Chris Riesbeck
Last updated: July 3, 2009

Defining C++ Iterators

Overview

Iterators are central to the generality and efficiency of the generic algorithms in the STL. All the STL containers (but not the adapters) define

- the **iterator types** for that container, e.g., `iterator` and `const_iterator`, e.g., `vector<int>::iterator`
- the **begin/end methods** for that container, i.e., `begin()` and `end()`

(Reverse iterators, returned by `rbegin()` and `rend()`, are beyond the scope of this discussion.)

Containers that don't define the above are second-class citizens. They can't be used with the generic algorithms. Defining iterator types and methods for your containers, **whether they're generic templated containers or not**, is often a very useful thing to do.

What follows is a simplified introduction in how to define iterators for your own containers. I wish it was shorter, but C++ requires a number of little details to be taken care of, even when ignoring many important factors.

Iterators from Nested Containers

Fortunately, the most common way to define the iterator types and begin/end methods is also the simplest. If your container uses an STL container in an internal data member to hold its elements, all you need to do is delegate the types and methods to the underlying container.

For example, suppose you have a class `CourseList` that is a list of the students taking a course, and internally you use a vector of `Student` to store the course list. Then all you need to do is:

```
class CourseList {
private:
  typedef vector<Student> StudentList;
  StudentList students;
public:
  typedef StudentList::iterator iterator;
  typedef StudentList::const_iterator const_iterator;
  iterator begin() { return students.begin(); }
  iterator end() { return students.end(); }
  ...
};
```

In other words, all we do is define

- `CourseList::iterator` and `CourseList::const_iterator` to be whatever our container's iterator types are, and
- `begin()` and `end()` to return whatever our containers's `begin()` and `end()` return

This is a common programming pattern called **delegation**, because we delegate the work to another class.

The `StudentList` typedef is optional, but it makes it easy to switch containers with just one line change.

Iterators from Pointers

C pointers are legal iterators, so if your internal container is a simple C array, then all you need to do is return the pointers. Thus if we used an array for our `StudentList`, the code would look like this:

```
class CourseList {
private:
  typedef Student StudentList[100];
  StudentList students;
public:
  typedef Student * iterator;
  typedef const Student * const_iterator;
  iterator begin() { return &students[0]; }
  iterator end() { return &students[100]; }
  ...
};
```

These two cases, iterators from internal STL containers, and iterators from pointers, cover 90% of the cases you need to worry about.

Iterators from Iterators

Sometimes you want a variation on an iterator for a normal container, e.g., you want an iterator that counts or one that checks to make sure it's within a legal range.

For example, Stroustrup's *The C++ Programming Language*, 3rd ed., gives an example of a range-checking iterator. Musser and Saini's *STL Reference and Tutorial Guide* gives an example of a counting iterator.

Defining this kind of iterator uses **delegation** a lot. Basically, the constructor for the new iterator takes some kind of existing iterator, plus whatever other kind of information it needs, and saves the existing iterator in a private data member. Then `operator++()`, `operator*()` and so on are defined mostly as calls to the same operators on the stored iterator, with whatever extra functionality you need.

Unfortunately, while conceptually simple, this kind of iterator definition turns out to require a lot of special C++ incantations, because it can be hard for the compiler to know when something's a type name. If you're trying to do this, definitely get a copy of Stroustrup or some other reference with a detailed example.

Defining Iterators for New Containers: Overview

Our last case of when you want to define an iterator is when you have a new kind of container and it needs an iterator.

The STL does not use class hierarchies and inheritance. Therefore, when defining an iterator, there is no iterator superclass to start from. Something qualifies as an iterator as long as it defines some or all of the operators described on the iterators page.

For this example, we're only going to describe and define the most basic operations.

- `operator*()` -- the dereferencing operator
- `operator++()` -- the incrementing operator
- `operator!=()` -- the inequality operator, needed for all those "<TT>while (begin != end) ... </TT>" loops

For our example container, we're going to implement a ring queue.

A Ring Queue

A ring queue is a finite-sized queue that never gets full and never grows past a certain size. Instead, new elements replace the oldest elements. A ring queue lets you keep track of the most recent values in some stream of numbers, automatically throwing away older ones as you add new ones. For example, you might use a ring queue to get the average of the N most recently processed numbers.

You create a ring queue like this:

```
RingQueue< int, 5 > ringQueue;
```

This constructs an empty 5-element ring queue.

