STL Iterators, Part II

Introduction

Last week's lecture introduced STL iterators and demonstrated how they both manipulate elements and define ranges. In preparation for STL algorithms, there are a few more iterator concepts we need to cover. This handout talks about the different types of STL iterators and introduces *iterator adapters*, objects that look like iterators perform other tasks.

Iterator Categories

If you'll recall from the discussion of the `vector` and `deque insert` functions, to specify an iterator to the nth element of a container, you use the syntax `myContainer.begin() + n`. While this works on a `vector` and `deque`, when working with other containers like `map` and `set` this notation will cause a compiler error because `map` and `set` iterators can't respond to the + operator. Intuitively, this makes sense, since `map` and `set` don't store elements sequentially, but when you realize that iterators are modeled after pointers, you'll notice that in this case standard pointers are more powerful than `map` and `set` iterators.

The STL categorizes its iterators based on their relative power. At the high-end are *random-access iterators* which can perform all of the functions of regular pointers, and at the bottom are the *input* and *output* iterators which guarantee only a minimum of functionality. While it might seem strange to create deliberately weakened iterators, as you'll see later in this handout, the restrictions actually make it much easier to write iterator-compatible code and pave the way for iterator adapters.

There are five different types of iterators, each of which is discussed in short detail below.[*]

Output Iterators

Of the five iterator types, output iterators are the weakest. With an output iterator, you can write values using the syntax `*myItr = value` (but not `value = *myItr`) and you can advance the iterator forward one step using the ++ operator. Output iterators are referenced frequently by the STL algorithms because they require the minimum functionality necessary to write values.

Input Iterators

Input iterators are the second-simplest type of iterator and represent iterators that can read from a source but not write to it. That is, you can write code along the lines of `value = *myItr`, but not `*myItr = value`. Input iterators can reread the same value multiple times. As you'll see in the next handout, many STL algorithms use input iterators because, like output iterators, they require little functionality but are powerful enough to read from a location.

---

[*]   Technically there's a sixth iterator type called a "trivial iterator" that just knows how to dereference to a value, but it's rare to actually encounter them in practice.

Forward Iterators

Forward iterators combine the functionality of input and output iterators so that most intuitive operations are well-defined. With a forward iterator, you can write both `*myItr = value` and `value = *myItr`. Forward iterators, as their name suggests, can only move forward. Thus `++myItr` is legal, but `--myItr` is not.

Bidirectional Iterators

Bidirectional iterators are some of the most common iterator types and are the ones employed by `map` and `set`. Bidirectional iterators can do anything forward iterators can and also support backwards movement with the decrement operator. Thus it's possible to write `--myItr` to go back to the last element you visited, or even to traverse a list in reverse order. However, bidirectional iterators cannot respond to the + or += operators.

Random-Access Iterators

Don't get tripped up by the name – random-access iterators don't move around randomly. Random-access iterators get their name from their ability to move forward and backward by arbitrary amounts at any point. These are the iterators employed by `vector` and `deque` and can do anything that a standard C++ pointer can, including pointer-from-pointer subtraction, bracket syntax, and incrementation with + and +=.

If you'll notice, each class of iterators is progressively more powerful than the previous one – that is, the iterators form a functionality hierarchy. This means that when a library function requires a certain class of iterator, you can provide it any iterator that's at least as powerful. For example, if a function requires a forward iterator, you can provide either a forward, bidirectional, or random-access iterator.

Iterator Adapters

Iterator adapters are some of the coolest features of the STL. Iterator adapters are objects that look like iterators – you can dereference and increment them as you would a regular iterator – but that perform special operations behind the scenes.

For example, consider the following code which uses an `ostream_iterator` to print values to `cout`:

```
ostream_iterator<int> myItr(cout, " ");
*myItr = 137; // Prints 137 to cout!
++myItr;
*myItr = 42;  // Prints 42 to cout!
++myItr
```

If you compile and run this code, you will notice that the numbers 137 and 42 get written to the console, separated by spaces. Although it *looks* like you're manipulating the contents of a container or writing to a memory location, what you're actually doing is writing bits and bytes to the `cout` stream.

The syntax for `ostream_iterator`s is not particularly complicated. `ostream_iterator` is a template type that requires you to specify what type of element you'd like to write to the stream. In

the constructor, you must also specify what `ostream` to use, which in the above example is `cout` but in practice can be any type of stream, including `ofstream`s and `stringstream`s. The final argument to the constructor specifies an optional string to print out between elements. You may omit this if you want the contents to run together, which is useful when printing `char`s.

Another useful iterator adapter is the `back_insert_iterator`. With `back_insert_iterator`, you can append elements to a container using iterator syntax. For example, the following code creates a `vector<int>` and uses a `back_insert_iterator` to fill it in:

```
vector<int> myVector; // Empty
back_insert_iterator<vector<int> > itr; // Template type is vector<int>

for(int i = 0; i < 10; i++)
{
    *itr = i;
     ++itr;
}

for(vector<int>::iterator itr = myVector.begin();
    itr != myVector.end(); ++itr)
    cout << *itr << endl; // Prints numbers zero through nine
```

Although we never explicitly added any elements to the `vector`, through the magic of iterator adapters we were able to load the `vector` with data.

