Templates

10

10.1. INTRODUCTION

We are already acquainted with the knowledge of function overloading. As we know, overloaded functions have same name and are normally used to perform similar kind of operations (i.e. identical operations) on different data types. Let us first write a simple program to recall the concept of function overloading.

Program 10.1

```
/* 10.1.cpp */
#include <iostream>
using namespace std;
void show (int, int);
void show (double, double);
int main ( )
{
   show (2.6, 5.6);
   show (2, 5);
   show (2.6, 7.6);
   return 0;
}
void show (double a, double b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
void show (int a, int b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
```

Output:

```
a = 2.6
b = 5.6
a = 2
b = 5
a = 2.6
b = 7.6
```

From the above program, it can be observed that the 'show ()' function is overloaded. There are two function definition for 'show ()' function. Though both 'show ()' function definition has the same name, their parameter type is different. Hence, though the functions have same name, the function call gets mapped to the corresponding function definition during compile time by looking at the function arguments or parameters. Function overloading definitely gives an advantage that, the name of the function can be same, provided either among their type of the parameters, number of the parameters and the order of the parameters passed to the function should be different.

But a careful observation of overloaded functions as in our program 10.1 above will show us the disadvantage of 'overloaded functions'. That is, each overloaded function definition does identical tasks. But the only change with the overloaded functions is that, they are handling arguments of different data types to do the identical tasks. Or to say in other words, each overloaded function is doing the same task on different function parameters. This is a disadvantage because, the data types of function arguments are different, we are writing separate code for function definition for performing the same task. So, now the question is "Can we overcome this disadvantage by replacing the different function definitions doing similar kind of task by a single generic function which does the same job by handling function arguments of different data types ? The answer is yes! By using **Function Templates**.

10.2. FUNCTION TEMPLATES

Function templates is a concept in C++, which allows the programmer to write a generic function which is independent of a data type but embodies a common algorithm (i.e. does same kind of task) for different data types. Using function templates reduces the code size and ease the maintenance of code.

The syntax for creating a function template is as follows:

```
template <class T>
return_type <function_name> (parameters of type T)
{
   //function body
}
```

where,

`template`	→	keyword
`class T`	→	template type parameter enclosed within a pair of angle brackets (<>), also called generic data type)
`function_name`	→	name of the function

Let us convert the program 10.1 from 'program containing overloaded functions' into 'program containing function templates'.

Program 10.2

```
/* 10.2.cpp */
#include <iostream>
using namespace std;
template <class T>
void show (T a, T b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
int main ( )
{
   show (2.6, 5.6); // new function created
   show (2, 5);     // new function created
   show (2.6, 7. 6);  // new function not created
   return 0;
}
```

Output:

```
2.6
5.6
2
5
2.6
7.6
```

When the above program 10.2 is compared with the program 10.1, we can notice that, the two 'show ( )' function definitions of program 10.1 is replaced by just one generic function or function template in program 10.2.

In the above program, in 'main ( )' function, when the compiler encounters the statement,

```
show (2.6, 5.6);
```

the compiler understands that, the 'show ( )' function is trying to pass two double type arguments and therefore the compiler generates an actual function definition for the 'show ( )' function by replacing each occurrence of 'T' in the function template by the keyword 'double'.

Similarly, when the compiler encounters the statement,

```
show (2, 5);
```

the compiler understands that the 'show ( )' function is trying to pass two int type parameters and therefore the compiler generates an actual function definition for the 'show ( )' function by replacing each occurrence of "T" in the function template by the keyword 'int' .

Now, look at the third call to the function 'show ( )'. When the compiler encounters the statement,

```
Show (2.6, 7.6);
```

the compiler understands that, the function 'show ( )' is again trying to pass two double type parameters. But this time the compiler does not generate another function definition for this call, because, it has already generated a 'show ( )' function definition out of the function template by replacing "T" by the keyword 'double' before.

Therefore it is to be kept in mind that, for subsequent calls with the same data type, the compiler will not generate the definition again and again. The main reason behind it is that, during compilation, the compiler first looks for an exact function definition match to resolve a function call before looking for a function template. If it finds an exact match, it does not look for a function template. Hence, in our program, since the first call to 'show ( )' function generated the function definition, subsequent funciton calls with the same data type will not do so.

During compilation,we can visualize the source code for the program 10.2 as shown below.

```
template <class T>
void show (T a, T b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
void show (double  a, double b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
void show (int a, int b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
```

```
int main ( )
{
   show (2.6, 5.6);
   show (2, 5);
   show (3.7, 5. 7);
}
```

It is to be noted that, apart from built-in data types like int, double, etc. , the template parameter 'T' can also be replaced by user-defined data types. Technically speaking, *"the process of substituting or replacing the template type with various built-in and user-defined data types is known as* ***template instantiation.*** The function template has to be defined above 'main ( )' function only. It is not mandatory that, the name of the template parameter should be 'T' itself. It can be other than 'T' , as chosen by the programmer. Let us prove that with an example.

