CS106L Handout #11
Spring 2009 April 23, 2009

STL Iterators, Part I

Introduction

The STL comprises several different container types, from the linear `vector` to the associative `map`. To provide a unified means for accessing and modifying elements in different container types, the STL uses *iterators*, objects that mimic pointers. While the benefits of iterators might not be apparent right now, when we cover algorithms next week you will see exactly how much power and flexibility iterators provide. This handout covers iterator basics and is primarily designed to get you up to speed on iterator syntax.

A Word on Safety, Syntax, and Readability

So far, in CS106 the only iterators you've been exposed to so far are the CS106 ADT iterators, like `Map<ElemType>::Iterator`. These iterators, like Java's `Iterator<T>`, support functions called `next` and `hasNext` to return values and identify whether they may safely advance. STL iterators, unfortunately, have nothing in common with this interface.

STL iterator syntax is complicated because it is designed to look like pointer arithmetic, which, as you've seen, is not pretty. As with pointers, STL iterators do not perform any bounds checking and consequently can lead to difficult bugs from "one past the end" operations. However, as with every other part of the STL, these frustrating features lead to incredible performance boosts. Iterators are optimized for speed, and commonly you can increase the performance of your code by switching to iterator loops over traditional for loops.

A Word on Templates

All STL iterators have a common interface – that is, an iterator for a `deque<string>` has the same syntax as an iterator for a `set<int>`. This commonality paves the way for STL algorithms, which we'll cover next week. Today we'll begin to cover how interchangeable STL iterators are, and I'll go over some of the cool things you can do with them.

However, because iterators are so commonly used in template functions, you are likely to encounter some very nasty compiler messages if you make syntax errors with iterators. A single mistake can cause a cascade of catastrophic compiler complaints that can really ruin your day. If you encounter these messages, be prepared to sift through them for a while before you find the root of your problem.

Now, without further ado, let's start talking about iterators!

Basic STL Iterators

Rather than jumping headfirst in to full-fledged iterator syntax, we'll start off at the basics and take smaller steps until we end up with the idiomatic STL iterator loop.

All STL container classes, except for `stack` and `queue`, define the `iterator` type.* For example, to declare an iterator for a `vector<int>`, we'd use the syntax

```
vector<int>::iterator myItr; // Correct, but iterator is uninitialized.
```

Note that unlike the CS106 libraries, the word `iterator` is in lowercase. Also, keep in mind that this iterator can only iterate over a `vector<int>`; if we wanted to iterate over a `vector<string>` or `deque<int>`, we'd have to declare iterators of the type `vector<string>::iterator` and `deque<int>::iterator`, respectively.

STL iterators are designed to work like C and C++ pointers. Thus, to get the value of the element pointed at by an iterator, you "dereference" it using the `*` operator. Unlike the CS106 iterator types, STL iterators can modify the contents of the container they are iterating over. Here is some code using iterators to read and write values:

```
// Assumes myItr is of type vector<int>::iterator
int value = *myItr; // Read the element pointed at by the iterator.
*myItr = 137;       // Set the element pointed to by the iterator to 137.
```

When working with containers of objects, `class`es, or `struct`s, you can use the `->` operator with iterators as you would with a standard pointer. For example, here's some code to print out the length of a string in a `vector<string>`:

```
// Assumes myItr is of type vector<string>::iterator
cout << myItr->length() << endl; // Print length of the current string.
```

As with pointers, forgetting to initialize an iterator leads to undefined behavior. Thus before you work with an iterator, you must initialize it to point to the elements of a container. Commonly you'll use the STL containers' `begin` member functions, which returns an iterator to the first element of the container. For example:

```
vector<int>::iterator itr = myVector.begin(); // Initialize the iterator
*itr = 137; // myVector[0] is now set to 137.
```

Note that unlike CS106 iterators' `next` function, getting the value of an STL iterator using `*` does not advance it to the next item in the container. To move forward with an STL iterator, use the ++ operator, as shown below.

```
vector<int>::iterator itr = myVector.begin();
*itr = 137; // myVector[0] is now 137.
++ itr;     // Advance to next element.
*itr = 42;  // myVector[1] is now 42.
```

You might be wondering why I wrote `++itr` and not `itr++`. For somewhat technical reasons involving operator overloading (which we'll cover in a few weeks), while both `++itr` and `itr++` will work correctly, the first version is much faster than the second. Thus when using STL iterators, remember to use the prefix version of ++ to iterate over a container.

* As mentioned in the STL Containers Part I handout, `stack` and `queue` are not technically containers (they're *container adapters*) and therefore do not have `iterator`s.

There's one last thing to consider right now, and it has to do with bounds checking. With CS106 iterators, you can easily tell when you've hit the end of a container by checking to see if `hasNext` returns `false`. However, STL iterators don't have a `hasNext` function or even rudimentary functionality like it. With STL iterators you need *two* iterators to define a range – one for the beginning and one for the end. While this makes the syntax a bit trickier, it makes STL iterators more flexible than CS106 iterators because it's possible to iterate over an arbitrary range instead of the entire container.

Each STL container class defines an `end` function that returns an iterator to the element *one past the end* of the container. Put another way, `end` returns the first iterator that is not in the container. While this seems confusing, it's actually quite useful because it lets you use iterators to define ranges of the type [start, stop).