The syntax `back_insert_iterator<vector<int> >` is a bit clunky, and in most cases when you're using `back_insert_iterator`s you'll only need to create a temporary object. For example, when using STL algorithms, you'll most likely want to create a `back_insert_iterator` only in the context of an algorithm. To do this, you can use the `back_inserter` function, which takes in a container and returns an initialized `back_insert_iterator` for use on that object.

Internally, `back_insert_iterator` calls `push_back` whenever it's dereferenced, so you can't use `back_insert_iterator`s to insert elements into containers that don't have a `push_back` member function.

All of these examples are interesting, but why would you ever want to use an iterator adapter? After all, you can just write values directly to `cout` instead of using an `ostream_iterator`, and you can always call `push_back` to insert elements into containers. But iterator adapters have the advantage that they are iterators – that is, if a function expects an iterator, you can pass in an iterator adapter instead of a regular iterator.

Suppose, for example, that you want to use an STL algorithm to perform a computation and print the result directly to `cout`. Unfortunately, STL algorithms aren't designed to write values to `cout` – they're written to store results in ranges defined by iterators. But by using an iterator adapter, you can trick the algorithm into "thinking" it's storing values but in reality is printing them to `cout`. Iterator adapters thus let you customize the behavior of STL algorithms by changing the way that they read and write data.

The following table lists the standard iterator adapters and provides some useful context. You'll likely refer to this table most when writing code that uses algorithms.

`back_insert_iterator<Container>`	`back_insert_iterator<vector<int> > itr(myVector);`
	 `back_insert_iterator<deque<char> > itr = back_inserter(myDeque);`
	
	 An output iterator that stores elements by calling `push_back` on the specified container. You can declare `back_insert_iterator`s explicitly, or can create them with the function `back_inserter`.

`front_insert_iterator<Container>`	`front_insert_iterator<deque<int> > itr(myIntDeque);`
	 `front_insert_iterator<deque<char> > itr = front_inserter(myDeque);`
	
	 An output iterator that stores elements by calling `push_front` on the specified container. Note that this means you cannot use a `front_insert_iterator` with a `vector`. As with `front_insert_iterator`, you can create `front_insert_iterator`s with the the `front_inserter` function.

`insert_iterator<Container>`	`insert_iterator<int> itr(mySet, mySet.begin());`
	 `insert_iterator<int> itr = inserter(mySet, mySet.begin());`
	
	 An output iterator that stores its elements by calling `insert` on the specified container to insert elements at the indicated position. You can use this iterator type to insert into any container, especially `set`. The special function `inserter` generates `insert_iterator`s for you.

`ostream_iterator<type>`	`ostream_iterator<int> itr(cout, " ");`
	 `ostream_iterator<char> itr(cout);`
	 `ostream_iterator<double> itr(myOFStream, "\n");`
	
	 An output iterator that writes elements into an output stream. In the constructor, you must initialize the `ostream_iterator` to point to an `ostream`, and can optionally provide a separator string written after every element.

`istream_iterator<type>`	`istream_iterator<int> itr(cin);  // Reads from cin`
	 `istream_iterator<int> endItr; // Special end value`
	
	 An input iterator that reads values from the specified `istream` when dereferenced. When `istream_iterator`s reach the end of their streams (for example, when reading from a file), they take on a special "end" value that you can get by creating an `istream_iterator` with no parameters. `istream_iterator`s are susceptible to stream failures and should be used with care.

`string` Iterators

The C++ `string` class exports its own `iterator` type and consequently is a container just like the `vector` or `map`. Like the `vector` and `deque`, `string` iterators are random-access iterators, so you can write expressions like `myString.begin() + n` to get an iterator to the `nth` element of a `string`.

Most of the `string` member functions described in the C++ `string` handout that require a start position and a length can also accept two iterators to define a range. So, for example, to replace characters three through five in a `string` with the string "STL," you can write

```
myString.replace(myString.begin() + 3, myString.begin() + 5, "STL");
```

The `string` class also has several member functions similar to those of the `vector`, so be sure to consult a reference for more information.

More to Explore

The two iterator handouts cover most of the practical iterator functions and scenarios you're likely to encounter in practice. While there are many other interesting iterator topics, most of them concern implementation techniques and are far beyond the scope of this class. However, there are a few topics that might be worth looking into, some of which I've listed here:

1. **`advance` and `distance`**: Because not all iterators support the += operator, the STL includes a nifty function called `advance` that efficiently advances any iterator by the specified distance. Similarly, there is a function called `distance` that mimics pointer subtraction for types that do not support it.
2. **The Boost Iterators**: The Boost C++ Libraries have many iterator adapters that perform a wide variety of tasks. For example, the `filter_iterator` type iterates over containers but skips over elements that don't match a certain predicate function. Also, the `transform_iterator` reads and writes elements only after first applying a transformation to them. If you're interesting in supercharging your code with iterator adapters, definitely look into the Boost libraries.

Practice Problems

1. Write a function `NumUniqueCharacters` that accepts a `string` and returns the number of unique characters in that `string`. *(Hint: Use iterators and a `set`)*.
2. Using iterator adapters, write a function `LoadAllTokens` that, given a filename, returns a `set` consisting of all of the tokens in that file. For our purposes, define a token to be a series of characters separated by any form of whitespace. While you can do this with a standard `ifstream` and a while loop, try to use `istream_iterator`s instead.