X3J16/96-0091 WG21/N0909

Distance Type for Output Iterators

Angelika Langer (email: langer@roguewave.com)

Abstract

The distance type of an output iterator usually is void, i.e. an output iterator does not have a distance type. This proposal suggest to give the output iterator's distance type a useful meaning. It makes the STL more consistent and easier to understand and extend.

1. Motivation

It was discussed to make input iterators and output iterators more consistent; for both iterator categories the notion of the distance between two iterator positions does not really make sense. Still, the distance type can be seen as a type that allows to represent counts on the number of increments which can be done on an iterator. In this sense the distance type of an input iterator is used in the count algorithm.

Input iterators have a distance type, which usually defaults to `ptrdiff_t`. Output iterators are not supposed to have an iterator type; an output iterator's distance type therefore is usually `void`. However, it is hard to explain why one can express the count of increments of an input iterator by means of the input iterator's distance type, but the same is not true for an output iterator.

In order to show the inconsistency, consider the algorithms search_n, fill_n, and generate_n. They all have a size type as a template parameter. From a logical point of view the best choice for the size type is the respective iterator's distance type, as is it for the count algorithm. However, fill_n and generate_n work with output iterators, which do not have a distance type.

Additionally, the lack of a distance type for output iterator makes it difficult to implement algorithms that count the elements that they insert into an output sequence. Such an algorithm would typically take an output iterator as an argument and return a count. The appropriate type of such a count would be the iterator's distance type. However, without a distance type for output iterators, you can't do it. You would have the same problem we originally had with the ordinary

count algorithm: the return type cannot be deduced from the function arguments.[1] I don't see a compelling reason why counting output should be more difficult than counting input.

Here is an example of such a algorithm : It puts values, which are generated by a generator, into an output sequence, until the generator decides that the job is done. The algorithm returns the number of values inserted to the output sequence.

```
template <class OutputIterator, class Generator>
iterator_traits<OutputIterator>::distance_type
output_count(OutputIterator iter, Generator gen)
{ for (n=0; !gen.stop(); n++, *iter++=gen());
  return n;
}
```

The interface of a generator to be used with my this algorithm is:

```
class Generator {
public:
  Generator(...);                       // any kind of constructor
  some_type operator()(void);           // the function call operator
  bool stop() const;                    // the end of generation indicator
};
```

Below is an example of an according generator. With each call it "generates" the next element from a container that has certain properties determined by a predicate. The generator decides how many values it generates.

```
template <class Container, class Predicate>
class CertainContainerElements {
public:
  CertainContainerElements(const Container& c, Predicate p)
  : _cont(c),_iter(c.begin()),_pred(p) {}
  typename Container::value_type operator()(void)
  {  while (!stop())
     { if (_pred(*_iter))   return *_iter++;
       else                 _iter++;
     }
  }
bool stop() const
  { return _iter == _cont.end(); }
private:
  const Container& _cont;
  typename Container::const_iterator _iter;
  Predicate _pred;
};
```

[1] See X3J16/96-0029 WG21/N0847 "Bring back the obvious definition of count()" for reference.

Below is an example of how the `output_count()` algorithm can be used. It takes all negative elements from a list of integers, inserts them into an output sequence, and counts them.

```
  list<int> l;
  ostream_iterator<int> ost(cout,"\t");
  // populate the list
  ostream_iterator<int>::distance_type n =
  output_count(ost,CertainContainerElements<list<int>,Negative>(l,Negative()));
```

Without a meaningful distance type for output iterators the return type of my output count algorithm would be `void` in the example above, because the distance type of an ostream iterator at present is `void`. I would have to add another template parameter for the return type. As return types cannot be deduced from the function arguments, the interface would look like this:

```
template <class OutputIterator, class Generator, class Count>
void
output_count(OutputIterator iter, Generator gen, Count& n)
{ for (n=0; !gen.stop(); n++, *iter++=gen());
  return n;
}
```

which is counter-intuitive. The count should be the return type rather than a reference parameter. This is exactly the problem that caused the introduction of `iterator_traits`, so that the standard `count()` didn't have to use the latter style.

2. Proposal

I suggest to modify the predefined output iterators so that they have a meaningful distance type. The iterators concerned are the insert iterator adaptors and the ostream and ostreambuf iterator.

The insert iterator adaptors shall take their distance type from the container they work on, e.g.

```
template <class Container>
    class back_insert_iterator : public
      iterator<output_iterator_tag,void,typename Container::difference_type>;
```

The ostream and ostreambuf iterators shall take their distance as a template parameter that defaults to ptrdiff_t, e.g.

```
template <class T, class Distance = ptrdiff_t>
    class ostream_iterator : public
      iterator<output_iterator_tag,void,Distance>;
```

3. Alternative

In Santa Cruz it was discussed to make input and output iterators more consistent by eliminating the distance type for input iterators as well, because the notion of a distance between two iterators does not make sense.
I think the distance type is used for two things:

- On the one hand it expresses the distance between the first and last iterator of a range.
- On the other hand it is used to express the count of increments to an iterator, like in `count()`.

With input and output iterators only the second usage makes sense. One might therefore consider to have two separate types: a distance type and a count type. I do not propose this, because it would introduce yet another type. However, I do suggest to (ab)use the distance type for both purposes. Following this line of logic, input as well as output iterators should have a meaningful distance type, i.e. a type different from `void`.

4. Caveat

The proposed changes depend on the availability of partial specialization. This is because I made use of the iterator traits, which require partial specialization. As I had no access to a compiler that supports this language feature, I had no chance to test the proposed changes.

5. Working Paper Changes

In clause 24.1.6 [lib.iterator.tags] change the description of the insert iterator adaptors and output stream and output stream buffer iterators in the header <iterator> synopsis to:

```
template <class Container>
  class back_insert_iterator : public
  iterator<output_iterator_tag,void,typename Container::difference_type>;

template <class Container>
  class front_insert_iterator : public
  iterator<output_iterator_tag,void,typename Container::difference_type>;

template <class Container>
  class insert_iterator : public
  iterator<output_iterator_tag,void,typename Container::difference_type>;

template <class T, class Distance = ptrdiff_t>
  class ostream_iterator : public
  iterator<output_iterator_tag,void,Distance>;

template <class charT, class traits = char_traits<charT>,
          class Distance = ptrdiff_t >
  class ostreambuf_iterator : public
  iterator<output_iterator_tag,void,Distance>;
```

Make according changes in clause 24.3.2. [lib.insert.iterators], i.e. change the base classes of class `back_insert_iterator`, `front_insert_iterator` and `insert_iterator` from `iterator <output_iterator_tag, void, void>` to `iterator <output_iterator_tag, void, typename Container::difference_type>`.

In clause 24.4.2 [lib.ostream.iterator] and clause 24.4.4 [lib.ostreambuf.iterator] add a second template parameter `Distance` to the class templates `ostream_iterator` and `ostreambuf_iterator`, and change the base classes from `iterator <output_iterator_tag, void, void>` to `iterator <output_iterator_tag, void, Distance>`.