PERIYAR INSTITUTE OF DISTANCE EDUCATION (PRIDE)

PERIYAR UNIVERSITY

SALEM - 636 011.

B.Sc. COMPUTER SCIENCE

SECOND YEAR

PRACTICAL – II : PROGRAMMING IN 'C++' USING OOPS

Prepared by:
Prof N. RAJENDRAN, M.C.A., M.Phil.
HOD of Computer Science,
Vivekanandha College of Arts and Sciences for Women,
Elayampalayam, Tiruchengode,
Namakkal (District) – 637 205.

FOREWORD

> "In human affairs we have reached a point where the problems that we must solve are no longer solvable without the aid of computers. I fear not computers but the lack of them"
>
> - **Issac Asimov**

Dear Students,

The importance of computers is felt in every field and it has become an integral part of our society. This booklet has been planned in a way that an in depth learning about practical programs in **Programming in C++ using OOPs.**

The list of practical has 7 programs. Mainly this booklet focused the concepts of OOPs (Object Oriented Programs) and how to implement it using C++ programs with output. It simply explained the programs with definitions, concepts and examples of OOPs, which will implement in program.

The first program classes and objects give simple introduction about classes and objects concept with example. The second program says about the concept of functions such as Inline, Friend, Virtual and Function with default argument.

The third program explained about the concept of special member function in c++ types of constructors such as Empty, Parameterized, Copy and Constructors with default arguments and destructors. The fourth program polymorphism deals the concept of Function and Operator overloading.

The fifth program Inheritance says about the concept and types of inheritance such as Single, Multilevel, Multiple, Hierarchical and Hybrid inheritance. The sixth program gives brief introduction about files and how to manipulate on file.

The seventh program explained about the generic programming concept Templates and its types such as Function, Class and Member Function templates.

PRIDE would be happy in you could make use of this learning material to enrich your knowledge and skills to serve the society.

SYLLABUS

i) **PRACTICAL – II**

ii) **Programming in 'C++' using OOPs**

List of Practical:

1. Classes and Objects
2. Functions
    a. Inline functions
    b. Friend functions
    c. Functions with default argument
    d. Virtual functions
3. Constructors and Destructors
    a. Empty constructor
    b. Parameterized constructor
    c. Constructors with default arguments
    d. Copy constructors
    e. Destructor
4. Polymorphism
    a. Function overloading
    b. Operator overloading
5. Inheritance
    a. Single
    b. Multilevel
    c. Multiple
    d. Hierarchical
    e. Hybrid
6. Files
7. Templates
    a. Function templates
    b. Class templates
    c. Member function templates

Programming in C++ using OOPS.

Section 1.02

Section 1.03 Classes and objects

Section 1.04 Class

A class is a way to bind the data and its associated functions together. It allows the data and functions to be hidden, if necessary, from external use. When defining a class, we are creating a new **abstract data type** that can be treated like any other build in data type.

Generally, a class specification has two parts.

1. Class declaration – It describes the type and scope of its member.
2. Class definition – It describes how the class functions are implemented.

Article II. Syntax of class

```
Class class_name
{
        private :
                Variable declaration;
                Function declaration;
        public :
                Variable declaration;
                Function declaration;
};
```

Where

Class – Keyword

Class_name – Valid user defined class name

Private, public – Visibility labels ended with colon (:)

Variables and functions – Class members

The body of a class should enclose within braces and terminated by a semicolon.

Article III. Object

Objects are variables of the type class. The objects are used to access class members from outside the class. The general format to create an object of type class is

Class-name objectname1, objectname2, …

EX:

```
//Class declaration
class student
{
 private :
        char name[20];
        int rno;
 public :
        void getinput();
        void putoutput();
};
//Object creation
student s1, s2, s3;
```

here s1, s2 and s3 are the object of type class student. These objects are used to access the member function and public data member of class.

Program – 1.

Program to implement Classes and objects in c++.

```
//Program to implement classes and object
#include <iostream.h>
#include <conio.h>
//class definition
class si_interest
{
  private : float principle;
            int year,rate; //rate should be percentage
  public : void acceptvalue(float,int,int); //funtion declaration
           float interest();
           void display(float);
};
//acceptvalue member function definition
void si_interest :: acceptvalue(float p,int y,int r)
```

```
{
  principle=p;
  year=y;
  rate=r;
}
//interest member function definition
float si_interest :: interest()
{
  return(principle*year*rate/100.0);
}
//display member function definition
void si_interest :: display(float si)
{
  cout<<"\nSimple interest  :  "<<si;
}
//Main function definition
int main()
{
   si_interest s; //create object s
   clrscr();
   cout<<"\n\tClasses and object implementation";
   cout<<"\n\t******* *** ****** **********\n";
   float p1;
   cout<<"\nEnter principle amount  :";
   cin>>p1;
   int y1,r1;
   cout<<"\nEnter no.of year   :";
   cin>>y1;
   cout<<"\nEnter rate of interest in percentage  :";
   cin>>r1;
   s.acceptvalue(p1,y1,r1); //invoke function using objetc
   float simple=s.interest();
   s.display(simple);
   getch();
```

```
    return 0;
}
```

ARTICLE IV. RESULT

Classes and object implementation

******* *** ****** ***********

Enter principle amount :10500

Enter no.of year :2

Enter rate of interest in percentage :12

Simple interest : 2520

Article V. Functions

a) Inline functions

An inline function is a function that is expanded in line when it is invoked. It is easy to make a function inline. Just add prefix the keyword inline to the function definition. All inline functions must be defined before they are called.

Syntax

```
inline function_header
 {
      function body;
 }
```

Inline keyword merely sends a request, not a command, to the compiler. The compiler may ignore this request if the function definition is too long or two complicated and compile the function as a normal function.

Some of the situations where inline expansion may not work are

1. For functions returning values, if a **loop**, a **switch** or a **goto** exists.
2. For functions not returning values, if a return statement exists.
3. If functions contain static variables.
4. If inline functions are recursive

b) Friend functions

Friend function is a special function in c++ that it provides a way to access the private member of the class. To make an outside function “friendly” to a class, we have to simply declare this function as a friend of the class.

Example :

```
Class test
{
        …..
        …..
    public :
            …..
            …..
             friend void func1();  //Friend function declaration
};
Section 5.01    Friend function definition
Void func1()
{
        function body;
}
```

Friend function call:

```
funcl();
```

The friend function declaration should be preceded by the keyword **friend**. It is defined elsewhere in the program like a normal c++ function. It can be declared as a friend in any number of classes. A friend function, although not a member function, has full access rights to the private members of the class. It can be invoked like a normal function without the help of any object.

c) Functions with default arguments

C++ allows us to assign default values to the function parameters when the function is declared. In such a case we can call a function without specifying all its arguments. The defaults are always added from right to left.

