CS165 Lab 12

Jason Dorweiler

November 23, 2013

1 STL containers

Vector
The vector container is an easy replacement for arrays. The memory is handled for you so no need to declare the size of a vector. The one downside of a vector is that when it is declared the compiler sections off a spot of memory larger then it needs. This is to allow for the vector to grow but also takes up more time and memory then it might need to.

```
// Some ways to use a vector
//Declare a vector
std::vector<type> vectorName
//Add to a vector
std::vectorName.push_back(value)
//Delete the last element
std::vectorName.pop_back()
//Access a specific element
std::vectorName.at or std::vectorName[]
```

Listing 1: Using Vectors

List
Lists are similar to arrays except that the memory for the list can be scattered around. Adding and removing elements of the list can be faster than an array or vector. The one big downside is that elements of the list can only be accessed sequentially with no way to directly access elements of the list.
Some interesting things I found with lists are they way they are assigned. The assignment of a list can be an array or even a section of an array. Another neat use of lists is the way they can be swapped. Just call list.swap(otherlist). I made a short program showing these two uses of lists.
see lists.cpp

Queue
A queue is similar to a list except that when you put something in the list you only get the oldest item in that list back (FIFO). Pretty simple so not much to say about these. I wish I had known about these a long time ago. A while ago I ended up writing a FIFO for a robotics project to store acceleration data from a very sensitive acceleration sensor. I sent the data into a FIFO and took the average of the queue as an easy way to do a low-pass filter to cut down some of the noise. It probably would have been a lot easier with stl::queue.

Deque
Similar to a queue except that these are double ended so that elements can be added and removed from both the front and back.
It seems to me that a deque is very similar to a vector and I'm actually trying to figure out why anyone would use a deque over a vector. They both have the same ways to add and access elements.
After a bit more investigation I found out that they are on the out side essentially the same but they way they work internally is very different. While a vector needs a continuous block of memory a deque can be scattered around. This will make a deque much easier in terms of time to shrink and grow.

2 Algorithm

Algorithm has lots of useful looking things in it. I'll go over some that I saw and bring them together in the end in a single test program.

Set_intersection
This does just what it sounds like. It takes two sets and returns the elements in the intersection. This takes two pre-sorted arrays and returns an iterator. And example use is:

```
     itter = set_intersection(array1, array1+lengtharray1, array2,
        lengtharray2);
```

Listing 2: Using set_intersection

Sort
Sort the elements in an array or vector. There is a separate STL::list::sort for doing this with a list. Sort has a pretty neat function other than just sorting the elements it can also call a function sort based other parameters that you define.

```
// sort a vector
 std::sort (myvector.begin(), myvector.end());

// sort and using another function
 std::sort (myvector.begin(), myvector.end(), myfunction);
```

Listing 3: Using sort

I brought the sort and set_intersection together in a single program below. It works by first declaring two arrays of integers, and integer vector, and iterator. Then the two arrays are sorted using Sort.
After sorting the two arrays they are sent to the set_intersection function which returns an iterator. After that can print out the elements of the intersection in the vector including the zeros at the end.
In order to get rid of the zeros I use the vector::shrink function. Sending the vector resize the two iterators, it and v.begin(), and subtracting which gave me address 0x215f01c - 0x215f010 = 3. This tells the resize function to shrink the vector to 3 elements.
see sets.cpp

Which gives the output:

```
Elements in the intersection stored in the vector
2 7 7 0 0 0 0 0 0 0
0x1bb701c - 0x1bb7010 = 3
The intersection has 3 elements:
 2 7 7
```

Listing 4: Output

3 Boost

Array
The boost array container looks like it will also be very useful. From the description on the website it sounds like the goal is for it to be similar to a std::vector. One of the big differences is that you can control the size of the boost array. This also gives the ability to work with a c-style array with out all the pain of one. The use of a boost array follow that of a std vector.

```
//declare an array
boost::array<int, 2> array = {{1,2}};
// get an element from the array
array[1] or array.at(1)
// size of the array
array.size()
```

Listing 5: boost array

circular buffer
This one just sounded interesting. I had to do a bit of digging for what it might be used for. Some examples I ran into are: a memory restricted error log and as a container for UART data in microcontrollers (μC).
The use of these for serial data is pretty interesting. I've never really though about how a μC handles the serial data it gets but a circular buffer makes sense.

As the data comes in it will go into the buffer. When the buffer is full the μC will start processing the data and if data is coming in faster than it can process the new data will just overwrite the old data in the buffer.

I think this is will be useful in any situation where you might want to store data but are only really concerned with the most recent data. So in the case of a μC you could process some of the serial data but if that processing takes too long you want to get the next newest data versus processing old data. This is kind of a way to prevent a buffer from filling up with data.

I wrote a simple program to show how they work.
see buffer.cpp

Which gives the following output:

```
Initial Buffer State:
1
2
3
4

Removing an element from the front:
2
3
4

Adding new data:
2
3
4
7

Fill buffer with new data:
8
8
8
8
```

Listing 6: circular buffer output

foreach

This one is great! One thing that bugged me about c++ is how much work it is to iterate over a sequence. In other languages like Python for example it is easy to just say for(element in array) which will iterate through each element. Boost::foreach gives basically this same functionality. Here is an example of how to use it.

```
#include <boost/foreach.hpp>
#include <iostream>

int main(int argc, char **argv)
{

    int array[5] = {1,2,3,4,5};

    int element;

    BOOST_FOREACH(element, array){
        std::cout << element << std::endl;
    }
}
```

Listing 7: foreach