Published on **O'Reilly Network** (http://www.oreillynet.com/)
http://www.oreillynet.com/pub/a/network/2005/11/21/what-is-iterator-in-c-plus-plus-part2.html
See this if you're having trouble printing code examples

What Is an Iterator in C++, Part 2

by Ryan Stephens, coauthor of C++ Cookbook
11/21/2005

Iterator

> An iterator in C++ is a concept that refines the iterator design pattern into a specific set of behaviors that work well with the C++ standard library. The standard library uses iterators to expose elements in a range, in a consistent, familiar way. Anything that implements this set of behaviors is called an iterator. This paves the way for generic algorithms and makes it easy to implement your own iterators and have them integrate smoothly with the standard library.

In This Article:

1. Reverse Iterators
2. Stream Iterators
3. Homemade Iterators

In part one of this two-part series, I described what an iterator is, both in terms of the iterator pattern and its implementation in C++. That explanation is sufficient when you are using the standard containers, but there are other kinds of iterators I didn't discuss that you should know about: reverse iterators, stream iterators, and custom iterators. I discuss each of these in this article.

Reverse Iterators

Say you want to start at the end of a range of elements in a standard container and move toward the beginning. No problem. If your container supports bidirectional or random access iterators, you can simply move backward using the decrement operator, right? Well, yes, you can, but this approach has its problems. Consider looping backward through a container of elements:

```
list<string> lst;
// fill lst with data...
list<string>::iterator it;
for (it = lst.end(), it--;    // Init on last
   it != lst.begin(); --it) { // element
      cout << *it << endl; // Uh-oh, skipped
                           // first element!
}
```

This snippet works, although it actually omits the first element in the list, which is the first pain-in-the-neck with this approach. The one-past-the-end semantics don't work when you're going backward, so you have to treat the first element as a special case. You can add an `if` statement here and there to accommodate this, but you're still not out of the woods.

If you write an algorithm that operates on a range, or you want to use standard algorithms on a range in reverse order (without copying it to another container in reverse), the naive approach above won't work. Say you write a simple function template to print each element in a range, like this:

```
template<typename Fwd>
void printRange(Fwd first, Fwd last,
   char delim = ',', ostream& out = cout) {
    out << "{";
    while (first != last) {
        out << *first;
        if (++first != last)
            out << delim << ' ';
    }
    out << "}" << endl;
}
```

This won't work on the same range forward and backward, because it uses the pre-increment operator to advance the iterator from the beginning to the end of the range. You have to write two functions, `printRangeForward` and `printRangeBackward`, to work in both directions. If you have lots of algorithms like this, then you will be writing a lot of duplicate code.

Reverse iterators solve this problem. A reverse iterator is an adapter class on a container's iterator class that lets you go backward using the same one-past-the-end semantics, except that it is one-past-the-beginning instead. You can retrieve a `reverse_iterator` using the `rbegin` and `rend` member functions (instead of `begin` and `end`), like this:

```
for (list<string>::reverse_iterator it = lst.rbegin();
     it != lst.rend(); ++it) {
   cout << *it << endl;
}
```

You can also use a `const_reverse_iterator` if you are working with a `const` container. With reverse iterators, the `printRange`
function template above can print the same container's elements forward or backward:

```
printRange(lst.begin(), lst.end());
printRange(lst.rbegin(), lst.rend());
```

You can also use the standard algorithms with reverse iterators. Consider the standard `copy` algorithm:

```
list<string> lst2(lst.size());
copy(lst.rbegin(), lst.rend(), lst2.begin());
```

This will copy the elements in `lst` to `lst2` in reverse order.

