Chapter 10

Template

Introduction

Templates are a mechanism that make it possible to use one function or class to handle many different data types. By using templates we can design a single class or function that operates on data of many types. This avoides creating a separate class or function for each data type. When templates are used with function they are known as function templates and when templates are used with class they are called as class templates.

Function Template

Suppose, we want to write a function which performs the addition of two nos. The function definition for integer data type can be written as

```
int add(int x,int y)
{
   return(x+y);
}
```

In the above function definition the function takes two arguments of type integer and returns a value of same type. If we want to perform the addition of two float nos. We have to write another add function. Similarly we have to write a separate function if we want to perform the addition of two long int nos. If the names of the function for different data types are same, then we just achieve function overloading.

We have to write the same function body again and again for different data types. It is a time consuming process. The program consumes more disk space. If any error occurs then we have to correct it in every function. It will be always better if we have to write only one function which will operate for different data types. Function Templates does exactly the same job.

Write a Program using Function Template to Addition of 2 Integer & 2 Float nos.

```
#include<iostream.h>
#include<conio.h>

template <class T>

T add(T x, T y)
{
   return(x+y);
}

main()
{
int a,b,c;
clrscr();
```

```
cout<<"Enter 2 Integer nos. = ";
cin>>a>>b;
c=add(a,b);
cout<<"Addition = "<<c<<endl;
float p,q,r;
cout<<"Enter 2 Float nos. = ";
cin>>p>>q;
r=add(p,q);
cout<<"Addition = "<<r;
getch();
return(0);
}
```

Write a Program using Function Template to find Smallest no of 2 Integer & 2 Float nos.

```
#include<iostream.h>
#include<conio.h>

template <class T>

T small(T x, T y)
{
   return(x<y?x:y);
}

main()
{
int a,b;
clrscr();
cout<<"Enter 2 Integer nos. = ";
cin>>a>>b;
cout<<"Smallest = "<<small(a,b)<<endl;
float p,q;
cout<<"Enter 2 Float nos. = ";
cin>>p>>q;
cout<<"Smallest = "<<small(p,q);
getch();
return(0);
}
```

Write a Program using Function Template to sort an Array of int & Float type.

```
#include<iostream.h>
#include<conio.h>

template <class T>

void sort(T a[])
{
   int i,j,temp;
```

```
   for(i=0;i<5;i++)
      for(j=i+1;j<5;j++)
      {
       if(a[i]>a[j])
       {
         temp=a[i];
         a[i]=a[j];
         a[j]=temp;
       }
      }
}

main()
{
int n[5],i;
clrscr();
cout<<"Enter 5 Integer nos. = ";
for(i=0;i<5;i++)
   cin>>n[i];
sort(n);
cout<<"Sorted Element = ";
for(i=0;i<5;i++)
   cout<<n[i]<<" ";
float m[5];
cout<<"\nEnter 5 Float nos. = ";
for(i=0;i<5;i++)
   cin>>m[i];
sort(m);
cout<<"Sorted Element = ";
for(i=0;i<5;i++)
   cout<<m[i]<<" ";
getch();
return(0);
}
```

Function Template with Multiple type Arguments

It is possible for us to create a function template to have arguments of different data types. In such cases for each data type, we have to create a separate template name.

Write a Program to Addition of Integer and Float nos. using Function Template.

```
#include<iostream.h>
#include<conio.h>

template <class T,class S>

S add(T x, S y)
{
   return(x+y);
}
```

```
main()
{
int a;
float b,c;
clrscr();
cout<<"Enter Integer nos. = ";
cin>>a;
cout<<"Enter Float nos. = ";
cin>>b;
c=add(a,b);
cout<<"Addition = "<<c<<endl;
getch();
return(0);
}
```

Class Template

It is possible to extend the concept of template to class also. By using the concept of class template it is possible to have a single class having data members of different data type. This allows us to create a single class instead of several classes for different data type members.

While declaring a data members of a class, the data types of the data members if not explicitly specified. Instead of that we will specify a template name that template name will work for different data types. The actual data type of data member is specified at the time of creation of objects of the class.