Like a regular queue, you can push on the back and pop off the front but unlike a regular queue:

- A ring queue keeps only the N most recently pushed elements.
- `push_back()` and `pop_front()` are used instead of `push()` and `pop()` to support using generic algorithms and `back_inserter`.
- `push_front()` on a full ring queue adds the new element and removes the oldest, i.e., the one at the front.
- Iterator access to the queue is supported with `begin()` and `end()`.

A common way to implement ring queues is as a simple array with two numbers:

- `myBuffer`: an array of size N to hold the queue
- `myBegin`: an integer indicating where the front of queue is
- `mySize`: an integer indicating how many elements are in the queue

Initially `myBegin` is 0, the beginning of the array. The end of the ring queue -- we'll call it `myEnd` but it's something we'll calculate as needed -- is given by the simple formula

```
myEnd = (myBegin + mySize) % N
```

We can define the following accessors:

- `front()` -- `myBuffer[ myBegin ]`, assuming `mySize` is not 0
- `back()` -- `myBuffer[ myEnd ]`, assuming `mySize` is not 0

The rules for changing `myBegin` and `mySize` (and hence `myEnd`) are simple but slightly surprising:

- `pop_front()` increments `myBegin` and decrements `mySize`.
- `push_back()` stores the value in `myBuffer[ myEnd ]` and increments `mySize` up to N; thereafter, it stores in the same place, but increments `myBegin`!
- Whenever `myBegin` reaches N, it's reset to 0. `mySize` is never incremented past N, nor decremented below 0.

The table shows what happens when we add and remove some elements to a 4 element queue.

Action	myBuffer	myBegin	mySize	myEnd

	[ ][ ][ ][ ]	0	0	0
push 1	[1][ ][ ][ ]	0	1	1
push 2	[1][2][ ][ ]	0	2	2
pop => 1	[1][2][ ][ ]	1	1	2
push 3	[1][2][3][ ]	1	2	3
push 4	[1][2][3][4]	1	3	0
push 5	[5][2][3][4]	1	4	1
push 6	[5][6][3][4]	2	4	2
pop => 3	[5][6][3][4]	3	3	2

Note that no data is ever shifted in the array. A `push_back()` when the queue is full simply replaces the oldest element with the newest. That's what makes this a ring queue.

A Ring Queue Iterator

Given our definitions of `front()` and `back()` above, you might think we could also define

- `begin()` as the address of `myBuffer[ myBegin ]`
- `end()` as the address of `myBuffer[ myEnd ]`

You can do this and it will compile, but then you try the following code:

```
RingQueue<int, 4> rq;

for ( int i = 0; i < 10 ++i ) rq.push_back( i * i );

cout << "There are " << rq.size() << " elements." << endl;
cout << "Here they are:" << endl;
copy( rq.begin(), rq.end(),
      ostream_iterator<int>( cout, "\n" ) );
cout << "Done" << endl;
```

This produces the following output:

```
There are 4 elements.
Here they are:
Done
```

What happened? Clearly the ring queue thinks it has the full 5 elements, but `copy()` doesn't seem to find them.

Look in the table above at what happens when the ring queue is full. When the queue is full, i.e., `mySize` is equal to 4, then `myBegin` is equal to `myEnd`! That's because when `mySize` is N, the following equation is true:

```
myEnd = (myBegin + mySize) % N = myBegin
```

That means that the end of the queue is equal to the beginning! Therefore, any loop that starts with

```
while ( rq.begin() != rq.end() ) ...
```

on a full ring queue will stop immediately, because it will look like the queue were empty.

Therefore, we need some other way to indicate where we are in the queue. An obvious solution is keep a simple offset from `myBegin`:

- An offset of 0 means the beginning of the queue.
- An offset of `mySize` means the end of the queue.
- An offset in between 0 and `mySize` means somewhere in the middle of the queue.

Our ring queue **iterator** therefore will need two things:

- a reference to a specific ring queue
- an offset variable

Each of the operators we need to define becomes simple, at least conceptually:

- `operator!=()` is true if two iterators have either different ring queues or different offsets.
- `operator++()` increments the offset.
- `operator*()` returns the array location indexed by `(myBegin + offset) % N`.

Conceptually, that's that. C++ being C++, there's a lot of little details that need to be taken care of, to make the compiler happy, but the basic ideas are what have just been described.

