**C++ Activities and Lecture Notes**

**Chapter 20 – The Standard Template Library**

We’ve already met templates: a means of overloading our functions to work with any data type so long as that type supports the operations in the function. The Standard Template Library is a library of container classes and associated functions that use generic algorithms and templates to allow these containers and functions to be used with almost any data type. As a result the containers themselves are quite simple, mainly providing iterators, and have no knowledge of the algorithms used on the data. The algorithms, on the other hand, don’t manipulate the containers directly but get their data through the iterators. Since templates are implemented at compile time, using the STL can produce fast, efficient run-time code as the compiler does all the work of writing the code specific to the data type used.

**Iterators**

We access data stored in containers through iterators. Iterators are simply a means of cycling through the container and giving us a reference to one data item at a time. Iterators can be thought of as doing the same job as pointers; in many containers, they *are* pointers.

For example, say we have defined a vector of ints called myVector. Consider the following code:

vector<int>::iterator first = myVector.begin();

Now we have an iterator to the vector and can use it as we please. For example,

cout << \*first << endl;

displays the first item in the vector. We could also write

vector<int>iterator last = myVector.end();

and loop through the vector like this:

for(vector<int>iterator i = first; i != last; ++i){

// do whatever to each member of the vector

}

Note that iterators, like pointers, don’t have any kind of bounds checking, so make sure your iterator points to legitimate data before using it.

[ code example: using an iterator to view a vector of strings]

**Pre-increment or post-increment?**

For the primitive data types, stand-alone ++i and i++ are identical in performance and outcome. However, with iterators it’s a different story…

iterator operator++(int) // post increment

{

iterator \_Tmp = \*this; // make a copy

++\*this; // pre increment the argument

return (\_Tmp); // return the copy

}  
  
iterator& operator++() // pre increment

{

\_Ptr = \_Acc::\_Next(\_Ptr); // increment

return (\*this); // return

}

Since iterators are objects, post-increment does more work than pre-increment.

Therefore, when looping with an iterator, you should use pre-increment, not post-increment.

However, note that compiler optimizers are smart and will throw away the extra line in the post increment code if the copy isn’t actually used. The same is true for ints in a for loop.

Programming habit: pre-increment *unless* you have a specific reason to post-increment.

**Fundamental Containers**

There are three categories of fundamental containers in the STL.

Category One: the sequential containers vector, list, and deque.

Category Two: the adapters stack, queue, and priority queue.

Category Three: the associative containers such as set and map.

All three share many of the same operations. For example, every STL container will return an iterator with begin or end. For example, consider this code:

XXXX<int> myContainer;

XXXX<int>::iterator current = myContainer.begin();

while(current != myContainer.end()){

if (\*current%2 == 0){

current = myContainer.erase(current);

}

else(

++current;

}

}

What does this code do, and what container types can be written in place of XXXX?

**STL Containers**

**Vector**

The vector is an indexed container; the most fundamental operation is subscripting with []. Elements may be added to the end of the vector in constant time, but adding to the middle of the vector is (*n*). Both size and capacity are known at all times. Iterators to a vector can be thought of as pointers into the buffer.

**List**

The principle advantage of list (which is implemented as a linked list) is that it supports constant-time insertion into the middle of the list. List also supports operations that the other container classes do not, such as merge and sort.

**Deque**

A deque is a double-ended queue, allowing for insertion or removal at either the front or the back. If you need random access to the middle of the list and also add and remove at both ends, use a deque; if you add and remove only at one end, use a vector. If you don’t care about random access but only add or remove at the ends, a deque is more efficient than a list.

**STL Container Adapters**

The adapter containers **stack** and **queue** are simply built on top of one of the fundamental containers. For example, writing

stack<int> myStack

is the same thing as

stack<int deque<int>> myStack

The default for both stack and queue is deque, but you can specify the underlying container:

stack<int, vector<int>> myStack

Priority queue defaults to vector to build a heap, so

priority\_queue<string> myPQ

is the same as

priority\_queue<string, vector<string>> myPQ

**Associative Containers**

The associative containers do not keep elements in the order in which they were inserted, but arrange them for fast insertion, search, and retrieval. The four associative containers are **set**, **multiset**, **map**, and **multimap**.

A set is an associative container that does not allow duplicates; if you attempt to insert a duplicate item into a set or multiset, it is ignored. A multiset allows duplicate keys. Searching is fast – O( *lg n*) – but is achieved by maintaining elements in order, so the elements must be comparable.

[ add a set of strings to our code demo. when we iterate through the set it will be sorted ]

A map is an indexed data structure that stores pairs of (key, value). Unlike the others, the index can be of any comparable data type – string, ComplexNumber, Person, whatever. In a map the keys must be unique; a multimap allows duplicate keys.

[ example: read a flat text and display the words and word counts ]

**Generic Algorithms**

What does this code do?

template<typename Iterator, typename Action>

void doSomething(Iterator current,

Iterator stop,

Action action){

while(current != stop){

action(\*current);

++current;

}

}

This is a *generic algorithm*; the outcome depends on the data type and the action specified. There are several generic algorithms in the STL. A few examples…

fill(a.begin(), a.end(), 0);

will take the object a (the type is irrelevant, as long as it is something with an iterator) and fill it with zeros.

generate(a.begin(), a.end(), RandomInt);

will fill the object with pseudorandom integers as defined by the function RandomInt.

Note that the generating function must have an empty parameter list (the text is wrong here) or be a reference to a “functor,” which is a function that overloads the () operation. We’ll keep it simple…

[ add this one to our first demo program ]

We can even sort…

sort(a.begin(), a.end());

[ add this one as well ]