**Templates and Template 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.

# Templated Functions

C++ templates can be used both for classes and for functions in C++. Templated functions are actually a bit easier to use than templated classes, as the compiler can often deduce the desired type from the function's argument list.

The syntax for declaring a templated function is similar to that for a templated class:

template <class type> type func\_name(type arg1, ...);

For instance, to declare a templated function to add two values together, you could use the following syntax:

template <class type> type add(type a, type b)

{

return a + b;

}

Now, when you actually use the add function, you can simply treat it like any other function because the desired type is also the type given for the arguments. This means that upon compiling the code, the compiler will know what type is desired:

int x = add(1, 2);

will correctly deduce that "type" should be int. This would be the equivalent of saying:

int x = add<int>(1, 2);

where the template is explicitly instantiated by giving the type as a template parameter.   
  
On the other hand, type inference of this sort isn't always possible because it's not always feasible to guess the desired types from the arguments to the function. For instance, if you wanted a function that performed some kind of cast on the arguments, you might have a template with multiple parameters:

template <class type1, class type2> type2 cast(type1 x)

{

return (type2)x;

}

Using this function without specifying the correct type for type2 would be impossible. On the other hand, it is possible to take advantage of some type inference if the template parameters are correctly ordered. In particular, if the first argument must be specified and the second deduced, it is only necessary to specify the first, and the second parameter can be deduced.   
  
For instance, given the following declaration

template <class rettype, class argtype> rettype cast(argtype x)

{

return (rettype)x;

}

this function call specifies everything that is necessary to allow the compiler deduce the correct type:

cast<double>(10);

which will cast an int to a double. Note that arguments to be deduced must always follow arguments to be specified. (This is similar to the way that default arguments to functions work.)   
  
You might wonder why you cannot use type inference for classes in C++. The problem is that it would be a much more complex process with classes, especially as constructors may have multiple versions that take different numbers of parameters, and not all of the necessary template parameters may be used in any given constructor.

## Templated Classes with Templated Functions

It is also possible to have a templated class that has a member function that is itself a template, separate from the class template. For instance,

template <class type> class TClass

{

// constructors, etc

template <class type2> type2 myFunc(type2 arg);

};

The function myFunc is a templated function inside of a templated class, and when you actually define the function, you must respect this by using the template keyword twice:

template <class type> // For the class

template <class type2> // For the function

type2 TClass<type>::myFunc(type2 arg)

{

// code

}

The following attempt to combine the two is **wrong** and will not work:

// bad code!

template <class type, class type2> type2 TClass<type>::myFunc(type2 arg)

{

// ...

}

because it suggests that the template is entirely the class template and not a function template at all.

url: http://www.cprogramming.com/tutorial/templated\_functions.html