Object Oriented Programming Using Cpp

Topic:

Templates In C++

Chapter 11: Templates in C++

11.1 Introduction

A significant advantage of C++ is code reusability, which eliminates redundant coding. An important feature of C++ known as templates also supports code reusability. It provides flexibility to the programmer in terms of efficient software development process. Before discussing templates in C++, we need to understand the necessity of templates.

11.2 Specific Programming vs Generic Programming

Till now the programs we have discussed in this subject are example of specific programming. This is because the user already knows the data type of the input used in the program. The programs where user already knows the input data type while writing the code, is known as specific program and the concept is specific programming.

Now, sometimes it may happen that the exact type of input is unknown at the time of coding i.e. user does not know the input type to be used. So, to develop a program where the input type is not known is the requirement of the user.

Formal definition of generic programming: The process where a generalized program is developed where any type of input can be used is known as generic programming. A generic program can work on any type of input i.e. user need not bother about the input data type while writing the code. The feature template, available in C++, supports the concept of generic programming.

11.3 Template in C++

A **template** in C++ allows an user to create a generalized function or class which can perform an operation on any type of input provided. A template uses generic data type which can handle a variety of data types during execution.

A template may be of following two types:

i. Function Template: It is also known as generic function. A generic function defines set of operations that will be applied to various types of data. A generic function has the type of data that it will operate upon passed to it as a parameter i.e.

A generic function can accept any type of input provided by the user. A generic function can be declared using following syntax:

Syntax:

```
template<class T>          //for one generic argument
return_type  function_name(T    arg_name)
{
        ……..;
        ……..;
}
```

Here, we can see that the syntax is similar to the normal function. The only difference is that this generic function contains a generic data type as argument. The exact input type is not known till the function is called. When user calls a generic function by providing exact input type then system replaces the generic type with the data type provided by the user and creates a copy of the function and executes the function. This copy of the function is created internally and it is unknown to the user.

Program 69: Write a program to implement generic function for swapping operation.

Code:

```
#include<iostream.h>
#include<conio.h>
template<class T>
void swap(T x,T y)
{
        T temp;
        temp=x;
        x=y;
        y=temp;
        cout<<x<<"  "<<y<<endl;
```

```
}

void main()
{
        int n1,n2;
        float f1,f2;
        char c1,c2;
        clrscr();
        cout<<"Enter two integers";
        cin>>n1>>n2;
        cout<<"For integer input"<<endl;
        swap(n1,n2);
        cout<<"Enter two float numbers";
        cin>>f1>>f2;
        cout<<"For float input"<<endl;
        swap(f1,f2);
        cout<<"Enter two characters";
        cin>>c1>>c2;
        cout<<"For character input"<<endl;
        swap(c1,c2);
        getch();
}
```

O/P:

Enter two integers

2 4

For integer input

4 2

Enter two float numbers

3.5 5.7

For float input

5.7 3.5

Enter two characters

A Z

For character input

Z A

Explanation: In the above program swap() function is a template function which contains a generic data type T. Now, when 1st function call is made with integer parameters system creates a copy of swap() which accepts integer parameter and performs the operation. During 2nd function call with float parameters, again a copy of swap() function is made which accepts float parameter and performs the operation. In 3rd function call with character parameters, again a copy of swap() function is created which accepts character parameter and performs the operation. These copies of functions are created internally and thus hidden from the user. So, we can see that the same swap function performs swapping for three different types of input which is the objective of code reusability.

Function with more than one generic type: It is possible to define a generic function with more than one generic data type in the template statement. These generic types are separated by comma.

Program 70: Write a program to implement a generic function with multiple generic data type.

Code:

```
#include<iostream.h>
#include<conio.h>
template<class T1,class T2>
void add(T1 x,T2 y)
{
        T1 res;
        res=x+y;
        cout<<"Result is "<<res<<endl;
```

```
}
void main()
{
        int n1;
        float f1;
        char c1;
        clrscr();
        cout<<"Enter an integer, a float and a character";
        cin>>n1>>f1>>c1;
        cout<<"1st function call"<<endl;
        swap(n1,f1);
        cout<<"2nd function call"<<endl;
        swap(f1,c1);
        cout<<"3rd function call"<<endl;
        swap(c1,n1);
        getch();
}
```


O/P:

Enter an integer, a float and a character

10 4.5 A

1st function call

Result is 14

2nd function call

Result is 69.5

3rd function call

Result is K

Explanation: Here, the add() function contains two generic data types T1 and T2. In main() function, an integer, a float and a character is taken as input. During 1st function call, add() function gets 1st parameter as integer and 2nd parameter as float i.e. T1 is replaced by int whereas T2 is replaced by float. Hence, the function performs the addition of an integer and float which gives a float number but since T1 type is integer, res data type is int due to which implicit type conversion will take place and result is printed as "Result is 14". Same procedure is followed for the rest two function calls.