Function declaration:

```
Float amount(float principal, int year, float rate=0.12);
```

Function definition:

```
Float amount(float principal, int year, float rate)
{
    function body;
}
```

Function call

i) Float value = amount(5750.00,2)

This function call takes the default value to parameter "rate".

ii) Float val = amount(7550.00,3,0.15)

This function call overwrites the existing rate value as 0.15.

In function only the trailing arguments can have default values. We cannot provide a default value to a particular argument in the middle of an argument list.

Ex: float amount(float principal, int year=2, float rate); //Illegal declaration.

d) Virtual functions

When we use the same function name in both the base and derived classes, the function in base class is declared as ***virtual*** using the keyword **virtual** preceding its normal declaration. When a function is made virtual, c++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus by making the base pointer to point to different objects, we can execute different versions of the virtual function.

Program - 2.

Functions:

a) Program to implement Inline function

```
//Program to implement inline function
#include <iostream.h>
#include <conio.h>
//Class definition
class mean
{
private : int a,b;
public : void getinput(int x,int y)
{
a=x; b=y; //auto make inside function as inline
}
float average();
};
//Make outside function as inline
inline float mean :: average()
```

```
{
return((a+b)/2.0);
}
//Main funtion definition
int main()
{
mean m; //create an object m
clrscr();
int r,s;
cout<<"\nEnter two values \n";
cin>>r>>s;
m.getinput(r,s);  //invoke function using object
cout<<"\nMean value  :  "<<m.average();
getch();
return 0;
}
```

Result

```
      INLINE FUNCTION
      ****** ************
Enter two values
5
6
Mean value  :  5.5
```

Program - 2.

b) Program to implement Friend function

```
//Program to implement friend function
#include <iostream.h>
#include <conio.h>
class two; //forward class declaration
//Class one definiton
class one
{
   private : int a;
   public : void input(int x)
```

```
                {
                      a=x;
                }
                friend int add(one,two);
};
//class two definition
class two
{
   private : int b;
   public  : void input(int y)
                {
                    b=y;
                }
                friend int add(one,two);
};
//friend function definition
int add(one b1,two b2)
{
  return(b1.a+b2.b);
}
//main function definition
int main()
{
  one m; //create object m of class one
  two n; //create object n of class two
  clrscr();
  cout<<"\n\tFRIEND FUNCTION IMPLEMENTATION";
  cout<<"\n\t****** ******** **************";
  int i,j;
  cout<<"\nEnter i value  : ";
  cin>>i;
  m.input(i); //invoke class one function
  cout<<"\nEnter j value  :";
  cin>>j;
```

```
  n.input(j); //invoke class two function
  cout<<"\nAdded value  :  "<<add(m,n); //invoke friend function
  getch();
  return 0;
}
```

(A) RESULT

```
FRIEND FUNCTION IMPLEMENTATION
****** ******** **********************
Enter i value  : 5
Enter j value  :8
Added value  :  13
```

Program - 2.

c) Program to implement Functions with default argument

```
//Program to implement function with default arguments
#include <iostream.h>
#include <conio.h>
#include <math.h>
//Class definition
class compound_interest
{
   private : float principle;
             int year,rate;
   public :  void input(float p,int y,int r=12)//Default value to r
             {
                principle=p;
                year=y;
                rate=r;
             }
             float compound();
             void display(float c)
             {
               cout<<"\nCompound interest = "<<c;
             }
};
```

```
//Compound function definition
float compound_interest :: compound()
{
  return((principle*pow((1.0+rate/100.0),year))-principle);
}
//Main function definition
int main()
{
   compound_interest c1; //Object c1 created
   clrscr();
   cout<<"\n\tFUNCTION WITH DEFAULT ARGUMENT
IMPLEMENTATION";
   cout<<"\n\t******** **** ******* ******** **********";
   int p1,y1,r1;
   cout<<"\nEnter principle and year  :  ";
   cin>>p1>>y1;
   c1.input(p1,y1);//call function r takes default value
   float x=c1.compound();
   c1.display(x);
   cout<<"\n\nEnter principle, year and rate : ";
   cin>>p1>>y1>>r1;
   c1.input(p1,y1,r1);//call function r1 overwrite on r
   x=c1.compound();
   c1.display(x);
   getch();
   return 0;
}
```

Result

FUNCTION WITH DEFAULT ARGUMENT IMPLEMENTATION

******** **** ******* ******** **********

Enter principle and year :

5550

2

Compound interest = 1411.920044

Enter principle, year and rate :

6750

3

8

Compound interest = 1753.05603

Program - 2.

d) Program to implement Virtual function

```
//Program to implement virtual function
#include<iostream.h>
#include <conio.h>
//Base class definition
class base
{
   public :  virtual void display()
                {
                    cout<<"\nBase class display";
                }
                virtual void show()
                {
                    cout<<"\nBase class show";
                }
};
//Derived class definition
//base class public members inhertied as derived class public member
class derived : public base
{
   public : void display()
               {
                    cout<<"\nDerived class display";
               }
               void show()
               {
                   cout<<"\nDerived class show";
               }
};
//Main function definition
int main()
{
  clrscr();
```

```
  cout<<"\n\tVIRTUAL FUNCTION IMPLEMENTATION";
  cout<<"\n\t******* ******** **********\n";
  base b;
  derived d;
  base *ptr; //Base class pointer object
  //Base object address is assigned to base pointer object
  ptr=&b;
  ptr->display();//invoke base class display function
  ptr->show();//invoke base class show function
  //Derived object address is assigned to base pointer object
  ptr=&d;
  ptr->display();//invoke derived class display function
  ptr->show();//invoke derived class show function
  getch();
  return 0;
}
```

(B)RESULT

```
VIRTUAL FUNCTION IMPLEMENTATION
******* ******** **********
Base class display
Base class show
Derived class display
Derived class show
```

Constructors and Destructors

Constructor

C++ provides a special member function called the constructor, which enables an object to initialize itself when it is created. This is known as automatic initialization of objects. The constructor name is same as the class name. It is invoked whenever an object of its associated class is created. It should be declared in the public section. It has no return type and it does not return any value.

Article VI. Declaration and definition of constructor

Ex:

```
Class sample
{
        private :
                int m, n;
        public :
                sample();  //constructor declared
                ……
                ……
};

sample :: sample()  //constructor defined
{
        m=0;
        n=0;
}
```

Article VII. Invocation of constructor

```
sample s1, s2;      /* Here the object s1 and s2 of type sample are created.
        Now it invoke the constructor sample() automatically */
```

Article VIII. Types of constructors

a) Empty constructor (or) Default constructor
b) Parameterized constructor
c) Copy constructor
d) Dynamic constructor

a) Empty constructor (or) Default constructor

A constructor that accepts no parameters is called the empty or default constructor.

Declaration

```
Sample();
```

Definition

```
Sample::sample()
{
        ….
        ….
}
```

Invocation

```
Sample s ; // invoke empty constructor
```

b) Parameterized constructor

The constructors that can take arguments are called parameterized constructor.

