UNIT II

SYLLABUS

Operator overloading – friend functions- type conversions- templates - Inheritance – virtual functions- runtime polymorphism.

OPERATOR OVERLOADING

- Defining a special meaning to an existing operator is called as operator overloading.
- Adding two variables of the basic data types (built-in data types such as int float etc) can be made successfully without any issue.
- But, to perform the same addition operation on the user-defined data type (objects of the class), C++ has the ability to provide a special meaning to the operators.
- The operator can be overloaded by using an **Operator Function,** which describes the task to be performed.
- Operators are overloaded in C++ by creating the operator function either as a member function or as a friend function of the class

Syntax for Operator Function:

```
return_type operator op(arg list)

{
// define the task;

}
```

UNARY OPERATOR OVERLOADING:

- Unary operator overloading operates with just **one operand of user-defined type (object) and one operator.**
- The unary operator function can either be the member function of the class or it can be a friend function (non-member function).
- If the unary overloaded operator function is a member function of the class it takes no arguments.
- If the unary overloaded operator function is a friend function of the class it takes one object as argument explicitly.

<u>EXAMPLE 1: OVERLOADING UNARY MINUS (-) OPERATOR:</u>

```
#include<iostream.h>
#include<conio.h>
class unary                          // Class definition
{
int x,y,z;
public:
void getdata(int a,int b,int c)      // Member function definition
{
x=a;
y=b;
z=c;
}
void display()                       // Member function definition
{
cout<<x<<" ";
cout<<y<<" ";
cout<<z<<" "<<endl;
}
void operator - ()                   // Operator function definition as member function
{
x= -x;
y= -y;
z= -z;
}
};                                   //End of class
void main()
{
clrscr();
unary ob;
ob.getdata(10,20,30);              // Assigning values to class members
cout<<"Before overloading"<<endl;
ob.display();
cout<<"After overloading"<<endl;
-ob;                               // Call to operator function
ob.display();
getch();
}
```

OUTPUT:

```
Before overloading
10  -20  30
After overloading
-10  20  -30
```

EXAMPLE 2: OVERLOADING UNARY * OPERATOR:

```
#include<iostream.h>
#include<conio.h>
class unary                        // Class definition
{
int x,y,z;
public:
void getdata(int a,int b,int c)    // Member function definition
{
x=a;
y=b;
z=c;
}
void display()                     // Member function definition
{
cout<<x<<" ";
cout<<y<<" ";
cout<<z<<" "<<endl;
}
void operator * ()                 // Operator function definition as member function
{
x*=x;
y*=y;
z*=z;
}
};                              //End of class
void main ()
{
clrscr();
unary ob;
ob.getdata(10,20,30);           // Assigning values to class members
cout<<"Before overloading"<<endl;
ob.display();
cout<<"After overloading"<<endl;
*ob;                            // Call to operator function
ob.display();
getch();
}
```

```
OUTPUT:
Before overloading
10   20   30
After overloading
100  400  900
```

EXAMPLE 3: OVERLOADING UNARY INCREMENT(++) OPERATOR

```
#include<iostream.h>
#include<conio.h>
class unary                     // Class definition
{
int x,y,z;
public:
void getdata(int a,int b,int c)   // Member function definition
{
x=a;
y=b;
z=c;
}
void display()                    // Member function definition
{
cout<<x<<endl;
cout<<y<<endl;
cout<<z<<endl;
}
void operator ++()                // Operator function definition as member function
{
x=++x;
y=++y;
z=++z;
}
};                                //End of class
void main()
{
clrscr();
unary ob;
ob.getdata(10,20,30);             // Assigning values to class members
cout<<"Before overloading"<<endl;
ob.display();
cout<<"After overloading"<<endl;
++ob;                             // Call to operator function
ob.display();
getch();
}
```

OUTPUT:

```
Before overloading
10
20
30
After overloading
11
21
31
```

EXAMPLE 3: OVERLOADING UNARY DECREMENT(--) OPERATOR

```
#include<iostream.h>
#include<conio.h>
class unary                       // Class definition
{
```

```
int x,y,z;
public:
void getdata(int a,int b,int c)      // Member function definition
{
x=a;
y=b;
z=c;
}
void display()                       // Member function definition
{
cout<<x<<endl;
cout<<y<<endl;
cout<<z<<endl;
}
void operator --()                   // Operator function definition as member function
{
x= --x;
y= --y;
z= --z;
}
};
void main()
{
clrscr();
unary ob;
ob.getdata(10,20,30);                // Assigning values to class members
cout<<"Before overloading"<<endl;
ob.display();
cout<<"After overloading"<<endl;
--ob;                                // Call to operator function
```

OUTPUT:

```
Before overloading
10
20
30
After overloading
9
19
29
```

```
ob.display();
getch();
}
```

BINARY OPERATOR OVERLOADING:

- Binary Operator overloading takes two operands of user-defined data types (objects) and one operator.
- The Binary overloaded operator function can either be the member function of the class or it can be a friend function.

1) BINARY OPERATOR OVERLOADING USING MEMBER FUNCTIONS

1. The binary overloaded operator function (which is a member function) takes the first operand (first object) directly and the second operand (second object) must be passed as argument to the operator function.
2. The data members of the first object are accessed without using the dot operator.
3. The second object is passed as the argument to the operator function and its data members are accessed using the object name and dot operator.

EXAMPLE: COMPLEX NUMBER ADDITION USING BINARY OPERATOR OVERLOADING USING MEMBER FUNCTIONS /*c3=c1+c2*/

```
#include<iostream.h>
#include<conio.h>
class complex                 // Class defintion
{
public:
float real,imag;
void getdata()                // Getting the input values for complex numbers
{
cout<<"\nReal=";
cin>>real;
```

```
    cout<<"\nImaginary=";
    cin>>imag;
    }
complex operator +(complex c2) //Operator function with two operands
{
complex t     ;                  // Creating temporary object to store the result
t.real=real+c2.real;             // Add the real parts of the two objects
t.imag=imag+c2.imag;             // Add the imaginary parts of the two objects
return t;                        // Return the object that stored the result
}
void outdata()
{
cout<<real<<"+j"<<imag<<"\n";
}
};                               //End of class
void main()
{
clrscr();
complex c1,c2,c3;
cout<<"\nEnter complex number c1";
c1.getdata();
cout<<"\nEnter complex number c2";
c2.getdata();
c3=c1+c2;                        //Invoke Binary overloaded operator+ () function
cout<<"\n c1=";
c1.outdata();
cout<<"\n c2=";
c2.outdata();
cout<<"\n c3=";
c3.outdata();
```

OUTPUT

```
Enter complex number c1
Real=2.2
Imaginary=3.2
Enter complex number c2
Real=2.3
Imaginary=3.5
C1=2.2+j3.2
C2=2.3+j3.5
C3=4.5+j6.7
```

```
getch();
}
```

2) BINARY OPERATOR OVERLOADING USING FRIEND FUNCTIONS

1. The binary overloaded operator function (which is a non-member function) takes both the operands as arguments in the operator function.
2. The data members of those two objects are accessed using the object name and dot operator.

