【C++】Day41

Class	C++
Date	@January 19, 2022
Material	
Series Number	
Summary	

【Ch10】Generic Algorithms

10.4 Revisiting Iterators

The library defines several additional kinds of iterators in the iterator header. These iterators include

- Insert iterators: These iterators are bound to a container and can be used to insert elements into the container
- Stream iterators: These iterators are bound to input or output streams and can be used to iterate through an associated IO stream
- Reverse iterators: These iterators move backward, rather than forward. The library containers, other than `forward_list`, have reverse iterators
- Move iterators: These special-purpose iterators move rather than copy their elements. We'll cover move iterators in future lectures.

10.4.1 Insert Iterators

An inserter is an iterator adaptor that takes a container and yields an iterator that adds elements to the specified container.

When we assign a value through an insert iterator, the iterator calls a container operation to add an element at a specified position in the given container.

`it = t`	Inserts the value `t` at the current position denoted by `it`. Depending on the kind of insert iterator, and assuming `c` is the container to which `it` is bound, calls `c.push_back(t)`, `c.push_front(t)`, or `c.insert(t, p)`, where `p` is the iterator position given to `inserter`.
`*it, ++it, it++`	These operations exist but do nothing to `it`. Each operator returns `it`.

There are three kinds of inserters. Each differs from the other as to where elements are inserted:

- `back_inserter` creates an iterator that uses `push_back`
- `front_inserter` creates an iterator that uses `push_front`
- `inserter` creates an iterator that uses `insert`. This function takes a second argument, which must be an iterator into the given container. Elements are inserted ahead of the element denoted by the given iterator.

Note: We can use front_inserter only if the container has push_front. Similarly, we can use back_inserter only if it has push_back.

It is important to understand that when we call `inserter(c, iter)`, we get an iterator that, when used successively, inserts elements ahead of the element originally denoted by `iter`. That is, if it is an iterator generated by inserter, then an assignment such as

```
*it = val;
```

behave as

```
it = c.insert(it, val); //it points to the newly added element
++it; //increment it so that it denotes the same element as before
```

The iterator generated by `front_inserter` behaves quite differently from the one created by inserter. When we use `front_inserter`, elements are always inserted ahead of the then first element in the container.

```
list<int> lst = {1, 2, 3, 4};
list<int> ls2, lst3; //empty lists
//after copy completes, lst2 is {4, 3, 2, 1}
copy(lst.begin(), lst.end(), front_inserter(lst2));
//after copy completes, lst3 is {1, 2, 3, 4}
copy(lst.begin(), lst.end(), inserter(lst3, lst3.begin()));
```

10.4.2 iostream Iterators

Even though the iostream types are not containers, there are iterators that can be used with objects of the IO types.

An `istream_iterator` reads an input stream, and an `ostream_iterator` writes an output stream. These iterators treat their corresponding stream as a sequence of elements of a specified type.

Using a stream iterator, we can use generic algorithms to read data from or write data to stream objects.

`istream_iterator<T> in(is);`	`in` reads values of type `T` from input stream `is`.
`istream_iterator<T> end;`	Off-the-end iterator for an `istream_iterator` that reads values of type `T`.
`in1 == in2` `in1 != in2`	`in1` and `in2` must read the same type. They are equal if they are both the end value or are bound to the same input stream.
`*in`	Returns the value read from the stream.
`in->mem`	Synonym for `(*in).mem`.
`++in, in++`	Reads the next value from the input stream using the `>>` operator for the element type. As usual, the prefix version returns a reference to the incremented iterator. The postfix version returns the old value.

Table 10.4. ostream Iterator Operations

`ostream_iterator<T> out(os);`	`out` writes values of type `T` to output stream `os`.
`ostream_iterator<T> out(os, d);`	`out` writes values of type `T` followed by `d` to output stream `os`. `d` points to a null-terminated character array.
`out = val`	Writes `val` to the `ostream` to which `out` is bound using the `<<` operator. `val` must have a type that is compatible with the type that `out` can write.
`*out, ++out, out++`	These operations exist but do nothing to `out`. Each operator returns `out`.

Operations on istream_iterators

When we create a stream iterator, we must specify the type of objects that the iterator will read or write.

An `istream_iterator` uses `>>` to read a stream. Therefore, the type that an `istream_iterator` reads must have an input operator defined.

When we create an `istream_iterator`, we can bind it to a stream. Alternatively, we can default initialize the iterator, which creates an iterator that we can use as the off-the-end value.

```
istream_iterator<int> int_it(cin); //reads ints from cin
istream_iterator<int> int_eof; //end iterator value
ifstream in("afile");
istream_iterator<string> str_it(in); //reads strings from "afile"
```

As an example, we can use an `istream_iterator` to read the standard input into a vector:

```
istream_iterator<int> in_iter(cin);
istream_iterator<int> eof;
//while there's valid input to read
//postfix increment reads the stream and returns the old value of the iterator
//we dereference taht iterator to get the previosu value read from the stream
wihle(in_iter != eof) {
  vec.push_back(*in_iter++);
}
```

What is more useful is that we can rewrite this program as

```
istream_iterator<int> in_iter(cin), eof;
vector<int> vec(in_iter, eof);
```

Using Stream Iterators with the Algorithms

Because algorithms operate in terms of iterator operations, and the stream iterators support at least some iterator operations, we can use stream iterators with at least some of the algorithms.

As one example, we can call accumulate with a pair of `istream_iterators:`

```
istream_iterator<int> in(cin), eof;
cout << accumualte(in, eof, 0); << endl;
```

Operations on ostream_iterators

An `ostream_iterator` can be defined for any type that has an output operator(the `<<` operator).

When we create an `ostream_iterator`, we may (optionally) provide a second argument that specifies a character string to print following each elements. That string must be a C-style character string.

We must bind an `ostream_iterator` to a specific stream. There is no empty or off-the-end `ostream_iterator`.

We can use an `ostream_iterator` to write a sequence of values:

```
ostream_iterator<int> out_iter(std::cout, " ");
for(auto &elem : vec)
  *out_iter++ = elem;
std::cout << endl;
```

Rather than writing the loop ourselves, we can more easily print the elements in `vec` by calling copy:

```
copy(vec.begin(), vec.end(), out_iter);
cout << endl;
```

Exercise

Exercise 10.29: Write a program using stream iterators to read a text file into a `vector` of `string`s.

See 10_29.cpp for code

Exercise 10.30: Use stream iterators, `sort`, and `copy` to read a sequence of integers from the standard input, sort them, and then write them back to the standard output.

See 10_30.cpp for code

Exercise 10.31: Update the program from the previous exercise so that it prints only the unique elements. Your program should use `unqiue_copy` (§ 10.4.1, p. 403).

See 10_31.cpp for code

Exercise 10.33: Write a program that takes the names of an input file and two output files. The input file should hold integers. Using an `istream_iterator` read the input file. Using `ostream_iterator`s, write the odd numbers into the first output file. Each value should be followed by a space. Write the even numbers into the second file. Each of these values should be placed on a separate line.

See 10_33.cpp for code