Zip Iterator

Author: David Abrahams, Thomas Becker
Contact: dave@boost-consulting.com, thomas@styleadvisor.com
Date: 2004-11-01
Copyright: Copyright David Abrahams and Thomas Becker 2003.

abstract: The zip iterator provides the ability to parallel-iterate over several controlled sequences simultaneously. A zip iterator is constructed from a tuple of iterators. Moving the zip iterator moves all the iterators in parallel. Dereferencing the zip iterator returns a tuple that contains the results of dereferencing the individual iterators.

Table of Contents

zip_iterator synopsis
zip_iterator requirements
zip_iterator models
zip_iterator operations
Examples

zip_iterator synopsis

template<typename IteratorTuple>
class zip_iterator
{

public:
   typedef /* see below */ reference;
   typedef reference value_type;
   typedef value_type* pointer;
   typedef /* see below */ difference_type;
   typedef /* see below */ iterator_category;

   zip_iterator();
   zip_iterator(IteratorTuple iterator_tuple);

   template<typename OtherIteratorTuple>
   zip_iterator(const zip_iterator<OtherIteratorTuple>& other,
                typename enable_if_convertible<
                   OtherIteratorTuple

1
zip_iterator<IteratorTuple>

make_zip_iterator(IteratorTuple t);

The reference member of zip_iterator is the type of the tuple made of the reference types of the iterator types in the IteratorTuple argument.

The difference_type member of zip_iterator is the difference_type of the first of the iterator types in the IteratorTuple argument.

The iterator_category member of zip_iterator is convertible to the minimum of the traversal categories of the iterator types in the IteratorTuple argument. For example, if the zip_iterator holds only vector iterators, then iterator_category is convertible to boost::random_access_traversal_tag. If you add a list iterator, then iterator_category will be convertible to boost::bidirectional_traversal_tag, but no longer to boost::random_access_traversal_tag.

zip_iterator requirements

All iterator types in the argument IteratorTuple shall model Readable Iterator.

zip_iterator models

The resulting zip_iterator models Readable Iterator.

The fact that the zip_iterator models only Readable Iterator does not prevent you from modifying the values that the individual iterators point to. The tuple returned by the zip_iterator's operator* is a tuple constructed from the reference types of the individual iterators, not their value types. For example, if zip_it is a zip_iterator whose first member iterator is an std::vector<double>::iterator, then the following line will modify the value which the first member iterator of zip_it currently points to:

zip_it->get<0>() = 42.0;

Consider the set of standard traversal concepts obtained by taking the most refined standard traversal concept modeled by each individual iterator type in the IteratorTuple argument. The zip_iterator models the least refined standard traversal concept in this set.

zip_iterator<IteratorTuple1> is interoperable with zip_iterator<IteratorTuple2> if and only if IteratorTuple1 is interoperable with IteratorTuple2.

zip_iterator operations

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

zip_iterator();

Returns: An instance of zip_iterator with m_iterator_tuple default constructed.
zip_iterator(IteratorTuple iterator_tuple);

**Returns:** An instance of zip_iterator with m_iterator_tuple initialized to iterator_tuple.

template<typename OtherIteratorTuple>
zip_iterator(
    const zip_iterator<OtherIteratorTuple>& other
    , typename enable_if_convertible<
        OtherIteratorTuple
        , IteratorTuple>::type* = 0 // exposition only
);

**Returns:** An instance of zip_iterator that is a copy of other.
**Requires:** OtherIteratorTuple is implicitly convertible to IteratorTuple.

cnst IteratorTuple& get_iterator_tuple() const;

**Returns:** m_iterator_tuple

reference operator*() const;

**Returns:** A tuple consisting of the results of dereferencing all iterators in m_iterator_tuple.

zip_iterator& operator++();

**Effects:** Increments each iterator in m_iterator_tuple.
**Returns:** *this

zip_iterator& operator--();

**Effects:** Decrements each iterator in m_iterator_tuple.
**Returns:** *this

template<typename IteratorTuple>
zip_iterator<IteratorTuple>
make_zip_iterator(IteratorTuple t);

**Returns:** An instance of zip_iterator<IteratorTuple> with m_iterator_tuple initialized to t.

template<typename IteratorTuple>
zip_iterator<IteratorTuple>
make_zip_iterator(IteratorTuple t);

**Returns:** An instance of zip_iterator<IteratorTuple> with m_iterator_tuple initialized to t.

**Examples**

There are two main types of applications of the zip_iterator. The first one concerns runtime efficiency: If one has several controlled sequences of the same length that must be somehow processed, e.g., with the for_each algorithm, then it is more efficient to perform just one parallel-iteration rather than several individual iterations. For an example, assume that vect_of_doubles and vect_of_ints are two vectors of equal length containing doubles and ints, respectively, and consider the following two iterations:
These two iterations can now be replaced with a single one as follows:

```cpp
std::for_each(
    boost::make_zip_iterator(
        boost::make_tuple(beg1, beg2)
    ),
    boost::make_zip_iterator(
        boost::make_tuple(end1, end2)
    ),
    zip_func()
);
```

A non-generic implementation of `zip_func` could look as follows:

```cpp
struct zip_func :
    public std::unary_function<const boost::tuple<const double&, const int&>&, void>
{
    void operator()(const boost::tuple<const double&, const int&>& t) const
    {
        m_f0(t.get<0>());
        m_f1(t.get<1>());
    }

    private:
        func_0 m_f0;
        func_1 m_f1;
};
```

The second important application of the `zip_iterator` is as a building block to make combining iterators. A combining iterator is an iterator that parallel-iterates over several controlled sequences and, upon dereferencing, returns the result of applying a functor to the values of the sequences at the respective positions. This can now be achieved by using the `zip_iterator` in conjunction with the `transform_iterator`.

Suppose, for example, that you have two vectors of doubles, say `vect_1` and `vect_2`, and you need to expose to a client a controlled sequence containing the products of the elements of `vect_1` and `vect_2`. Rather than placing these products in a third vector, you can use a combining iterator that calculates the products on the fly. Let us assume that `tuple_multiplies` is a functor that works like `std::multiplies`, except that it takes its two arguments packaged in a tuple. Then the two iterators `it_begin` and `it_end` defined below delimit a controlled sequence containing the products of the elements of `vect_1` and `vect_2`:

```cpp
typedef boost::tuple<
    std::vector<double>::const_iterator,
    std::vector<double>::const_iterator
> the_iterator_tuple;
```
typedef boost::zip_iterator<
    the_iterator_tuple
  > the_zip_iterator;

typedef boost::transform_iterator<
    tuple_multiplies<double>,
    the_zip_iterator
  > the_transform_iterator;

the_transform_iterator it_begin(
    the_zip_iterator(
      the_iterator_tuple(
        vect_1.begin(),
        vect_2.begin()
      ),
      tuple_multiplies<double>()
    ),
    tuple_multiplies<double>()
);

the_transform_iterator it_end(
    the_zip_iterator(
      the_iterator_tuple(
        vect_1.end(),
        vect_2.end()
      ),
      tuple_multiplies<double>()
    );