Lecture 20

“Standard” Template Library

What is the Standard Template Library?
- A note about the “STL”
  - Presented to the C++ Standards Committee by Alex Stepnov, Spring 1994
  - Alex was working for Hewlett-Packard at the time
  - The standards committee adopted it after making a large number of changes
  - HP made Alex’s version available for public downloads, now maintained on an SGI site:
  - This means that the “STL” is not part of the C++ Standard, despite its name!
  - The Standard has a variation of the STL included.

What is the Standard Template Library?
- A collection of C++ classes (templates)
  - containers
    - vectors
    - lists
    - stacks
    - queue/deques
    - sets/multisets
    - maps/multimaps
  - iterators
  - algorithms
- Each container is a class template, taking the type of element as a template argument:
  - A list of strings is: list<string>
  - A vector of integers is: vector<int>
  - A deque of vectors of float: deque<vector<float> >

The Standard Template Library: Iterators
- Each template (container) defines a public type name called iterator which can be used for iterations of objects in the container.
- In the STL, an iterator is a generalization of a pointer.
- Think of an iterator as a “pointer” to any object in the container at a given time.
- The “operator” is defined to return the actual element currently being “pointed at”.
- For unidirectional iterators, ++ is defined to advance to the next element.
- For bidirectional iterators, - - is also defined to back up to the previous element.
- Any container has member functions named begin() and end() which point at the first element and one past the last element, respectively.

The STL: Iterators
- Now, you can visit each element successively with the following:

```cpp
define the vector here...
for (vector<int>::iterator i = v.begin(); i < v.end(); ++i)
    cout << "Element at index: \"" << i << \"\n;" << endl;
```

- The vector container has methods available for performing “stack” operations (among others).
  - push_back(const T& x) // Add an element to end
  - pop_back() // removes last element
- So, consider the following code used to declare and populate a vector of strings:

```cpp
vector<string> someStringVector; // Declare a new vector
someStringVector.push_back("Hello");
someStringVector.push_back("World");
```

The STL: vectors
- Now, recalling our use of iterators to create a for loop which cycles through a vector:

```cpp
for (vector<int>::iterator i = v.begin(); i < v.end(); ++i)
    cout << *i << \"\n;" << endl;
```
Demonstration #1

The vector in action

Demonstration #2

The vector with list operations

The STL: vectors

```cpp
void main()
{
    vector<string> stringVector; // Declare a new vector
    stringVector.push_back("This");
    stringVector.push_back("Is");
    stringVector.push_back("The");
    stringVector.push_back("Test");

    for (vector<string>::iterator p = stringVector.begin(),
        p < stringVector.end(); ++p)
    {
        cout << "Next Vector Element is: " << *p << endl;
    }
}
```

Let's verify that this works as expected...

Demonstration #1

The STL: vectors

The vector in action

Demonstration #2

The vector with list operations

The STL: vectors

The vector<T>::iterator also allows "random" access

```cpp
vector<string> v; // A vector of strings
vector<string>::iterator p = v.begin();
for (int k=0; k<v.size(); k++)
    cout << "Next Vector Element is: " << p[k] << endl;
```

This gives us a way to access specific elements in the vector.

In the expression p[k] refers to the kth element after the element "pointed at" by p.

This is consistent with how pointer arithmetic works and maintains the "illusion" that p is actually a pointer.

The only difference is that there is range checking going on, so if you attempt to access an item that is out of range an exception will be thrown.

The STL: vectors

Vectors also have "list" manipulation methods

```cpp
vector<string> v; // A vector of strings
vector<string>::iterator p = v.begin();
vector<string>::iterator iter = v.begin();
vector<string>::iterator iter2 = v.end();
```

As you can see, the insertions are a little uglier because we have to provide iterators to tell the vector where to insert.

This is more powerful because we can insert into the middle of the list instead of the end.

The STL: vectors

```cpp
vector<string> v; // A vector of strings
v.insert(v.end(),"This"); // inserts new element before
v.insert(v.end(),"Is"); // specified iterator
v.insert(v.end(),"The");
vector<string>::iterator p = v.begin();
```

```cpp
v.erase(p); // erase 3rd element
v.erase(p); // erases first two elements
```

The erase methods take either a single iterator representing the element to erase or a range of elements to delete...
The STL: vectors
- Most STL containers have a built-in sort mechanism...

```cpp
tag::string> v; // A vector of integers
v.insert(v.end(), "This"); // inserts new element before
v.insert(v.end(), "is"); // specified iterator
v.insert(v.end(), "a");
sort(v.begin(), v.end()); // sorts the specified range of
// elements
for (vector<string>::iterator p = v.begin(); p != v.end(); p++)
    cout << *p << endl;
```
- sort takes two iterators and sorts all elements in that range.
- Let's make sure this works...

