Objectives:

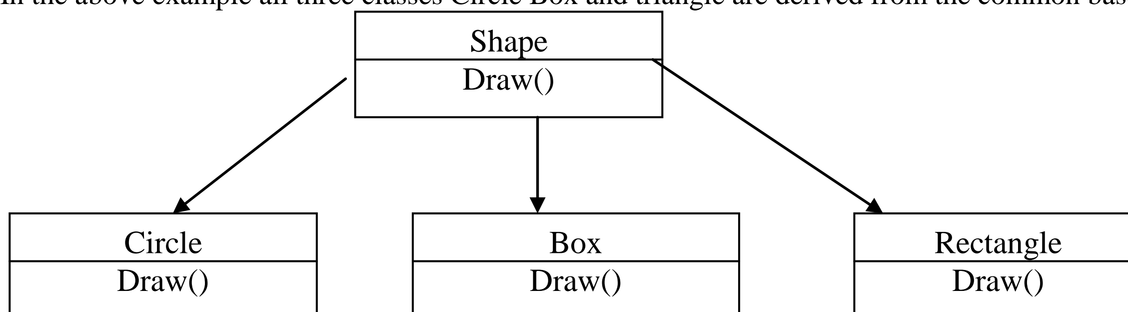
- Polymorphism concept & its types.
- Program for overloading operators & functions.

6.1l

INTRODUCTION :

Ques – Explain the concept of polymorphism. (4M)

- Polymorphism is one of the crucial features of OOP.
- "Polymorphism is a Greek term means "The ability to take more than one forms". An operation may exhibit different behavior in different instances. The behavior depends upon the types of data used in the operation.
- The polymorphism can be 1) Compile time polymorphism or 2) runtime polymorphism
- Operator over loading and function over loading are the best examples of Compile time polymorphism.
- C++ allows operators and functions to operate in different ways, depending upon on what they are operating.
- Consider the example of operator overloading. For example if operands are two numbers, the + operator will generate a sum of two numbers. But if operands are strings the + operator generates the third string by concatenation. This mechanism is called operator overloading.
- Similarly functions can be overloaded to perform different task for different situation, depending upon the data types of the arguments. More than one function can have same name but different number of arguments and / or with different data types. At the time of compilation compiler matches the number and data type of arguments. And accordingly binds the function definition with each function calls. This mechanism is called function overloading.
- In compile time polymorphism the linking of code to be executed in response to a particular function call is done at the compilation that is why it is also called early binding.
- In Run time polymorphism the linking of code to be executed in response to a particular function call is done at the run time that is why it is also called late binding.
- Example of late binding is virtual functions.
- To understand the concept of virtual function considers the following example.
- In the above example all three classes Circle Box and triangle are derived from the common base class Shape.



- All the classes can have the same function Draw () which will draw the respective shapes.
- Now suppose you want to draw a number of shapes on the screen using a one-function call statement. To do so you can create an array of base class pointer, which contains the addresses of different object such as circle, box and rectangle.

And by putting all together in loop we can draw various objects with single function call statement.

