Templates - Creating generic functions

[Much of the material in these notes is derived from the Hubbard book (see course outline).]

The swap function is a specific function that swaps two objects of a particular type. For example, swapping two integers:

```
void swap (int& m, int& n){
   int temp = m;
   m = n;
   n = temp;
}
```

or two strings:

```
void swap (string& s1, string& s2){
   string temp = s1;
   s1 = s2;
   s2 = temp;
}
```

etc.

So, we generally would expect to have to overload the swap function to account for swapping objects of different types (classes).

Wouldn't it be nice if we could write a single swap function that has, as one of its parameters, the type (or, class) of the items being swapped? We would like to create a ***generic*** swap function that would work on any two objects of the same type (for **any** type).

We can do this by creating a ***template***, a placeholder for a type to be known later.

Swap Function Template - Generic Swap Function:

```
template <class T>
void swap (T& x, T& y){
   T temp = x;
   x = y;
   y = temp;
}
```

In this case,

T ≡ a type parameter

≡ a placeholder that is replaced with an actual type when the function is invoked. The call looks like any ordinary call.

```
//swapfn.cpp
//using template
#include <iostream>
#include <string>
using namespace std;

template <class T>
void swap (T& x, T& y){
   T temp = x;
   x = y;
   y = temp;
}

int main(){
   int a=1,b=2;
   string s1, s2;
   s1= "string 1";
   s2 = "string 2";
   cout << "a= " << a << " and b= " << b << endl;
   swap(a,b);
   cout << "After swap, \na= " << a << " and b= " << b << endl;
   cout << "\ns1= " << s1 << " and s2= " << s2 << endl;
   swap (s1, s2);
   cout << "After swap, \ns1= " << s1 << " and s2= " << s2 << endl;

   char c; cin >> c; //just to hold console
   return 0;
}
```

```
a= 1 and b= 2
After swap,
a= 2 and b= 1

s1= string 1 and s2= string 2
After swap,
s1= string 2 and s2= string 1
```

We may also create a function template that has more than one type parameter, e.g.:
template <class T, class U, class V>

Class Templates - Generic Classes

We can create a generic class with a class template for, e.g., a stack. Instead of limiting a program to processing a stack of, say, integers, or a stack of whatever, why not create a generic stack class using a stack template that will work on stack of any type. In the following example the stack is housed in an array:

```
//stack.cpp
//using template
//modified from Hubbard p. 359 ex 13.3
#include <iostream>
#include <string>
using namespace std;

template <class T>
class stack {
public:
   stack (int s=100) : size(s), top(-1) {data = new T[size];}
   ~stack(){delete [] data;}
   void push (const T& x) {data[++top] = x;}
   T pop() {return data[top--];}
   bool isEmpty() const {return top == -1;}
   bool isFull() const {return top == size-1;}
private:
   int size;
   int top;
   T* data;
};


int main(){
   stack<int> intStack1(5);
   stack<int> intStack2(10);
   stack<string> strStack(8);
   intStack1.push(77);
   strStack.push("apple");
   intStack2.push(22);
   strStack.push("banana");
   strStack.push("orange");
   intStack2.push(44);
   cout << intStack2.pop() << endl;     // just pop
   if (intStack2.isEmpty())             // test first, then pop
      cout << "intStack2 is empty." << endl;
      else cout << intStack2.pop() << endl;
   if (intStack2.isEmpty())
      cout << "intStack2 is empty." << endl;
      else cout << intStack2.pop() << endl;

   char c; cin >> c; //just to hold console
   return 0;
}
```

Output:

```
44
22
intStack2 is empty.
```

stack<int> and stack<string> are two distinct classes, generated by the compiler from the single generic stack class template.