21.3 — STL iterators overview

ALEX DECEMBER 21, 2020

An Iterator is an object that can traverse (iterate over) a container class without the user having to know how the container is implemented. With many classes (particularly lists and the associative classes), iterators are the primary way elements of these classes are accessed.

An iterator is best visualized as a pointer to a given element in the container, with a set of overloaded operators to provide a set of well-defined functions:

- Operator* -- Dereferencing the iterator returns the element that the iterator is currently pointing at.
- Operator++ -- Moves the iterator to the next element in the container. Most iterators also provide `Operator--` to move to the previous element.
- Operator== and Operator!= -- Basic comparison operators to determine if two iterators point to the same element. To compare the values that two iterators are pointing at, dereference the iterators first, and then use a comparison operator.
- Operator= -- Assign the iterator to a new position (typically the start or end of the container's elements). To assign the value of the element the iterator is pointing at, dereference the iterator first, then use the assign operator.

Each container includes four basic member functions for use with Operator=:

- begin() returns an iterator representing the beginning of the elements in the container.
- end() returns an iterator representing the element just past the end of the elements.
- cbegin() returns a const (read-only) iterator representing the beginning of the elements in the container.
- cend() returns a const (read-only) iterator representing the element just past the end of the elements.

It might seem weird that end() doesn't point to the last element in the list, but this is done primarily to make looping easy: iterating over the elements can continue until the iterator reaches end(), and then you know you're done.

Finally, all containers provide (at least) two types of iterators:

- container::iterator provides a read/write iterator
- container::const_iterator provides a read-only iterator

Lets take a look at some examples of using iterators.

Iterating through a vector

```cpp
#include <iostream>
#include <vector>

int main()
{
    std::vector<int> vect;
    for (int count=0; count < 6; ++count)
        vect.push_back(count);

    std::vector<int>::const_iterator it; // declare a read-only iterator
    it = vect.cbegin(); // assign it to the start of the vector
    while (it != vect.cend()) // while it hasn't reach the end
        {
        std::cout << *it << " "; // print the value of the element it points to
        ++it; // and iterate to the next element
        }

    std::cout << '\n';
}
```

This prints the following:

0 1 2 3 4 5

Iterating through a list

Now let's do the same thing with a list:

```cpp
#include <iostream>
#include <list>

int main()
{

    std::list<int> li;
    for (int count=0; count < 6; ++count)
        li.push_back(count);

    std::list<int>::const_iterator it; // declare an iterator
    it = li.cbegin(); // assign it to the start of the list
    while (it != li.cend()) // while it hasn't reach the end
    {
        std::cout << *it << " "; // print the value of the element it points to
        ++it; // and iterate to the next element
    }

    std::cout << '\n';
}
```

This prints:

0 1 2 3 4 5

Note the code is almost identical to the vector case, even though vectors and lists have almost completely different internal implementations!

Iterating through a set

In the following example, we're going to create a set from 6 numbers and use an iterator to print the values in the set:

```cpp
#include <iostream>
#include <set>

int main()
{
    std::set<int> myset;
    myset.insert(7);
    myset.insert(2);
    myset.insert(-6);
    myset.insert(8);
    myset.insert(1);
    myset.insert(-4);

    std::set<int>::const_iterator it; // declare an iterator
    it = myset.cbegin(); // assign it to the start of the set
    while (it != myset.cend()) // while it hasn't reach the end
    {
        std::cout << *it << " "; // print the value of the element it points to
        ++it; // and iterate to the next element
    }

    std::cout << '\n';
}
```

This program produces the following result:

```
-6 -4 1 2 7 8
```

Note that although populating the set differs from the way we populate the vector and list, the code used to iterate through the elements of the set was essentially identical.

Iterating through a map

This one is a little trickier. Maps and multimaps take pairs of elements (defined as a std::pair). We use the make_pair() helper function to easily create pairs. std::pair allows access to the elements of the pair via the first and second members. In our map, we use first as the key, and second as the value.

```cpp
#include <iostream>
#include <map>
#include <string>

int main()
{
 std::map<int, std::string> mymap;
 mymap.insert(std::make_pair(4, "apple"));
 mymap.insert(std::make_pair(2, "orange"));
 mymap.insert(std::make_pair(1, "banana"));
 mymap.insert(std::make_pair(3, "grapes"));
 mymap.insert(std::make_pair(6, "mango"));
 mymap.insert(std::make_pair(5, "peach"));

 auto it{ mymap.cbegin() }; // declare a const iterator and assign to start of vector
 while (it != mymap.cend()) // while it hasn't reach the end
 {
  std::cout << it->first << "=" << it->second << " "; // print the value of the element it points to
  ++it; // and iterate to the next element
 }

 std::cout << '\n';
}
```

This program produces the result:

```
1=banana 2=orange 3=grapes 4=apple 5=peach 6=mango
```

Notice here how easy iterators make it to step through each of the elements of the container. You don't have to care at all how map stores its data!

Conclusion

Iterators provide an easy way to step through the elements of a container class without having to understand how the container class is implemented. When combined with STL's algorithms and the member functions of the container classes, iterators become even more powerful. In the next lesson, you'll see an example of using an iterator to insert elements into a list (which doesn't provide an overloaded operator[] to access its elements directly).

One point worth noting: Iterators must be implemented on a per-class basis, because the iterator does need to know how a class is implemented. Thus iterators are always tied to specific container classes.

Next lesson
21.4 STL algorithms overview

Back to table of contents

Previous lesson
21.2 STL containers overview

Notify me about replies: POST COMMENT

DP N N FOUT

Newest

X

X