Difference between Binary overloaded operator function acting as member function and friend function.

Binary operator function as member function	**Binary operator function as friend function**
• Takes a single object as argument in the operator function.	• Takes both the objects as argument in the operator function.

EXAMPLE: COMPLEX NUMBER ADDITION USING BINARY OPERATOR OVERLOADING USING FRIEND FUNCTIONS /*c3=c1+c2*/

```
#include<iostream.h>
#include<conio.h>
class complex                    //Class definition
{
public:
float real,imag;
void getdata()                   // Getting the input values for complex numbers
{
cout<<"\nReal=";
cin>>real;
cout<<"\nImaginary=";
cin>>imag;
}
void outdata()
{
cout<<real<<"+j"<<imag<<"\n";
}
friend complex operator +(complex, complex);
};                                  //End of class
complex operator+(complex c1, complex c2)  /*Operator function performing binary
                                            operator overloading using friend funcs*/
{
complex c3;                         // Creating temporary object to store the result
c3.real=c1.real+c2.real;            // Add the real parts of the two objects
c3.imag=c1.imag+c2.imag;            // Add the imaginary parts of the two objects
return  c3;                         // Return the object that stored the result
}
void main()
{
clrscr();
complex c1,c2,c3;                   // Creating object
cout<<"\nEnter complex number c1";
c1.getdata();
cout<<"\nEnter complex number c2";
c2.getdata();
c3=c1+c2;                           //Invoke Binary overloaded operator +() function
cout<<"\n c1=";
c1.outdata();
cout<<"\n c2=";
```

OUTPUT

```
Enter complex number c1
Real=2.2
Imaginary=3.2
Enter complex number c2
Real=2.3
Imaginary=3.5
C1=2.2+j3.2
C2=2.3+j3.5
C3=4.5+j6.7
```

```
c2.outdata();
cout<<"\n c3=";
c3.outdata();
getch();
}
```

OVERLOADING ASSIGNMENT OPERATORS USING MEMBER FUNCTIONS

- The assignment operator such as '=' and the compound assignment operators such as '+=,*= etc., can be overloaded to be assign an object to the object of the same class.
- Assignment operators **cannot be overloaded through friend function. The operator '=' can only be a member function because it is associated to the object on the left side of the =.**

EXAMPLE: ASSIGNMENT OPERATOR OVERLOADING TO PERFORM C1+=C2 (i.e. C1=C1+C2)

```
#include<iostream.h>
#include<conio.h>
class complex                    //Class definition
{
public:
float real,imag;
void getdata()                   // Member Function definition
{
cout<<"\nReal=";
cin>>real;
cout<<"\nImaginary=";
cin>>imag;
}
void operator +=(complex c2)     // Operator Function as Member function.
 {
real=real+c2.real;
imag=imag+c2.imag;  }
void outdata()
{
cout<<real<<"+j"<<imag<<"\n";
}
};                              //End of class
void main()
{
clrscr();
complex c1,c2,c3;               // Creating object
cout<<"\nEnter complex number c1";
c1.getdata();
cout<<"\nEnter complex number c2";
c2.getdata();
c3=c1;
c3+=c2;                         //Invoke Binary overloaded operator function
cout<<"\n c1=";
c1.outdata();
cout<<"\n c2=";
c2.outdata();
cout<<"\n c3=";
c3.outdata();
getch();
}
```

OUTPUT

```
Enter complex number c1
Real=2.2
Imaginary=3.2
Enter complex number c2
Real=2.3
Imaginary=3.5
C1=2.2+j3.2
C2=2.3+j3.5
C3=4.5+j6.7
```

ISTREAM(>>) AND OSTREAM(<<) OPERATOR OVERLOADING (or) Overloading Input/Output Operators:

- C++ is able to input and output the built-in data types using the stream insertion operator (<<) and the stream extraction operator (>>).
- The insertion (<<) and stream extraction (>>) operators also can be overloaded to perform input and output for user-defined types like an object.

Example:

```
#include <iostream.h>
#include<conio.h>
class stream                              //Class Definition
{
private:
     int x;
     int y;

public:
     stream()                             //Default Constructor
     {
       x = 0;
       y = 0;
     }
     stream(int tx, int ty)               //Parameterized Constructor
     {
        x=tx;
        y=ty;
     }
     friend ostream &operator<<(ostream &output, stream &D)
     {
         output<< "X=" <<D.x <<"&"<<" Y="<<D.y;
         return output;
     }
     friend istream &operator>>(istream  &input,stream &D)
     {
         input >>D.x >>D.y;
         return input;
     }
};
int main()
{
   clrscr();
   stream D1(2,4);
   stream D2(6,8);
   stream D3;
  cout <<"\n\nEnter the value of object : ";
   cin>>D3;
   cout <<"The Value of:"<<D1<<endl;
   cout <<"The Value of:"<<D2<<endl;
   cout <<"The Value of:" <<D3<<endl;
  getch();
  return 0;}
```

OUTPUT:

```
Enter the value of object :
10
12
The Value of: X=2 & Y=4
The Value of: X=6 & Y=8
The Value of: X=10 & Y=12
```

RULES FOR OVERLOADING OPERATORS

1. Only existing operators can be overloaded. New operators cannot be created.
2. The overloaded operator must have at least one operand that is of user-defined type(object).
3. The basic meaning of the operator cannot be changed. i.e. We cannot redefine the plus (+) operator to subtract one value from the other.
4. Overloaded operators must follow the syntax rules of the original operators. They cannot be overridden.
5. Unary operators overloaded by means of member function takes no explicit arguments and return no explicit values. But the unary operator overloaded through friend function, take one reference argument (object of the class).
6. Binary operators overloaded through a member function takes one explicit arguments and those which are overloaded through friend function take two explicit arguments.

- There are some operators that cannot be overloaded, listed as follows

Sizeof	Size of operator
.	Membership operator
.*	Pointer-to-member operator
::	Scope Resolution operator
?:	Conditional operator

- Friend functions cannot be used to overload certain operators listed as follows

=	Assignment operator
()	Function call operator
[]	Subscripting operator
->	Class member access operator

FRIEND FUNCTION

- A class can allow non-member functions to access its own private data by making them as friends.
- To make an outside function friendly to the class, declare the function by the following **syntax:**

  ***friend* return-type function-name (classname);**// *Friend function declaration*

- The friend function declaration must be made inside the class which gives access to its private data.

