Templates and Templated Classes in C++

What's better than having several classes that do the same thing to different datatypes? One class that lets you choose which datatype it acts on.

Templates are a way of making your classes more abstract by letting you define the behavior of the class without actually knowing what datatype will be handled by the operations of the class. In essence, this is what is known as generic programming; this term is a useful way to think about templates because it helps remind the programmer that a templated class does not depend on the datatype (or types) it deals with. To a large degree, a templated class is more focused on the algorithmic thought rather than the specific nuances of a single datatype. Templates can be used in conjunction with abstract datatypes in order to allow them to handle any type of data. For example, you could make a templated stack class that can handle a stack of any datatype, rather than having to create a stack class for every different datatype for which you want the stack to function. The ability to have a single class that can handle several different datatypes means the code is easier to maintain, and it makes classes more reusable.

The basic syntax for declaring a templated class is as follows:

```
template <class a_type> class a_class {...};
```

The keyword 'class' above simply means that the identifier **a_type** will stand for a datatype. NB: **a_type** is not a keyword; it is an identifier that during the execution of the program will represent a single datatype. For example, you could, when defining variables in the class, use the following line:

```
a_type a_var;
```

and when the programmer defines which datatype '**a_type**' is to be when the program instantiates a particular instance of **a_class**, **a_var** will be of that type.

When defining a function as a member of a templated class, it is necessary to define it as a templated function:

```
template<class a_type> void a_class<a_type>::a_function(){...}
```

When declaring an instance of a templated class, the syntax is as follows:

```
a_class<int> an_example_class;
```

An instantiated object of a templated class is called a specialization; the term specialization is useful to remember because it reminds us that the original class is a generic class, whereas a specific instantiation of a class is specialized for a single datatype (although it is possible to template multiple types).

Usually when writing code it is easiest to precede from concrete to abstract; therefore, it is easier to write a class for a specific datatype and then proceed to a templated - generic - class. For that brevity is the soul of wit, this example will be brief and therefore of little practical application.

We will define the first class to act only on integers.

```
class calc
{
	public:
		int multiply(int x, int y);
		int add(int x, int y);
};

int calc::multiply(int x, int y)
{
	return x * y;
}

int calc::add(int x, int y)
{
	return x + y;
}
```

We now have a perfectly harmless little class that functions perfectly well for integers; but what if we decided we wanted a generic class that would work equally well for floating point numbers? We would use a template.

```
template <class A_Type> class calc
{
	public:
		A_Type multiply(A_Type x, A_Type y);
		A_Type add(A_Type x, A_Type y);
};

template <class A_Type> A_Type calc<A_Type>::multiply(A_Type x,A_Type y)
{
	return x * y;
}

template <class A_Type> A_Type calc<A_Type>::add(A_Type x, A_Type y)
{
	return x + y;
}
```

To understand the templated class, just think about replacing the identifier **A_Type** everywhere it appears, except as part of the template or class definition, with the keyword int. It would be the same as the above class; now when you instantiate an object of class calc you can choose which datatype the class will handle.

```
calc <double> a_calc_class;
```

Templates are handy for making your programs more generic and allowing your code to be reused later.