C++ templates

Templates are a mechanism for classes and functions in C++ to have type parameters. This is a concept similar to generics in Java 1.5. In this course, we won't bother much with the details of how to create templatized classes and functions; we will simply use the ones provided by the C++ Standard Template Library (STL).

Nonetheless, it's nice to know how templates work, so here's a small example of what a function template looks like. Consider a simple *min()* function that returns the minimum of two integers.

```
int min( int a, int b ) {
    return a < b ? a : b;
}
```

Using templates, we can write a function that will work on not only integers, but also chars, doubles, strings, and anything else that has the '<' operator defined on it. Here is the code.

```
template< class C >
C min( C a, C b ) {
    return a < b ? a : b;
}
```

The function takes two parameters of type C and returns their minimum (also of type C), where C is whatever a and b happen to be at the place where the function *min()* is called. This is how you can call the function from *main()* or anywhere else.

```
int x = 2, y = 3, z = min( x, y );
double d = 0.5, e = -1, f = min( d, e );
string s = "Xander", t = "Willow", u = min( s, t );
```

The variable z will be 2, f will be -1.0 and u will be "Willow". The parameter type for C in each of the 3 function calls are *int*, *double* and *string*, respectively. In reality, when the code is compiled, there are 3 *min()* function created. Note that the parameters *a* and *b* are both of type C, and C can only be one thing at a time. This means that you can't call *min()* and pass to it an *int* and a string like this:

```
min( 15, "Giles" );  // compile error
```

You can define classes with template parameters, too. For example, the STL defines a class `pair` that simply stores 2 things.

```
template< class A, class B >
class pair {
 public:
    A first;
    B second;
    pair( A a, B b ) { first = a; second = b; }
};
```

`pair` has 2 member variables (or fields): `first` is of type A and `second` is of type B. To create a `pair`, you need to manually specify what A and B are, like this.

```
pair< string, double > p( "pi", 3.14 );
p.second = 3.14159;
```

The first line calls the constructor. To use `pair`, you need to #include <map>.

STL collections

The STL provides a large number of data structures that can greatly simplify many programming tasks. Some of the more commonly used ones are

vector:	like Java's Vector or ArrayList.
set:	like Java's TreeSet, implemented as a balanced binary search tree.
list:	a doubly-linked list with O(1) insertion and removal of elements at the front or the back.
map:	like Java's TreeMap - basically, a set of key-value pairs, sorted by the key.
queue:	allows O(1) insertion at the back and O(1) removal at the front.
stack:	allows O(1) insertion and removal at the front (top).

There are more (priority_queue, rope, etc.), but we will focus on these 6 because they appear most often in writing complex algorithms.

All of these collections are templatized classes. For example, to create a vector of integers, you need to specify the template parameter (`int`) in the constructor.

Example 1:
```
vector< int > v( 10 );
for( int j = 0; j < 10; j++ ) v[j] = j * j;
```

This will create a vector of 10 integers (all garbage initially) and fill it up with the first 10 perfect squares (0, 1, 4, 9, ..., 81). Note that the square brackets operator is overloaded for vectors, so`v` can be used just like an array. Now we will go into more details about using vector and set.

Vectors

```
#include <vector>
```

The STL **vector** class is designed to provide all the functionalities of normal arrays, but with several extra features that make vector easier to use and handle. Like an array, a vector holds a collection of elements of the same *data type*, in a contiguous sequence for O(1) random access. The elements of a vector are accessed using the **square bracket** operator "`[]`". You can set and retrieve these elements just like you would in a normal array. Example 1 above shows how a vector is used.

STL vectors have several convenient features. You can resize, shrink, or grow a vector **dynamically**, while the program is running. For example, the following code uses the **resize()** method to increase the size of the vector after it is declared.

Example 2:
```
vector< int > v( 10 );
for( int j = 0; j < 10; j++ )
  v[j] = j * j;        // v is of size 10, filled with squares
v.resize( 20 );        // After resizing to size 20, the first 10 elements
                       // remain unchanged.  The rest are undefined.
for (int j = 10; j < 20; j++ )
  v[j] = j * j;        // v is now size 20, filled with squares
```

Question: What would be the complexity of **resize()**?

Another useful feature of vector is the **push_back()** method. This method appends an element to the *end* of the vector. If the size of the vector is not big enough to accommodate the new element, the vector is **automatically** resized! This means that you do not have to worry about how large you should initialize your vector to be. The following example demonstrates this:

Example 3:
```
vector< int > v;          // Creates an empty vector
for( int j = 0; j < 10; j++ )
  v.push_back( j * j );   // append one element to v
/**
 * At this point, the vector v has automatically grown to size 10.
 * You can add more elements to v using more push_back calls.
 **/
```

Very often, you would like to know what the size of a vector is. This is done via the **size()** method. You can also clear an array using the **clear()** method, which is like calling **resize(0)**.

Example 4:
```
vector< int > v( 3, 1 );   // Creates a vector of size 3, with all 1's
for ( int j = 0; j < 2; j++ )
  v.push_back( j*2 );      // Append elements "0", and "2" to v
int sz = v.size();         // sz is set to the size of v, which is 5.
for ( int i = 0; i < sz; i++ )
  cout << v[i] << endl;    // Prints the array line by line
v.clear();                 // Make the array empty
sz = v.size();             // v is now empty, so sz is set to 0.
```

Stacks, Queues, and Deques

```
#include <stack>
#include <queue>
// either one will provide deque
```

Stacks, queues, and deques (for double-ended queues) are simple containers that allows O(1) insertion and deletion at the beginning and/or end. You cannot modify any elements inside the container, or insert/remove elements other than at the ends. However, because they are templatized, they can hold just about any data types, and are termendously useful for many purposes.

A queue is a First-In-First-Out (FIFO) container, whereas a stack is Last-In-First-Out (LIFO). A deque is **both** a stack and a queue. The following demonstrates just about all you can do with a queue:

Example 5:
```
queue< int > Q;        // Construct an empty queue
for ( int i = 0; i < 3; i++ )
  Q.push( i );         // Pushes i to the end of the queue

// Q is now { "0", "1", "2" }
int sz = Q.size();     // Size of queue is 3
while( !Q.empty() ) {      // Print until Q is empty
  int element = Q.front(); // Retrieve the front of the queue
  Q.pop();                 // REMEMBER to remove the element!
  cout << element << endl; // Prints queue line by line
}
```

A stack has pretty much all the methods of a queue, including **push(), pop(), size(), empty()**. However, instead of **front()**, a stack accesses the top of the stack with **top()**. And of course, **pop()** will retrive the element at the **top** (or **end**) of the stack, not the **front**.

Finally, a deque has both the features of a stack and a queue. It has the member methods **size()** and **empty()**, but instead of the **push(), pop()** combination, it now provides 4 different methods, all pretty much selft-explanatory: **push_front(), push_back(), pop_front(), pop_back()**.