Section 8.01 Declaration

```
Sample(int m, int n);
```

Definition

```
Sample::sample(int m, int n)
{
        ….
….
}
```

Invocation

Sample s(5, 10) ; // invoke parameterized constructor

c) Constructors with default arguments

We can possible to define constructors with default arguments. It may cause ambiguity for some cases.

Section 8.02 Declaration

Sample(int x=0);

Definition

Sample::sample(int x)

{

….

….

}

Invocation

Sample s ; // invoke parameterized constructor with default argument

We consider the default constructor sample::sample() and parameterized constructor sample::sample(int x=0) in same class. We have to create an object of class sample **sample s.** Now the ambiguity is whether to call constructor **sample::sample() or sample::sample(int x=0)**. So when we use a constructor carefully avoid such ambiguity.

d) Copy constructor

A copy constructor is used to declare and initialize an object from another object.

Section 8.03 Declaration

Sample(sample &);

Definition

Sample::sample(sample &s1)

{

….

….

}

Invocation

Sample s, ss(s) ; // ‘s’ invoke empty constructor and ‘ss’ invoke copy constructor

(a) Destructor

C++ also provides another member function called the destructor that destroys the objects when they are no longer required. The destructor name also same as the class name but is preceded by a tilde symbol. It never takes any argument nor does it return any value. It invoked implicitly by the compiler upon exit from the program or block or function as the case may be to clean up storage that is no longer accessible.

(i) Declaration

```
~sample();
```

Definition

```
Sample::~sample()
{
        ….
        ….
}
```

Invocation

Ex:

```
Void main()
{
        ….
        ….
        {
                sample s;
                ….
                ….
        } //Here it invoke destructor implicitly
}
```

Program - 3.

(ii) Constructors and Destructors

a) Program to implement Empty constructor

```
//Program to implement empty constructor
#include <iostream.h>
#include <conio.h>
#include <math.h>

//Class definition
class empty_constructor
{
  private :  int x,y;
  public  :  empty_constructor()//definition of empty constructor
                {
                    cout<<"\nEnter x and y value  : \n";
                    cin>>x>>y;
                }
                long int power();
                void display(long int p)
                {
                   cout<<"Value of x to the power y = "<<p;
                }
};
//Power function definition
long int empty_constructor :: power()
{
   return(pow(x,y));
}
//Main function definition
int main()
{
  clrscr();
  cout<<"\n\tEMPTY CONSTRUCTOR IMPLEMENTATION";
  cout<<"\n\t***** ********** **********";
```

```
  empty_constructor e; //Constructor invoked automatically
  long int m=e.power();
  e.display(m);
  getch();
  return 0;
}
```

(B) RESULT

EMPTY CONSTRUCTOR IMPLEMENTATION

***** ********** **********

```
Enter x and y value  :
5
7
Value of x to the power y = 78125
```

Program - 3.

b) Program to implement Parameterized constructor

```
//Program to implement parameterized constructor
#include <iostream.h>
#include <conio.h>

//Class definition
class parameter_constructor
{
  private :  int add,sub;
  public  :  //Definition of parameterized constructor
               parameter_constructor(int x,int y)
               {
                  add=x+y;
                  sub=x-y;
               }
               void display()
               {
                 cout<<"\nAdd Value = "<<add;
                 cout<<"\nSub value = "<<sub;
               }
```

```
};
//Main function definition
int main()
{
  clrscr();
  cout<<"\n\tPARAMETERIZED  CONSTRUCTOR  IMPLEMENTATION";
  cout<<"\n\t************* ********** **************";
  int a,b;
  cout<<"\nEnter a and b value  :\n";
  cin>>a>>b;
  //Arguments passed at the time of object creation
  parameter_constructor p(a,b); //Constructor invoked automatically
  p.display();
  getch();
  return 0;
}
```

(C) RESULT

```
PARAMETERIZED  CONSTRUCTOR  IMPLEMENTATION
************* ********** *************************
Enter a and b value  :
4
5
Add Value = 9
Sub value = -1
```

Program - 3.

c) Program to implement Constructors with default arguments

```
//Program to implement constructor with default argument
#include <iostream.h>
#include <conio.h>
//Class definition
class dargument_constructor
{
  private :  float principle;
             int year,rate;
  public  :  //constructor has default argumet r
             dargument_constructor(float p,int y,int r=10)
             {
                principle=p;
                year=y;
                rate=r;
             }
             float si();
             void display(float s)
             {
                cout<<"\nSimple interest = "<<s;
             }
};
//si function definition
float dargument_constructor :: si()
{
   return(principle*rate*year/100.0);
}
//Main function definition
int main()
{
  clrscr();
  cout<<"\n\tCONSTRUCTOR WITH DEFAULT ARGUMENT
IMPLEMENTATION";
```

```
  cout<<"\n\t********** **** ******* ******** **********\n";
  int p1,y1;
  cout<<"\nEnter principle and year value :\n";
  cin>>p1>>y1;
  //invoke three parameter constructor
  //Third argumetn already set as default
  dargument_constructor d(p1,y1);
  float m=d.si();
  d.display(m);
  getch();
  return 0;
}
```

(D)

(E)RESULT

```
CONSTRUCTOR WITH DEFAULT ARGUMENT IMPLEMENTATION
********** **** ******* ******** **************************
Enter principle and year value :
11550
2
Simple interest = 2310
```

Program - 3.

d) Program to implement Copy constructor

```
//Program to implement copy constructor
#include <iostream.h>
#include <conio.h>
//Class definition
class copy_constructor
{
  private :  int data1,data2;
  public  :  copy_constructor() //Default constructor definition
             {
                cout<<"\nEnter data1 and data2 value :\n";
                cin>>data1>>data2;
             }
             //Declaration of copy constructor
             copy_constructor(copy_constructor &);
             void display()
             {
               cout<<"\nData1 value = "<<data1;
               cout<<"\nData2 value = "<<data2;
             }
};
//copy_constructor definition
copy_constructor :: copy_constructor(copy_constructor & c1)
{
  data1=c1.data1*2;
  data2=c1.data2*3;
}
//Main function definition
int main()
{
  clrscr();
  cout<<"\n\tCOPY  CONSTRUCTOR  IMPLEMENTATION";
  cout<<"\n\t****  ***********  **************\n";
```

```
  copy_constructor c; //invoke default constructor
  //Object type argument passed at the time of object creation
  copy_constructor cc(c); //invoke copu constructor
  cout<<"\nValue of object c ";
  cout<<"\n~~~~~ ~~ ~~~~~~ ~";
  c.display();
  cout<<"\n\nValue of object cc";
  cout<<"\n~~~~~ ~~ ~~~~~~ ~~";
  cc.display();
  getch();
  return 0;
}
```

(F)

RESULT

```
      COPY CONSTRUCTOR IMPLEMENTATION
      ****  ***********  **********************
Enter data1 and data2 value :
7
3
Value of object c
~~~~~ ~~ ~~~~~~ ~
Data1 value = 7
Data2 value = 3

