
9.6. Writing User-Defined Iterators

Let's write an iterator. As mentioned in the previous section, you need iterator traits provided for the user-defined iterator. You can provide them in one of two ways:

1. Provide the necessary five type definitions for the general `iterator_traits` structure (see Section 9.5, page 467).
2. Provide a (partial) specialization of the `iterator_traits` structure.

For the first way, the C++ standard library provides a special base class, `iterator<>`, that does the type definitions. You need only pass the types:

Click here to view code image

```
class MyIterator
 : public std::iterator <std::bidirectional_iterator_tag,
                         type, std::ptrdiff_t, type*, type&> {
    ...
};
```

The first template parameter defines the iterator category, the second defines the element type *type*, the third defines the difference type, the fourth defines the pointer type, and the fifth defines the reference type. The last three arguments are optional and have the default values `ptrdiff_t`, *type* `*`, and *type* `&`. Thus, often it is enough to use the following definition:

Click here to view code image

```
class MyIterator
 : public std::iterator <std::bidirectional_iterator_tag, type> {
    ...
};
```

The following example demonstrates how to write a user-defined iterator. It is an insert iterator for associative and unordered containers. Unlike insert iterators of the C++ standard library (see Section 9.4.2, page 454), no insert position is used.

Here is the implementation of the iterator class:

Click here to view code image

```
// iter/assoiter.hpp
#include <iterator>

// class template for insert iterator for associative and unordered containers
template <typename Container>
class asso_insert_iterator
  : public std::iterator <std::output_iterator_tag,
                          typename Container::value_type>
{
   protected:
     Container& container;      // container in which elements are inserted

   public:
     // constructor
     explicit asso_insert_iterator (Container& c) : container(c) {
     }

     // assignment operator
     // - inserts a value into the container
     asso_insert_iterator<Container>&
     operator= (const typename Container::value_type& value) {
         container.insert(value);
         return *this;
     }

     // dereferencing is a no-op that returns the iterator itself
     asso_insert_iterator<Container>& operator* () {
         return *this;
     }

     // increment operation is a no-op that returns the iterator itself
     asso_insert_iterator<Container>& operator++ () {
         return *this;
     }
     asso_insert_iterator<Container>& operator++ (int) {
```

```
            return *this;
        }
};

// convenience function to create the inserter
template <typename Container>
inline asso_insert_iterator<Container> asso_inserter (Container& c)
{
    return asso_insert_iterator<Container>(c);
}
```

The `asso_insert_iterator` class is derived from the `iterator` class, where corresponding types are defined. The first template argument passed is `output_iterator_tag` to specify the iterator category. The second argument is the type of the values the iterator refers to, which is the `value_type` of the container. Because output iterators can be used only to write something, this type definition is not necessary, so you can pass `void` here. However, passing the value type as demonstrated here works for any iterator category.

At creation time the iterator stores its container in its `container` member. Any value that gets assigned is inserted into the container by `insert()`. Operators `*` and `++` are no-ops that simply return the iterator itself. Thus, the iterator maintains control. If the usual iterator interface

```
*pos = value
```

is used, the `*pos` expression returns `*this`, to which the new value is assigned. That assignment is transferred into a call of `insert(value)` for the container.

After the definition of the inserter class, the usual convenient function `asso_inserter` is defined as a convenience function to create and initialize an inserter. The following program uses such an inserter to insert some elements into an unordered set:

Click here to view code image

```
// iter/assoiter1.cpp

#include <iostream>
#include <unordered_set>
#include <vector>
#include <algorithm>
#include "print.hpp"
#include "assoiter.hpp"

int main()
{
    std::unordered_set<int> coll;

    // create inserter for coll
    // - inconvenient way
    asso_insert_iterator<decltype(coll)> iter(coll);

    // insert elements with the usual iterator interface
    *iter = 1;
    iter++;
    *iter = 2;
    iter++;
    *iter = 3;

    PRINT_ELEMENTS(coll);

    // create inserter for coll and insert elements
    // - convenient way
    asso_inserter(coll) = 44;
    asso_inserter(coll) = 55;

    PRINT_ELEMENTS(coll);

    // use inserter with an algorithm
    std::vector<int> vals = { 33, 67, -4, 13, 5, 2 };
    std::copy (vals.begin(), vals.end(),    // source
               asso_inserter(coll));        // destination

    PRINT_ELEMENTS(coll);
}
```

The normal application of the `asso_inserter` demonstrates the `copy()` call:

```
std::copy (vals.begin(), vals.end(),     // source
           asso_inserter(coll));         // destination
```

Here, `asso_inserter(coll)` creates an inserter that inserts any argument passed into `coll`, calling coll.insert(*val*).

The other statements demonstrate the behavior of the inserter in detail. The output of the program is as follows:

```
1 2 3
55 44 1 2 3
-4 33 55 44 67 1 13 2 3 5
```

Note that this iterator could also be used by associative containers. Thus, if you replace `unordered_set` by `set` in both the include directive and the declaration of `coll`, the program would still work (although the elements in the container would be sorted then).