Need for friend function

- When a data member is declared as private inside the class, it can be accessed only by the member functions in the class. It is not accessible outside the class.
- But, in some situations, some non-member functions need to access the private data of a class.
- In such cases, the concept of friend functions is a useful tool.

Special characteristics of friend functions

1. It is not in the scope of the class to which it has been declared as friend.
2. Though, it is not in the scope of the class, it can be defined outside the class without using the '::' operator, because it is a non-member function.
3. It can be declared either in the public or private part of the class without affecting the meaning.

4. Since it is not in the scope of the class, it cannot be called using the object of the class.
5. It can be invoked or called like a normal function without the help of any object.
6. The friend function takes objects as arguments
7. Unlike the member functions, it cannot access the members directly. The object name and the dot membership (.) operator is to be used to access the members.

EXAMPLE1: FRIEND FUNCTION

```
#include<iostream.h>
#include<conio.h>
class sample                      // Class definition
{
int a;
int b;
public:
void setvalue()                   // Member function definition
{
a=30;
b=45;
}
friend float mean(sample s);      // Friend function declaration
};                                // End of class
float mean(sample s)              // Friend function definition with object as argument
{
return float(s.a+s.b)/2.0;
}
void main()
{
sample x;
```

OUTPUT:

```
Mean value: 37.5
```

```
x.setvalue();
cout<<"Mean value:"<<mean(x)<<"\n";
getch();                              /*Calling friend function by
}                                     passing object as argument*/
```

EXAMPLE 2: FRIEND FUNCTION COMMON TO TWO CLASSES

```
#include<iostream.h>
#include<conio.h>
class two;                         //Forward Declaration of class
class one                          // Definition of class1
{
private:
int x;
public:
void setdata(int a)                // Member function definition
{
x=a;
}
friend float mean(one,two);        //Friend function declaration
};                                 //End of class1

class two                          // Definition of class2
{
private:
int y;
public:
void setdata(int b)                // Member function definition
{
y=b;
}
friend float mean(one,two);        //Friend function declaration
};
float mean(one ob1,two ob2)        //Friend function definition with objects as arguments
{
return (ob1.x+ob2.y)/2.0;
}
void main()
{
clrscr();
one ob1;
two ob2;
ob1.setdata(5);
ob2.setdata(10);
cout<<"Mean Value:"<<mean(ob1,ob2);
getch();                          /*Calling friend function by
}                                  passing objects as arguments*/
```

OUTPUT:

```
Mean Value: 7.5
```

TYPE CONVERSION

AUTOMATIC TYPE CONVERSION

- An assignment operator causes the automatic type conversion. The type of data to the right of an assignment operator '=' is automatically converted to the type of the variable on the left.
- For example,

```
int m;

float x=3.14;

m=x;
```

****These statements converts the float value x into an integer value m, before assigned to m.

- Consider the following statement,that adds two objects and then assigns the result to a third object where all the three objects belong to the same class.

```
a3=a1+a2;        //a1,a2,a3 are the class type objects
```

****If the objects are of the same class type, the operations of addition and assignment may be done without causing any errors by overloading the operator.

➢ TYPE CONVERSION OF INCOMPATIBLE TYPES:

Three types of situations that require data conversion between incompatible types are:

1. Conversion from Basic type to Class type
2. Conversion from Class type to Basic type
3. Conversion from one Class type to another Class type.

/* Here Basic type denotes the built in data types such as int, float, double etc.,Class type denotes the objects for the relevant class*/

1. Conversion from Basic Type to Class Type

Conversion from basic data type to object type can be done with the help of constructors. **The conversion constructor takes a single argument whose type is to be converted.**

Syntax:

```
Constructor_name(one argument)
{
}
```

2. Conversion from Object to Basic Data Type

An object can be converted into a basic data type by using the **conversion operator function**. The syntax for defining the conversion operator function is defined as follows:

Syntax:

```
operator type-name()
{
Function body
}
```

Here type-name denotes the basic data type to which the object is to converted.

Conditions that the conversion operator function should satisfy:

- It must be a member function of the class
- It must not have any return type
- It must not have any arguments

3. CONVERSION OF ONE CLASS TYPE TO ANOTHER CLASS TYPE

- The object of one type can be converted into an object of another class can be done using **either conversion constructor or conversion operator function**. (Here both are objects belong to different classes).
- For example:

```
objx=objy;          // Objects of different class
```

*** Here objy is the source and objx is the destination.

- If conversion needs to take place in **S**ource class, use conversion **O**perator function in the source class /*S-O*/
- If conversion needs to take place in **D**estination class, use **C**onversion constructor in the destination class /* D-C*/

Example program for type conversion:

```
#include<iostream.h>
#include<conio.h>
class invent2;                  // Forward Declaration of class

class invent1                   //SOURCE CLASS
{
int code;
int items;
float price;
public:
invent1(int a, int b, int c)    // Parameterized Constructor
{
code=a;
items=b;
price=c;
}
void putdata()
{
cout<<"Code:"<<code<<endl;
cout<<"Items:"<<items<<endl;
cout<<"Value:"<<price<<endl;
}
int getcode()
{
return code;
}
int getitems()
{
return items;
}
float getprice()
{
return (items*price);
}
operator float()                // Operator function
{
return (items*price);
}
};                              //End of source class
class invent2                   // DESTINATION CLASS
{
int code;
float value;
```

```
public:
invent2()                          //Default Constructor
{
code=0;
value=0;
}
invent2(int x, float y)            //Parameterized Constructor
{
code=x;
value=y;
}
void putdata()
{
cout<<"Code="<<code<<endl;
cout<<"Value="<<value<<endl;
}
invent2(invent1 p)                 // Conversion constructor
{
code=p.getcode();
value=p.getitems();
}
};
void main()
{
clrscr();
float total_value;
invent1 s1(100,5,140.0);           //Call the parameterized constructor of class invent1
invent2 d1;                        //Call the default constructor of class invent2
total_value=s1; /* Object converted into basic type. Call Conversion operator function.*/
```

```
d1=s1;                  /* One class type converted into another class type. Conversion
                        takes place at destination class. So Conversion constructor will
                        be called*/
cout<<"TYPE CONVERSION";
cout<<"\n\n";
cout<<"Product details of invent1 type"<<endl;
s1.putdata ();
cout<<"Total value:"<<total_value<<endl;
cout<<"\n";
cout<<"Product details-invent2 type"<<endl;
d1.putdata ();
getch ();
}
```

OUTPUT:

```
TYPE CONVERSION

Product details of invent1 type
Code:100
Items:5
Value:140
Total value: 700

Product details-invent2 type
Code=100
Value=5
```

===

TEMPLATES

TEMPLATE DEFINITION:

- **Templates enable to define generic classes and functions (common to all data types) and thus support generic programming.**
- Generic programming is an approach where generic types are used as parameters, so that they can work for a variety of data types.
- A template can be used to create a family of classes or functions.
- Since a template is defined with a parameter that would be replaced by a specified data type at the time of actual use of function or class, the templates can also be called as *parameterized classes or functions*.