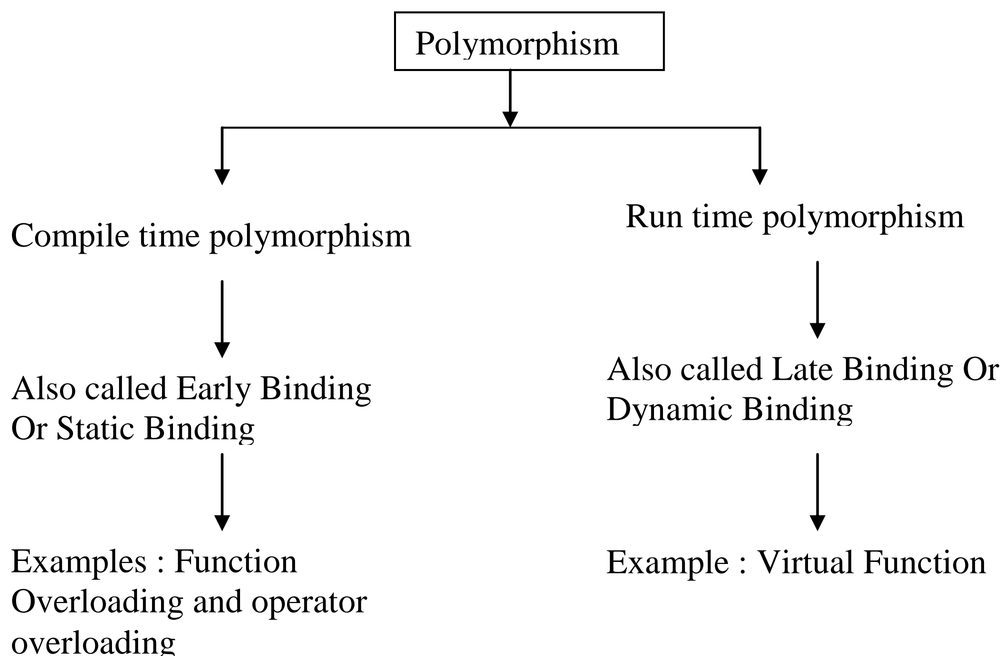
```
For (I=0;I<=10;I++)
{
  p-> draw();
  p++;
}
```

- When pointer contains the address of the object of circle it calls the draw() of circle.
- When pointer contains the address of the object of Box it calls the draw() of Box and so on.
- Polymorphism is extensively used in implementing inheritance.

TYPES OF POLYMORPHISM:

There are two types of polymorphism

1) Compile time polymorphism (Early Binding / static binding)
2) Run time Polymorphism (Late binding / Dynamic binding)



Early Binding / Compile Time Polymorphism / static binding

Ques : Explain static binding using example.

" The term binding refers to the process of linking a function call to the code to be executed in response to that call."

- In compile time polymorphism or early binding the linking of code to be executed in response to a particular function call is done at the compilation time and that is why it is called early binding.
- Operator over loading and function over loading are the best examples of Compile time polymorphism.
- C++ allows operators and functions to operate in different ways, depending upon on what they are operating.

Operator overloading :

- Consider the example of operator overloading. For example the + operator can be overloaded to perform different task depending upon the operands on which it is operating.
- If operands are two numbers, the + operator will generate a sum of two numbers. But if operands are strings the + operator generates the third string by concatenation. This mechanism is called operator overloading.

Function Overloading:

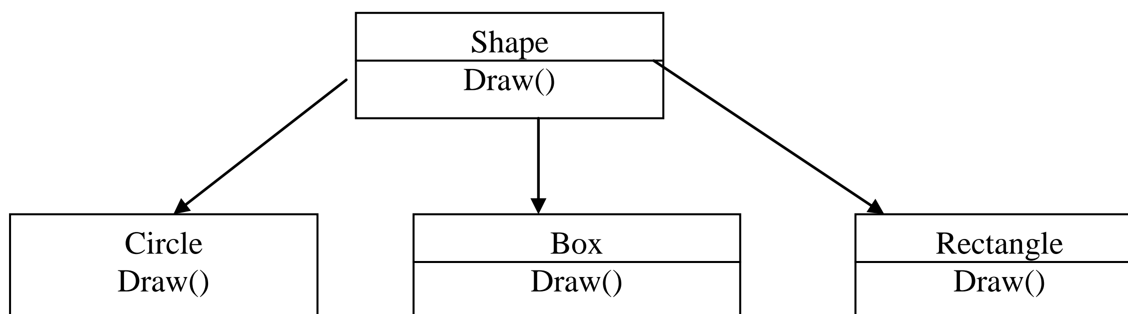
- Similarly functions can be overloaded to perform different task for different situation, depending upon the data types of the arguments passed to it. More than one function can have same name but different number of arguments and / or with different data types. At the time of compilation compiler matches the number and data type of arguments and accordingly binds the function definition with each function call. This mechanism is called function overloading.
- Because in both cases (function overloading and operator overloading) the linking of a code to be executed in response to function call is done at compile time, both the function and operator overloading are called compile time polymorphism or early binding.
- The early binding/ compile time polymorphism is also sometimes called static binding.