All is not rosy with reverse iterators though, because some container member functions take an `iterator` parameter, and arguments of type `reverse_iterator` won't compile. Consider finding and erasing an element that you know to be near the end of a container. You would probably want to begin your search at the end of the iterator, if you use the standard `find` algorithm, like this:

```
lst.erase(find(lst.rbegin(), // Won't compile
   lst.rend(), "boat"));
```

This won't work, though, because `find`'s return value is the same type as its arguments, which, in this case, are `reverse_iterators`, and `list::erase` requires a parameter of the type `iterator`. There's a quick way around this, though, with the `base` member function, which is provided by `reverse_iterator` to let you get at its underlying `iterator` for just this reason. Here's a version that will compile and run:

```
lst.erase(find(lst.rbegin(), lst.rend(), "boat").base());
```

Using `base` may feel like a hack, but it works.

The moral of this discussion on reverse iterators is that you should keep them in mind so that you don't waste processor cycles by copying your containers into reverse order, or waste your mental cycles writing goofy `for` loops to iterate backward using `for` loops.

Stream Iterators

The iterator pattern applies to any sort of sequence of elements, whatever that sequence's implementation may be. An interesting take on this is found in the stream iterators in the standard library.

A stream iterator is an iterator interface to a stream. Here's a simple example of reading a few strings from `cin`:

```
list<string> strs;
cout << "Enter some strings, (enter, ctrl-z, enter to quit): ";
istream_iterator<string> input(cin);

while (!cin.eof())
   strs.push_back(*input++);
```

This creates an iterator named `input`
that iterates over elements read from the stream. The elements read from the stream are those that would be read by using the >> operator, as in `cin >> str`. It reads until `cin.eof` returns true, which, on my Windows XP machine, is when I press `Enter`, `Ctrl-Z`, `Enter`.

You can also use the one-past-the-end semantics on input streams. If you invoke the stream constructor with no arguments, the `istream_iterator` object it creates is an end marker. For example, instead of using `cin.eof` above, I could have done this instead:

```
istream_iterator<string> input(cin);
istream_iterator<string> inputEnd;

while (input != inputEnd)
   strs.push_back(*input++);
```

In fact, this is what makes stream iterators work so well with standard algorithms. Consider the opposite situation of writing the contents of a container to an output stream. Here's a way to do that:

```
list<string> lst;
lst.push_back("banana");
lst.push_back("apple");
lst.push_back("strawberry");
```

```
ostream_iterator<string> outStr(cout, " ");

copy(lst.begin(), lst.end(), outStr);
```

This will dump the contents of `lst` to `cout`, and delimit each one with a space. The constructor of `ostream_iterator` accepts an optional delimiter as its second argument.

Finally, stream iterators work with any kind of stream, so you can use them for file streams, too. Here's an example of reading elements from a file into a `vector`:

```
ifstream in("data.txt");
istream_iterator<string> start(in);
istream_iterator<string> end;
vector<string> v(start, end);
```

These few lines do quite a bit. The first line opens a file stream from my text file named *data.txt*. Then I create two input stream iterators, `start` and `end`, which I described a moment ago. The last line emphasizes how iterators make the details of their sources irrelevant. I create a string, `vector v`, using the range constructor (which copies the elements in the range into the new `vector`), and give it the two stream iterators I just created. This works the same as the first example in this section: it reads elements from the input stream until the stream reaches an end marker. This is a quick way to read elements from a stream into a container.

I find stream iterators more useful for debugging than anything else, but they illustrate an important point: that any sort of sequence of elements is a candidate for an iterator interface. The next section explains how you can add an iterator interface to your own classes.

Homemade Iterators

Since a C++ iterator provides a familiar, standard interface, at some point you will no doubt want to add one to your own classes. This is a good idea, because it allows you to plug-and-play with standard library algorithms and it lets consumers of your data structure use the same iterator interface they're probably already familiar with. Thankfully, writing your own iterators is a straightforward process, with only a couple of subtleties you need to be aware of.

Before you begin implementing your own iterator, you should decide which category you can support. Use the same design parameter as the standard library and only support an iterator category that you can implement efficiently. "Efficiently" means different things to different people, but typically this means logarithmic complexity or better. For example, among the standard containers, `vector` provides random access iterators, while `list` offers only bidirectional iterators. A `list` is often implemented as a doubly linked list, so you know it's *possible*
to implement random access iterators (by advancing forward and backward to support array-style indexing of elements or iterator arithmetic), so why not provide them? Because doing so would provide linear complexity at best, and that would result in terrible performance for clients who use the random access capability to navigate back and forth.