***CLASS TEMPLATE:

Definition:

The template that allows us to define generic classes is called as ***class template***. Using class template, a family of classes that contains the data members and member functions of different data types can be defined.

CLASS TEMPLATE WITH SINGLE ARGUMENT

Syntax for defining a class template with single argument:

```
template<class T>
class class_name
{
/* Class member specification with anonymous type T*/
}
```

Syntax to invoke the class template

```
classname <datatype> objectname;
```

EXAMPLE FOR CLASS TEMPLATE WITH ONE ARGUMENT

//To add two numbers of the same type using class template

```
#include<iostream.h>

#include<conio.h>

template<class T>          // Template declaration with one generic type T
```

```
class add
{
T a,b,c;                        // Data member declaration with type T
public:
void getdata(T x, T y)// Member Function definition with template type arguments
{
a=x;
b=y;  }
void display()
{
c=a+b;
cout<<c<<endl;
}
};
void main()
{
clrscr();
add <int> obj1;                           // Invoking the class template add.
cout<<"Sum of integers:";
obj1.getdata(5,4);
obj1.display();
add <float> obj2;               // Invoking the class template again with another data type
cout<<"Sum of float type:";
obj2.getdata(1.5,3.1);
obj2.display();
getch();
}
```

```
OUTPUT:
Sum of integers: 9
Sum of float type: 4.6
```

In this program,

- The statement, **add <int> obj1;** Invoke the class template with datatype 'int'. i.e all the template type variables and arguments(T) are replaced with integer type.
- The statement, **add <float> obj2;** Invoke the class template with datatype 'float'. i.e all the template type variables and arguments(T) are replaced with float type

CLASS TEMPLATE WITH MULTIPLE ARGUMENTS

Definition:

- More than one generic datatype can be used in a class template. They are declared as a comma-seperated list within the template specification

Syntax for defining a class template with multiple arguments:

```
template<class T1,class T2,… >
class class_name
{
/* Class member specification with anonymous type T1, T2…*/
}
```

Syntax to invoke the class template with multiple arguments:

- classname <datatype1, datatype2,…> objectname;

EXAMPLE FOR CLASS TEMPLATE WITH MULTIPLE ARGUMENTS

//To add two numbers of different types using class template

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

```
template<class T1, class T2>      // Template specification with two arguments
class add
{
T1 a;
T2 b,c;                           // Data member declaration with type T
public:
void getdata(T1 x, T2 y)// Member Function definition with template type arguments
{
a=x;
b=y;
}
void display()
{
c=a+b;
cout<<c<<endl;
}
};
void main()
{
clrscr();
add <int,float> obj1;             // Invoking the class template add
cout<<"Sum of int and float:";
obj1.getdata(5,4.2);
obj1.display();
add <float, long> obj2;     // Invoking the class template with different data types
cout<<"Sum of float and long:";
```

OUTPUT:

```
Sum of integers: 9.2

Sum of float type: 30002
```

```
obj2.getdata(2.5,30000);
obj2.display();
getch();
}
```

In this program,

- The statement, **add <int,float> obj1;** Invoke the class template by replacing the first template type (T1) by 'int' type and the second template type (T2) by the 'float' type
- The statement, **add <float, long> obj2;** Invoke the class template by replacing the first template type (T1) by 'float' type and the second template type (T2) by the 'long' type

***FUNCTION TEMPLATE

Definition:

Function templates can be used to create a family of functions with different argument types.

FUNCTION TEMPLATE WITH SINGLE ARGUMENT

Syntax:

```
template<class T>
returntype functionname(argument of type T)
{
/* Body of the function with type T wherever appropriate*/
}
```

EXAMPLE PROGRAM FOR FUNCTION TEMPLATE WITH ONE ARGUMENT

1). PROGRAM TO SWAP TWO VALUES

```
#include<iostream.h>
#include<conio.h>
template<class T>           // Template specification with one argument
```

```
void swap(T  &x, T  &y)     // Function declaration with template type arguments
{
T t;                        //Temporary variable declaration of type T
t=&x;
&x=&y;
&y=t;
}
void main()
{
clrscr();
char ch1,ch2;
cout<<"Enter the characters to swap"<<endl;
cin>>ch1>>ch2;       // Input the characters
swap(ch1,ch2);       // Swap the characters
cout<<"After swapping"<<endl;
cout<<ch1<<" "<<ch2<<endl;
int a,b;
cout<<"Enter the integers to swap"<<endl;
cin>>a>>b;           // Input the integers
swap(a,b);           // Swap the characters
cout<<"After swapping"<<endl;
cout<<a<<" "<<b<<endl;
getch();
}
```

OUTPUT:

```
Enter the characters to swap
a
b
After swapping
b   a
Enter the integers to swap
1
2
After swapping
2   1
```

In this program,

- The statement, **swap(ch1,ch2);** Invokes the function template by replacing the template datatype(T) to the 'char' datatype.
- The statement, **swap(a,b);** Invokes the function template by replacing the template datatype(T) to the 'int' datatype.

2). PROGRAM TO PERFORM BUBBLE SORT

```
#include<conio.h>
#include<iostream.h>
template<class T>              // Template specification with one argument
void sort(T  a[], int  n)      /*Function with one template argument and one
                               non-template type argument*/
{
   int i, j;
   for(i=0;i<n-1;i++)
   {
       for(j=i+1;j<n;j++)
       {
          if(a[i]>a[j])
          {
               T t;
               t = a[i];
               a[i] = a[j];
               a[j] = t;
          }
       }
   }
}

void main()
{
   int a[6]={1,2,3,4,4,3};
   char b[4]={'s','b','d','e'};
   clrscr();
   sort(a,6);                  // Sort the integer array
   cout<<"\nSorted Order Integers: ";
   for(int i=0;i<6;i++)
       cout<<a[i]<<"\t";
   sort(b,4);                  // Sort the character array

   cout<<"\nSorted Order Characters: ";
   for(int j=0;j<4;j++)
       cout<<b[j]<<"\t";
```

```
  getch();
}
```

OUTPUT:

```
Sorted Order Integers: 1    2       3       3       4       4
Sorted Order Characters: b          d       e       s
```

FUNCTION TEMPLATE WITH MULTIPLE ARGUMENTS

Definition:

More than one generic data type can be used in the function template using a comma-separated list.