Value of object cc
~~~~~ ~~ ~~~~~~ ~~
Data1 value = 14
Data2 value = 9
```

Program 3:

e) Program to implement destructor.

```
//Program to implement the destructor concept
#include <iostream.h>
#include <conio.h>
int count=0;
//Class definition
class destructor
{
        public :
                destructor() //Constructor defintion
                {
                    count++;
                    cout<<"\nThe count value : "<<count;
                }
                ~destructor() //Destructor defintion
                {
                   count--;
                   cout<<"\nThe count value : "<<count;
                }
};
//Main function definition
int main()
{
    clrscr();
    cout<<"\n\t\tIMPLEMENT DESTRUCTOR CONCEPT";
    cout<<"\n\t\t********* ********** *******\n";
    cout<<"\nEnter into main program";
    {
        cout<<"\nEnter into block I";
        destructor d1, d2; //Constructor called here
        {
          cout<<"\nEnter into inner block I ";
          destructor d3, d4; //Constructor called here
```

```
            {
              cout<<"\nEnter into inner block II";
              destructor d5; //Constructor called here
              cout<<"\nEnd of inner block II";
            } //Destructor called here
            cout<<"\nEnd of inner block I";
          } //Destructor called here
          destructor d6; //Constructor called here
          cout<<"\nEnd of block I";
    } //Destructor called here
    cout<<"\nEnd of main program";
          getch();
          return 0;
}
```

Result

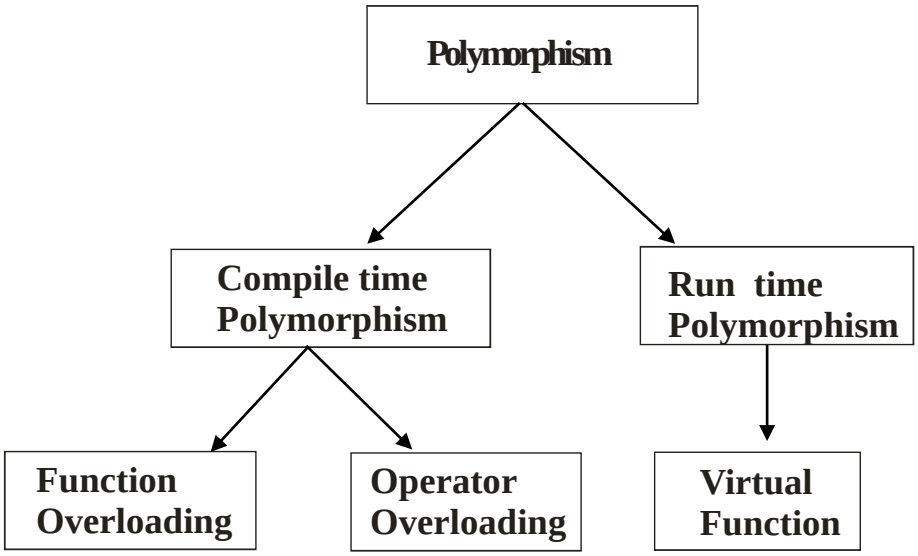
IMPLEMENT DESTRUCTOR CONCEPT

********* ********** ***************

Enter into main program
Enter into block I
The count value : 1
The count value : 2
Enter into inner block I
The count value : 3
The count value : 4
Enter into inner block II
The count value : 5
End of inner block II
The count value : 4
End of inner block I
The count value : 3
The count value : 2
The count value : 3
End of block I
The count value : 2
The count value : 1
The count value : 0
End of main program

Article IX. Polymorphism

Polymorphism is one of the special features in object-oriented programming (OOP). It means one name having multiple forms. There are two types of polymorphism, namely, compile time polymorphism and run time polymorphism.



Article X. Compile time polymorphism

Functions and operators overloading are examples of compile time polymorphism. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler knows this information at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early or static binding or static linking. It means that an object is bound to its function call at compile time.

Article XI. Function overloading

Overloading refers to the use of the same thing for different purposes. That is, we can have more than one function with the same name in our program to perform a variety of different tasks. The compiler matches the function call with the exact function code by checking the number and type of the arguments.

Ex:

```
Function decalaration
int add(int a, int b);              //Prototype 1
int add(int a, int b, int c);       //Prototype 2
double add(int p, double q)   //Prototype 3
Function calls
add(5, 7.5);            //Call prototype 3
add(7, 4, 9);           //Call prototype 2
add(5, 1);              //Call prototype 1
```

Article XII. Operator overloading

The mechanism of giving such special meanings to an operator is known as operator overloading.

We can overload almost all the C++ operators except the following

- Class member access operators (., .*)
- Scope resolution operator (::)
- Size of operator (sizeof())
- Conditional operator (?:)

Operator overloading is done with the help of a special function, called operator function, which describes the special task to an operator.

Section 12.01 Operator function syntax

```
return-type class-name :: operator op(op-arglist)
{
    Function body
}
```

where return-type - type of value returned by the specified operation

op – operator being overloaded. It preceded by the keyword operator.

operator op – function name.

Operator function must be either member functions (non-static) or friend functions. A basic difference between them is that a friend function will have only one argument for unary operators and two for binary operators, while a member function has no arguments for unary operators and only one for binary operators.

Section 12.02 Syntax for overloaded operator function calls

Article XIII. Friend function

operator op (x) //Unary operators

operator op (x, y) //Binary operators

Article XIV. Member function

op x (or) x op //Unary operators

x.operator op (y) //Binary operators

Article XV. Run time polymorphism

The virtual function is example of run time polymorphism. In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports run time polymorphism with the help of virtual functions. It is called late or dynamic binding because the appropriate function is selected dynamically at run time. Dynamic binding requires use of pointers to objects. It is useful in creating objects at run time.

Program - 4.

(i) Polymorphism

a) Program to implement Function overloading

```
//Program to implement function overloading
#include<iostream.h>
#include<conio.h>
float area(float a)
{
   return(a*a);
}
float area(float l,float b)
{
  return(l*b);
}
float area(int r)
{
  return(3.14*r*r);
}
//Main function definition
int main()
```

```
{
   clrscr();
   float a1;
   cout<<"\n\tFUNCTION OVERLOADING IMPLEMENTATION";
   cout<<"\n\t******** ********** **********\n";
   cout<<"\nEnter side of square  :\n";
   cin>>a1;
   cout<<"\nArea of square :  ";
   cout<<area(a1);//invoke one float argument function
   float l1,b1;
   cout<<"\nEnter side of rectangle  :\n";
   cin>>l1>>b1;
   cout<<"\nArea of rectangle  :  ";
   cout<<area(l1,b1);//invoke two float argument function
   int r1;
   cout<<"\nEnter radius of circle :\n";
   cin>>r1;
   cout<<"\nArea of circle  :  ";
   cout<<area(r1);//invoke one int argument function
   getch();
   return 0;
}
```

