Templates

Templates are a way to abstract a function or class such that various types are allowed to use the same function or class. This reduces code as you do not have to create a function or class for each and every type you want to use. For example, if you want to swap two integers, you can make a function to explicitly do so. If you want to swap two characters, you would have to make a separate function to handle this. Two functions to do the same task is extraneous. With templates, you only need one function and this function can be used on any valid type.

```
template <typename T>
template <class T>
```

To start a template function, you need to first write one of the two lines shown in the picture. They both do the same thing and the bottom is valid for legacy reasons. The **T** will represent the generic type for the function. This can be thought of as a variable and can be named whatever you would like (such as **Type** in the example below). Remember that this header must come before all function declarations and definitions. Here is an example of a function to swap two items whether they are both integers, characters, doubles, etc:

```
template <typename Type> //Type serves as our base type placeholder
void swapValues(Type &x, Type &y)
{
    Type temp;  //Can also use Type in function to create new vars
    temp = x;
    x = y;
    y = temp;
}
```

Whatever you pass into this function must be of the same base type. You can think of **Type** as being replaced by **int** if you pass in two integers, or **char** if you pass in two characters.

Remember what is being done in a template function or class must be valid for the base type you choose. For example, you cannot pass in an array to the **swapValues** function, because you cannot swap arrays as such.

In order to use a template function, you would invoke the function the same way you would any other function.

Templates for classes start out the same way as a template function:

```
template <typename T>
class ClassName
```

Inside the class definition, you use **T** in place of the actual base type you want the class to have. This ensures you are able to use **int**, **char**, **double**, etc (so long as what you are doing in the class can be done to the base type of your choice).

When it comes to the implementation, templates should be kept in the same file at the class definition (header file). This has something to do with the compilation process and this is the easiest way to compile your program. Each member function must have `template <typename T>` just as the template function on the previous page. Additionally, the class name before the scope resolution operator must have `<T>` immediately after the name. Here is a generic example of the beginning aspect of a member function:

```
template <typename T>
void className<T>::memberFunction(T value)
```

In your main function, declaring an object of a template class also deals with angle brackets. If you are familiar with vectors or the Standard Template Library, it is the same syntax. If you are not familiar, angle brackets come directly after the class name and inside specifies the type you want for that instance (object) of the class. For example:

```
className<char> object;
className<int> object2;
```

The convenient thing is you are able to have objects of the same class but with different types. Here, there is one object of type `char` and one object of type `int` in the same program. You are able to use the objects the same way you would any object of a class.

A small example of a class definition/implementation is:

```
template <typename T> //Required before class declaration
class className<T>
{
    public:
        display();
    private:
        T firstItem;
        T secondItem;
};

template <typename T> //Required before every member function
void className<T>::display()
{
    cout << firstItem << ' ' << secondItem << endl;
}
```

This incomplete example was to show you what needs to be done to every class definition as well as every member function. This class would work with many base types as all this class does is display the items passed into it (you would need to implement constructors).