

## CS 213 -- Lecture #20

“Late Night Guide to C++”  
Chapter 10 pg 278 - 290

### “STANDARD” TEMPLATE LIBRARY

### *Administrative...*

- Assignment #9 due Thursday
- 2 more assignments to be done
- Remember, second prelim on 11/23

### *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 > >`

### *What is the Standard Template Library?*

- A note about the “STL”
  - Presented to the C++ Standards Committee by Alex Stepanov, 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:
    - <http://www.sgi.com/tech/stl/>
  - This means that the “STL” is not part of the C++ Standard, despite its name!
  - The Standard has a variation of the STL included.

### *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 (cont)*

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

```
vector<int> v;           // A vector of integers
// Populate the vector here...
for (vector<int>::iterator p = v.begin(); p < v.end(); ++p)
    cout << "Next Vector Element is: " << *p << endl;
```

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

### *The STL: vectors (cont)*

```
Void main()
{
    vector<string> stringVector;    // Declare a new vector
    stringVector.push_back("Test");
    stringVector.push_back("a");
    stringVector.push_back("is");
    stringVector.push_back("This");
}
```

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

### *The STL: vectors*

```
Void main()
{
    vector<string> stringVector;    // Declare a new vector
    stringVector.push_back("This");
    stringVector.push_back("is");
    stringVector.push_back("a");
    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 vector in Action

### *The STL: vectors (cont)*

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

```
vector<string> v;                // A vector of strings
// Populate the vector here...
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 (cont)*

- Vectors also have "list" manipulation methods

```
vector<string> v;                // A vector of strings
// Populate the vector here...
v.insert(v.end(), "This");       // inserts new element before
v.insert(v.end(), "is");         // specified iterator
v.insert(v.end(), "a");
v.insert(v.end(), "test");
vector<string>::iterator p = v.begin();
v.erase(p+2);                   // erase 3rd element
v.erase(p, p+2);                // erases first two elements
```

- 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 (cont)*

```
vector<string> v;                // A vector of strings
// Populate the vector here...
v.insert(v.end(), "This");       // inserts new element before
v.insert(v.end(), "is");         // specified iterator
v.insert(v.end(), "a");
v.insert(v.end(), "test");
vector<string>::iterator p = v.begin();
v.erase(p+2);                   // erase 3rd element
v.erase(p, p+2);                // erases first two elements
```

- The erase methods take either a single iterator representing the element to erase or a range of elements to delete...

## Demonstration #2

The vector with list operations

### The STL: vectors (cont)

- Most STL containers have a built in sort mechanism...

```
vector<string> v; // A vector of integers
// Populate the vector here...
v.insert(v.end(),"This"); // inserts new element before
v.insert(v.end(),"is"); // specified iterator
v.insert(v.end(),"a");
v.insert(v.end(),"test");
sort(v.begin(),v.end()); // sort 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: vectors--summary

- Let's summarize what we've covered with vector

```
vector<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
// the vector and removes it
v.insert(iterator p,const T&)// Adds "T" to the vector just
// before p
v.erase(iterator p) // Removes the item "pointed at"
// by "p" from the vector
v.erase(iterator p,iterator q)//removes all elements between
// p and q from the vector
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"

```
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:

```
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 (cont)*

- It is based on `pair`
- More specifically, it looks like this:

```
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:
  - `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 (cont)*

```
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.

```
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 (cont)*

- 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:
  - `map<int, string>::iterator`
- 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

### *The STL: map (cont)*

- Let's look at a complete example

```
typedef map<int, string>::value_type IDRecord;
typedef map<int, string>::iterator IDRecordIterator;
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;
}
```

### *The STL: map (cont)*

- The map iterator syntax is cumbersome, so there is a shorthand

```
typedef map<int, string>::value_type IDRecord;
typedef map<int, string>::iterator IDRecordIterator;
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;
    cout << "ID 34564 belongs to: " << ids[34564] << endl;
}
```

## *Demonstration #4*

Using the map

### *The STL: map (cont)*

- Suppose we didn't have an `int` as the "key type"
- The `operator[ ]` functionality still works!

```
typedef map<string,string>::value_type IDRecord;  
typedef map<string,string>::iterator IDRecordIterator;  
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

### *Final Thoughts*

- Assignment #9 due on Thursday
- Prelim 11/23
- We're winding down, but some more interesting topics on the way