(B)RESULT

FUNCTION OVERLOADING IMPLEMENTATION

******** ********** **********

Enter side of square :

3.5

Area of square : 12.25

Enter side of rectangle :

2.7

1.2

Area of rectangle : 3.24

Enter radius of circle :

4

Area of circle : 50.240002

Program - 4.

b) Program to implement Operator overloading

```
//Program to implement operator overloading
#include <iostream.h>
#include <conio.h>
//Class definition
class complex
{
   float real,imag;
   public : complex() //Do_nothing constructor
            {
            }
            complex(float x,float y)//Assign values to data members
            {
               real=x;
               imag=y;
            }
            void operator -() //Unary operator oveloaded
            {
               real=-real;   //defined as member function
               imag=-imag;
            }
            //binary operator oveloaded using friend function
            friend complex operator +(complex,complex);
            void display();
};
//binary operator overloaded function definition
complex operator +(complex c1,complex c2)
{
   complex temp;
   temp.real=c1.real+c2.real;
   temp.imag=c1.imag+c2.imag;
   return(temp);
}
```

```
//Display member function definition
void complex :: display()
{
  if(imag>0)
     cout<<real<<"+"<<imag<<"j";
  else
     cout<<real<<imag<<"j";
  cout<<endl;
}
//Main function definition
int main()
{
   complex a1,a2,a3,a4;
   clrscr();
   cout<<"\n\tOPERATOR OVELOADING IMPLEMENTATION";
   cout<<"\n\t******** ********** **********\n";
   float r1,i1;
   cout<<"\nEnter first complex number real and imaginary value:\n";
   cin>>r1>>i1;
   a1=complex(r1,i1);
   cout<<"\nEnter second complex number real and imaginary value:\n";
   float r2,i2;
   cin>>r2>>i2;
   a2=complex(r2,i2);
   cout<<"\nDisplay complex number one before unary operator calling\n";
   a1.display();
   cout<<"\nDisplay complex number two before unary operator calling\n";
   a2.display();
   //Call binary operator overloaded friend function
   a3=operator +(a1,a2);
   cout<<"\nDisplay added value of complex number\n";
   a3.display();
   -a1;//call overloaded unary member function operator
   cout<<"\nDisplay complex number one after unary operator calling\n";
```

```
    a1.display();
    -a2;//call overloaded unary member funtion operator
    cout<<"\nDisplay complex number two after unary operator calling\n";
    a2.display();
    a4=operator +(a1,a2);//call binary oprator overloaded friend function
    cout<<"\nDisplay added value of complex number after unary ";
    cout<<"operator calling\n";
    a4.display();
    getch();
    return 0;
}
```

(C) RESULT

OPERATOR OVELOADING IMPLEMENTATION

******** ********** **********

Enter first complex number real and imaginary value:

-3

6

Enter second complex number real and imaginary value:

-1

4

Display complex number one before unary operator calling

-3+6j

Display complex number two before unary operator calling

-1+4j

Display added value of complex number

-4+10j

Display complex number one after unary operator calling

3-6j

Display complex number two after unary operator calling

1-4j

Display added value of complex number after unary operator calling

4-10j

Section 15.02 Inheritance

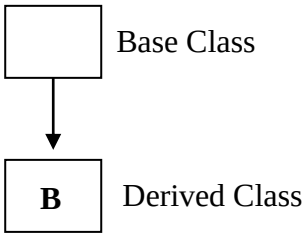
Reusability is highly important feature of OOP. C++ strongly supports the concept of ***reusability***. Creating new classes, reusing the properties of the existing ones, basically does it. The mechanism of deriving a new class from an old one is called ***inheritance (or derivation)***. The old class is referred to as the ***base class*** and the new one is called the ***derived class*** or ***subclass***. The derived class inherits some or all the properties of the base class.

Inheritance classified as the following types

a) Single inheritance
b) Multilevel inheritance
c) Multiple inheritance
d) Hierarchical inheritance
e) Hybrid inheritance

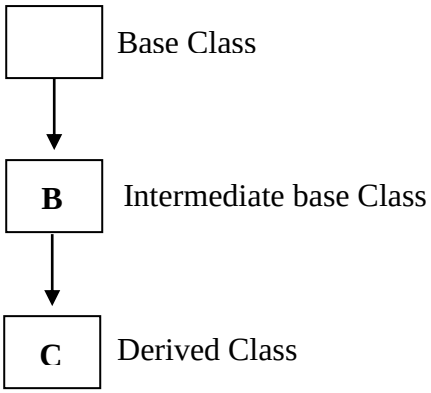
Section 15.03 Single inheritance

A derived class with only one base class is called single inheritance.



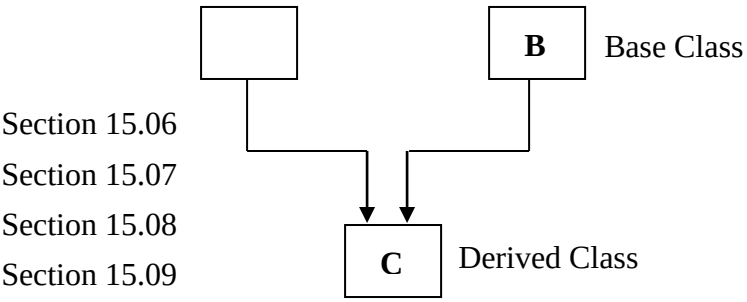
Section 15.04 Multilevel inheritance

The mechanism of deriving a class from another ‘derived class’ is known as multilevel inheritance.



Section 15.05 Multiple inheritance

A derived class with several base classes is called multiple inheritance.