Program 10.3

```
/* 10.3.cpp */
#include <iostream>
using namespace std;
template <class SUB>
void show (SUB a, SUB b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
int main ( )
{
   show (2.6, 5.6);
   show (2, 5);
   show (3.7, 5. 7);
   return 0;
}
```

Output:

```
2.6
5.6
2
5
3.7
5.7
```

0.3. FUNCTION TEMPLATES WITH MULTIPLE PARAMETERS

Function templates can have more than one template parameter. In other words,

we can use more than one generic data type in the template statement as shown in the example below.

Program 10.4

```
/* 10.4.cpp */
#include <iostream>
using namespace std;
template < class T1, class T2 >
void show (T1 a, T2 b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
int main ( )
{
   show (2.6, 1);
   show ("SUBHASH", 1.1);
   return 0;
}
```

Output:

```
a = 2.6
b = 1
a = SUBHASH
b = 1.1
```

Let us look into another example program

Program 10.5

```
/* 10.5.cpp */
#include <iostream>
using namespace std;
template < class T1 >
void show (T1 a, int b)
{
   cout << "a = " << a << endl;
   cout << "b = " << b << endl;
}
int main ( )
{
   show (2.6, 1);
   show ("SUBHASH", 1);
   return 0;
}
```

Output:

```
a = 2.6
b = 1
a = SUBHASH
b = 1
```

10.4. OVERLOADING FUNCTION TEMPLATES

Like normal functions, even function templates can be overloaded. Overloaded function templates must differ in either the number of parameters it is expecting or the data types of these parameters or both. In such a scenario, the compiler resolves the function calls in the following order:

Step 1 : Firstly, the function call is resolved by trying to call an ordinary function definition that matches the function call exactly and the control returns.

Step 2 : Only if the above step 1 fails, the compiler tries to call a template function definition that can create a function definition that exactly matches with the call and returns.

Step 3 : Only if even the above step 2 fails, the compiler tries normal overloading resolution to ordinary functions and calls the function definition that matches and returns.

Step 4 : Only if even the above step 3 fails, an error message is generated.

Let us look at an example program containing overloaded function templates.

Program 10.6

```
#include <iostream>
using namespace std;
template <class T>
void show (T a)
{
   cout << "a = " << a << endl;
}
void show (int a)
{
   cout << "a = " << a << endl;
}
int main ( )
{
   show ('A'); // will create a template function
   show (2);  // will not create a template function
```

```
    return 0;
}
```

Output:

```
a = A
a = 2
```

10.5. CLASS TEMPLATES

Like function templates, we can also have **class templates.** Before learning *how to make use of class templates,* it is a nice idea to understand *why to make use of class templates.* For this, let us consider a situation, where, we make use of three set of classes. The member functions of each of the three classes does similar kind of operation, but the only difference being that, each are operating on different data members. This situation is shown in the below program 10.7.

Program 10.7

```
/* 10.7.cpp */
#include <iostream>
using namespace std;
class A
{
   private :
        int a, b;
   public :
        A ( )
        {
           a = 1;
           b = 2;
        }
        void show ( )
        {
           cout << "a = " << a << endl;
           cout << " b = " << b << endl;
        }
};
class B
{
   private :
        double a, b;
   public :
      B ( )
      {
        a = 2.6;
```

```
        b = 3.6;
    }
    void show ( )
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};
class C
{
  private :
        char a, b;
  public :
    C ( )
    {
        a = 'S';
        b = 'A';
    }
    void show ( )
    {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
    }
};
int main ( )
{
  A a;
  B b;
  C c;
  a.show ( );
  b.show ( );
  c.show ( );
  return 0;
}
```

Output:

```
a = 1
b = 2
a = 2.6
b = 3.6
a = S
b = A
```

As can be noticed from the above program that, there are three classes 'A', 'B', and 'C'. Each have a member function 'show ()' which is doing an identical task of displaying

data on the monitor. But the only difference is that, they are operating on different data types. This increases the program size and hence maintenance of the code becomes more tedious. Hence to avoid these kind of situations we make use of **class templates.** Using class templates, we can write a single generic class which does the job of all the three classes defined in the program 10.7.

The syntax for creating a class template is as follows:

```
template<class T>
class class_name
{
   //class definition
};
```

A class created from a class template is called as template class. The syntax for defining an object of a template class is as follows.

```
   class_name <data_type> objectname;
```

The syntax for creating a member function template is as follows.

```
template <class T>
return_type class_name <T> :: function_definition (<arguments>)
{
   // member function definition.
}
```

Let us take an example program to understand this.

