Lesson 11 - Templates
Programming for Code Reuse

Summary

In this lesson we demonstrate the power of templates for writing generic code.

New Concepts

Templates, static class fields, default arguments, inline methods.

Templates

One of the features that C++ provides is the Template syntax for writing generic, reusable code. For instance, let's imagine we have the class `MatrixInt` as shown in lines 1-15. This class provides a matrix abstraction for integers. Ideally, we would also like to use the same abstraction for other types, say floats, doubles, perhaps even our own defined classes. Rather than re-writing the same code for each type, we can use the power of templates to give us a versatile definition that allows us to declare Matrix objects of any type. In this lesson we will step through the process of converting the `MatrixInt` class to a generic template class, but first let's start by examining `MatrixInt` class.

We begin declaring the class as usual but this class possesses some features that we haven't seen before so let's explain them first. Note on lines 3 and 4 we have declared two constant integer types with the prefix keyword `static`. The `static` keyword denotes that these data members are defined with respect to the class, as opposed to objects of the class. What does that mean? Well, let's suppose we have two `MatrixInt` objects, 'a' and 'b'. It's possible that 'a' might possess the value 4 for the non-static field `x_size` while 'b' might possess the value 5; these variables are local to the objects. Both 'a' and 'b' however will always possess the same value for a static field like `default_x`; this variable belongs to the class.

Next, take a look at the constructor for `MatrixInt` on line 5. Note the syntax using our static fields `default_x` and `default_y` here. We are using these values as 'default arguments' for the `x` and `y` parameters. This basically means that we can omit values for these parameters when we instantiate an object of this class. If we do then the compiler will assign the values of the `default_x` and `default_y` fields.

Following the constructor and destructor declarations we provide two 'getter' methods `get_x_size` and `get_y_size` on lines 7 and 8 respectively. Note that we provide the definition of these methods in the actual class declaration. If we do this it is possible (although not guaranteed) that the compiler will 'inline' these functions. That means that whenever these functions are called in a program, the compiler will replace the function call with the actual function statements in order to avoid an execution jump. When our methods are short, this can provide an increase in efficiency with respect to execution time. Note that the actual keyword `inline` is available to explicitly label a function as such. There is no guarantee, however, that the compiler will actually 'inline' that method (rather, we simply provide the hint).

```
1  class MatrixInt {
2  public:
3     static const int default_x = 3;
4     static const int default_y = 3;
5     MatrixInt(int x = default_x, int y = default_y);
6     ~MatrixInt();
7     int get_x_size() const { return x_size;}
8     int get_y_size() const { return y_size;}
9     int get_element(int x, int y) const;
10    void set_element(int x, int y, int elem);
11 protected:
```

```
12      int** cells;
13      int x_size;
14      int y_size;
15  };
```

templates.cpp

Following the class declaration we provide the usual method definitions to complete our `MatrixInt` Class. On line 16 we provide the constructor, following by the destructor on line 25. As is customary, we assign our heap memory in the constructor and release it in the destructor. We use the `cells` data member (a pointer to an integer pointer) to hold our matrix elements. We initialise this variable by first assign an array of integer pointers on the heap (line 18). Then for each integer pointer, we assign an array of integers on the heap (lines 19 and 20). This provides us with a dynamically allocated 2-D array for our matrix abstraction.

Note that on line 21 we call a library function called `memset`. This function takes a pointer to a memory location, a value and a number of bytes. `memset` simply sets the number of bytes in a block of memory to the value supplied, beginning at the pointer. We use the `sizeof` operator to get the size of an `int` and multiply this by the size of the array. This gives us the size of the array in bytes.

On lines 32 and 36 we provide method definitions for our `get_element` and `set_element` methods.

```
16  MatrixInt::MatrixInt(int x, int y):
17  x_size(x), y_size(y) {
18      cells = new int*[x_size];
19      for(int i = 0; i < x_size; ++i)  {
20            cells[i] = new int[y_size];
21            memset(cells[i], 0, (y_size * sizeof(int)));
22      }
23  }
24
25  MatrixInt::~MatrixInt() {
26      for(int i = 0; i < x_size; ++i)  {
27         delete[] cells[i];
28      }
29      delete[] cells;
30  }
31
32  int MatrixInt::get_element(int x, int y) const {
33      return (cells[x][y]);
34  }
35
36  void MatrixInt::set_element(int x, int y, int elem) {
37      cells[x][y] = elem;
38  }
```

templates.cpp

On line 39 we have jumped to our main function. Here we instantiate an object of type `MatrixInt` on the heap setting `x_size` and `y_size` to 3 and 4 respectively. On line 40 we set the element at position (2,2) to the value of 8. Finally on line 41 we verify this took place by printing the value at this position to screen.

```
39      MatrixInt m(3,4);
40      m.set_element(2,2,8);
41      cout << "element at 2,2 is " << m.get_element(2,2) << "\n";
```

templates.cpp

Now let's convert our `MatrixInt` class to provide a generic template implementation that we can use for any type. Take a look at line 42. We begin our template declaration with the keyword `template` followed by `typename T` in angle brackets. `typename` is another C++ keyword and `T` is simply a label allowing us to substitute a specific type when the template is instantiated.