Section 15.06

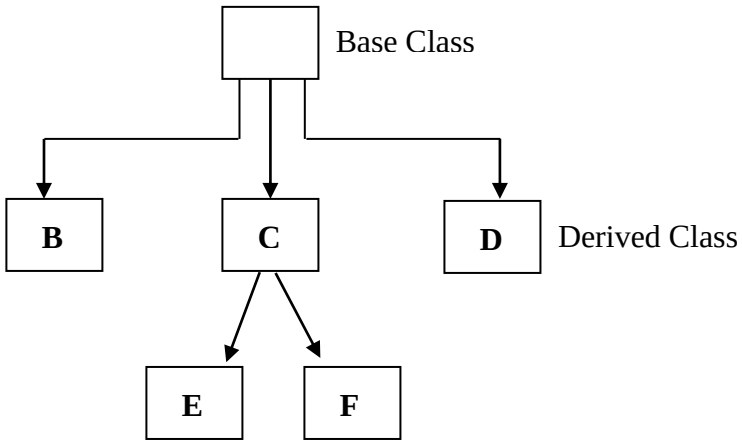
Section 15.07

Section 15.08

Section 15.09

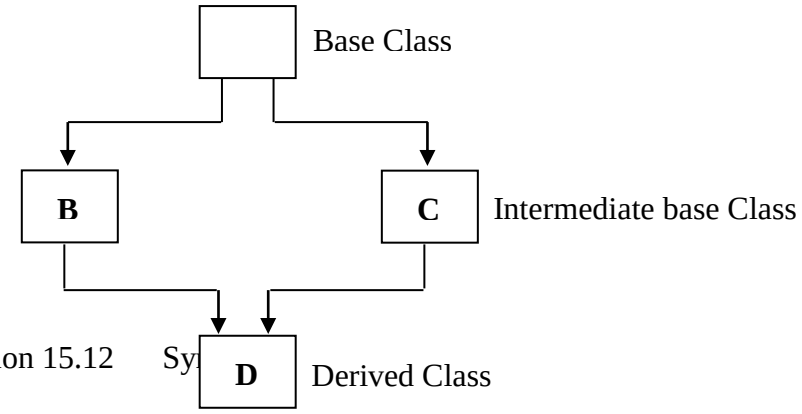
Section 15.10 Hierarchical inheritance

The classes may inherit the traits of one class. This is known as hierarchical inheritance.



Section 15.11 Hybrid inheritance

Combination of the above four types of inheritance is known as hybrid inheritance.



Section 15.12 Sy

class derived-class-name : visibility-mode base-class-name

{

Members of derived class

};

where **class** – keyword

derived-class-name, base-class-name – meaningful class name

: - indicated that the **derived-class-name** is derived from

base-class-name.

visibility mode – optional, if present may be either **private** or **protected**

or **public**. The default visibility mode is private.

(a) Visibility of inherited member

The given table represents the visibility of derived properties from base class to derived class.

Base class visibility	**Derived class visibility**		
	Private derivation	**Protected derivation**	**Public derivation**
Private	Not inherited	Not inherited	Not inherited
Protected	Private	Protected	Protected
Public	Private	Protected	Public

Program - 5.

(i) Inheritance

a) Program to implement Single inheritance

```
//Program to implement single inheritance
#include <iostream.h>
#include <conio.h>
//Base class definition
class single_base
{
   private : int a; //private members cannot inherited
   public : int b; //public members can inherited
                void input(int x,int y)
                {
                    a=x;
                    b=y;
                }
                int returna()
                {
                     return a;
                }
                int addab()
                {
                    return(a+b);
                }
};
//Derived class definition
class single_derived : public single_base
{
   private : int c;
   public  : void getc(int m)
                {
                    c=m;
                }
                int mul_ca()
```

```
                {
            //call base class private data using its own function
                return(c*returna());
                }
};
//Main function defintion
int main()
{
   clrscr();
   single_derived d;
   cout<<"\n\tSINGLE INHERITANCE IMPLEMENTATION";
   cout<<"\n\t****** ********** **********\n";
   cout<<"\nEnter a and b value to base class\n";
   int i,j;
   cin>>i>>j;
   //call base function using derived class object
   d.input(i,j);
   cout<<"\nEnter c value to derived class\n";
   int k;
   cin>>k;
   d.getc(k);
   cout<<"\nDisplay added value of a and b in base class\n";
   cout<<d.addab();
   cout<<"\nDisplay multiplied value of a in base class c in";
   cout<<" derived class\n";
   cout<<d.mul_ca();
   getch();
   return 0;
}
```

(b) RESULT

SINGLE INHERITANCE IMPLEMENTATION

****** ********** **********

Enter a and b value to base class

3

7

Enter c value to derived class

5

Display added value of a and b in base class

10

Display multiplied value of a in base class c in derived class

15

Program - 5.

b) Program to implement Multilevel inheritance

```
//Program to implement multilevel inheritance
#include <iostream.h>
#include <conio.h>
//Base class definition
class mlevel_student
{
   protected : int roll;
   public : //public members can inherited
                void getroll(int r)
                {
                    roll=r;
                }
                int return_roll()
                {
                     return roll;
                }
                void display()
                {
                    cout<<"\nRoll number  :  "<<roll;
                }
```

```
};
//Inherit mlevel_student class as public member of Derived class
class mlevel_mark : public mlevel_student
{
   protected : int m1,m2;
   public  : void getmark(int x,int y)
                {
                   m1=x;
                   m2=y;
                }
                void show_mark()
                {
                   cout<<"\nMark  1     :  "<<m1;
                   cout<<"\nMark  2     :  "<<m2;
                }
};
//Inherit mlevel_mark class as public member of derived class total
class mlevel_total : public mlevel_mark
{
     int total;
     public : void show_total()
                 {
                    total=m1+m2;
                    cout<<"\nTotal       :  "<<total;
                 }
};
//Main function defintion
int main()
{
   clrscr();
   mlevel_total ml;
   cout<<"\n\tMULTILEVEL INHERITANCE IMPLEMENTATION";
   cout<<"\n\t********** ********** **********\n";
   cout<<"\n\nEnter rollnumber of student :  ";
```

```
    int r;
    cin>>r;
    ml.getroll(r); //call mlevel_student class member
    cout<<"\nEnter mark1 and mark2  :\n";
    int s1,s2;
    cin>>s1>>s2;
    ml.getmark(s1,s2); //call mlevel_mark class function
    cout<<"\nDisplay Student details \n";
    ml.display();
    ml.show_mark();
    ml.show_total();
    getch();
    return 0;
}
```

(C) RESULT

```
MULTILEVEL INHERITANCE IMPLEMENTATION
********** ********** ************************
Enter rollnumber of student :  101
Enter mark1 and mark2  :
67
85
Display Student details
Roll number  :  101
Mark  1      :  67
Mark  2      :  85
Total        :  152
```

Program - 5.

c) Program to implement Multiple inheritance

```
//Program to implement multiple inheritance
#include<iostream.h>
#include<conio.h>
//Base class one defintion
class one
{
  protected : int a;
  public : void geta(int x)
                {
                  a=x;
                }
};
//Base class two definition
class two
{
   protected : int b;
   public : void getb(int y)
                 {
                   b=y;
                 }
};
//Derived class definition
class three : public one, public two
{
    public : void display()
                  {
                    cout<<"\nValue   a  :  "<<a;
                    cout<<"\nValue   b  :  "<<b;
                    cout<<"\nMultiplied value a and b  :  "<<a*b;
                  }
};
//Main function definition
```

```
int main()
{
  clrscr();
  three m;
  cout<<"\n\tMULTIPLE INHERITANCE IMPLEMENTATION";
  cout<<"\n\t******** ********** **********\n";
  int x,y;
  cout<<"\nEnter value a to base class one  :  ";
  cin>>x;
  m.geta(x); //call class one member
  cout<<"\nEnter value b to base class two  :  ";
  cin>>y;
  m.getb(y); //call class two member
  cout<<"\nDisplay values";
  cout<<"\n~~~~~~~ ~~~~~~\n";
  m.display(); //call class three member
  getch();
  return 0;
}
```

