CS115 Lab: Templates

Highlights of this lab:

It is often useful to be able to reuse the same algorithms with many different data types. This is called generic programming. This week you will learn generic programming with templates. By the end of the lab you should be able to take a function or class, create a templated version of it and apply multiple data types to it.

Lab Exercise:

- Make and use a template based on a function
- Transform the Matrix class into a templated class and use it with different data types.

Click the little computer above for a detailed description.
NOTE: Your lab instructor will tell you what will be marked for this lab.

---

1. What are Templates?

A template is a mechanism in C++ that lets you write a function or a class that uses a generic data type. A placeholder is used instead of a real type and a substitution is done by the compiler whenever a new version of the function or class is needed by your program. The compiler literally fills in the blanks in the template.

In this lab we will take the simple `Matrix` ADT we started in a previous lab and use it for 2D arrays of integers, floats, and strings. Without templates if you needed all three in one program you would need to write three versions. With templates you would need only the template and a few simple calls.

Using templates can:

- save you typing
- help you reuse old code without introducing new mistakes

You can use templates on functions or classes. Because the syntax is different for the two we will discuss them in seperate sections.

2. Function Templates

Consider the simple function `printIt`:

```
#include <iostream>
using namespace std;

void printIt(int a, int b)
{
   int c = a + b;
   cout << "You gave me " << a << " and " << b << ".\n";
   cout << "Together they make " << c << "." << endl;
}

int main()
{
   string aA = "Oh ";
   string aB = "noes!";
   printIt(1,2);
   printIt(2.6, 3.7);
   printIt('A','1');
   //printIt(aA, aB);
}
```

Here's the output for this program:

```
You gave me 1 and 2.
Together they make 3.
You gave me 2 and 3.
Together they make 5.
You gave me 65 and 49.
Together they make 114.
```

The program compiles, but `printIt` only accepts integers so the floating point numbers 2.6 and 3.7 and the characters `'A'` and `'1'` are coerced to int and are not treated as they should be. There is no good coercion for the string type to integer, so the program won't compile if you try to use it.

We can fix all these problems by making printit generic with templates. Here is the templated version of `printIt`:

```
template <typename T>
void printIt( T a, T b)
{
   T c = a + b;
   cout << "You gave me " << a << " and " << b << ".\n";
   cout << "Together they make " << c << "." << endl;
}
```

Here's the output. The call to `printIt` with string arguments has been uncommented as well.

```
You gave me 1 and 2.
Together they make 3.
You gave me 2.6 and 3.7.
Together they make 6.3.
You gave me A and 1.
Together they make r.
You gave me Oh  and noes!.
Together they make Oh noes!.
```

A new version of `printIt` is generated for each data type used as a parameter. Notice, though, that operations on the data types are bound by the normal rules that apply to them. The string concatenation would not work for C strings and the behaviour shown by the characters is a bit odd.

A few notes on the syntax.

- The `template <typename T>` bit can be on the same line as the function type declaration, but it is usually on the line above.
- The value `T` stands for the type the template will be ***instantiated*** with. `T` can be any valid token, but watch out for namespace clashes.
- **T**'s scope is limited to just one function, in this case `printIt`.
- Notice that we don't need to do anything special to make `printIt` work for new types.
- If a placeholder appears more than once in a function's parameter list the types you use in their place in the function call must match. e.g.

```
...
int a,b;
double c;
printIt(a,b); //OK
printIt(a,c); //bad
...
```

Note: prototypes

To create a prototype for a templated function remember to include the template specifier like so:

```
template <typename T>
void printIt(T,T);
```

3. Class Templates

Class Templates should be considered in three parts:

1. Class Definition
2. Class Instantiation
3. Member Function Definition

3.1 Class Definition

Here's a simplified `Matrix` class's definition:

```
class Matrix
{
  private:
    int doubleArray[MAXROWS][MAXCOLS];
    int rows;
    int cols;
  public:
    Matrix();
    void printMatrix();
    void setElement(int row, int col, int value); //set an element of the matrix
    void setMatrix(int [][MAXCOLS]); //set the doubleArray to what is sent
    void addMatrix(int [][MAXCOLS]); //add an array to doubleArray
    void addMatrix(int [][MAXCOLS], int[][MAXCOLS]); //add two arrays together
};
```

Making a template from this is just like making a template from a function:

```
template <typename M_type>
class Matrix
{
  private:
    M_type doubleArray[MAXROWS][MAXCOLS];
    int rows;
    int cols;
  public:
    Matrix();
    void printMatrix();
    void setElement(int row, int col, M_type value); //set an element of the matrix
    void setMatrix(M_type [][MAXCOLS]); //set the doubleArray to what is sent
    void addMatrix(M_type [][MAXCOLS]); //add an array to doubleArray
    void addMatrix(M_type [][MAXCOLS], M_type[][MAXCOLS]); //add two arrays together
};
```

Notice that although `rows` and `cols` are of type int they have not been templated. They need to be int to represent the dimensions of the matrix no matter what type is stored in the matrix.