A forward iterator is a good start. In most cases, a forward iterator suffices, and its requirements are, uh, straightforward. At a high level, a forward iterator supports assignment, tests for equality, forward advancement using the prefix and postfix forms of the `++` operator, and dereferencing that returns an `rvalue` or an `lvalue`, meaning the caller of the dereference operator can assign from the value returned or assign to it.

To use a simple example, let's say you've implemented a queue as a singly linked list, where consumers can add elements to the back with a `pushBack` member function, or remove elements from the front with the `popFront` member function. Example 1 presents some code for this class, named `SQueue`.

Example 1. A singly linked queue

```
#ifndef SQUEUE_H__
#define SQUEUE_H__

#include <stdexcept>
#include <iterator>

// A simple queue as a singly-linked list
// A simple queue as a singly-linked list
template<typename T>
class SQueue {

private:
   // The Node class holds the value
   class Node {
   public:
      Node(const T& val) : next_(NULL), val_(val) {}
      T& getVal() {return(val_);}
      Node* next_;
   private:
      T val_;
   };

   Node* root_;
   Node* last_;

public:
   SQueue() : root_(NULL), last_(NULL) {}
  ~SQueue()
   {
      delete root_;
   }

   void pushBack(const T& val)

   {
      if (root_ == NULL)
      {
         root_ = new Node(val);
         last_ = root_;
      }
      else
      {
         Node* p = new Node(val);
         last_->next_ = p;
         last_ = p;
      }
   }

   T popFront()
   {
      if (root_ == NULL)
      {
         throw std::out_of_range("Queue is empty");
      }
      else
      {
         Node* p = root_;
         root_ = root_->next_;
         return(root_->getVal());
      }
   }
};
```

The implementation of `SQueue`
is not important. What is important here is that you want to provide an interface to `SQueue` whereby client code can examine each of the elements one at a time, in the same way that you might iterate through a `list`. Since this is a singly linked structure, the only type of iterator you can reasonably provide is a forward iterator.

As I said earlier, there are two good reasons to add an iterator to `SQueue`. First, a user of `SQueue` can iterate

through elements in the queue in the same way that he could iterate through a standard container. For example, printing out each element in an `SQueue` might look like this:

```
SQueue<int> q;
// Fill up q with ints

for (SQueue<int>::Iterator it = q.begin();
     it != q.end(); ++it) {
   cout << "*it = " << *it << endl;
}
```

Second, you want users of `SQueue` to be able to use it with standard algorithms, like this:

```
list<int> lst;
std::copy(q.begin(), q.end(),
   back_inserter (lst));
```

This example uses the standard library algorithm copy (declared in `<algorithm>`) to copy each element in the `SQueue` `q` into the `list` named `lst`. `back_inserter` is a standard library function template defined in `<iterator>` that returns an output iterator, which repeatedly calls `push_back` on its container argument.

A custom iterator will satisfy both of the use cases. At this point, you know which category of iterator you want to support, and therefore you should be aware of all of the member functions you need to implement (see part one of this series for more information about iterator categories). Here is a simple `Iterator` class that meets the first use case:

```
class Iterator {
   public:
      Iterator(Node* p) : node_(p) {}
     ~Iterator() {}

      // The assignment and relational operators are straightforward
      Iterator& operator=(const Iterator& other)
      {
         node_ = other.node_;
         return(*this);
      }

      bool operator==(const Iterator& other)
      {
         return(node_ == other.node_);
      }

      bool operator!=(const Iterator& other)
      {
         return(node_ != other.node_);
      }

      // Update my state such that I refer to the next element in the
      // SQueue.
      Iterator& operator++()
      {
         if (node_ != NULL)
         {
            node_ = node_->next_;
         }
         return(*this);
      }

      Iterator& operator++(int)
      {
         Iterator tmp(*this);
         ++(*this);
         return(tmp);
      }

      // Return a reference to the value in the node.  I do this instead
      // of returning by value so a caller can update the value in the
      // node directly.
```

```
      T& operator*()
      {
         return(node_->getVal());
      }

      // Return the address of the value referred to.
      T* operator->()
      {
         return(&*(SQueue<T>::Iterator)*this);
      }

   private:
      Node* node_;
   };
```