(D) RESULT

```
MULTIPLE INHERITANCE IMPLEMENTATION
******** ********** **********
Enter value a to base class one  :  7
Enter value b to base class two  :  4
Display values
~~~~~~~ ~~~~~~
Value   a  :  7
Value   b  :  4
Multiplied value a and b  :  28
```

Program - 5.

d) Program to implement Hierarchical inheritance

```
//Program to implement hierarchical inheritance
#include <iostream.h>
#include <conio.h>
//Base class defintion
class student
{
  protected : int rno;
  public : void getrno(int r)
           {
             rno=r;
           }
           void show_rno()
           {
               cout<<"\nStudent roll number  :  ";
               cout<<rno;
           }
};
//Arts class defintion
class arts : public student
{
  protected : int maxa;
  public  : void getmaxa(int m)
            {
               maxa=m;
            }
            void show_arts()
            {
               show_rno();
               cout<<"\nEligible mark for arts  :  "<<maxa;
            }
};
//engg class defintion
```

```
class engg : public student
{
  protected : int maxe;
  public : void getmaxe(int e)
             {
               maxe=e;
             }
};
//subclass1 of engg definition
class ece : public engg
{
   public  :  void show_ece()
               {
                    show_rno();
                    cout<<"\nCommon eligibility mark  :  "<<maxe;
                    float t=maxe-(maxe*(5.0/100.0));
                    cout<<"\nEligible mark for ECE  :  "<<t;
               }
};
//Subclass2 of engg defnition
class eee : public engg
{
    public : void show_eee()
              {
                   show_rno();
                   cout<<"\nCommon eligible mark  :  "<<maxe;
                   float t=maxe-(maxe*(7.5/100));
                   cout<<"\nEligible mark for EEE  :  "<<t;
              }
};
//Main function definition
int main()
{
   clrscr();
```

```
    arts a;
    ece ec;
    eee ee;
    cout<<"\n\tHIERARCHICAL INHERITANCE IMPLEMENTATION";
    cout<<"\n\t************ *********** **************\n";
    cout<<"\nEnter roll number  :  ";
    int r,ar,egg;
    cin>>r;
    a.getrno(r);
    ec.getrno(r);
    ee.getrno(r);
    cout<<"\nEnter eligible mark for arts   :  ";
    cin>>ar;
    a.getmaxa(ar);
    cout<<"\nEnter eligible mark for engineering :  ";
    cin>>egg;
    ec.getmaxe(egg);
    ee.getmaxe(egg);
    cout<<"\nDisplay arts eligibility detail";
    cout<<"\n~~~~~~~~ ~~~~ ~~~~~~~~~~~~ ~~~~~~";
    a.show_arts();
    cout<<"\n\nDisplay ECE eligibility detail";
    cout<<"\n~~~~~~~~ ~~~ ~~~~~~~~~~~~ ~~~~~~";
    ec.show_ece();
    cout<<"\n\nDisplay EEE eligibility detail";
    cout<<"\n~~~~~~~~ ~~~ ~~~~~~~~~~~~ ~~~~~~";
    ee.show_eee();
    getch();
    return 0;
}
```

(e) RESULT

HIERARCHICAL INHERITANCE IMPLEMENTATION

********** ********** **********

Enter roll number : 105

Enter eligible mark for arts : 65

Enter eligible mark for engineering : 86

Display arts eligibility detail

~~~~~~~ ~~~~ ~~~~~~~~~~ ~~~~~~

Student roll number : 105

Eligible mark for arts : 65

Display ECE eligibility detail

~~~~~~~ ~~~ ~~~~~~~~~~ ~~~~~~

Student roll number : 105

Common eligibility mark : 86

Eligible mark for ECE : 81.699997

Display EEE eligibility detail

~~~~~~~ ~~~ ~~~~~~~~~~ ~~~~~~

Student roll number : 105

Common eligible mark : 86

Eligible mark for EEE : 79.550003
~~~~~~~

Program - 5.

e) Program to implement Hybrid inheritance

```
//Program to implement hybrid inheritance
#include <iostream.h>
#include <conio.h>
//Base class definition
class hybrid_student
{
   protected : int roll;
   public : void getroll(int r)
                {
                   roll=r;
                }
                int return_roll()
                {
                    return roll;
                }
                void display()
                {
                   cout<<"\nRoll number   :  "<<roll;
                }
};
//Inherit hybrid_student class as public member of Derived class
class hybrid_mark : public hybrid_student
{
   protected : int m1,m2;
   public  : void getmark(int x,int y)
                {
                   m1=x;
                   m2=y;
                }
                void show_mark()
                {
                   cout<<"\nMark  1       :  "<<m1;
```

```
                cout<<"\nMark  2       :  "<<m2;
              }
};
//Class hybrid_sports definition
class hybrid_sports
{
  protected : int point;
  public : void getpoint(int p)
              {
                point=p;
              }
              void show_point()
              {
                cout<<"\nSports point  :  "<<point;
              }
};
//Inherit hybrid_mark and hybrid_sports class to derived class total
class hybrid_total : public hybrid_mark, public hybrid_sports
{
    int total;
    public : void show_total()
               {
                  total=m1+m2+point;
                  cout<<"\nTotal        :  "<<total;
               }
};

//Main function defintion
int main()
{
   clrscr();
   hybrid_total h;
   cout<<"\n\tHYBRID INHERITANCE IMPLEMENTATION";
   cout<<"\n\t****** ********** **********\n";
```

```
    cout<<"\n\nEnter rollnumber of student :  ";
    int r;
    cin>>r;
    h.getroll(r); //call mlevel_student class member
    cout<<"\nEnter mark1 and mark2  :\n";
    int s1,s2;
    cin>>s1>>s2;
    h.getmark(s1,s2); //call mlevel_mark class function
    cout<<"\nEnter sports point   :  ";
    int s;
    cin>>s;
    h.getpoint(s);
    cout<<"\nDisplay Student details \n";
    h.display();
    h.show_mark();
    h.show_point();
    h.show_total();
    getch();
    return 0;
}
```

(f) RESULT

```
HYBRID INHERITANCE IMPLEMENTATION
****** *********** **********************
Enter rollnumber of student :  107
Enter mark1 and mark2  :
55
89
Enter sports point   :  7
Display Student details
Roll number   :  107
Mark  1       :  55
Mark  2       :  89
Sports point  :  7
Total         :  151
```

Article XVI. Files

A file is a place in the disk where a collection of related data is stored permanently. We can perform read and write operation on file whenever necessary that it means used to make data communication between the program and a disk file. A file is a collection of related information ordered as records. A record is nothing but a collection of fields.

General syntax to open a file

```
File-stream-class stream-object;
Stream-object.open ("file-name", mode);
```

Where

file-stream-object –either ofstream or ifstream or fstream.

Stream-object – valid user specified object name.

open() –member function.

file-name – valid user defined file name.

mode – this is optional one. It may be any one of or combination of the following modes.

ios :: app	ios :: in
ios :: ate	ios :: out
ios :: binary	ios :: nocreate
ios ::trunc	ios :: noreplace

Section 16.01 General syntax to close a file

