6.5. Iterator Adapters

Iterators are *pure abstractions*: Anything that *behaves* like an iterator *is* an iterator. For this reason, you can write classes that have the interface of iterators but do something completely different. The C++ standard library provides several predefined special iterators: *iterator adapters*. They are more than auxiliary classes; they give the whole concept a lot more power.

The following subsections introduce the following iterator adapters:

1. Insert iterators
2. Stream iterators
3. Reverse iterators
4. Move iterators (since C++11)

Section 9.4, page 448, covers them in detail.

6.5.1. Insert Iterators

Insert iterators, or *inserters*, are used to let algorithms operate in insert mode rather than in overwrite mode. In particular, inserters solve the problem of algorithms that write to a destination that does not have enough room: They let the destination grow accordingly.

Insert iterators redefine their interface internally as follows:

- If you assign a value to their element, they insert that value into the collection to which they belong. Three different insert iterators have different abilities with regard to where the elements are inserted — at the front, at the end, or at a given position.
- A call to step forward is a no-op.

With this interface, they fall under the category of output iterators, which are able to write/assign values only while iterating forward (see Section 9.2, page 433, for details of iterator categories).

Consider the following example:

Click here to view code image

```
// stl/copy2.cpp

#include <algorithm>
#include <iterator>
#include <list>
#include <vector>
#include <deque>
#include <set>
#include <iostream>
using namespace std;

int main()
{
    list<int> coll1 = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

    // copy the elements of coll1 into coll2 by appending them
    vector<int> coll2;
    copy (coll1.cbegin(), coll1.cend(),        // source
          back_inserter(coll2));               // destination

    // copy the elements of coll1 into coll3 by inserting them at the front
    // - reverses the order of the elements
    deque<int> coll3;
    copy (coll1.cbegin(), coll1.cend(),        // source
          front_inserter(coll3));              // destination

    // copy elements of coll1 into coll4
    // - only inserter that works for associative collections
    set<int> coll4;
    copy (coll1.cbegin(), coll1.cend(),      // source
          inserter(coll4,coll4.begin()));    // destination
}
```

This example uses all three predefined insert iterators:

1. **Back inserters** insert the elements at the back of their container (appends them) by calling `push_back()`. For example, with the following statement, all elements of `coll1` are appended into `coll2`:

```
copy (coll1.cbegin(), coll1.cend(),        // source
               back_inserter(coll2));                    // destination
```

Of course, back inserters can be used only for containers that provide `push_back()` as a member function. In the C++ standard library, these containers are `vector`, `deque`, `list`, and strings.

2. **Front inserters** insert the elements at the front of their container by calling `push_front()`. For example, with the following statement, all elements of `coll1` are inserted into `coll3`:

```
copy (coll1.cbegin(), coll1.cend(),         // source
      front_inserter(coll3));               // destination
```

Note that this kind of insertion reverses the order of the inserted elements. If you insert `1` at the front and then `2` at the front, the `1` is after the `2`.

Front inserters can be used only for containers that provide `push_front()` as a member function. In the C++ standard library, these containers are `deque`, `list`, and `forward_list`.

3. **General inserters**, or simply *inserters*, insert elements directly in front of the position that is passed as the second argument of its initialization. A general inserter calls the `insert()` member function with the new value and the new position as arguments. Note that all predefined containers except `array` and `forward_list` have such an `insert()` member function. Thus, this is the only predefined inserter for associative and unordered containers.

   But wait a moment. Passing a position to insert a new element doesn't sound useful for associative and unordered containers, does it? Within associative containers, the position depends on the *value* of the elements, and in unordered containers the position of an element is undefined. The solution is simple: For associative and unordered containers, the position is taken as a *hint* to start the search for the correct position. However, the containers are free to ignore it. Section 9.6, page 471, describes a user-defined inserter that is more useful for associative and unordered containers.

Table 6.1 lists the functionality of insert iterators. Additional details are described in Section 9.4.2, page 454.

Table 6.1. Predefined Insert Iterators

Expression	Kind of Inserter
`back_inserter(`*container*`)`	Appends in the same order by using `push_back(`*val*`)`
`front_inserter(`*container*`)`	Inserts at the front in reverse order by using `push_front(`*val*`)`
`inserter(`*container*`,`*pos*`)`	Inserts at *pos* (in the same order) by using `insert(`*pos*`,`*val*`)`

6.5.2. Stream Iterators

Stream iterators read from or write to a stream.[9] Thus, they provide an abstraction that lets the input from the keyboard behave as a collection from which you can read. Similarly, you can redirect the output of an algorithm directly into a file or onto the screen.

[9] A stream is an object that represents I/O channels (see Chapter 15).