Note: Like other functions, a generic function can also return a value. In this case the return_type must be a generic type.

ii. Class Template: It is also known as generic class. Like functions, classes can also be declared to operate on different data types. Such classes are known as class templates.

Syntax:

```
template<class T1>
class  <class_name>
{
        T1 data1;
};
```

In case of class template the association of the exact data type to the generic type is provided during the object declaration. It is done using following syntax **<class_name> <datatype> object_name;**

Program 71: Write a program to implement class template.

Code:

```
#include<iostream.h>
#include<conio.h>
template<class T>
class sample
{
        T n1,n2;
```

```
    public:

        void input()
        {
            cout<<"Enter n1 and n2";
            cin>>n1>>n2;
        }
        T add()
        {
            T sum;
            sum=n1+n2;
            return(sum);
        }
        void display()
        {
            cout<<"n1="<<n1<<"  "<<"n2="<<n2<<endl;
        }
};
void main()
{
    sample<int> ob1;
    ob1.input();
    cout<<"1st input is"<<endl;
    ob1.display();
    int r1=ob1.add();
    cout<<"Sum is "<<r1<<endl;
    sample<float> ob2;
```

```
        ob2.input();
        cout<<"2nd input is"<<endl;
        ob2.display();
        float r2;
        r2=ob2.add();
        cout<<"Sum is "<<r2<<endl;
        getch();
}
```

O/P:

Enter n1 and n2

2 4

1st input is

n1=2 n2=4

Sum is 6

Enter n1 and n2

2.5 3.4

2nd input is

n1=2.5 n2=3.4

Sum is 5.9

Explanation: In this program, class sample contains a generic data type T. The class contains 2 data member n1 and n2 of generic type T and few member functions. Now, during object declaration, for ob1 the input type was mentioned as int. So, T is replaced by int and the n1 and n2 is considered as integer. The member functions are executed using object ob1 and result is displayed. Similarly, in ob2 declaration, the input type is mentioned as float. So, the generic type T is replaced as float and n1,n2 becomes float. The member functions are executed using ob2 and result is displayed.

Important Note: In case of class template, the member functions are usually defined inside the class. This is because the member functions defined inside the class are

by default function template. In case, if the member functions are defined outside the class then each member function must be defined using template header so that they can work as function template.

11.4 Standard Template Library

The standard Template Library, commonly known as STL, is an advanced application of templates. It contains several in-built functions and operators that help the programmer to develop complex programs. The user only needs to include appropriate header file to use the function or operator from file like library functions.

STL is a new feature available in all advanced and famous C++ compilers. STL is vast and heterogeneous collection of reusable container classes which consists of vectors, lists, queues and stacks. STL is portable with various OS. STL was introduced by Meng Lee and Alexander Stepanov (Both from HP) and was accepted in 1994 to the standard C++ library.

11.4.1 STL Programming Model

STL is divided into three parts namely containers, algorithms and iterators. All these three components are associated with each other.

i. Containers: A container is an object that contains data or other objects. The standard C++ library has a number of container classes that allow a programmer to perform common tasks. These containers support generic programming and so they support different data types. All STL container classes are declared in namespace std. Container classes are of two types: **Sequence Containers and Associative containers.**

- **Sequence Containers** are created to allow sequential and random access to members. Ex: List, vector etc.
- **Associative Container** allows access to their elements through key. Ex: Map, Multimap, Set and Multiset.

ii. Algorithm: An algorithm is a technique used to handle the data stored in containers. The STL comprises approximately 60 standard algorithms which supports frequent and primary operations like copying, sorting, merging etc. The standard algorithms are defined in the header file <algorithm>.

iii. Iterators: An iterator is an object. It exactly behaves like pointers. It indicates or points to data element in a container. Iterators are utilized to move between the data items of containers. Like pointers, iterators can also be incremented or decremented. The task of the iterator is to link algorithms with containers.

Note: For examination point of view go through the templates in detail and basics of STL. Then if time permits go through the STL details from OOP with ANSI and Turbo C++ by Ashok N. Kamthane (Chapter 17).

