STL ALGORITHMS

Today

- Today we look more at the *Standard Template Library*.
- First we will look at how the STL implements stacks, queues and priority queues.
- Then we'll look at some algorithms that the STL provides for use with its templates.
- Good references continue to be:
  - `http://www.learncpp.com/cpp-tutorial`
  - `http://www.cplusplus.com/doc/tutorial`
  - `http://www.cppreference.com/index.html`
- This material isn't covered in the textbook.

Container adaptors

- Container adaptors implement other commonly used kind of datastructure:
  - `stack`,
  - `queue` and
  - `priority_queue`

  are containers that are adapted from sequence containers (`vector`, `list` and `deque`)
- They define how elements are added and removed
- They canot be accessed using iterators.

Stack

- A stack is a "LIFO" data structure: "last in, first out"
- Which means that items are added to the front of the stack and also removed from the front of the stack.
- We have talked about stacks in the past this semester and used the analogy of a stack of plates in a cafeteria: new plates are added to the top; plates are also removed from the top
- The STL stack has the following members:

```
constructor
empty()
pop()
push()
size()
top()
```

- Here is an example of using a stack from `adaptor.cpp`.

```
stack<int> myStack;

for (int i=0; i<10; i++) {
  r = rand() % 100;
  myStack.push(r);
}

while(!myStack.empty()) {
  cout << myStack.top() << " ";
  myStack.pop();
}
cout << endl;
```

Queue

- A queue is a "FIFO" data structure: "first in, first out"
- Which means that items are added to the back of the queue and are removed from the front of the queue
- A queue is just like a conventional line (of humans) (also called a "queue" if you live in the UK)
- Has the following members:

```
constructor
back()
empty()
front()
pop()
push()
size()
```

- Here is an example of using a queue from `adaptor.cpp`.

```
queue<int> myQueue;

for (int i=0; i<10; i++) {
  r = rand() % 100;
  myQueue.push(r);
}

while(!myQueue.empty()) {
  cout << myQueue.front() << " ";
  myQueue.pop();
}
cout << endl;
```

Priority Queue

- Like a queue, except that the items are ordered according to a comparison operator that is specified when a priority queue object is instantiated.
- So elements are inserted in order.
- Has the following members:

```
constructor
empty()
pop()
push()
size()
top()
```

- Here is an example of using a priority queue from `adaptor.cpp`.

```
priority_queue<int> myPriority;

for (int i=0; i<10; i++) {
  r = rand() % 100;
  myPriority.push(r);
}

while(!myPriority.empty()) {
    cout << myPriority.top() << " ";
    myPriority.pop();
}
cout << endl;
```

- If we push the same set of random numbers into the three containers, we will get something like:

```
21 49 92 86 35 93 15 77 86 83
83 86 77 15 93 35 86 92 49 21
93 92 86 86 83 77 49 35 21 15
```

- So the queue and the stack output the numbers in the opposite order and the priority queue has automatically sorted them.

STL algorithms

- The STL also contains some useful algorithms that can be used on containers.
- The require the containers to be accessible by iterators, so they won't work on `stack`, `queue`, or `priority_queue`.
- You can find all these examples in the file `algorithms.cpp`.

- There is a set of algorithms that just look stuff up in the container.
- For example `count` and `find`

```
cout << "The number 86 appears: "
     << count(theVector.begin(), theVector.end(), 86)
     <<  " times" << endl;

cout << "The number following the "
     << "first occurance of 86 is: ";
p = find(theVector.begin(), theVector.end(), 86);
p++;
cout << *p << endl;
```

- As you can see, count counts the number of instances of its third argument, and `find` returns an iterator with a location.

- With the same vector as above:

  ```
  21 49 92 86 35 93 15 77 86 83
  ```

  this will produce:

  ```
  The number 86 appears: 2 times
  The number following the first
            occurance of 86 is: 77
  ```

- (Note that the spacing was altered so that the text would fit on the page).

