elt; // exposition only
    IndexIterator m_order; // exposition only
};

template <class ElementIterator, class IndexIterator>
permutation_iterator<ElementIterator, IndexIterator>
make_permutation_iterator( ElementIterator e, IndexIterator i);

permutation_iterator requirements

ElementIterator shall model Random Access Traversal Iterator. IndexIterator shall model Readable Iterator. The value type of the IndexIterator must be convertible to the difference type of ElementIterator.

permutation_iterator models

permutation_iterator models the same iterator traversal concepts as IndexIterator and the same iterator access concepts as ElementIterator.

  If IndexIterator models Single Pass Iterator and ElementIterator models Readable Iterator then permutation_iterator models Input Iterator.
  If IndexIterator models Forward Traversal Iterator and ElementIterator models Readable Lvalue Iterator then permutation_iterator models Forward Iterator.
  If IndexIterator models Bidirectional Traversal Iterator and ElementIterator models Readable Lvalue Iterator then permutation_iterator models Bidirectional Iterator.
  If IndexIterator models Random Access Traversal Iterator and ElementIterator models Readable Lvalue Iterator then permutation_iterator models Random Access Iterator.

permutation_iterator interoperability

permutation_iterator<E1, X, V1, C2, R1, D1> is interoperable with permutation_iterator<E2, Y, V2, C2, R2, D2> if and only if X is interoperable with Y and E1 is convertible to E2.

permutation_iterator operations

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

    permutation_iterator();
Effects: Default constructs m_elt and m_order.

explicit permutation_iterator(ElementIterator x, IndexIterator y);
Effects: Constructs m_elt from x and m_order from y.

template< class OEIter, class OIIter, class V, class C, class R, class D >
permutation_iterator<OEIter, OIIter, V, C, R, D> const& r
 , typename enable_if_convertible<OEIter, ElementIterator>::type* = 0
 , typename enable_if_convertible<OIIter, IndexIterator>::type* = 0
 );
Effects: Constructs m_elt from r.m_elt and m_order from y.m_order.

reference operator*() const;
Returns: *(m_elt + *m_order)

permutation_iterator& operator++();
Effects: ++m_order
Returns: *this

ElementIterator const& base() const;
Returns: m_order

template <class ElementIterator, class IndexIterator>
permutation_iterator<ElementIterator, IndexIterator>
make_permutation_iterator(ElementIterator e, IndexIterator i);
Returns: permutation_iterator<ElementIterator, IndexIterator>(e, i)

Example

using namespace boost;
int i = 0;

typedef std::vector< int > element_range_type;
typedef std::list< int > index_type;

static const int element_range_size = 10;
static const int index_size = 4;

element_range_type elements( element_range_size );
for(element_range_type::iterator el_it = elements.begin(); el_it != ele-
ments.end(); ++el_it)  
*el_it = std::distance(elements.begin(), el_it);

index_type indices( index_size );
for(index_type::iterator i_it = indices.begin(); i_it != in-
dices.end(); ++i_it )  
*i_it = element_range_size -
index_size + std::distance(indices.begin(), i_it);
std::reverse( indices.begin(), indices.end() );

typedef permutation_iterator< element_range_type::iterator, index_type::iterator > permutation_type;
permutation_type begin = make_permutation_iterator( elements.begin(), indices.begin() );
permutation_type it = begin;
permutation_type end = make_permutation_iterator( elements.begin(), indices.end() );

std::cout << "The original range is : ";
std::copy( elements.begin(), elements.end(), std::ostream_iterator< int >( std::cout, " ") );
std::cout << "\n";

std::cout << "The reindexing scheme is : ";
std::copy( indices.begin(), indices.end(), std::ostream_iterator< int >( std::cout, " ") );
std::cout << "\n";

std::cout << "The permutated range is : ";
std::copy( begin, end, std::ostream_iterator< int >( std::cout, " ") );
std::cout << "\n";

std::cout << "Elements at even indices in the permutation : ";
it = begin;
for(i = 0; i < index_size / 2 ; ++i, it+=2 ) std::cout << *it << " ";
std::cout << "\n";

std::cout << "Permutation backwards : ";
it = begin + (index_size);
assert( it != begin );
for( ; it-- != begin ; ) std::cout << *it << " ";
std::cout << "\n";

std::cout << "Iterate backward with stride 2 : ";
it = begin + (index_size - 1);
for(i = 0 ; i < index_size / 2 ; ++i, it-=2 ) std::cout << *it << " ";
std::cout << "\n";

The output is:

The original range is : 0 1 2 3 4 5 6 7 8 9
The reindexing scheme is : 9 8 7 6
The permutated range is : 9 8 7 6
Elements at even indices in the permutation : 9 7
Permutation backwards : 6 7 8 9
Iterate backward with stride 2 : 6 8

The source code for this example can be found here.