The following example is typical for the power of the whole STL. Compared with ordinary C or C++, the example does a lot of complex processing by using only a few statements:

```
// stl/ioiter1.cpp

#include <iterator>
#include <algorithm>
#include <vector>
#include <string>
#include <iostream>
using namespace std;

int main()
{
    vector<string> coll;

    // read all words from the standard input
    // - source: all strings until end-of-file (or error)
    // - destination: coll (inserting)
    copy (istream_iterator<string>(cin),    // start of source
          istream_iterator<string>(),       // end of source
          back_inserter(coll));             // destination

    // sort elements
    sort (coll.begin(), coll.end());

    // print all elements without duplicates
    // - source: coll
    // - destination: standard output (with newline between elements)
    unique_copy (coll.cbegin(), coll.cend(),             // source
                         ostream_iterator<string>(cout,"\n"));     // destination
}
```

The program has only three statements that read all words from the standard input and print a sorted list of them. Let's consider the three statements step-by-step. In the statement

```
copy (istream_iterator<string>(cin),
      istream_iterator<string>(),
      back_inserter(coll));
```

two input stream iterators are used:

1. The expression

   ```
   istream_iterator<string>(cin)
   ```

   creates a stream iterator that reads from the standard input stream `cin`. The template argument `string` specifies that the stream iterator reads elements of this type (string types are covered in Chapter 13). These elements are read with the usual input operator `>>`. Thus, each time the algorithm wants to process the next element, the istream iterator transforms that desire into a call of

   ```
   cin >> string
   ```

   The input operator for strings usually reads one word separated by whitespaces (see Section 13.2.10, page 677), so the algorithm reads word by word.

2. The expression

   ```
   istream_iterator<string>()
   ```

   calls the default constructor of istream iterators that creates a so-called *end-of-stream iterator*. It represents a stream from which you can no longer read.

As usual, the `copy()` algorithm operates as long as the (incremented) first argument differs from the second argument. The end-of-stream iterator is used as the *end of the range,* so the algorithm reads all strings from `cin` until it can no longer read any more (owing to end-of-stream or an error). To summarize, the source of the algorithm is "all words read from `cin`." These words are copied by inserting them into `coll` with the help of a back inserter.

The `sort()` algorithm sorts all elements:

```
sort (coll.begin(), coll.end());
```

Finally, the statement

```
unique_copy (coll.cbegin(), coll.cend(),
             ostream_iterator<string>(cout,"\n"));
```

copies all elements from the collection into the destination `cout`. During this process, the `unique_copy()` algorithm eliminates adjacent duplicate values. The expression

```
ostream_iterator<string>(cout,"\n")
```

creates an output stream iterator that writes `string`s to `cout` by calling operator `<<` for each element. The second argument behind `cout` is optional and serves as a separator between the elements. In this example, it is a newline character, so every element is written on a separate line.

All components of the program are templates, so you can change the program easily to sort other value types, such as integers or more complex objects. Section 9.4.3, page 460, explains more and gives more examples about iostream iterators.

In this example, one declaration and three statements were used to sort all words read from standard input. However, you could do the same by using only one declaration and one statement. See Section 1, page 394, for an example.

Note that since C++11, you can pass empty curly braces instead of a default constructed stream iterator as end of the range. This works because the type of the argument that defines the end of the source range is deduced from the previous argument that defines the begin of the source range:[10]

[10] Thanks to Jonathan Wakely for pointing this out.

```
copy (istream_iterator<string>(cin),    // start of source
      {},                               // end of source (default constructed istream iterator)
      back_inserter(coll));             // destination
```

6.5.3. Reverse Iterators

Reverse iterators let algorithms operate backward by switching the call of an increment operator internally into a call of the decrement operator, and vice versa. All containers with bidirectional iterators or random-access iterators (all sequence containers except `forward_list` and all associative containers) can create reverse iterators via their member functions `rbegin()` and `rend()`. Since C++11, the corresponding member functions returning read-only iterators, `crbegin()` and `crend()`, are also provided.

For `forward_list`s and unordered containers, no backward-iteration interface (`rbegin()`, `rend()`, etc.) is provided. The reason is that the implementation requires only singly linked lists to go through the elements.

Consider the following example:

Click here to view code image

```
// stl/reviter1.cpp

#include <iterator>
#include <algorithm>
#include <vector>
#include <iostream>
using namespace std;

int main()
{
    vector<int> coll;

    // insert elements from 1 to 9
    for (int i=1; i<=9; ++i) {
        coll.push_back(i);
    }

    // print all element in reverse order
    copy (coll.crbegin(), coll.crend(),          // source
          ostream_iterator<int>(cout," "));      // destination
    cout << endl;
}
```

The following expression returns a read-only reverse iterator for `coll:`

```
coll.crbegin()
```

This iterator may be used as the beginning of a reverse iteration over the elements of the collection. The iterator's position is the last element of the collection. Thus, the following expression returns the value of the last element:

```
*coll.crbegin()
```

Accordingly, the following expression returns for `coll` a reverse iterator that may be used as the end of a reverse iteration.

```
coll.crend()
```

As usual for ranges, the iterator's position is past the end of the range but from the opposite direction; that is, it is the position *before* the first element in the collection.

Again, you should never use operator `*` (or operator `->`) for a position that does not represent a valid element. Thus, the expression

```
*coll.crend()
```

is as undefined as `*coll.end()` or `*coll.cend()`.

The advantage of using reverse iterators is that all algorithms are able to operate in the opposite direction without special code. A step to the next element with operator `++` is redefined into a step backward with operator `--`. For example, in this case, `copy()` iterates over the elements of `coll` from the last to the first element. So, the output of the program is as follows:

```
9 8 7 6 5 4 3 2 1
```

You can also switch "normal" iterators into reverse iterators, and vice versa. However, the referenced value of an iterator changes in doing so. This and other details about reverse iterators are covered in Section 9.4.1, page 448.

6.5.4. Move Iterators

Move iterators are provided since C++11. They convert any access to the underlying element into a move operation. As a result, they allow moving elements from one container into another either in constructors or while applying algorithms. See Section 9.4.4, page 466, for details.