Note : If asked for example, also write a small program demonstrating function overloading / operator overloading.

Late Binding / Run Time Polymorphism / Dynamic Binding

Ques : Explain dynamic binding using example:

- " The term binding refers to the process of linking a function call to the code to be executed in response to that call."
- In run time polymorphism (also known as dynamic binding) the linking of code to be executed in response to a particular function call is done at the run time and that is why it is also called late binding.
- Virtual functions are good example of run time polymorphism.
- Consider the following example where there is one base class shape and three derived classes circle , Box and rectangle. All are having the function draw() that draw respective figure.



We know that the base class pointer is always compatible with derived class object. In other words base class pointer can store the address of derived class object.

- But doing so, compiler always checks the type of a pointer (it ignores the contents of pointer) and always call the base class function.
- But if you make the base class function virtual then compiler checks the contents of a pointer rather than its type.
- Here the compiler doesn't know the pointer is going to contain the address of object of which class. It can be address of the object of circle class or object of rectangle. So which version is to call is not known at compile time. When actually the program runs then it comes to know at what object the pointer points to and accordingly it binds the appropriate function.
- Late binding requires some overhead but provides increased power and flexibility.

Note : If asked with example , Write a program demonstrating use of virtual function.
(write the program of shape class given below)

Differentiate Late binding and Early binding

Early Binding	Late Binding
Linking of function call and the code to be executed in response to that call is done at compile time	Linking of function call and the code to be executed in response to that call is done at Run time
Which function to be executed is decided by matching the type and number of arguments passed to the function	Which function is to be executed is decided by looking at the contents of the base class pointer at run time.
Operator overloading and function overloading are the examples of early binding	Virtual functions is the example of late binding
It requires less execution time because process of binding is done at compile time i.e. prior to the execution of program.	Late binding requires some overhead at run time but provides increased power and flexibility
Also known as compile time polymorphism or static binding	Also known as rum time polymorphism or Dynamic binding
This does not require use of pointers to objects	This requires use of pointers to object
Function calls are faster	Function calls execution are slower

FUNCTION OVERLOADING

Ques : Polymorphism is implemented using function overloading . Justify the statement. The general meaning of polymorphism is "The ability to take more than one forms" . Polymorphism can be achieved by using function overloading.

- Function overloading is a feature of c++ that allows a function to behave in more than one way depending upon the number and / or type of arguments passed to it.
- There can be more than one function -definitions with the same name but with the different number of arguments or different data types in a program.
- The compiler determines which function code is be executed in response to the function call, just by matching the number and /or type of arguments passed to it. Since the compiler knows this information at compile time, it is called as compile time polymorphism or early binding.
- Thus, more than one the functions with the same name but with the different no. of arguments or different data types can exist in program.

Example: consider the following function prototypes.

1) int ad (int ,int);   // function prototype to add two integers
2) double add (double ,double); // function prototype to add two doubles
3) int add (int ,int ,int); // function prototype to add three integers

- Therefore when a function is called as add (10,20) ; the first definition will be called.
- For function call add(12.3 ,56.8); the second definition will be called.
- And for function call add (5,6,7); the third definition will be called.

NOTE : when functions are overloaded , compiler does not check for return data type. It only matches the no. and the type of arguments passed to it.