- We can use any range, indicated by iterators, within a vector, as input to the algorithms.
- For example:

```
q = p;
s = theVector.end() - 1;

cout << "The number 86 appears "
     << count(q, s, 86)
     <<  " times in the subsequence" << endl;

cout << "The number following the first occurance"
     << " of 86 in the subsequence is: ";
p = find(q, s, 86);
p++;
cout << *p << endl;
```

- This time the output will be:

```
The number 86 appears 1 times in the subsequence
The number following the first occurance of 86
        in the subsequence is: 92
```

- The function `reverse` reverses the contents of the container between the two iterators that we give it as an argument:

```
reverse(theVector.begin(), theVector.end());
cout << "After reversal, we have: " << endl;
for ( p = theVector.begin(); p != theVector.end(); p++ ) {
  cout << *p << " ";
}
cout << endl;

reverse(q,s);
cout << "After partial re-reversal, we have: " << endl;
for ( p = theVector.begin(); p != theVector.end(); p++ ) {
  cout << *p << " ";
}
cout << endl;
```

- Here we will get:

```
After reversal, we have:
21 49 92 86 35 93 15 77 86 83
After partial re-reversal, we have:
21 49 86 77 15 93 35 86 92 83
```

- Note how the positions of `q` and `s` were changed by the reversal.

- We can also `sort`

```
sort(theVector.begin(), theVector.end());
cout << "After sorting, we have: " << endl;
for ( p = theVector.begin(); p != theVector.end(); p++ ) {
  cout << *p << " ";
}
cout << endl;
```

and do a `random_shuffle`

```
random_shuffle(q, s);
cout << "After randomly shuffling part, we have: " << endl;
for ( p = theVector.begin(); p != theVector.end(); p++ ) {
  cout << *p << " ";
}
cout << endl;
```

- Giving us:

```
After sorting, we have:
15 21 35 49 77 83 86 86 92 93
After randomly shuffling part, we have:
15 21 92 77 83 35 86 86 49 93
```

- Naturally we can also do these operations on any range we like.

- The last two functions we will look at are `rotate` and `transform`.
- `rotate` takes three arguments. The first two define the range we will rotate.
- The last argument is another iterator which identifies the point that will, after rotation, be at the first position in the range.
- The rest of the range will be moved so that the items are still in the same order relative to each other as before.
- It is easier to understand this with an example.

- This code

```
sort(theVector.begin(), theVector.end());
cout << "After resorting, we have: " << endl;
for ( p = theVector.begin(); p != theVector.end(); p++ ) {
  cout << *p << " ";
}
cout << endl;

s = s - 1;
rotate(theVector.begin(), theVector.end(), s);
cout << "After rotating, we have: " << endl;
for ( p = theVector.begin(); p != theVector.end(); p++ ) {
  cout << *p << " ";
}
cout << endl;
```

generates this:

```
After resorting, we have:
15 21 35 49 77 83 86 86 92 93
After rotating, we have:
35 49 77 83 86 86 92 93 15 21
```

since `s` identifies the number in the third place to be the one about which the vector is rotated.

- `transform` applies a function to every element in the range.
- So with a function `doubleIt` that doubles its argument, this:

```
vector<int> theOther;
theOther.resize(theVector.size());

transform(theVector.begin(),theVector.end(),
          theOther.begin(),doubleIt);

cout << "After transforming, we have: " << endl;
for ( p = theOther.begin(); p != theOther.end(); p++ ) {
  cout << *p << " ";
}
cout << endl;
```

generates:

```
After transforming, we have:
70 98 154 166 172 172 184 186 30 42
```

- There are a few things to notice here.
- First, `transform` takes the result of applying the function and puts it in a separate container, identified by the third argument.
- In fact the third argument identifies the position in the container at which the results should start.
- You have to make sure there is enough space in the container you are copying into.
- The use of `resize` is an easy way to do this.
- This line

  ```
  theOther.resize(theVector.size());
  ```

  finds the size of `theVector` and then alters the size of `theOther` to match.

Summary

- This lecture focussed on more of the C++ Standard Template Library.
- We started by looking at container adapters, `stack`, `queue` and `priority_queue`.
- Then we took a quick look at some of the STL algorithms.
- There are many more that we did not have time to look at, but what we saw was pretty representative.
- Hopefully this suggested some of the work that the STL can save you doing as you write more complex C++ programs.