```
stream-object.close();
```

Program : 6

Program to implement file concept

```
//Program for file implementation
#include <iostream.h>
#include <conio.h>
#include<fstream.h>

//Class definition
class elig
{
char name[10];
int age,w;
public:
        void getdata();
        void putdata();
};

//Getdata function defintion
void elig :: getdata()
{
cout<<"Enter the Name : ";
cin>>name;
cout<<"Enter the Age  : ";
cin>>age;
}

//Putdata function definition
void elig ::putdata()
{
if (age<18)
{
cout<<"Name : \t"<<name<<endl;
cout<<"Age  : \t"<<age<<endl;
cout<<name<<" is not eligible to vote \n";
```

```
w = 18 - age;
cout<<name<<" has to wait for "<<w;
cout<<" year to vote \n\n";
}
else
{
cout<<"Name : \t"<<name<<endl;
cout<<"Age  : \t"<<age<<endl;
cout<<name<<" is eligible to vote \n\n";
}
}

//Main funtion definition
void main()
{
clrscr();
int i;
elig e[4];
fstream outf; //file stream object outf created
outf.open("outdata.dat", ios::in | ios::out);  //outdata.dat file opened
cout<<"\t\t INPUT \n";
cout<<"\t\t ***** \n";
for(i=0; i<3; i++)
{
e[i].getdata();
outf.write( (char*) &e[i], sizeof(e[i]) ); //write data on file
}
outf.seekg(0);
cout<<"\n\n\n\t\t OUTPUT \n";
cout<<"\t\t*******\n";
for(i=0; i<3; i++)
{
outf.read( (char*) &e[i], sizeof(e[i]) ); //read data from file
e[i].putdata();
```

```
}
getch();
}
```

Result

```
            INPUT
            *****
Enter the Name : Vishnu
Enter the Age  : 41
Enter the Name : Meera
Enter the Age  : 16
Enter the Name : Radha
Enter the Age  : 23

            OUTPUT
           *******
Name :  Vishnu
Age  :  41
Vishnu is eligible to vote
Name :  Meera
Age  :  16
Meera is not eligible to vote
Meera has to wait for 2 year to vote
Name :  Radha
Age  :  23
Radha is eligible to vote
```

Article XVII. Templates

Template is a new concept included in C++. It is used to define generic classes and functions and thus provides support for generic programming. Generic programming is an approach where generic types are used as parameters in algorithms so that they work for a variety of suitable data types and data structures.

Templates allow to generate a family of classes or a family of functions to handle different data types. A template can be considered as a kind of macro. When an object of a specific type is defined for actual use, the

template definition for that class is substituted with the required data type. Since a template is defined with a parameter that would he replaced by a specified data type at the time of actual use of the class or function, the templates are sometimes called parameterized classes or functions.

We have three types of templates

a) Function templates
b) Class templates
c) Member function templates

Article XVIII. Function templates

It is used to create a family of functions with different argument types.

Section 18.01 General format

```
template <class T>
return-type function-name (arguments of type T)
{
        Body of function
        with type T
        whenever appropriate.
}
```

Article XIX. Class templates

It is used to create a generic class using a template with an anonymous type.

General format

```
template <class T>
        class class-name
        {
                class member specification
                with anonymous type T
                wherever appropriate.
        };
```

(a) Syntax for defining an object of a template class

```
class-name <type> object-name (arglist);
```

(i) Member function templates

Member functions of a class template must be defined as function templates using the parameters of the class template. Those functions are known as template member functions.

General format

```
        template <class T>
return-type class-name <T> :: function-name (arglist)
{
        Body of function
        with type T
        whenever appropriate.
}
```

Program : 7

a) Program to implement function template concept in c++.

```
//Program for function template
#include <iostream.h>
#include <conio.h>
//template function definition
template <class t>
void interchange(t&a, t&b)
{
  t temp;
  temp=a;
  a=b;
  b=temp;
}
//input function definition
void input(int m,int n, float p, float q)
{
  cout<<"\nFUNCTION TEMPLATE";
  cout<<"\n******** ********\n";
  cout<<"\nDisplay m & n before interchange \n";
  cout<<m<<endl<<n;
  interchange(m,n);
  cout<<"\nDisplay m & n after interchange \n";
  cout<<m<<endl<<n;
  cout<<"\nDisplay p & q before interchange \n";
  cout<<p<<endl<<q;
  cout<<"\nDisplay p & q after interchange \n";
  interchange(p,q);
  cout<<p<<endl<<q;
}
//Main program definition
int main()
{
        clrscr();
```

```
        input(5,10,7.5,9.8);
        getch();
        return 0;
}
```

Result

```
FUNCTION TEMPLATE
******** **********
Display m & n before interchange
5
10
Display m & n after interchange
10
5
Display p & q before interchange
7.5
9.8
Display p & q after interchange
9.8
7.5
```

Program : 7

b) Program to implement class template

```
//Program for class template
#include <iostream.h>
#include <conio.h>
//Class template definition
template<class t1, class t2>
class vector
{
   t1 a;
   t2 b;
   public : vector(t1 m, t2 n)
                {
                    a=m;
                    b=n;
                }
                void display()
                {
                   cout<<a<<"\t"<<b<<endl;
                }
};
//Main function definition
int main()
{
   clrscr();
   cout<<"\n\tCLASS TEMPLATE";
   cout<<"\n\t***** ********\n";
   vector<int, float>s1(100,5.7);
   vector<char *, int>s2("Gandhi",87);
   cout<<"\nInteger, float values : ";
   s1.display();
   cout<<"\nCharacter, integer values : ";
   s2.display();
   getch();
```

```
   return 0;
}
```

Result

```
      CLASS TEMPLATE
      ***** ***********
Integer, float values : 100     5.7
Character, integer values : Gandhi      87
```

Program : 7

c) Program to implement member function template

```
//Program for member function template
#include <iostream.h>
#include <conio.h>
template<class T>
class student
{
    T d;
    public : void getdata();
                void display();
};

template<class t>
void student<t>::getdata()
{
   cout<<"\nEnter the data : ";
   cin>>d;
}
template<class t>
void student<t>::display()
{
  cout<<d<<endl;
}
//main function definition
int main()
{
```

```
    clrscr();
    cout<<"\n\tMEMBER FUNCTION TEMPLATE";
    cout<<"\n\t****** ******** ********\n";
    student<int>s;
    student<int>s1;
    student<int>s2;
    s.getdata();
    s.display();
    s1.getdata();
    s1.display();
    s2.getdata();
    s2.display();
    getch();
    return 0;
}
```

Result

```
        MEMBER FUNCTION TEMPLATE
        ****** ******** ****************
Enter the data : 20
20
Enter the data : 30
30
Enter the data : 40
40
```