The following lines of the Matrix declaration bear little difference to `MatrixInt`. Note on line 51, however, that we have substituted the integer return type for `T` and again with the integer parameter of the `set_element` function on line 52. We are saying that the type that is returned and passed as a parameter here will be defined when this template is instantiated to a specific type, be it an `int`, a `double` or whatever.

Finally, on line 54 we change the cells type from an integer to `T` to denote that the 2-D array here will hold the type assigned to this template.

```
42  template <typename T>
43  class Matrix {
44  public:
45      static const int default_x = 3;
```

```
46    static const int default_y = 3;
47    Matrix(int x = default_x, int y = default_y);
48    ~Matrix();
49    int get_x_size() const { return x_size;}
50    int get_y_size() const { return y_size;}
51    T get_element(int x, int y) const;
52    void set_element(int x, int y, T elem);
53 protected:
54    T** cells;
55    int x_size;
56    int y_size;
57 };
```

templates.cpp

Now we have to define the methods of our template. Let's start with the constructor on line 59. Notice again we use the `template <typename T>` prefix here. We need this before each method we define in our template. Notice that the class Name `MatrixInt` has changed to `Matrix<T>` before the scope operator (::), denoting that the class is a template parametrised by type `T`. In the constructor we substitute the `int` type for `T` (lines 62, 64 and 65).

```
59 template <typename T>
60 Matrix<T>::Matrix(int x, int y):
61 x_size(x), y_size(y) {
62    cells = new T*[x_size];
63    for(int i = 0; i < x_size; ++i)  {
64          cells[i] = new T[y_size];
65          memset(cells[i], 0, (y_size * sizeof(T)));
66    }
67 }
68
69 template <typename T>
70 Matrix<T>::~Matrix() {
71    for(int i = 0; i < x_size; ++i)  {
72       delete[] cells[i];
73    }
74    delete[] cells;
75 }
```

templates.cpp

Finally we define our `get_element` and `set_element` methods for our Matrix template.

```
77 template <typename T>
78 T Matrix<T>::get_element(int x, int y) const {
79    return (cells[x][y]);
80 }
81
82 template <typename T>
83 void Matrix<T>::set_element(int x, int y, T elem) {
84    cells[x][y] = elem;
85 }
```

templates.cpp

Now let's jump to the main function and instantiate some variations on our matrix template. On line 87 we define another matrix of type `int`. We call the `set_element` and `get_element` methods on lines 88 and 89. On line 91 we instantiate a new matrix of type `double` to provide a floating point matrix. Notice we only have to change the parameter between the angle brackets now to give us a matrix that holds floating point values; on lines 92 and 93 we can call the same methods.

```
87    Matrix<int> m_int;
88    m_int.set_element(1,1,4);
89    cout << "element at 1,1 is " << m_int.get_element(1,1) << "\n";
90
91    Matrix<double> m_doub;
92    m_doub.set_element(2,2,8.421);
93    cout << "element at 2,2 is " << m_doub.get_element(2,2) << "\n";
```

templates.cpp

Here is the complete main function.

```
1  int main() {
2     MatrixInt m(3,4);
3     m.set_element(2,2,8);
4     cout << "element at 2,2 is " << m.get_element(2,2) << "\n";
5
6     Matrix<int> m_int;
7     m_int.set_element(1,1,4);
```

```
8     cout << "element at 1,1 is " << m_int.get_element(1,1) << "\n";
9
10    Matrix<double> m_doub;
11    m_doub.set_element(2,2,8.421);
12    cout << "element at 2,2 is " << m_doub.get_element(2,2) << "\n";
13    return 0;
14 }
```

main function

C++ Style

Templates are treated differently by the compiler than that of regular C++ classes. The compiler only generates code for a template if, and when a particular template is instantiated (with the type argument specifying the template type). At this point, the compiler also needs access to all the code which describes the functionality of that template (i.e. template declarations and definitions). You must therefore make all the code pertaining to a template available to the compiler when needed. Hence you would typically write all the code for a template, complete with definitions, in the template header file (see lines 1 to 9 below).

```
1 template <typename T>
2 class Foo {
3 // code omitted for brevity
4 };
5
6 template <typename T>
7 void Foo<T>::bar() {
8 // method declaration
9 }
```

Foo.h

Alternatively, if you don't like the thought of mixing declaration code with definitions, you can separate your template code into two header files and add an include statement after the class declaration of a template (see lines 10 and 11).

```
10 // mytemplate class declaration here
11 #include "FooDefinitions.h"
```

Foo.h

Whichever approach you use, the compiler must have access to all the code it requires at the point when a template is first instantiated (see lines 1 to 4, below).

```
1 #include "Foo.h"
2 //...code omitted for brevity
3 Foo<int> intFoo; // code for an integer type 'Foo' is generated
4 Foo<double> doubleFoo; // code for a double type 'Foo' is generated
```

somesource.cpp