Templates

Sisoft Technologies Pvt Ltd
SRC E7, Shipra Riviera Bazar, Gyan Khand-3, Indirapuram, Ghaziabad
Website: www.sisoft.in Email:info@sisoft.in
Phone: +91-9999-283-283

A template is a blueprint or formula for creating a generic class or a function to work with generic types. It is the one of the feature which added to C++ recently.

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. Template allows a function or class to work on many different data types without being rewritten for each one.

A template can be used to create a family of classes or functions.

For ex: a class template for an array class would enable us to create array of various data types such as int array and float array. Similarly we can define a template for a function, say add() , that would help us create various versions of add() for adding int, float,double types values.

As we know that template allows us to define generic calsses. It is a simple process to create a generic class using a template with an anonymous type.

Syntax:

```
Template <class T>
Class class_name
{
…….
…….          //   Class member specification with anonymous
…….          //    type T whereever appropriate
……..
};
```

Note:

The class template definition is similar to an ordinary class definition except the prefix template<class T> and the use of type T.

This prefix tells the compiler that we are going to declare a template and use T as a type name in the declaration.

Template class

A class created from a class template is called a template class.

Syntax for defining an object of a template class is:

Class_name <type> object_name (arg_list);

This process of creating a specific class from a class template is called instantiation.

Class Template with multiple parameters:

We can use more than one generic data types in a class template. They are declared as a comma separated list within the template specification as shown below:

Syntax :

```
Template < class T1, class T2,.....>
Class class_name
{
.....
.....
.....
};
```

Program:

```
template <class T1, class T2>

    class Test
    {
       T1 a;
       T2 b;

  public:
     Test (T1 x , T2 y)
     {
        a = x;
        b = y;
     }

     void show ()
     {
cout << "a:\t"<<a <<"\n"<<"b:\t"<<b<<endl;
     }
    };
```

```
int main()
     {
cout<< "Instantiation the class template as
     test 1 with float and int data types"
     <<endl;

        Test <float, int> test1(1.23 , 123);
        test1.show();

cout<< "Instantiation the class template as
     test 2 with int and char data types"
     <<endl;

        Test <int , char> test2(100 , 'W');
        test2.show();
     }
```

Function Templates:

Like class templates, we can also define function templates that could be used to create a family of functions with different argument types.

Syntax :

```
Template < class T>
return_type function_name ( arguments of Type T )
{
{
…….
…….                    //    Class member specification with anonymous
…….                    //     type T whereever appropriate
……..
};
```

The function template syntax is similar to class template syntax except that defining the function instead of classes. Here we necessary to use the template parameter T as and when necessary in the function body and in its argument list.

Function Templates with multiple parameters:

Like tempaltes calsses, we can use more than one generic data type in the template statement using a comma separated list.

Syntax:

```
Template < class T1, class T2,..... >
Return_type function_name ( arguments of Type T1, T2, ..... )
{
{
.......
.......                    //    Body of Function
.......
.......
};
```

Program:

```
template <class T>
 T GetMax (T a, T b)
{
T result;
result = (a>b)? a : b;
return (result);
}
 int main ()
{
 int i=5, j=6, k;
 long l=10, m=5, n;
 k=GetMax<int>(i,j);
 n=GetMax<long>(l,m);
 cout << k << endl;
cout << n << endl;
return 0;
}
```

Output:

6

10

Overloading of Template Functions:

A template function can be overload either by template functions or ordinary functions of its name.

The procedure are given as below :

1) Call an ordinary function that has an exact match.

2) Call a template fucion that could be cretaed with an exact match.

3) Try normal overloading resolution to ordinary functions and call the one that matches.

An error is generated if no match found.

Note: There is no automatic conversions are applied to arguments on the template functions.

Program:

```
#include <iostream>
using namespace std;
template<class T>
void f(T x, T y)
 {
cout << "Template" << endl;
}
void f(int w, int z)
 {
cout << "Non-template" << endl;
}
 int main()
 {
 f( 1 , 2 );
 f('a', 'b');
 f( 1 , 'b');
 }
```

Output:

Non-template

Template

Non-template