Write a Program to Demonstration of Class Template.

```
#include<iostream.h>
#include<conio.h>

template <class T>

class array
{
  private:
      T n[5];
  public:
      void getdata()
      {
       for(int i=0;i<5;i++)
          cin>>n[i];
      }
      void putdata()
      {
       for(int i=0;i<5;i++)
          cout<<n[i]<<" ";
      }
};
```

```
main()
{
array <int>a1;
array <float>a2;
clrscr();
cout<<"Enter Integer nos. =";
a1.getdata();
cout<<"Enter Float nos. =";
a2.getdata();
cout<<"Array Integer nos. =";
a1.putdata();
cout<<"\nArray Float nos. =";
a2.putdata();
getch();
return(0);
}
```

Member Function Templates

When we created a class template for array, all member function were defined as inline which was not necessary. We could have defined them outside the class as well. But remember that the member function of the template classes themselves are parameterized by the type argument (to their template classes) and therefore these functions must be defined by the function templates.

Syntax

```
template <class T>

Return Type Class Name<T> : : Function Name(arglist)
{
        // . . . . .
        Function Body
        // . . . . .
}
```

Write a Program to Demonstration of Member Function Template.

```
#include<iostream.h>
#include<conio.h>

template <class T>

class array
{
  private:
      T n[5];
  public:
      void getdata();
      void putdata();
};
```

```
template <class T>

void array<T> :: getdata()
{
   for(int i=0;i<5;i++)
   {
      cin>>n[i];
   }
}

template <class T>

void array<T> :: putdata()
{
   for(int i=0;i<5;i++)
      cout<<n[i]<<" ";
}

main()
{
array <int>a1;
array <float>a2;
clrscr();
cout<<"Enter Integer nos. =";
a1.getdata();
cout<<"Enter Float nos. =";
a2.getdata();
cout<<"Array Integer nos. =";
a1.putdata();
cout<<"\nArray Float nos. =";
a2.putdata();
getch();
return(0);
}
```

Template Arguments

A template can have multiple arguments. That is, in addition to the type argument T, we can also use other arguments such as string, function names, constant expression and built-in types.

e.g.

```
template <class T, int size>

class array
{
     T a[size];
     // . . . . .
     // . . . . .
};
```

This template supplies the size of the array as an argument. This implies that the size of the array is known to the complier at the compile time itself. The arguments must be specified whenever a template class is created.

e.g.

```
array<int,10> a1;
array<float, 5> a2;
array<char, 20> a3;
```

Exception Handling

Exceptions are errors that occur at run time. There are many reasons for runtime errors. Some of the most common reasons are

1. Falling short of memory.
2. Exceeding the bounds of an array.
3. Inability to open a file.
4. Division of a nos. by zero.

When such exceptions occurs, the programmer has to decide a specific way according to which the user can handle the exception the user can handle the exception (error). The various strategies of handling errors are displaying error messages on the screen or requesting the user to supply new values or simply terminating the program.

Exception Handling in C++

C++ provides a systematic object oriented approach for handling run time errors generated by C++ classes. The exception mechanism of C++ uses three keywords as

1. Throw
2. Catch
3. Try

Also we have to create a new kind of class known as an exception class.

e.g.

Suppose we have a program that works with objects of certain class. If during the execution of a member function of this class an error occurs, then this member function informs the program that an error has occurred. This process of informing is called Throwing & Exceptions. In the program we have to create a separate section of code to handle the error. This section of code is called an Exception Handle or a Catch Block. Any code in the program that uses the objects of the class is enclosed in a 'Try' Block. Errors generated in the Try Block are called in the Catch Block.

The general syntax of these blocks is as shown below.

```
class sample
{
        . . . .
        . . . .
    public:
```

```
        class error         //Exception Class
        {
        };
        get()
        {
            if(some error occur)
                throw error( )
        }
};

main( )
{
    sample s
    try
    {
        s.get( )
    }
    catch(sample : : error)
    {
        . . . .     // Do something about error
    }
}
```

Here sample is any class in which error might occur. An Exception class called error is specified in the public part of sample. In main function the statement which calls the function of the class is enclosed in a try block. If an error occurs in the member function, the keyword 'throw' followed by the constructor of the exception class throws the exceptions. When an exception is thrown control goes to the catch block and the errors are handle in the catch block.