Nov 16, 2015

Templates in C++

ABDUL AZIZ

FAST NATIONAL UNIVERSITY OF COMPUTER AND EMERGING SCIENCES |

Lab Instructor	**Abdul Aziz**
Course	**Computer Programming Lab**
Duration	**2hrs**

Objectives:

In this lab, following topics will be covered:

- Introduction to Templates
- Function Templates
- Class Templates
- Explicit Specialization

1. Templates

C++ has a mechanism called templates to reduce code duplication when supporting numerous data types. A C++ function or C++ class with functions which operates on integers, float and double data types can be unified with a single template function or class with functions which is flexible enough to use all three data types. This mechanism in C++ is called the "Template".

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. In other words you can say a template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

Function Templates:

In C++, **function templates** are functions that serve as a pattern for creating other similar functions. The basic idea behind function templates is to create a function without having to specify the exact type(s) of some or all of the variables. Instead, we define the function using placeholder types, called **template type parameters**. Once we have created a function using these placeholder types, we have effectively created a “function stencil”.

<u>Syntax:</u>

```
template <class type> ret-type func-name(parameter list)
{
  // body of function
}
```

<u>Example:</u>

```
#include <iostream>
using namespace std;
template <class T>
T square(T x)
{
   T result;
   result = x * x;
   return result;
};
int main()
{
   int    i, ii;
   float  x, xx;
   double y, yy;
   i = 2;
   x = 3.3;
   y = 4.4;

   ii = square<int>(i);
   cout << i << ": " << ii << endl;

   xx = square<float>(x);
   cout << x << ": " << xx << endl;
                 // Explicit use of template
   yy = square<int>(y);
   cout << y << ": " << yy << endl;
                         // Implicit use of template
   yy = square(y);
   cout << y << ": " << yy << endl;
}
```

- The template function works using either the explicit or implicit template expression square<int>(value) or square(value).
- In the template definition, "T" represents the data type. The compiler will generate the type specific functions required. This results in a more compact code base which is easier to maintain.
- The code and logic of the functions only has to be specified once in the template function and the parameter "T" is used to represent the argument type.

Another Example:

```
#include <iostream>
using namespace  std;
                    //Interchanges the values of variable1 and variable2.
                    //The assignment operator must work for the type T.
template<class T>
void swapValues(T &variable1, T &variable2)
{
     T temp;
     temp = variable1;
     variable1 = variable2;
     variable2 = temp;
}
int main( )
{
     int integer1 = 1, integer2 = 2;
   cout << "Original integer values are "<< integer1 << " " << integer2 << endl;

   swapValues(integer1, integer2);
   cout << "Swapped integer values are " << integer1 << " " << integer2 << endl;
   char symbol1 = 'A', symbol2 = 'B';
   cout << "Original character values are: "<< symbol1 << " " << symbol2 << endl;
   swapValues(symbol1, symbol2);
   cout << "Swapped character values are: " << symbol1 << " " << symbol2 << endl;
   return 0;

}
```

Class Templates

The concept of template functions can be extended to template classes. The general form of a generic class declaration is shown here:

Syntax:

```
template <class type> class class-name {
.
.
.
}
```

Here, **type** is the placeholder type name, which will be specified when a class is instantiated. You can define more than one generic data type by using a **comma-separated list.**

```
#include<iostream>
using namespace std;
template <class T> class calc
{
  public:
    T multiply(T x,T y);
    T add(T x,T y);
};
template <class T> T calc<T>::multiply(T x,T y)
{
  return x*y;
}
template <class T> T calc<T>::add(T x, T y)
{
  return x+y;
}
int main()
{
 calc<int> obj1;
 calc<float> obj2;
 cout<<"6+7 = "<< obj1.add(6,7)<<endl;
 cout<<"6 * 7 = "<< obj1.multiply(6,7)<<endl;
 cout<<"6.54 + 7.13 = "<< obj2.add(6.54,7.13)<<endl;
 cout<<"6.54 * 7.13 = "<< obj2.multiply(6.54,7.13)<<endl;
}
```

Explicit Specialization:

When you instantiate a template with a given set of template arguments the compiler generates a new definition based on those template arguments. You can override this behavior of definition generation. You can instead specify the definition the compiler uses for a given set of template arguments. This is called explicit specialization.

Example:

```
#include<iostream>
 using namespace std;
 template<class T = float, int i = 5> class A
 {
 public: A();
 int value;
 };
 template<> class A<> { public: A(); };
 template<> class A<double, 10> { public: A(); };
template<class T, int i> A<T, i>::A() : value(i)
 { cout << "Primary template, "
          << "non-type argument is " << value << endl;
 }
 A<>::A()
{
 cout << "Explicit specialization "<< "default arguments" << endl;
 }
 A<double, 10>::A()
{
 cout << "Explicit specialization "<< "<double, 10>" << endl;
 }

 int main()
{
A<int,6> x; A<> y;
 A<double, 10> z;
 return 0;
 }
```

EXERCISES

1. Write a program which creates a function template to show next in the series, provided two initial values in the series. Include two arguments of template type for the function template and set return type as void i.e.

void function (Type x, Type y);

For Example the function call:

- function(1,3) would print 5

2. A class template to construct two data members of template type and print their addition.
For Example:
- className obj(4,5) would print 9
- className obj2("Now","Then") would print NowThen

3. Write a template function that returns the average of all the elements of an array. The arguments to the function should be the array name and the size of the array (type int).