STL Iterators

By Alex Allain

The concept of an iterator is fundamental to understanding the C++ Standard Template Library (STL) because iterators provide a means for accessing data stored in container classes such a vector, map, list, etc.

You can think of an iterator as pointing to an item that is part of a larger container of items. For instance, all containers support a function called begin, which will return an iterator pointing to the beginning of the container (the first element) and function, end, that returns an iterator corresponding to having reached the end of the container. In fact, you can access the element by "dereferencing" the iterator with a *, just as you would dereference a pointer.

To request an iterator appropriate for a particular STL templated class, you use the syntax

```
std::class_name<template_parameters>::iterator name
```

where *name* is the name of the iterator variable you wish to create and the *class_name* is the name of the STL container you are using, and the *template_parameters* are the parameters to the template used to declare objects that will work with this iterator. Note that because the STL classes are part of the std namespace, you will need to either prefix every container class type with "std::", as in the example, or include "using namespace std;" at the top of your program.

For instance, if you had an STL vector storing integers, you could create an iterator for it as follows:

```
std::vector<int> myIntVector;
std::vector<int>::iterator myIntVectorIterator;
```

Different operations and containers support different types of iterator behavior. In fact, there are several different classes of iterators, each with slightly different properties. First, iterators are distinguished by whether you can use them for reading or writing data in the container. Some types of iterators allow for both reading and writing behavior, though not necessarily at the same time.

Some of the most important are the forward, backward and the bidirectional iterators. Both of these iterators can be used as either input or output iterators, meaning you can use them for either writing or reading. The forward iterator only allows movement one way -- from the front of the container to the back. To move from one element to the next, the increment operator, ++, can be used.

For instance, if you want to access the elements of an STL vector, it's best to use an iterator instead of the traditional C-style code. The strategy is fairly straightforward: call the container's begin function to get an iterator, use ++ to step through the objects in the container, access each object with the * operator ("*iterator") similar to the way you would access an object by dereferencing a pointer, and stop iterating when the iterator equals the container's end iterator. You can compare iterators using != to check for inequality, == to check for equality. (This only works when the iterators are operating on the same container!)

The old approach (avoid)

```
using namespace std;

vector<int> myIntVector;
// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);
```

```
for(int y=0; y<myIntVector.size(); y++)
{
    cout<<myIntVector[y]<<" ";
    //Should output 1 4 8
}
```

The STL approach (use this)

```
using namespace std;

vector<int> myIntVector;
vector<int>::iterator myIntVectorIterator;

// Add some elements to myIntVector
myIntVector.push_back(1);
myIntVector.push_back(4);
myIntVector.push_back(8);

for(myIntVectorIterator = myIntVector.begin();
        myIntVectorIterator != myIntVector.end();
        myIntVectorIterator++)
{
    cout<<*myIntVectorIterator<<" ";
    //Should output 1 4 8
}
```

As you might imagine, you can use the decrement operator, --, when working with a bidirectional iterator or a backward operator.

Iterators are often handy for specifying a particular range of things to operate on. For instance, the range *item*.begin(), *item*.end() is the entire container, but smaller slices can be used. This is particularly easy with one other, extremely general class of iterator, the random access iterator, which is functionally equivalent to a pointer in C or C++ in the sense that you can not only increment or decrement but also move an arbitrary distance in constant time (for instance, jump multiple elements down a vector).

For instance, the iterators associated with vectors are random access iterators so you could use arithmetic of the form

```
iterator + n
```

where n is an integer. The result will be the element corresponding to the nth item after the item pointed to be the current iterator. This can be a problem if you happen to exceed the bounds of your iterator by stepping forward (or backward) by too many elements.

The following code demonstrates both the use of random access iterators and exceeding the bounds of the array (don't run it!):

```
vector<int> myIntVector;
vector<int>::iterator myIntVectorIterator;
myIntVectorIterator = myIntVector.begin() + 2;
```

You can also use the standard arithmetic shortcuts for addition and subtraction, += and -=, with random access iterators. Moreover, with random access iterators you can use <, >, <=, and >= to compare iterator positions within the container.

Iterators are also useful for some functions that belong to container classes that require operating on a range of values. A simple but useful example is the erase function. The vector template supports this function, which takes a range as specified by two iterators -- every element in the range is erased. For instance, to erase an entire vector:

```
vector<int>::iterator myIntVectorIterator;
myIntVector.erase(myIntVectorIterator.begin(), myIntVectorIterator.end());
```

which would delete all elements in the vector. If you only wanted to delete the first two elements, you could use

```
myIntVector.erase(myIntVectorIterator.begin(), myIntVectorIterator.begin()+2);
```

Note that various container class support different types of iterators -- the vector class, which has served as our model for iterators, supports a random access iterator, the most general kind. Another container, the list container (to be discussed later), only supports bidirectional iterators.

So why use iterators? First, they're a flexible way to access the data in containers that don't have obvious means of accessing all of the data (for instance, maps [to be discussed later]). They're also quite flexible -- if you change the underlying container, it's easy to change the associated iterator so long as you only use features associated with the iterator supported by both classes. Finally, the STL algorithms defined in <algorithm> (to be discussed later) use iterators.

Summary

The Good

- The STL provides iterators as a convenient abstraction for accessing many different types of containers.
- Iterators for templated classes are generated inside the class scope with the syntax

  ```
  class_name<parameters>::iterator
  ```

- Iterators can be thought of as limited pointers (or, in the case of random access iterators, as nearly equivalent to pointers)

The Gotchas

- Iterators do not provide bounds checking; it is possible to overstep the bounds of a container, resulting in segmentation faults
- Different containers support different iterators, so it is not always possible to change the underlying container type without making changes to your code
- Iterators can be invalidated if the underlying container (the container being iterated over) is changed significantly

Previous: STL Vectors
Next: STL Associative Arrays (Maps)

Read more...

C++11 range-based for loops takes iterators to the next level of usability and simplicity

C++11's Auto Keyword really helps make working with interators easier