To best see how to use the `end` function, consider the following code snippet, which is the idiomatic "loop over a container" for loop:

```
for(vector<int>::iterator itr = myVector.begin();itr != myVector.end(); ++itr)
    cout << *itr << endl;
```

Here, we simply crawl over the container by stepping forward one element at a time until the iterator reaches `end`. Remember that when using iterators in loops, you'll want to use the `!=` operator for the looping condition instead of the more familiar <.

If you'll remember from our discussion of the `vector` container class, the syntax to remove single elements was `myVector.erase(myVector.begin() + n)`, where `n` represented the index of the element to remove. The reason this code works correctly is because you can add integer values to `vector` iterators as though they were pointers. Thus to get an iterator that points to the nth element of the `vector`, we simply get an iterator to the beginning of the `vector` with `begin` and add `n` to it. This same trick works with a `deque`.

Iterator Generality

As I mentioned last lecture, the `deque` class is capable of performing any task that a `vector` can do. This includes iteration. Here's some basic code to print out all elements of a `deque`:

```
for(deque<int>::iterator itr = myDeque.begin(); itr != myDeque.end(); ++ itr)
    cout << *itr << endl;
```

Recall, however, that `deque`s and `vector`s are implemented in completely different ways – the `vector` with a single contiguous array, the `deque` with many chained arrays. But despite these differences, the iterator loops to crawl over them have exactly the same structure. Somehow each iterator "knows" how to get from one element to the next, even if those elements aren't stored in consecutive memory locations. This is the real beauty of STL iterators – no matter how the data is stored, the iterators will access them correctly and with minimal syntax changes on your end. This is the magic of *operator overloading*, a technique we'll return to later in the quarter.

Using Iterators to Define Ranges

The STL relies heavily on iterators to define ranges within containers. Currently, all the examples we've seen so far have used the range [`begin`, `end`) for iteration, but it is possible (and also quite useful) to narrow that range in a variety of ways.

For example, suppose you want to write a loop that will print the first ten values of a `deque` to a file. Using iterators, this can be accomplished quite simply:

```
for(deque<int>::iterator itr = myDeque.begin();
    itr != myDeque().begin + 10;  // Ten steps down from the beginning.
    ++itr)
    myStream << *itr << endl;
```

Similarly, most containers support several functions that let you access, manipulate, or add a range of data. For example, there is a second form of the common function `insert` that inserts values from the specified range of iterators into a container. Here's an example:

```
// Create a vector, fill it in with the first NUM_INTS integers.
vector<int> myVector;
for(vector<int>::size_type h = 0; h < NUM_INTS; h++)
    myVector.push_back(h);

// Copy the first five elements of the vector into the deque
deque<int> myDeque;
myDeque.insert(myDeque.begin(), myVector.begin(), myVector.begin() + 5);
```

Interestingly, even though the `vector`'s iterators are of type `vector<int>::iterator` and not `deque<int>::iterator`, the code will compile. This is your first real glimpse of the magic of the STL: the fact that all iterators have the same interface means that the `deque` can accept iterators from any container, not just other `deque`s.

What's even more interesting is that you can specify a range of iterators as arguments to the constructors of most containers. When the new container is constructed, it will contain all of the elements specified by the range. This next code snippet fills in a `set` with the contents of a `vector`. Of course, since it's a `set`, it will automatically sort the entries and remove duplicates:

```
vector<int> myVector;
// Vector contents get filled in, etc, and then:
set<int> mySet(myVector.begin(), myVector.end());
```

This is really where the "template" in "Standard Template Library" begins to show its strength, and next week when we cover algorithms you'll see exactly how powerful the common iterator interface can be.

Iterating Backwards

At some point you might want to traverse the elements of a container backwards. All STL containers define a type called `reverse_iterator` that represents an iterator that responds to the normal `++` operator backwards. For example, the statement `++myReverseItr` would result in `myReverseItr` pointing to the element that came *before* the previous one. Similarly, containers have functions `rbegin` and `rend` that act as reversed versions of the traditional `begin` and `end`. Note, however, that `rbegin` does *not* point one past the end as does `end` – instead it points to the very last element in the container. Similarly, `rend` points one position *before* the first element, not to the same element as `begin`.

Here's some code showing off reverse iterators:

```
// Print a vector backwards
for(vector<int>::reverse_iterator itr = myVector.rbegin();
    itr != myVector.rend(); ++itr)
    cout << *itr << endl;
```

Arrays and Iterators

As I mentioned at the start of this handout, STL iterators are modeled after array and pointer syntax. Through the magic of templates, it turns out that you can actually use standard pointers in places where STL iterators would normally be expected. This means that if you have a regular C++ array, you can treat it as though it were a sort of "container."

For example, the next section constructs a `vector` whose contents are equal to the numbers in a regular C++ array:

```
int myArray[] = {0, 1, 2, 3, 4};
vector<int> myVector(myArray, myArray + 5);
```

When we cover STL algorithms next week this same trick will apply, so you can use STL algorithms to rapidly manipulate arrays as you would containers.

Practice Problems

1. Given that arrays and pointers can act as STL iterators, rewrite the idiomatic iterator for loop `for(iterator itr = container.begin(); itr != container.end(); ++itr)` using arrays and pointers. That is, given an array, write a for loop that uses pointers to iterate over it.
2. Some STL containers contain member functions to return iterators to specific elements. As a sentinel value, these functions return the value of the container's `end()` function. Why do you think this is?
3. Write a function `DuplicateReversed` that accepts a `vector<int>` and returns a new `vector<int>` with the same values as the original `vector` but in reverse order.