Syntax:

```
template<class T1, class T2,…>
Returntype functionname(arguments of types T1 and T2…)
{
        //Body of the function
}
```

EXAMPLE:

```
#include<iostream.h>
#include<conio.h>
template<class T1,class T2>   // Template specification with two arguments.
void display(T1 x, T2 y) //Function definition with template type arguments T1,T2.
{
cout<<x<<" "<<y<<"\n";
}
void main()
{
clrscr();
display(100,"ABC");
display(12.34, 123);
getch();
}
```

OUTPUT:

```
100 ABC

12.34 123
```

In this program,

- The statement, display(100,"ABC"); invokes the function template by replacing the first template type in the argument (T1) by ‘int’ datatype and the second template type in the argument (T2) by the ‘string’ datatype

- The statement, display(12.34,123); invokes the function template by replacing the first template type in the argument(T1) by 'float' datatype and the second template type in the argument (T2) by the 'int' type.

*** MEMBER FUNCTION TEMPLATES

Defintion:

When a class is defined, the member functions of the class can also be defined outside the class. The member function templates can be used for this purpose.

Syntax:

```
template<class T>
returntype classname <T> :: functionname(arglist)
{
//Function body;
}
```

EXAMPLE

```
#include<iostream.h>
#include<conio.h>
template<class T>                // Template specification with one argument
class add
{
T a,b,c;
public:
void getdata(T x, T y);                //Member function declaration inside class
void display()
{
c=a+b;
cout<<c<<endl;
}
};
/* Defining member function outside class using member function template*/
template<class T>
void add <T>::getdata(T x, T y)          /*Member function template*/
{
a=x;
b=y;
}
void main()
{
clrscr();
add <int> obj1;
cout<<"Sum of integers:";
obj1.getdata(5,4);
obj1.display();
add <float> obj2;
cout<<"Sum of float type:";
obj2.getdata(1.5,3.1);
obj2.display();
getch();
}
```

OUTPUT:

```
Sum of integers: 9

Sum of float type: 4.6
```

In this program,

- The statement, **obj1.getdata(5,4)**; Invoke the member function template by replacing the template datatype (T) by ‘int’ datatype.
- The statement, **obj2.getdata(1.5,3.1);** Invoke the member function template by replacing the template datatype(T) by ‘float’ datatype.

*** OVERLOADING OF TEMPLATE FUNCTIONS

A template function may be overloaded either by template functions or ordinary functions.

EXAMPLE

```
#Iinclude<iostream.h>

#include<conio.h>

template<class T>

void display(T  x)              // Generic display() with template type argument

{

cout<<”Template Display:”<<x<<”\n”;

}

void display(int  x)                    // Overloads the generic display() function
{
cout<<”Explicit Display:”<<x<<”\n”;
}
void main()
{
clrscr();
display(100);
display(12.34);
display(‘C’);
getch();
}
```

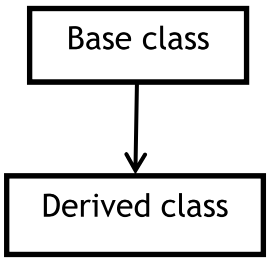
OUTPUT:

```
Explicit Display: 100

Template Display: 12.34

Template Display: C
```

In this program, the call **display (100)**; invokes the ordinary version of display () and not the template version.
Whereas, **display (12.34)** and **display (‘C’)** invokes the template version of display ().
===

INHERITANCE

- Inheritance is the process by which objects of one class acquire the properties of the objects of another class. This supports the concept of hierarchical classification.
- Inheritance is the object-oriented feature that supports **reusability.**



Base class:

It is the class whose members (variables + functions) are inherited by another class.

Derived class:

It is the class that inherits the members (variables + functions) from the base class. It contains the inherited members as well as its own members.

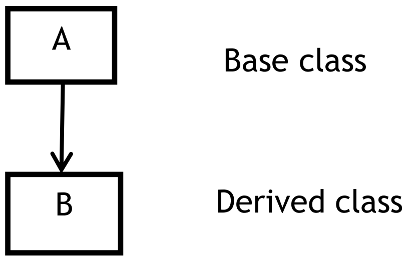
ADVANTAGES:

1. The concept of inheritance allows the code to be reused as many times as needed.
2. Saves the time and effort in program development.

TYPES OF INHERITANCE

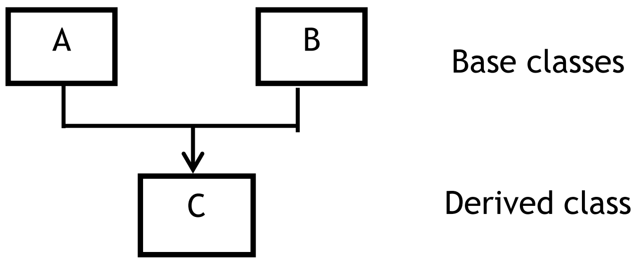
1. SINGLE INHERITANCE

- Deriving a class from only one base class is called as single inheritance.



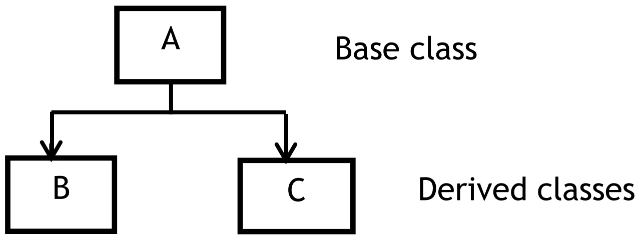
2. MULTIPLE INHERITANCE

- Deriving a class from two or more base classes is called as multiple inheritance.



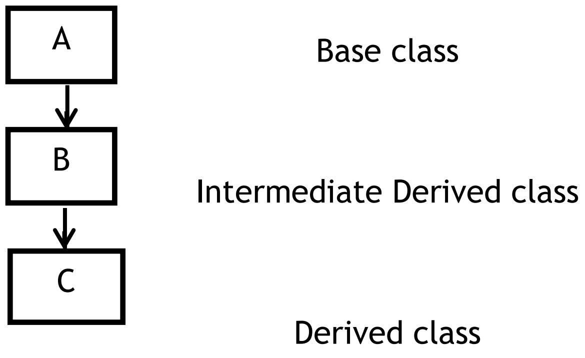
3. HIERARCHICAL INHERITANCE

- Deriving two or more classes from a single base class is called as Hierarchical inheritance.



