Department of Computing

CS212: Object Oriented Programming

Class: BSCS-5AB

Lab10: Template functions and classes

Date: 09-05-2015

Time: 09:00 am – 12:00 pm / 2pm to 5pm

Instructor: Ms. Hirra Anwar

Lab10: Template functions and classes

Introduction

In this lab we will learn about templates in c++.

Objectives

To learn about the basics of templates in c++, template classes and templates functions.

Tools/Software Requirement

Visual Studio C++

Description

Template classes

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.

Template 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.

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.

Tasks

Perform following tasks

1) Create a function template that returns a maximum of two values.
2) Create template for a class calculator that has generic functions for performing multiplication and addition.
3) Create template for a function that calculates the average of the elements of an array passed to it and returns a double type number.
4) Create template for a function that takes two different types of arguments and prints their values.

5) Create template for a class 'Item' that set, gets and prints the value stored.
6) Create template for a function that swaps the values of two variables passed to it.

Deliverables

Source code for all the tasks on LMS.