Transform 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
Copyright: Copyright David Abrahams, Jeremy Siek, and Thomas Witt 2003.

abstract: The transform iterator adapts an iterator by modifying the operator* to apply a function object to the result of dereferencing the iterator and returning the result.

Table of Contents
transform_iterator synopsis
transform_iterator requirements
transform_iterator models
transform_iterator operations
Example

transform_iterator synopsis

template <class UnaryFunction, 
    class Iterator, 
    class Reference = use_default, 
    class Value = use_default>
class transform_iterator
{
 public:
typedef /* see below */ value_type;
typedef /* see below */ reference;
typedef /* see below */ pointer;
typedef iterator_traits<Iterator>::difference_type difference_type;
typedef /* see below */ iterator_category;

transform_iterator();
transform_iterator(Iterator const & x, UnaryFunction f);

template<class F2, class I2, class R2, class V2>
transform_iterator(
    transform_iterator<F2, I2, R2, V2> const & t

1
transform_iterator requirements

The type UnaryFunction must be Assignable, Copy Constructible, and the expression f(*i) must be valid where f is an object of type UnaryFunction, i is an object of type Iterator, and where the type of f(*i) must be result_of<UnaryFunction(iterator_traits<Iterator>::reference)::type.

The argument Iterator shall model Readable Iterator.

transform_iterator models

The resulting transform_iterator models the most refined of the following that is also modeled by Iterator.

- Writable Lvalue Iterator if transform_iterator::reference is a non-const reference.
- Readable Lvalue Iterator if transform_iterator::reference is a const reference.
- Readable Iterator otherwise.

The transform_iterator models the most refined standard traversal concept that is modeled by the Iterator argument.

If transform_iterator is a model of Readable Lvalue Iterator then it models the following original iterator concepts depending on what the Iterator argument models.

<table>
<thead>
<tr>
<th>If Iterator models</th>
<th>then transform_iterator models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single Pass Iterator</td>
<td>Input Iterator</td>
</tr>
<tr>
<td>Forward Traversal Iterator</td>
<td>Forward Iterator</td>
</tr>
<tr>
<td>Bidirectional Traversal Iterator</td>
<td>Bidirectional Iterator</td>
</tr>
<tr>
<td>Random Access Traversal Iterator</td>
<td>Random Access Iterator</td>
</tr>
</tbody>
</table>
If `transform_iterator` models Writable Lvalue Iterator then it is a mutable iterator (as defined in the old iterator requirements).

`transform_iterator<F1, X, R1, V1>` is interoperable with `transform_iterator<F2, Y, R2, V2>` if and only if `X` is interoperable with `Y`.

### `transform_iterator` operations

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

```cpp
transform_iterator();
Returns: An instance of `transform_iterator` with `m_f` and `m_iterator` default constructed.
```

```cpp
transform_iterator(Iterator const& x, UnaryFunction f);
Returns: An instance of `transform_iterator` with `m_f` initialized to `f` and `m_iterator` initialized to `x`.
```

```cpp
template<class F2, class I2, class R2, class V2>
transform_iterator(transform_iterator<F2, I2, R2, V2> const& t,
                    typename enable_if_convertible<I2, Iterator>::type* = 0 // exposition only,
                    typename enable_if_convertible<F2, UnaryFunction>::type* = 0 // exposition only);
Returns: An instance of `transform_iterator` with `m_f` initialized to `t.functor()` and `m_iterator` initialized to `t.base()`.
Requires: OtherIterator is implicitly convertible to Iterator.
```

```cpp
UnaryFunction functor() const;
Returns: `m_f`
```

```cpp
Iterator const& base() const;
Returns: `m_iterator`
```

```cpp
reference operator*() const;
Returns: `m_f(*m_iterator)`
```

```cpp
transform_iterator& operator++();
Effects: `++m_iterator`
Returns: *this
```

```cpp
transform_iterator& operator--();
Effects: `--m_iterator`
Returns: *this
```

```cpp
template <class UnaryFunction, class Iterator>
transform_iterator<UnaryFunction, Iterator>
make_transform_iterator(Iterator it, UnaryFunction fun);
```
Returns: An instance of `transform_iterator<UnaryFunction, Iterator>` with `m_f` initialized to `f` and `m_iterator` initialized to `x`.

```cpp
template <class UnaryFunction, class Iterator>
transform_iterator<UnaryFunction, Iterator>
make_transform_iterator(Iterator it);
```

Returns: An instance of `transform_iterator<UnaryFunction, Iterator>` with `m_f` default constructed and `m_iterator` initialized to `x`.

Example

This is a simple example of using the `transform_iterator` class to generate iterators that multiply (or add to) the value returned by dereferencing the iterator. It would be cooler to use lambda library in this example.

```cpp
int x[] = { 1, 2, 3, 4, 5, 6, 7, 8 };
const int N = sizeof(x)/sizeof(int);

typedef boost::binder1st< std::multiplies<int> > Function;
typedef boost::transform_iterator<Function, int*> doubling_iterator;

doubling_iterator i(x, boost::bind1st(std::multiplies<int>(), 2)),
i_end(x + N, boost::bind1st(std::multiplies<int>(), 2));

std::cout << "multiplying the array by 2:" << std::endl;
while (i != i_end)
    std::cout << *i++ << " ";
std::cout << std::endl;

std::cout << "adding 4 to each element in the array:" << std::endl;
std::copy(boost::make_transform_iterator(x, boost::bind1st(std::plus<int>(), 4)),
          boost::make_transform_iterator(x + N, boost::bind1st(std::plus<int>(), 4)),
          std::ostream_iterator<int>(std::cout, " "));
std::cout << std::endl;
```

The output is:

```
multiplying the array by 2:
 2 4 6 8 10 12 14 16

adding 4 to each element in the array:
 5 6 7 8 9 10 11 12
```

The source code for this example can be found [here](#).