Object Oriented Programming Using Cpp

Topic:

Dynamic Memory Management

Chapter 12 Dynamic Memory Management and Namespaces

12.1 Introduction

Memory is one of the critical resources of a computer system. One must be conscious about the efficient use of memory while developing an application. In programming, memory can be allocated in static allocation as well as dynamic allocation. In static allocation method, once the memory is allocated then it can't be changed irrespective of it is used or not. This leads to the need of allocation of memory dynamically. In C programming, dynamic memory manangement was performed with the help of some predefined functions known as malloc(), calloc(), realloc() and free(). Though functions are also valid in C++, C++ provides two important predefined operators known as new and delete operators for this purpose.

12.2 The new and delete operator

new operator: In C++, new operator is used to create objects as well as allocating memory to them dynamically. It can be used using following syntax:

Syntax:

```
Datatype    *ptr;
ptr= new    datatype;    or, ptr= new    datatype[size];
```

The above syntax will allocate the memory as per the datatype used and assign the first address to the pointer ptr. In another case, new operator will allocate the block of address (applicable for array) as per the data type and assign the first address to the pointer.

Delete operator: delete operator is used to delete the object as well as it release the memory allocated so that it may be used for other purpose.

Syntax:

```
delete      ptr;
or, delete  [ ]ptr;
```

In the 1st syntax, delete operator will delete the memory allocated which is pointed by ptr. In 2nd case, delete operator will delete the complete block of address allocated which is referred by pointer ptr. Here, important thing to note is that if pointer

ptrrefers to a block of address and if 1st syntax is used for deletion of memory then it deletes only the first address pointed by ptr and the rest of the memory block remains blocked as there is no way to refer that blocked memory address. This is known as **"memory leak".**

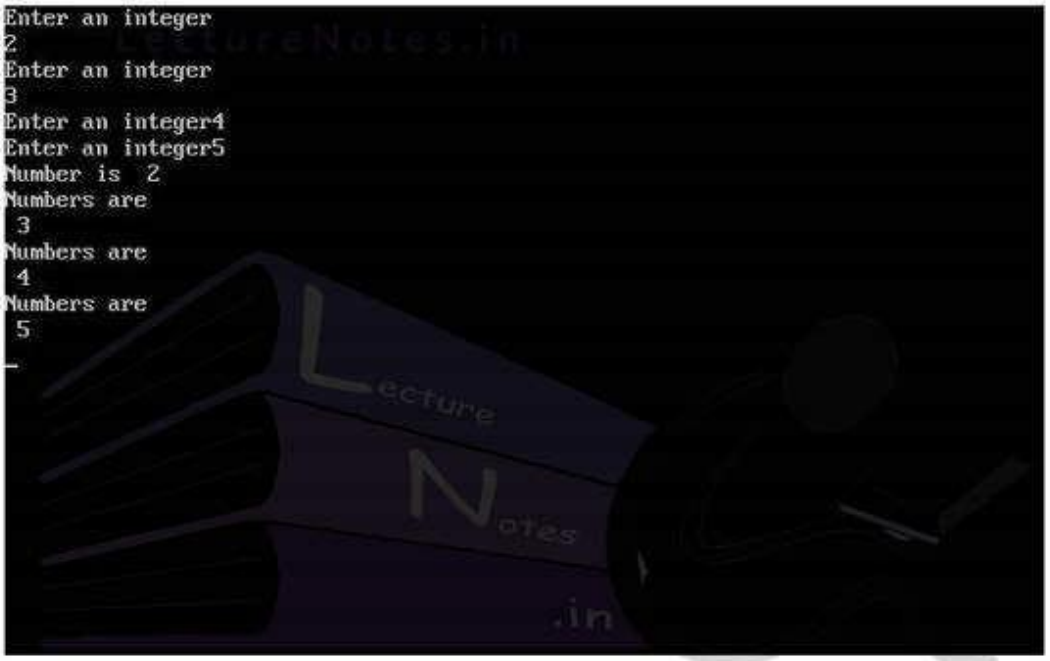
Program 72: Write a C++ program to illustrate the concept of dynamic memory management using new and delete operator.