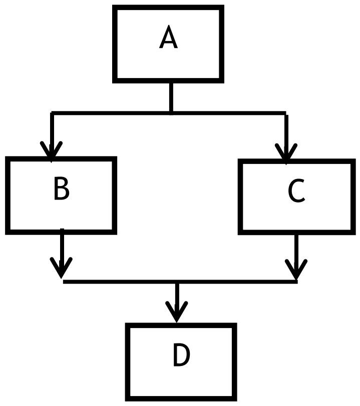
4. MULTILEVEL INHERITANCE

- Derivation of a class from another derived class is called as multi-level inheritance.



5. HYBRID INHERITANCE

- Derivation of a class involving more than one form of inheritance is called as hybrid inheritance.



- This is the combination of three types of inheritance: hierarchical, multiple and multi-level inheritance.

DERIVED CLASS DECLARATION

- A derived class extends its features by inheriting the properties of another class (base class) and adding features of its own.
- The derivation of a derived class specifies its relationship with the base class, in addition to its own features.

Syntax

Class derivedclassname: visibilitymode baseclassname

{

/*Members of the derived class, they can also access the members of the base class*/

}

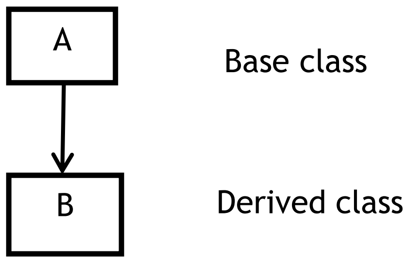
- The ':' denotes that the class is derived from another class
- The visibility mode specifies the inheritance type: **public,private or protected.**
- The visibility mode is optional. The default visibility mode is *private*.

VISIBILITY OF INHERITED MEMBERS

Base Class Members	Derived Class Visibility		
	Public Derivation	**Private Derivation**	**Protected Derivation**
Private members	Not inherited	Not inherited	Not inherited
Public members	Public	Private	Protected
Protected members	Protected	Private	protected

SINGLE INHERITANCE

- Deriving a class from only one base class is called as single inheritance.



```
#include<iostream.h>
#include<conio.h>
class base                    // Base class definition
{
int a;
public:
int b;
void get_ab()
{
a=5;
b=10;
}
int get_a()
{
return a;
}
void show_a()
{
cout<<"a="<<a<<"\n";  }  };
```

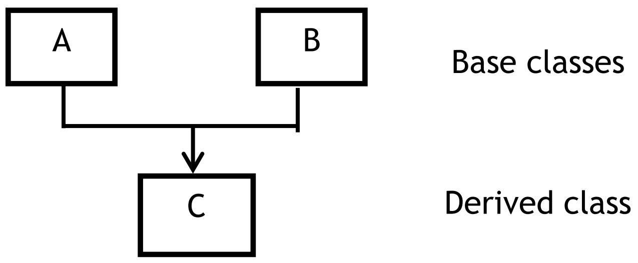
```
class derived: public base                    // Derived class definition
{
int c;
public:
void mul()
{
c=b*get_a();                   /* Cannot access 'a' directly here. i.e. c=b*a is not
                               possible, since 'a' is private*/
}
void display()
{
cout<<"a="<<get_a()<<endl;
cout<<"b="<<b<<endl;
cout<<"c="<<c<<endl;
}
};
void main()
{
clrscr();
derived d;              // Creating object for derived class
d.get_ab();
d.show_a();
d.mul();
d.display();
d.b=20;
d.mul();
d.display();
getch();
}
```

OUTPUT:

```
a=5

a=5

b=10

c=50

a=5

b=20

c=100
```

MULTIPLE INHERITANCE

Multiple inheritance contains more than one derived class and a single base class. The following example illustrates the multiple inheritance.



Example 1:

```
#include<iostream.h>
#include<conio.h>
class roll                           // Base class 1
{
protected:
        int rollno;
public:
        void getroll(int x)
        {
        rollno=x;
        } };
class marks                          // Base class 1
{
protected:
        int mark1,mark2;
public:
        void getmarks(int x, int y)
        {
        mark1=x;
        mark2=y;
        }
        };
```

```
class student: public roll, public marks / *Both the base class members are publicly
                                             inherited to student class*/
{
protected:
        float avg;
public:
        void display()
        {
        avg=(mark1+mark2)/2;
        cout<<"\t***MULTIPLE INHERITANCE PROGRAM***"<<endl;
        cout<<"\n Student Details \n";
        cout<<"The roll no    :"<<rollno<<endl;
        cout<<"Subject1 marks :"<<mark1<<endl;
        cout<<"Subject2 marks :"<<mark2<<endl;
        cout<<"Average marks  :"<<avg<<endl;
        }
};
void main()
{
clrscr();
student ob;                // Object created for derived class
ob.getroll(101);
ob.getmarks(90, 95);
ob.display();
getch();
}
```

OUTPUT:

```
   ***MULTIPLE INHERITANCE PROGRAM***
 Student Details
The roll no    :101
Subject1 marks :90
Subject2 marks :95
Average marks  :92
```

Multiple Inheritance: Example 2:

```
#include<iostream.h>

#include<conio.h>

class one                           //Base class 1

{

protected:

int m;

public:

void get_m(int x)

{

m=x;

}

};


class two                           //Base class 2

{

protected:

int n;

public:

void get_n(int y)

{

n=y;

}

};

class three : public one, public  two      // Derived class, inherits both classes pubicly

{

public:

void display()

{
```

```
cout<<"m="<<m<<"\n";
cout<<"n="<<n<<"\n";
cout<<"m*n="<<m*n<<"\n";
}
};
void main()
{
clrscr();
three ob;
ob.get_m(10);
ob.get_n(20);
ob.display();
getch();
}
```

OUTPUT:

```
m=10

n=20

m*n=200
```

AMBIGUITY RESOLUTION IN MULTIPLE INHERITANCE

- Problem may occur in multiple inheritance, when a function with the same name appears in more than one base class. In this case, the function to be used by the derived class will conflict. This causes error during compilation.

- This problem can be solved by defining a named instance within the derived class using scope resolution operator

Syntax to resolve this conflict:

Class_name::function_name();

EXAMPLE

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

class one                              //Base class 1

{
protected:
int m;
```

```
public:
void get_m(int x)
{
m=x;
}
void display()
{
cout<<"m="<<m<<"\n";
}
};

class two                                  //Base class 2

{
protected:
int n;
public:
void get_n(int y)
{
n=y; }
void display()
{
cout<<"n="<<n<<"\n";
}
};

class three:public one,public two         // Derived class , inherits both classes publicly.