The first thing you will probably notice in my `Iterator` class is that it uses private members of `SQueue`. It has to, since it needs to be able to advance from one element to the next, which can't be done without knowing `SQueue`'s internal structure. I could make `Iterator` go through a public interface to `SQueue`, but that would be a convoluted design, since the only reason for `Iterator`'s existence is to know the details of how `SQueue`
works and to provide an interface to the outside world: creating an interface for the iterator itself to use would be redundant.

I embedded the definition of `Iterator` within the definition of `SQueue` to provide it access to `SQueue`'s private members. Doing so gives `Iterator` access to `SQueue`'s private data. It makes more sense to do it this way than to create a standalone `Iterator` class and declare it a friend of `SQueue`, because `Iterator` is only used by `SQueue`, so there's no reason to make it a standalone class.

Once you have added the `Iterator`
definition as I just did, you have to provide member functions that return an `Iterator` so clients can use it. The standard containers' `begin` and `end`
functions provide a good model for this. Here's how I would implement them:

```
Iterator begin() {
   return(Iterator(root_));
}

Iterator end() {
   return(Iterator(NULL));
}
```

The `begin`
member function returns an iterator that refers to the first element in the sequence, which is pointed to by `root_`. `end` returns an `Iterator`
that points to nothing, which satisfies the one-past-the-end semantics of the standard containers, since I also implemented the `++` operator such that when the user advances past the end of the sequence, he gets an `Iterator` that refers to `NULL`. This allows the equality test of the `Iterator` returned by `end` and an `Iterator` that has been advanced past the last element to return `true`.

Note that the definition of `Iterator` in the body of `SQueue` must come before the `begin` and `end` member functions (or you must use a forward declaration of `Iterator`), since they both refer to the `Iterator` class. Not to worry: if you refer to `Iterator`
before declaring it, your compiler will be sure to point out your oversight. There are two ways to dereference my iterator, with the `*` or `->` operators. I implemented them like this:

```
T& operator*()
{
   return(node_->getVal());
}

T* operator->()
{
   return(&*(SQueue<T>::Iterator)*this);
}
```

The first member function returns a reference to the value referred to by the iterator. This differs from the standard containers, which return copies of the objects they contain, not references to them. You can do it either way, but be aware of the implications of departing from the semantics the standard containers follow, because it can be misleading for users of your classes.

The second member function implements the right-arrow operator for referring to an object's members. There is a lot going on in the `return` statement, so here's a more explicit version:

```
T* operator->()
{
   SQueue<T>::Iterator tmp = *this; // *this is an Iterator
   T& theVal = *tmp; // dereference *this to get the value
   return (&theVal); // Return the address of the referent
}
```

This lets users store objects in your queue, then invoke members on those objects like this:

```
SQueue<MyClass> q;
// fill up q with MyClass instances
SQueue<MyClass>::Iterator p = q.begin();

// This calls MyClass::foo on the first element in q
p->foo();
```

Of course, this only works if the type of element in the container is a `class` or `struct` and can therefore have its members referred to using the `->`
operator. Don't sit back and relax just yet. At this point, you are functionally only halfway there, because `Iterator`
as defined above will not work with standard library algorithms. This only requires a subtle change, though, so don't worry.