Defining Iterators for New Containers: C++ Details

C++ Detail #1: Linking Containers and Iterators

Container and iterator classes typically need to be closely linked. In particular

- In the container, `begin()` and `end()` need to construct and return iterators.
- In the iterator, there needs to be a data member with a reference to a container.

This kind of mutual referencing implies that we need to define the container before the iterator and vice versa. Furthermore the iterator typically needs access to private container methods and data members in order to do its job.

For this reason, the typical pattern for defining containers and their iterators is this:

- Forward declare the iterator class.
- Define the container.
- In the container, make the iterator a friend class.
- Define the iterator.

Therefore, we have this code

```
// forward declare the iterator

template < class T, int N > class RingIter;


// define the container, make the iterator a friend

template < class T, int N >
class RingQueue {

  friend class RingIter< T, N >;
  ...
};


// define the iterator

template < class T, int N >
class RingIter {
...
};
```

Notice that the ring queue and its iterator have the same template parameters. This is typical.

Digression: Integer Template Parameters

The `int N` template argument used above is a special feature of C++ that lets us write

```
RingQueue< int, 5 > ringQueue;
```

to make a 5-element ring queue of integers. The `N` parameter is "hard-wired" to 5 at compile-time, just like the `T` type parameter is hard-wired to `int`. This lets us declare the array in the ring queue at compile-time instead of dynamically allocating it. Of course, this also means you can't make a ring queue of some variable size N.

C++ Detail #2: Making the Container Iterator-Friendly

Some of the STL functions, e.g., `back_inserter()`, need to know a few key pieces of information about a container in order to construct iterators. This information is communicated by a standardized set of `typedef`'s at the start of the container. For example, a container should define `value_type` and `iterator` so that other functions can tell what kind of thing this container can hold and what kind of iterator it uses.

Fortunately, these type definitions are all pretty simple and usually identical from container to container. For our ring queue, we need

```
  typedef RingIter<T, N> iterator;
  typedef ptrdiff_t difference_type;
  typedef size_t size_type;
  typedef T value_type;
  typedef T * pointer;
  typedef T & reference;
```

The only thing specific to our ring queue is the iterator type definition.

C++ Detail #3: `begin()` and `end()`

The main functions in a container that return iterators are `begin()` and `end()` (we're ignoring reverse and constant iterators here).

Typically, these simply call the constructor for the iterator class. There are usually at least two arguments:

- the container itself, to give the iterator access to the container
- something else that indicates where the iterator is pointing

In our case, the something else is the offset, so our definitions are simply

```
iterator begin() { return iterator( *this, 0 ); }

iterator end() { return iterator( *this, mySize ); }
```

Notice that this is both readable and fairly safe from minor implementation changes, because we used the `iterator` typename instead of `RingIter<T, N>`.

C++ Detail #3: Storing the Container in the Iterator

Constructing the iterator is relatively straightforward except for one detail. We have to make sure that the iterator contains a *reference* to the actual container, not a copy. In C++, it's very easy to inadvertently get a copy. If the constructor was defined with

```
RingIter( RingQueue rq, int size ) { ... }
```

we'd get a copy of the ring queue because `rq` is passed by value.

We can fix this with

```
RingIter( RingQueue & rq;, int size ) { ... }
```

but suppose the whole class looks like this:

```
template < class T, int N >
class RingIter {
private:
  RingQueue<T, N> myRingQueue;
  int mySize;
public:
  RingIter( RingQueue & rq, int size )
  {
      myRingQueue = rq;
      ...
```

In this case, we will **still** get a copy because `myRingQueue = rq` will make a copy of `rq`.

Aha, we say, let's make `myRingQueue` a **reference variable**, i.e.,

```
template < class T, int N >
class RingIter {
private:
  RingQueue<T, N> & myRingQueue;
  int mySize;
public:
  RingIter( RingQueue & rq, int size )
  {
      myRingQueue = rq;
      ...
```

This won't compile. Reference variables must be initialized to the thing they reference. E.g.,

```
int x = 10, &y = x;   // fine, y references x
y = 4;                // sets x to 4

int a = 10, &b;       // illegal, b has nothing to reference
```

Fortunately, there is a solution. There are two places where variables can be initialized:

- when they're declared
- in a constructor's initialization clauses

Thus, the following code works fine:

```
template < class T, int N >
class RingIter {
private:
  RingQueue<T, N> & myRingQueue;
  int mySize;
public:
  RingIter( RingQueue & rq, int size )
   : myRingQueue( rq ), mySize ( size )
  {}
```

Initialization clauses are not assignment. They're initializations and work just like initializations in declarations.