### Demonstration #3
**Sorting the vector**

The STL: vector--summary
- Let's summarize what we've covered with vector

```cpp
tag::T> v; // Declares a vector of type "T"
    v.push_back(const T&); // Adds a new "T" to end of vector
    const T& v.pop_back(); // Returns the last element in
    v.insert(iterator p, const T&); // Adds "T" to the vector just
    v.erase(iterator p); // Removes the item "pointed at"
    v.erase(iterator p, iterator q); // removes all elements between
    sort(iterator p, iterator q); // sorts elements between p/q
```
- Remember, v.erase() does not include the last iterator in the
  "erase range"

The STL: map
- Another interesting STL container is called map
- The primary concept here is that a map allows the management of a key-value pair.
- Its declaration, therefore, allows you to specify types for the
  "key" and the "value"

```cpp
    void main()
    {
        map<int, string> ids; // Map Ids to a name
    }
```
- OK, but how to we add key-value pairs to the map?
- The first thing we need to understand is the concept of a pair.
  A pair is a simple data type which simply groups together
  two other types: in this case a key and a value:

### The STL: definition of pair
- The definition looks something like this:

```cpp
    template <class T1, class T2> // Simplified
    struct pair
    {
        T1 first;
        T2 second;
    };
```
- For each map that is defined there is a new type created
  within the scope of the map to represent the key-value pair
  for that particular map.

### The STL: map
- It is based on pair
- More specifically, it looks like this:

```cpp
    template <class Key, class Value> // Simplified
    class map
    {
    public:
        typedef pair<const Key, Value> value_type;
    ...
    };
```
- This means that for any map declared, the following type is also
  publicly declared:
  ```cpp
  typedef map<Key, Value>::value_type
  ```
- It is quite common to typedef this to some simpler type so
  that it may be referred to easily in your source code:
The STL: map

typedef map<int,string>::value_type IDRecord;

With this I can now begin to write some code that will allow us to populate a map.

```cpp
typedef map<int,string>::value_type IDRecord;
void main()
{
    map<int,string> ids;
    IDRecord rec1(12345, "Ron DiNapoli");
    IDRecord rec2(34564, "Albert Eskenazi");
    ids.insert(rec1);
    ids.insert(rec2);
}
```

The STL: map

OK, so we can put stuff into the map. How do we get it out?

With an iterator, of course!

Along with a custom pair for each map, there is a custom iterator.

It is given the name iterator and is defined locally to the scope of the particular map you are defining:

```cpp
typedef map<int,string>::iterator IDRecordIterator;
```

This iterator will represent a pair. You have to remember that pair.first is the key and pair.second is the value.

But how to you get one of these iterators?

The map<Key,Value>::find() method.

find() takes a key type and returns the corresponding value.

This reinforces that a map must contain unique keys.

The multimap allows duplicate keys

```cpp
IDRecordIterator p = ids.find(12345);
cout << "ID 12345 belongs to: " << (*p).second << endl;
```

The STL: map

Let's look at a complete example

```cpp
typedef map<int,string>::value_type IDRecord;
void main()
{
    map<int,string> ids;
    IDRecord rec1(12345, "Ron DiNapoli");
    IDRecord rec2(34564, "Albert Eskenazi");
    ids.insert(rec1);
    ids.insert(rec2);
    IDRecordIterator p = ids.find(12345);
    cout << "ID 12345 belongs to: " << (*p).second << endl;
}
```

Demonstration #4

Using the map

```cpp
typedef map<string,string>::value_type IDRecord;
void main()
{
    map<string,string> ids;
    IDRecord rec1("abcde", "Ron DiNapoli");
    IDRecord rec2("fghij", "Albert Eskenazi");
    ids.insert(rec1);
    ids.insert(rec2);
    cout << "ID abcde belongs to: " << ids["abcde"] << endl;
}
```

The STL: map

Suppose we didn't have an int as the "key type"

The operator[] functionality still works!

```cpp
typedef map<string,string>::value_type IDRecord;
void main()
{
    map<string,string> ids;
    IDRecord rec1("abcde", "Ron DiNapoli");
    IDRecord rec2("fghij", "Albert Eskenazi");
    ids.insert(rec1);
    ids.insert(rec2);
    cout << "ID abcde belongs to: " << ids["abcde"] << endl;
}
```
Demonstration #5

More fun with the map

Lecture 20

Final Thoughts