Function Templates

Before we look at what function templates are, let's look at a problem that will lead to an understanding of why we would use them. Say we have a function that computes the sum of the squares of two values and takes integers as parameters.

```
// Take two integers and returns the sum of their squares
int SumOfAllSquares(int opp, int adj)
{
    return opp*opp + adj*adj;
}
```

Now let's assume that we also want to perform this operation on floats and doubles. To do this, we would have to rewrite the same function for both data types.

```
// Take two floats and returns the sum of their squares
float SumOfAllSquares(float opp, float adj)
{
    return opp*opp + adj*adj;
}

// Take two doubles and returns the sum of their squares
double SumOfAllSquares(double opp, double adj)
{
    return opp*opp + adj*adj;
}
```

Although this method works, it's not very elegant. If we had some larger functions and we wanted them to be able to handle a whole set of different data types, we'd be doing a lot of copy-pasting and we'd have a lot of repetitive code.

This is where the benefit of templates comes in. Templates are a C++ feature that allow you to write code that can handle general data types. By "general datatypes", I mean that templates allow you to write functions that can work on multiple types.

The syntax for declaring a template is

```
template <class T>
```

followed by the declaration of the function. What this statement does is declares that our function will work with a data type that is specified when the function is called. class T denotes the variable T as a general data type.

Here's a complete example of how we would use a templated function. Note that the variable T can have any name, it doesnt have to be called T.

```
// Take two general values and return the sum of their squares
template <class DataType>
DataType SumOfAllSquares(DataType opp, DataType adj)
{
    return opp*opp + adj*adj;
}

// We call the function as if it's defined for whatever data type
we're using
int iX = 8, iY = 9;
float iX = 4, iY = 6;

int iR = SumOfAllSquares(iX, iY);
float fR = SumOfAllSquares(fX, fY);
```

Note that we specified that both parameters are of type DataType and that the return type is also of type DataType. We don't necessarily have to have all of the parameters or the return value be of general type (we did here because we're returning the same type that's passed in), but we must be sure that any operations we perform on the types are defined.

For example, this function works fine with numerical types, but what if we have a class that represents a person in a game? If we haven't specified numerical operations on the person class, our program will have no idea what to do when we tell it to add and multiply two of them together.

See the downloadable source code for this tutorial to see templated functions in action.

Class Templates

Now that we have seen how templated functions work, let's take a look at templated classes.

Templated classes are mostly used when building large data structures that need to be able to store different data types. The same reasoning works for classes as for functions. We can either make a class that uses say, an int, and then copy-paste when we want to use other data types, or we can template the class and use any data type we want.

Like I said before, templates are widely used with data structures (see my Linked Lists Tutorial for an example of a templated data structure), but I'm going to use a simpler class example here.

```
// Declare a class that uses templated data types
template <class DataType>
```

```
class NumberHolder
{
public:
    // The constructor takes three pieces of meaningless data
    NumberHolder(DataType num1, DataType num2, DataType num3) :
       m_Num1(num1), m_Num2(num2), m_Num3(num3) {}

    // Note that each accessor returns a 'DataType'
    DataType GetNum1() { return m_Num1; }
    DataType GetNum2() { return m_Num2; }
    DataType GetNum3() { return m_Num3; }

    // Note that each mutator takes a 'DataType'
    void SetNum1(DataType num) { m_Num1 = num; }
    void SetNum2(DataType num) { m_Num2 = num; }
    void SetNum3(DataType num) { m_Num3 = num; }

private:
    DataType m_Num1;
    DataType m_Num2;
    DataType m_Num3;
};
```

First of all, sorry for the lame example. I didn't want to build a data structure here so I took the easy way out and wrote a meaningless class. If you want to see an example of templated classes in real use, see my Linked List Tutorial.

Notice that in the above code we were able to use DataType just like it was a normal data type (like an int or a char). This allowed us to use this class with any reasonable type of data. To instantiate an object from this class, we would have to specify the type of data we want to use. After we specify the data type, DataType will be replaced by the specified type.

```
// We specify the type within the <> brackets
NumberHolder<int> iNumHolder(2, 4, 5);
NumberHolder<float> fNumHolder(2.0f, 4.0f, 5.0f);
NumberHolder<double> dNumHolder(2.0f, 4.0f, 5.0f);

// Now we can use the objects like normal
int x = iNumberHolder.GetNum1();
fNumHolder.SetNum2(9.0f);
```

There are two more things you should know about templates.

First, you can only write a templated classes within a single file. If you try to separate

the definition and the implemention into .h and .cpp files, you'll get link errors. So be sure to define your templated classes within indepenent header files.

Second, you aren't restricted to only using one data type for templates. You can have your templates take as many different general types as you like. For example, putting

```
template <class Type1, class Type2>
```

before a class would allow you to specify two different types when instantiating objects from it. This is great for data structures that need to hold multiple data types.

I hope you now have a good idea about how templates work. As always, if you need help just ask!

Click here to download the source code for this tutorial.