Thus following function definitions will lead to an error.

```
int f (int );
void f (int );
```

Ex:

```
#include<iostream.h>
int volume(int s)
{ return (s*s*s);
}
double volume( double r,int h)
{ return (3.14*r*r*h);
}
void main()
{
cout<<"volume of cube is"<<volume(5);
cout<<"volume of cylinder is"<<volume(10.5,8);
getch();
}
```

PROGRAM BASED ON FUNCTION OVERLOADING:

❖ **Write a program to find out area of a circle and rectangle using Inline function overloading**.

```
# include <iostream.h>
double area (int);              // function prototype
double area ( int ,int );   // function prototype
void main()
{ cout << area ( 4 );       // Invokes first definition
  cout << area (  4, 6 );   // Invokes second definition
}
inline double area ( int r)
{
double A;
A = 3.14 * r * r;// area of a circle = Pi * R *R
return A;
}
inline double area (int l ,int b)
{
double A;
A = l * b;
return A;
}
```

❖ Write a program for function overloading which will find the maximum number between three integers and maximum number between three float numbers.

```
# include <iostream.h>
   int max (int , int ,int);              // function prototype for three integers
   float max ( float, float , float  );            // function prototype for three floats
void main()
{   cout << max ( 3, 6, 8);           // Invokes first definition
    cout << max  ( 3.4 , 6.7 , 1.2);// Invokes second definition
}
int max ( int a ,int b,int c)
{
if ( a > b && a >c)
        return a;
elseif ( b > a && b > c)
        return b;
else
        return c;
```

```
}
float max (float a , float  b , float c)
{
if ( a > b && a >c)
        return a;
elseif ( b > a && b > c)
        return b;
else
        return c;
}
```

Program: Compile time polymorphism using function overloading: (Logic: 2Marks Syntax: 2Marks)

```
#include<iostream.h>   #include<conio.h>
Class Example
{ private:
char ch;
int num;
public:
void show(char a)
{
ch = a;
cout<< " The character is " <<num << endl;
}
Void show(int b)
{
num = b;
cout<<"the integer is" << num << endl;
}};
Void main()
{
clrscr();
Example ob; // object of class example created
ob.show(12);
ob.show(„x‟);
getch();
}
```

O/p :
The integer is : 12
The character is : x

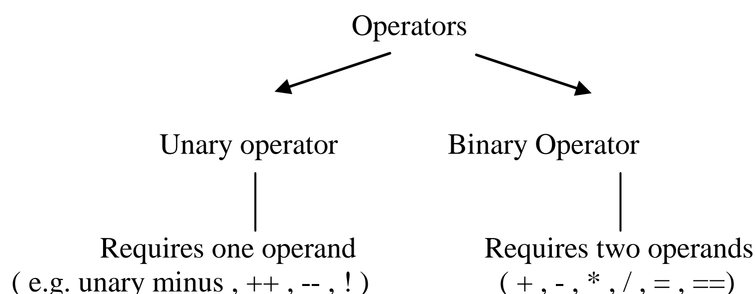
In above program class example has two functions with same name show() but different parameters. Hence we can say that show() function has been overloaded. So it is called as function overloading to implement compile time polymorphism.

OPERATOR OVERLOADING

- Operator overloading refers to giving normal c++ operators such as +, - ,*, / additional meaning when they are applied to the user defined data types.
- It allows us to extend the functionality of an existing operator to operate on user defined data type.
- For example you can create your own data type complex and can overload + operator to add two complex numbers.

Advantages of operator overloading

- Extends the functionality of existing operators so that they can work with user defined data type.
- Operator overloading makes the program more readable and easy to understand.



RULES FOR OVERLOADING OPERATORS:

1) Only existing operators can be overloaded.
2) The overloaded operators must have at least one operand that is of user defined data type.
3) We cannot change the basic meaning of an operator i.e. we cannot redefine the + operator to SunTrust the values.
4) We cannot overload the following operators.
   . (Dot operator)
   .* (pointer to member access op)
   :: (scope resolution op)
   ? : (conditional op)
   sizeof()
5) we cannot use friend function to overload = , ( ) , [ ] , -> however member function can be used.
6) Unary operators overloaded by means of member function take no argument. Unary operators overloaded by means of friend function take one argument.
7) Binary operators overloaded by means of member function take one argument. **B**inary operators overloaded by means of friend function take two arguments.
8) When using binary operators overloaded through member function, the LHS operand must be an object of the same class.