3.2 Class Instantiation

To make an instance of a class you use this form:

```
class_name<type> variablename;
```

To create a `Matrix` with float you would type:

```
Matrix<float> floatMatrix;
```

Taken together `Matrix<float>` becomes the name of a new class. This will help you understand how member function definition works with templates.

3.3 Member Function Definition

The definition for a templated member function is a little surprising at first. Recall that a member function starts like this:

```
return_type class_name::function_name(parameter_list,...)
```

Pay attention to class name. The class name of a templated class is partly defined by the type it was instantiated with. The class name of the `floatMatrix` object above is `Matrix.` To refer to the class in a generic way you must include the placeholder in the class name like so:

```
template <typename T>
return_type class_name<T>::function_name(parameter_list,...)
```

Given the following definition of `addMatrix`:

```
void Matrix::addMatrix(int otherArray[][MAXCOLS])
{
  for (int i=0; i< rows; i++)
  {
    for(int j=0; j< cols; j++)
    {
      doubleArray[i][j] += otherArray[i][j];
    }
  }
}
```

The templated function would look like this:

```
template <typename M_type>
void Matrix<M_type>::addMatrix(M_type otherArray[][MAXCOLS])
{
  for (int i=0; i< rows; i++)
  {
    for(int j=0; j< cols; j++)
    {
      doubleArray[i][j] += otherArray[i][j];
    }
  }
}
```

Bringing it All Together

Normally when you write a C++ class you break it into two parts: a header file with the interface, and a .cpp file with the implementation. With templates this doesn't work so well because the compiler needs to see the definition of the member functions to create new instances of the templated class. Some compilers are smart enough to figure out what to do. Others have a mechanism to give them hints. These are usually the most efficient way to use templates. We are going to add our template types to the .cpp file.

A Word of Warning

Templates are powerful, but they are not magical. They do not give data types features that they did not have before. When you design or use a template you should be aware of what operations the data types you will use need to support.

---

4. Lab Exercise — Templates

Exercise 1 - Function Templating

For this part of the lab make a template out of the `myMax` function and test it on different data types.

- Start with the repl code provided to you.
- Compile and run the program to see how it works.
- Make a template out of `myMax`. Don't forget the return type.
- Modify the prototype appropriately.
- Test your `myMax` template on int, double, and string types.

When you are done your output should resemble this:

```
The max of 3 and 5 is 5
The max of 5.6 and 7.3 is 7.3
The max of donkey and apple is donkey
```

Exercise 2 - Class Templating

For this part of the lab you will make a template out of the `Matrix` class, test it on different data types, and add a generic sort member function.

Part 1: Setup

- Start with the repl code provided to you.
- Add the template types to `matrix.cpp`
- Template the definition of the `Matrix` class in `matrix.h`.
- Template the member functions in `matrix.cpp`.
- The Makefile is set up for you. Use it to compile and run the program to make sure everything is working.

Part 2: Extend

- Add two more 2D arrays to `tempMain.cpp` that hold a new data type.
- Create an instances of `Matrix` that can hold that data type and use the templated `demoTemplate` function to show that your templated class works.
- Test your changes.

Your completed exercise should resemble this:

```
Demonstrating with string matrix:

Matrix set to first array
Congra   y   ar
alm   don   La

Matrix incremented by second array
Congratulations   you   are
almost   done   the   Lab!

Demonstrating with int matrix:

Matrix set to first array
1   2   3
4   5   6

Matrix incremented by second array
7   7   7
7   7   7

Demonstrating with float matrix:

Matrix set to first array
1.6   2.5   3.4
4.3   5.2   6.1

Matrix incremented by second array
7.7   7.7   7.7
7.7   7.7   7.7
```

[an error occurred while processing this directive]