The above code makes `myRingQueue` a reference to whatever `rq` is. Since `rq` is in turn a reference to some container, `myRingQueue` is a reference to the container. Anything we do with or to `myRingQueue` is applied directly to the original container, not a copy.

C++ Detail #4: Defining `operator!=()` for an iterator

The inequality operator is usually trivial. If your iterator class just contains pointers and/or integer indices, then the default equality operator that C++ defines will work just fine.

We also need to define

- `operator==()` to make sure we have the same container and offset
- `operator!=()` to call `operator==()` appropriately

The code for both is straightforward.

C++ Detail #5: Defining `operator*()` for an iterator

Consider this simple piece of C++ code using an iterator:

```
*dest = *source;
```

In this statement,

- `*source` returns the **value** pointed to by `source`
- `*dest` returns the **location** pointed to by `dest`

`*dest` is said to return an **l-value**. The "l" stands for "left" as in "left-hand sides of an assignment statement." An l-value is something that can be used to store a value.

What all this means is that when we define `operator*()` for an iterator class, we have to make sure that it returns a **reference** to a location. As long as we do that, the right thing will happen no matter which side of the assignment statement the dereferencing occurs.

In our code, all we need to do is return the array reference, but we have to make sure that the return type of `operator*()` is a reference.

```
T & operator*() { return myRingQueue[ myIncrement ]; }
```

We defined `operator[]` in `RingQueue` to make the above code simple.

> **Warning: watch your ampersands!** The following won't compile:
>
> `T & operator*() { return & myRingQueue[ myIncrement ]; }`
>
> The first ampersand says "return the address of the value specified by any `return` statements. The second ampersand says "return the address of `myRingQueue[ myIncrement ]`." So we end up trying to return the address of the address of `myRingQueue[ myIncrement ]`, which the compiler will correctly say is not the address of an object of type `T`.

C++ Detail #6: Defining `operator++()` for an iterator

The increment operator in C++ can be called in two ways:

- prefix, e.g., `++iter`
- postfix, e.g., `iter++`

The prefix case is simple. We just increment the iterator's internal data members appropriately and return the modified iterator. In our case, that means

```
iterator & operator++() { ++myOffset; return *this; }
```

The `return *this;` is standard in iterator definitions.

The postfix case is a little more confusing, for two reasons:

- We have to know when we have the postfix case and not the prefix case.
- The postfix case has to modify the iterator, but return what it was before it changed.

C++ has a kludge for the first problem. When `operator++()` is used postfix, the compiler generates a call with a dummy integer. This integer has no useful value. It's simply there so that you can define `operator++( int )` to handle the postfix case, and keep it separate from the code for the prefix case.

To handle the second problem, we need to create a copy of the iterator and return it.

```
iterator operator++ ( int )
{
  RingIter<T, N> clone( *this );
  ++myOffset;
  return clone;
}
```

> **Warning:** Hackers will notice that two copies actually get created: one when `clone` is built, and another when `clone` is returned by value. Do **not** try to fix this by returning a reference to the `clone`! This will crash your code, because it will return the address of an object that doesn't exist after this code exits.

C++ Details: What's Missing?

For our purposes in this class, the above should suffice for defining iterators for new containers. There are however many other details, including:

- **`const` iterators:** We really need to define two iterator classes, one as above and another, call it `RingCIter`, that is almost identical but wherever `RingIter` returns a reference to the container, `RingCIter` needs to return a `const` reference. We also need to change begin() and end() in RingQueue to return the right iterators, e.g.,

  ```
  typedef RingIter<T, N> iterator;

  typedef RingCIter<T, N> const_iterator;

  ...

  iterator begin() { return iterator( *this, 0 ); }

  iterator begin() const { return const_iterator( *this, 0 ); }
  ```

- **Reverse iterators:** We should defined `rbegin()` and `rend()` to reverse version of the iterators that go backward. Again, we need both regular and `const` versions.
- **Iterator traits:** Some of the more complex STL algorithms and adapters figure out what to do with containers and iterators by looking at a data structure that holds an iterator's **traits**. Traits say things like "this iterator is bidirectional, points to integers, ..." Such a data structure is created as a data member of the iterator.

For this and other issues in defining iterators, see an advanced book on C++ programming, such as Stroustrup's *The C++ Programming Language*, 3rd ed.

Comments? Let me know!