Standard algorithms need to know more about the iterators they are using than my `Iterator` class provides. Specifically, they need to know the category of iterator, the type of the difference between two iterators, and other sorts of things. Usually, this is used by standard library implementations to write function templates that are specific to each kind of template argument, where applicable. This information is packaged up using the standard class `iterator_traits`. You can create a specialization of `iterator_traits` explicitly, but it is far easier to let the standard library do it for you by inheriting from the `Iterator` class in `<iterator>`. Here's how I did it:

```
class Iterator : public
   std::iterator<std::forward_iterator_tag, T> {
Now, your iterator will work with the standard algorithms, like this:
list<int> lst;
copy(q.begin(), q.end(),
   back_inserter(lst));
```

The `Iterator`
class does not provide virtual functions (as you might expect) that you're supposed to override. It is used only to package up information about the iterator so it can be accessed by consumers. All you need to do is inherit from it to get this information for free; once you do, you can operate with the standard algorithms (and other non-standard libraries that do the same thing, such as those in the Boost project) with minimal effort.

I have presented the solution in pieces in the previous examples. See the code at the end of this article for a complete implementation of `SQueue` with the `Iterator` class. Adopting the iterator pattern to your data structure is a relatively simple process that gives you a number of benefits. The two most apparent are the familiar interface iterators offer and the interoperability with the standard library. The examples I gave above should get you on your way to implementing iterators effectively.

Example 2. A complete implementation

```
#ifndef SQUEUE_H__
#define SQUEUE_H__

#include <stdexcept>
#include <iterator>

// A simple queue as a singly-linked list
template<typename T>
class SQueue {

private:
   // The Node class holds the value
   class Node {
   public:
      Node(const T& val) : next_(NULL), val_(val) {}
      T& getVal() {return(val_);}
      Node* next_;
   private:
      T val_;
   };

   Node* root_;
   Node* last_;

public:
   SQueue() : root_(NULL), last_(NULL) {}
  ~SQueue()
   {
      delete root_;
   }

   void pushBack(const T& val)
   {
      if (root_ == NULL)
      {
         root_ = new Node(val);
         last_ = root_;
      }
      else
      {
         Node* p = new Node(val);
         last_->next_ = p;
         last_ = p;
      }
   }

   T popFront()
   {
      if (root_ == NULL)
      {
         throw std::out_of_range("Queue is empty");
      }
      else
      {
         Node* p = root_;
         root_ = root_->next_;
         return(root_->getVal());
      }
   }

   // Here is my custom iterator.  The only kind of iterator this data
   // structure can reasonably support is a forward iterator, so that's
   // what I provide.  I embedded the definition of the iterator within
   // the class it will iterate through for convenience.
   class Iterator :
      public std::iterator<std::forward_iterator_tag, T> {
   public:
      Iterator(Node* p) : node_(p) {}
     ~Iterator() {}

      // The assignment and relational operators are straightforward
      Iterator& operator=(const Iterator& other)
      {
         node_ = other.node_;
```

```
            return(*this);
        }

        bool operator==(const Iterator& other)
        {
            return(node_ == other.node_);
        }

        bool operator!=(const Iterator& other)
        {
            return(node_ != other.node_);
        }

        // Update my state such that I refer to the next element in the
        // SQueue.
        Iterator& operator++()
        {
            if (node_ != NULL)
            {
                node_ = node_->next_;
            }
            return(*this);
        }

        Iterator& operator++(int)
        {
            Iterator tmp(*this);
            ++(*this);
            return(tmp);
        }

        // Return a reference to the value in the node.  I do this instead
        // of returning by value so a caller can update the value in the
        // node directly.
        T& operator*()
        {
            return(node_->getVal());
        }

        T* operator->()
        {
            return(&*(SQueue<T>::Iterator)*this);
        }

    private:
        Node* node_;
    };

    Iterator begin()
    {
        return(Iterator(root_));
    }

    Iterator end()
    {
        return(Iterator(NULL));
    }
};

#endif // SQUEUE_H__
```

Ryan Stephens
is a software engineer, writer, and student living in Tempe, Arizona. He enjoys programming in virtually any language, especially C++.

Return to the O'Reilly Network