{
public:
void display()
{
m::display();               // Calls the display function of class m
n::display();               // Calls the display function of class n

cout<<"m*n="<<m*n<<"\n";
}
};
void main()
{
clrscr();
```

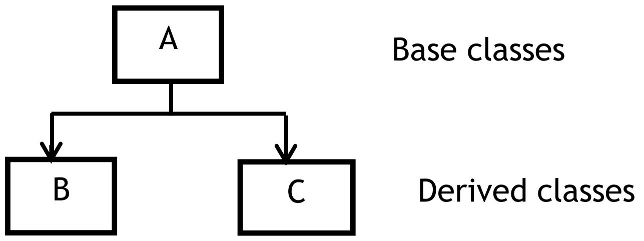
OUTPUT:

```
m=10

n=20

m*n=200
```

```
    three ob;      // Creating object for derived class
    ob.get_m(10);
    ob.get_n(20);
    ob.display();
    getch();
    }
```

HIERARCHICAL INHERITANCE

This contains one base class and more than one derived class.



EXAMPLE:

```
#include<iostream.h>
#include<conio.h>
class measure                    //Base class
{
protected:
int length;
public:
void getlength(int x)
{
length=x;
}
};
```

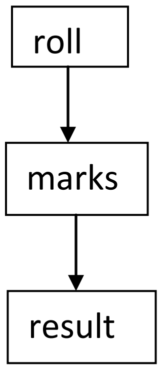
```
class square: public measure                // Derived class 1
{
public:
void squarearea()
{
cout<<"The area of square:"<<length*length<<endl;
}
};
class cube: public measure                  // Derived class 2
{
public:
void cubevolume()
{
cout<<"The volume of cube:"<<length*length*length<<endl;
}
};
void main()
{
clrscr();
cout<<"\t ***HIERARCHICAL INHERITANCE***"<<endl;
cout<<"\n";
square ob1;
ob1.getlength (5);
ob1.squarearea ();
cube ob2;
ob2.getlength (10);
ob2.cubevolume ();
getch();
}
```

OUTPUT:

```
        ***HIERARCHICAL INHERITANCE***
The area of square:25
The volume of cube:1000
```

MULTILEVEL INHERITANCE

Deriving a class from another derived class is stated as multilevel inheritance. The following program illustrates the multilevel inheritance



EXAMPLE

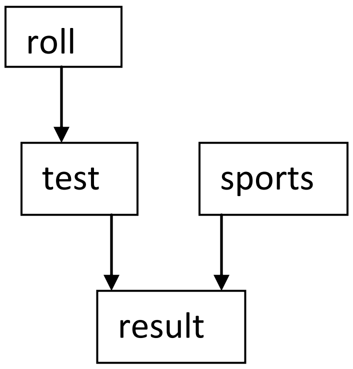
```
#include <iostream.h>
#include<conio.h>
class roll                              //Base class
  {
protected:
introllno;
public:
void get_num(int a)
        {
        rollno = a;
        }
};
```

```
class marks : public roll              //Intermediate derived class
{
protected:
int sub1;
int sub2;
public:
void get_marks(intx,int y)
        {
          sub1 = x;
          sub2 = y;
        }
void put_marks(void)
        {
        cout<<"Roll no:"<<rollno<<"\n";    //Protected member can be accessed here
        cout<< "Subject1:" << sub1 << "\n";
        cout<< "Subject2:" << sub2 << "\n";
        }
  };
class result : public marks
  {
protected:
float total;
public:
void disp()
         {
        tot = sub1+sub2;
        put_marks();
        cout<< "Total:"<< total;
         }
  };
void main()
 {
clrscr();
result ob;                    // Object is created for the finally derived class
ob.get_num(101);
ob.get_marks(90,80);
ob.disp();
getch();
 }
```

```
OUTPUT:
Roll no: 101
Subject 1: 90
Subject 2: 80
Total: 85
```

HYBRID INHERITANCE

- The combination of more than one form of inheritance is called as hybrid inheritance.



This hierarchy is a combination of single, multiple and multilevel inheritance

Example:

```
#include <iostream.h>
#include<conio.h>
class roll                              // Base class
 {
   protected:
     int rollno;
   public:
     void get_num(int a)
         {
         rollno = a;
         }
 };
class test : public roll                // Single inheritance
 {
   protected:
     int sub1;
     int sub2;

 public:
     void get_marks(int x,int y)
         {
            sub1 = x;
            sub2 = y;
         }
     void put_marks()
         {
           cout<<"Roll no:"<<rollno<<"\n";
           cout << "Subject1:" << sub1 << "\n";
           cout << "Subject2:" << sub2 << "\n";
         }
  };
class sports
  {
    protected:
      float score;
    public:
        void get_score(float s)
            {
            score=s;
            }
            void put_score()
            {
            cout<<"Sports credit:"<<score<<"\n";
            }
            };
class result: public test, public sports           // Multiple & multi-level inheritance
        {
        float total;
        public:
          void disp()
                  {
                    total = sub1+sub2+score;
                    put_marks();
                    put_score();
                    cout << "Total Score:"<< total<<"\n";
                  }
        };
void main ()
{
            clrscr();
            result ob;
            ob.get_num(101);
            ob.get_marks(90,95);
            ob.get_score(80);
            ob.disp();
            getch();
}
```

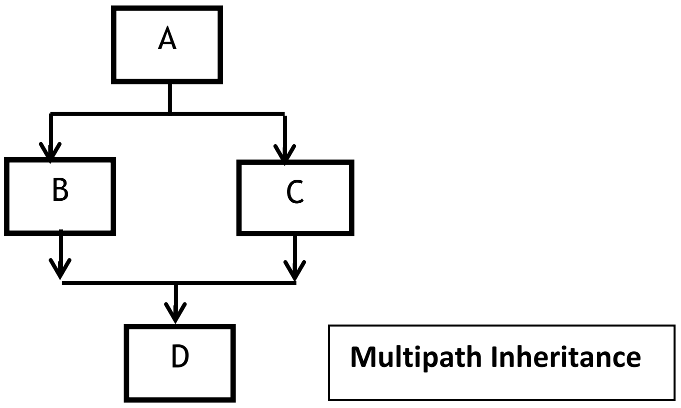
OUTPUT:

