All You Want To Know About C++ Iterators

June 30, 2016

1 Introduction

This week we will focus on C++ iterators. When I first encountered C++ iterators I was intimidated. I found the syntax strange and felt there was no way they could be more efficient than my non-iterator based code. I felt I was never going to be able to understand them, but was going to be forced to use them because they were so embedded into so many of the C++ standard library classes. I felt I was just going to have to "use this code because it works" without ever understanding the details.

I could not have been more wrong. Iterators are nothing more than a straightforward abstraction of a commonly used technique. They turned out to be very efficient – I could not beat their performance. Once you see the abstraction they are not difficult to understand and, while there are many details, can be fully mastered by an average programmer.

2 The Abstraction

Recall that in C/C++ the compiler converts `a[i]` to `*(a+i)`. `(a+i)` is a pointer to the i^{th} element of `a`, and the `*` dereferences that pointer. This is as tricky as iterators get.

Suppose you have an integer array, `a`, of n elements that you want to initialize to 0. A novice's approach to the problem would be this code snippet:

```
1    for (unsigned int i = 0u; i < n; ++i)
2    {
3      a[i] = 0;
4    }
```

If you showed this code to a more experienced friend, they may point out that the code performs indexing for each loop iteration. They may suggest using pointers as in this code snippet:

```
1  int *a_ptr = a;
2  for (unsigned int i = 0u; i < n; ++i)
3  {
4    *a_ptr++ = 0;
5  }
```

Proud of your optimized code, you show your new code to an even more experienced friend, and they point out the post increment on line 4 is a bit inefficient. They suggest this code snippet:

```
1  int *a_ptr = a;
2  for (unsigned int i = 0u; i < n; ++i, ++a_ptr)
3  {
4    *a_ptr = 0;
5  }
```

Next you show the code to a member of the BOOST team, and they wonder how this could be abstracted. The most obvious problem is two variables of differing types are used: one is an unsigned int to control the number of times through the loop, and the other is the pointer you dereference to set the elements to 0. They suggest an approach that only uses the pointer:

```
1  for (int *a_ptr = a + 0; a_ptr != a + n; ++a_ptr)
2  {
3    *a_ptr = 0;
4  }
```

Recall that `a + 0` is simply a pointer to the first element in the array, and `a + n` is a pointer one past the last element in the array. If you consider it for a while, you can see the above code is the most efficient version of those presented thus far.

Now that we efficiently use only one variable we are ready for the abstraction. Suppose we abstract `a + 0` to an inline function called `begin()`, and `a + n` to an inline function called `end()`. Now the snippet looks like:

```
1   inline int *begin(void) {return a + 0;};
2   inline int *end(void) {return a + n;};
3   . . .
4   for (int *a_ptr = begin(); a_ptr != end(); ++a_ptr)
5   {
6     *a_ptr = 0;
7   }
```

The final step in the abstraction, is to define the iterator. That's a simple `typedef`. We will rename `a_ptr` to `it` to complete the abstraction.

```
1   typedef int *iterator;
2   inline iterator begin(void) {return a + 0;};
3   inline iterator end(void) {return a + n;};
4   . . .
5   for (iterator it = begin(); it != end(); ++it)
6   {
7     *it = 0;
8   }
```

If you compare this code to examples of iterators, you will see the similarities. The only real difference is this example was done outside the context of a class. The above is actually the basic model for the iterators in `std::vector`. `std::vector` actually defines `begin()` and `end()` functions, and an `iterator` type.

The sample code demonstrates the above with the class `cVectorSimpleIterator`.

3 A More Detailed Look

The abstraction for iterators extends beyond simple pointers. The generalized abstraction requires implementing the pre/post increment operators with the meaning *move to the next element*. The general abstraction requires `begin()` refer to the first element and `end()` refer to the stopping value. We have seen what this means for a simple reference iterator.

An iterator can be abstracted for a linked list. In that case, `begin()` would be a pointer to the front of the list, `end()` would be `nullptr`, and the pre/post increment operators would chase the next pointer. In this way, the exact same `for` loop syntax on line 5 above could be used to traverse a linked list.

Iterators are usually implemented as part of a class containing nodes to be traversed. Such classes are call *container* classes. Examples from the standard library are *std::vector*, *std::queue* and *std::list*. A good overview of the standard library container classes can be found here.

Notice that some of the containers are *associative containers*. Those are covered in more detail later.

Iterators that traverse the nodes in the forward direction are called *forward iterators*, and use `begin()` and `end()`. There type is `iterator`.

Iterators that traverse the nodes in the backward direction are called *reverse iterators*, and use `rbegin()` and `rend()`. There type is `reverse_iterator`.

The above forward and reverse iterators allow the programmer to change the contents of the nodes. There are constant variants of the above with types `const_iterator` and `const_reverse_iterator`. The use the functions `cbegin()`, `cend()`, `crbegin()` and `crend()`.

Iterators can be *bi-directional*. These iterators implement pre/post decrement operators.

Iterators can also be *random access*. Random access operators implement the additional operators +, +=, -, -=, [] and comparison operators.

For more details, refer to:

- More information on forward iterators can be found here.
- More information on bi-directional iterators can be found here.
- More information on random access iterators can be found here.

A good overview of the iterator categories and the operators that must implemented for each category can be found here. In addition to the operators, some helper types must be defined. These types are used by the standard library. There is a special std::iterator class that defines these types.

```
1   template <class Category, class T, class Distance = ptrdiff_t,
2             class Pointer = T*, class Reference = T&>
3     struct iterator {
4       typedef T         value_type;
5       typedef Distance  difference_type;
6       typedef Pointer   pointer;
7       typedef Reference reference;
8        typedef Category   iterator_category;
9      };
```

The types are defined on lines 4-8. The iterator_category type is one of the following:

- input_iterator_tag
- output_iterator_tag
- forward_iterator_tag
- bidirectional_iterator_tag
- random_access_iterator_tag

The iterator_category type is used by the standard library to optimize iterator implementation.

In practice, the above means that container classes must implement at least the class types: `iterator`, `const_iterator`. If reverse iterators are needed, then the class types `reverse_iterator` and `const_reverse_iterator` must also be defined. The appropriate `begin()` and `end()` functions must be implemented as class functions.

The iterator classes must implement the operators appropriate for the iterator type (forward, bi-directional or random access).

4 Range For Loops

If you have implemented `iterator`, `const_iterator` that are at least forward iterators, then you can use range for loops.

5 Associative Containers

There is a special class of containers called associative containers. They differ from standard containers in that each node contains a key and value. The standard library handles these uniformly via the `std::pair` class. Each pair has a first (which is the key) and a second (which is the value). The example program shows how these work for a simple string map using `std::map`.