Program 10.8

```
/* 10.8.cpp */
#include <iostream>
using namespace std;
template <class T>
class A
{
   private :
      T a, b;
   public :
      A (T x, T y)
      {
         a = x;
         b = y;
      }
      void show ( )
      {
         cout << "a = " << a << endl;
```

```
        cout << "b = " << b << endl;
    }
};
int main ( )
{
  A <int> a1 (1, 2);
  a1.show ( );
  A <double> a2 (1.1, 2.2);
  a2.show ( );
  return 0;
}
```

Output:

```
a = 1
b = 2
a = 1.1
b = 2.2
```

In the above program, in the 'main ( )' function, when the compiler encounters the statement,

```
A <int> a1 (1, 2);
```

the compiler creates a new class definition by replacing each occurrence of the template parameter 'T' in the template class definition by the keyword 'int', and normal operation is carried on. Similarly, when the compiler encounters the statement,

```
A <double> a2 (1.1, 2.2);
```

the compiler creates a new class definition by replacing each occurrence of the template parameter 'T' in the template class definition by the keyword 'double'.

Let us look into another example on member function templates.

Program 10.9

```
/* 10.9.cpp */
#include <iostream>
using namespace std;
template <class T>
class A
{
   private :
      T a, b;
   public :
      A (T x, T y)
      {
```

```
        a = x;
        b = y;
    }
    void show ( ) ;
};
template <class T>
void A <T> :: show ( )
{
   cout << "a = " << a << endl;
   cout << "b = " << b << end;
}
int main ( )
{
   A <int> a1 (1, 2);
   a1.show ( );
   A <double> a2 (1.1, 2.2);
   a2.show ( );
   A <int> a3 (2, 3);
   a3.show ( );
   return 0;
}
```

Output:

```
a = 1
b = 2
a = 1.1
b = 2.2
a = 2
b = 3
```

10.6. CLASS TEMPLATES WITH MULTIPLE PARAMETERS

Like function templates, the class templates can also have more than one parameter. The example for this is shown below.

Program 10.10

```
/* 10.10.cpp */
#include <iostream>
using namespace std;
template <class T1, class T2>
class A
{
   private :
      T1 a;
```

```
    T2 b;
  public :
    A (T1 x, T2 y)
    {
       a = x;
       b = y;
    }
    void show ( )
    {
       cout << "a = " << a << endl;
       cout << "b = " << b << endl;
    }
};
int main ( )
{
   A <int,double> a1 (1, 2.1);
   a1.show ( );
   A <double,int> a2 (1.1, 2);
   a2.show ( );
   A <int,double> a3 (2, 3.2);
   a3.show ( );
   return 0;
}
```

Output:

```
a = 1
b = 2.1
a = 1.1
b = 2
a = 2
b = 3.2
```

Let us look into another important example program where a class template have **non-type template argument.**

Program 10.11

```
/* 10.11.cpp */
#include <iostream>
using namespace std;
template <class T, int a >
class A
{
   private :
      int x, y;
   public :
```

```
    A  ( )
    {
       x = 5;
       y = a;
    }
    void show ( )
    {
       cout << "x = " << x << endl;
       cout << "y = " << y << endl;
    }
};
int main ( )
{
   A <int, 10>a1;
   a1.show ( );
   return 0;
}
```

Output:

```
5
10
```

10.7. NESTED CLASS TEMPLATES

Templates can be embedded or defined within a class or within class templates. Now, they are referred to as member templates. Member templates that are classes are referred to as nested class templates.

Nested class templates are declared as class templates inside the scope of the outer class. The outer class can itself be a normal class or a class template. The following programs will illustrate how the class templates work.

Program 10.12

```
/* 10.12.cpp */
#include <iostream>
using namespace std;
class Outer
{
   template <class T>
   class inner
   {
      public :
         T var1;
```

```
        inner ( )
        {
    }
    inner(T t)
    {
          var1 = t;
        }
};
  inner <int> i1;
  inner <char> i2;
public:
  Outer(int i, char c)
  {
    i1 = i;
    i2 = c;
  }
  void display( )
  {
    cout << i1.var1 << " " << i2.var1 << endl;
  }
};
int main( )
{
  Outer O(25, 'S');
  O.display ( );
  return 0;
}
```

Output:

25 S

10.8. ADVANTAGES OF USING TEMPLATES

- Reduced source code
- Less disk space needed to store the source files
- Easy to debug the program
- Good documentability

10.9. SUMMARY

- Function templates are generic functions which are independent of a data type but embody a common algorithm for different data types.
- Like function templates, we can also have class templates.
- Function templates can be overloaded.
- The process of substituting or replacing the template type with various built-in and user-defined data types is known as template instantiation.

10.10. EXERCISES :

1. What are Templates in C++ and why do we need it ?
2. What are the types of Templates available in C++ ?
3. Write a function 'add( )' to add two data of type integers and doubles using templates.
4. Write a simple program to demonstrate class templates.
5. Explain the internal working of Function templates.
6. Can class templates be nested ? Support your answer with an example.
7. Can class templates have multiple parameters ? Support your answer with an example.
8. What are the advantages of templates ?

* * * * *