```
Roll no: 101
Subject1: 90
Subject2:95
Sports credit: 80
Total score: 88.3
```

VIRTUAL BASE CLASS (OR) MULTIPATH INHERTANCE

Consider a situation where three kinds of inheritance such as multiple, multilevel and hierarchical inheritance are involved.

- Derivation of class form another derived classes, which are derived from the same base class is called as **multipath inheritance**. The following figure depicts multipath inheritance.



The syntax for the above hierarchy of inheritance is

```
Class A
{ };
Class B:public A
{ };
Class C:public A
{ };
Class D:public B,public C
{ };
```

- In this hierarchy, the class D has two direct base classes ‘B’ and ‘C’, which has the common base class ‘A’.
- Class D inherits the properties of Class A via two separate paths.
- Class A is called as indirect base class.

- The members of Class A are inherited to class D twice, first via class B and then via class C.
- So class D have duplicate copy of the class A members which leads to error due to compilation.
- The duplication of the inherited members due to these multiple paths can be avoided by making the common base class(class A) as the virtual base class while declaring the direct or intermediate base class.

Syntax to achieve virtual base class concept:

Class A
{ };
Class B: **virtual public** A
{ };
Class C: **public virtual** A
{ };
Class D: public B, public C
{
*/*Only one copy of A will be inherited and the single copy will be shared by class B & class C, since the common base class is made virtual*/*
};

The keywords virtual and public may be used in either

- When a class is made a virtual base class, only one copy of that class is inherited and shared by the derived classes, regardless of the number of inheritance paths exists between Virtual base class and the derived class.

EXAMPLE

```
#include<iostream.h>
#include<conio.h>
class A                          // Class A is made virtual
{
    public:
        int i;  };
```

```
class B : virtual public A          /*Class B and C derives from the virtual base class
                                    A, so a single copy of class A is inherited and is
                                    shared by both the classes.*/
{
      public:
            int j;
};
class C: virtual public A
{
       public:
           int k;
};
class D: public B, public C
{
public:
     int sum;
};

void main()
{
         clrscr();
          D ob;          // object is created for the finally derived class.
          ob.i = 10; //Does not cause error, since only one copy of i is inherited by D
          ob.j = 20;
          ob.k = 30;
          ob.sum = ob.i + ob.j + ob.k;
          cout <<"Value of i is : "<< ob.i<<"\n";
          cout <<"Value of j is : "<< ob.j<<"\n";
          cout <<"Value of k is :"<< ob.k<<"\n";
          cout <<"Sum is : "<< ob.sum <<"\n";
          getch();
}
```

OUTPUT:

```
Value of i is : 10

Value of j is : 20

Value of k is :30

Sum is : 60
```

===

RUN TIME POLYMORPHISM

- The run-time polymorphism can also be called as dynamic binding or late binding. This is implemented using **Virtual functions.**

- When the same function with the same prototype is declared both in the base class and derived class, it is called as **function overriding**.
- The appropriate member function could be selected at the time of running the program. So, this is called as Runtime polymorphism.

VIRTUAL FUNCTIONS

- In virtual functions, the pointer to the base class is used to refer to all the derived class objects.
- The class that contains a virtual function is called as the **polymorphic class**.
- Even though, a base pointer is made to contain the address of the derived class object; it always executes the function in the base class.
- This is because, the compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer

Rules for Virtual Functions:

- The virtual functions must be members of the class
- They cannot be static members.
- They are accessed by using object pointers.
- A virtual function can be a friend of another class.
- A virtual function must be defined in the base class.
- Constructors cannot be virtual but destructors can be virtual.

EXAMPLE

```
#include <iostream.h>
#include<conio.h>
class Polygon                        //Base class
 {
 protected:
   int width, height;
 public:
   void set_values (int a, int b)
     {
     width=a; height=b;
     }
     virtual int area ()             // Base class function declared as  virtual
     {
     return (0);
```

```
      }
  };
class Rectangle: public Polygon            //Derived class1
{
  public:
    int area ()
      {
      return (width * height);
      }
};

class Triangle: public Polygon             //Derived class2
{
  public:
    int area ()
      {
         return (width * height / 2);
      }
  };
void main ()
{
  clrscr();
  Rectangle rect;
  Triangle trgl;
  Polygon poly;
  Polygon * bptr1 = &rect;
  Polygon * bptr2 = &trgl;
  Polygon * bptr3 = &poly;
  bptr1->set_values (4,5);         // '->' Used to access the member from the pointer
  bptr2->set_values (6,4);
  bptr3->set_values (10,2);
  cout << "Area of rect:"<<bptr1->area() << endl;      // Area() of rectangle is called
  cout << "Area of triangle:"<<bptr2->area() << endl; // Area() of triangle is called
  cout << "Area of polygon:"<<bptr3->area() << endl; // Area() of polygon is called
  getch();
}
```

OUTPUT:

```
Area of rect:20

Area of triangle:12

Area of polygon: 0
```

<u>*NOTE:</u>**

- **Without mentioning the base class function as virtual**, the output will be

```
0

0

0
```

This is because the base class's function would be repeatedly called, as the base pointer executes the function in the base class.

PURE VIRTUAL FUNCTION

- The pure virtual function (or abstract function) is a virtual function which **does not have a function definition**.The function is assigned to the value 0.
- '0' is appended to the virtual function instead of specifying an implementation for the function.

Syntax:

Virtual return_type function_name()=0;

ABSTRACT CLASS

- The class that contains atleast one pure virtual function is known as the abstract class.
- The object cannot be created for the abstract class.
- The main objectives of abstract class is
  - ✓ To provide some traits(features) to the derived classes
  - ✓ To create base pointer required for achieving run-time polymorphism.

EXAMPLE

```
#include <iostream.h>
#include<conio.h>
class Polygon                              // Abstract base class
{
  protected:
    int width, height;
  public:
    void set_values (int a, int b)
      {
      width=a; height=b;
      }
    virtual int area ()=0;           // Pure virtual function
  };

class Rectangle: public Polygon   // Derived class1
{
  public:
    int area ()
      {
      return (width * height);
      }
  };
class Triangle: public Polygon    // Derived class2
{
  public:
    int area ()
      {
      return (width * height / 2);
      }
  };
void main ()
{
  Rectangle rect;
  Triangle trgl;
  Polygon * bptr1 = &rect;
  Polygon * bptr2 = &trgl;
  bptr1->set_values (4,5);
  bptr2->set_values (6,4);
  cout << “Area of rect:”<<bptr1->area() << endl;
  cout << “Area of triangle:”<<bptr2->area() << endl;
  getch();
}
```

OUTPUT:

```
Area of rect: 20

Area of triangle:10
```