Code:

```
#include<iostream.h>
#include<conio.h>
void main()
{
        int *ptr1,*ptr2;
        clrscr();
        ptr1=new int;
        ptr2=new int[3];
        cout<<"Enter an integer";
        cin>>*ptr1;
        for(int i=0;i<3;i++)
        {
                cout<<"Enter an integer";
                cin>>*(ptr2+i);
        }
        cout<<"Number is  "<<*ptr1<<endl;
        for(i=0;i<3;i++)
        {
                cout<<"Numbers are\n ";
                cout<<*(ptr2+i)<<endl;
        }
```

```
        delete ptr1;
        delete []ptr2;
        getch();
    }
```

O/P:



12.3 Dynamic Object

C++ supports dynamic memory management. So, it is possible to create and destroy an object dynamically. When an object is created using new operator, it is known as dynamic object. The syntax is as below:

Syntax:

```
<class_name>    *ptr;          //Declaration of object pointer
ptr=new    <class_name1>    //Dynamic Object
```

In the above syntax, new operator first creates a dynamic object and then memory is allocated to that object. The new operator returns the address of object and it is assigned to the pointer. In this case the class members can be accessed using pointer ptr with the help of, →, arrow operator. The dynamic object must be deleted using delete operator using the statement as shown below:

```
delete          ptr;
```

Program 73: Write a C++ program to implement a dynamic object

Code:

```
#include<iostream.h>
#include<conio.h>
class SAMPLE
{
        int n1;
public:
        void fun()
        {
                n1=10;
                cout<<"n1="<<n1<<endl;
        }
};
void main()
{
        clrscr();
        SAMPLE *ptr;
        ptr=new SAMPLE;
        ptr->fun();
        deleteptr;
        getch();
}
```

O/P:

n1=1

12.4 Some important pointer definitions

12.4.1 The this pointer: In C++, an object can access its own address using a pointer known as this pointer. The this pointer is an implicit parameter to all non-static member functions which are invoked using that object. However, since friend function is not a class member, it does not have this pointer.

12.4.2 Dangling pointer: When an object, which is referred using a pointer, is deleted without modifying the pointer value, the pointer is known as dangling pointer. It means that the pointer points to a memory location which is already deleted.

12.4.3 Wild pointer: A pointer which is not initialized is known as wild pointer. A wild pointer may create disturbance in the program as it may point to the garbage address. Due to wild pointer, program may show unexpected behaviour.

12.5 Namespace

Creating name is one of the most basic activities in programming. For example, variable names, function names or class names etc.

While writing large application based programs which involve several persons may cause problems due to the same name used.

Consider the following example:

```
mylib.h              //My library header file
    void display()
    {


    }
    Class BIG
    {


    };
Somelib.h            //Some other header file
    void display()
    {
```

```
}
Class BIG
{

};
```


Now, if both the header files are included in a program there would be a clash of names for display() function and class BIG as both header files have these names. C++ provides the solution to the above problem through a keyword called namespace.

Consider the same example but this time using namespace:

```
namespace   mylib.h              //My library header file
        void display()
        {

        }
        Class BIG
        {

        };
namespace   Somelib.h            //Some other header file
        void display()
        {

        }
        Class BIG
        {
```

```
};
```

In this case, the problem may not occur as the classes now become mylib::BIG and somelib::BIG respectively.

12.5.2 Defining a namespace: A name space can be defined using following **syntax:**

```
namespace        <name>
{
        //namespace members
}
```

Note:

i. Any thing defined inside namespace is considered as namespace member.

ii. The namespace members can also be defined outside the namespace definition using namespace name and scope resolution operator.

iii. A namespace definition must be global

12.5.3 Using a namespace

There are following two ways to use a namespace member:

i. Using scope resolution: We can specify any name in a namespace using the scope resolution operator, as shown below:

```
namespace  my_code
{
        int     m;
        void display(int n)
        {
                cout<<n<<endl;
        }
}
```

Now, if we want to access the variable m, then the statement should be

my_code: :m=10; //To access the variable m

Similarly, display() function can be used using my_code: :display(20);

ii. Using namespace statement: Use of scope resolution operator is not a suitable option to access a namespace member as for each access to namespace member we need to use scope resolution operator. An alternate option is to use namespace statement so that all the members defined inside that namespace can be used efficiently.

Syntax:

using namespace <name>;

Example:

```
namespace  my_code
{
        int     m;
        void display(int n)
        {
                cout<<n<<endl;
        }
}
```


The above namespace can be used as below:

```
using  namespace  my_code;
        m=10;
        display(20);
```


Here, both members are used without using any scope resolution symbol.

12.5.4 Nested namespace

When a namespace is defined inside another namespace then it is known as nested namespace.

Example:

```
namespace   EXTERNAL
{
    int     n1=6;
    namespace   INTERNAL
    {
        float   a=3.14;
    }
}
```

Here, to access the variable n1, statement is cout<<n1; whereas to access the variable a, the statement is cout<<EXTERNAL: :INTERNAL: :a;