Overloading Unary operators :

Unary operator overloading requires no parameters to the operator member function

❖ **Program to overload unary minus operator <u>to negate data members</u> of class.**

```
# include<iostream.h>
class sample
{
 private :
        int a ,b;
public :
        sample (int x ,int y)
        {
        a= x ;
        b = y;
        }
        void display()
       {
        cout << " a= " << a;
        cout << "b= " << b;
        }
        void operator – (void)
       {
        a = -a;
        b= -b;
        }
};
void main()
{
sample s ( 3,4);
cout << "Before negation ";
s.display ();
cout << "After negation ";
-s ;        // operator fun is invoked
s.display();
}
```

output

Before negation a= 3 b =4

After negation a = -3 b =-4

❖ **Program to overload ++ operator to increment data member of class.**

```
# include <iostream.h>
class increment
{
 private :
        int no;
public :
        increment (int   x) // constructor to initialize no
        {
        no = x ;
        }
        void display()
       {
       cout << " no= " << no;
       }
        void operator ++ (void)
       {
        no++;
        }
};
void main()
{
increment  s ( 7 );
cout << "Before Increment ";
s.display ();
cout << "After increment ";
++s ;     // operator fun is invoked
s.display();
}
```

output
Before increment, no = 7
After increment, no = 8

```
//Program to overload ++ operator to  increment data member of  class.
#include<iostream>
using namespace std;
class increment
{
private:
        int no;
public:
        increment(int x) // constructor to initialize no
        {
        no = x ;
   }
        void display()
   {
   cout << "\n  no= " << no;
   }
```

```
            void operator++(void)
    {
            no++;
            }
            void operator--(void)
    {
            no--;
            }
};
int main()
{
increment  s(7);
cout << "\n Before Increment ";
s.display ();
cout << "\n After Increment ";
++s ;     // operator fun is invoked
s.display();
increment  s(6);
cout << "\n Before Decrement ";
s.display ();
cout << "\n After Decrement  ";
--s ;       // operator fun is invoked
s.display();
}
```

- **Program to overload the ! operator to find out factorial of a number (specimen paper)**

```
# include <iostream.h>
class fact
{
int no;
public:
fact (int a)
{
 no=a;
}
void operator !(void)
{
 long  I = 1;
  for (int I =1; I<=no ; I++)
 {
   f = f * I;
```

```
 }
cout << “Factorial = “ << f;
}
};
void main()
{
 fact f1 (5);
 !f1;
}
```

❖ **Write a program to overload ++ operator to increment the distance by one (distance is in feet and inches)**

```
# include <iostream.h>
Class distance
{
        int feet , inches;
public :
distance ()        // default constructor
{ }
distance (int f , int I)
{
 feet = f;
inches = I;
}
void operator ++ (void)
{
  inches = inches + 1;      // increment inches by 1
  if (inches >= 12 )        // if inches > 12 then convert inches into feet
 {
  inches = inches - 12;
  feet++;
  }
}
void showdata(void)
{
cout << feet << “Feet and  ” << inches << “ Inches ” ;
}
};
void main()
{
distance d1(3,6);
++d1;
d1.showdata();
}
OUTPUT
3 feet and 7 inches
```

Overloading Binary operators:

Binary operator overloading requires one parameter (object of the same class) to the operator member function.

❖ **Write a program to overload + operator to add two Time objects.**

```
# include <iostream.h>
Class time
{
 int hr , min;
public :
        time();     // default constructor
        time (int , int);     // constructor to initialize he and min
        void showdata (void);
        time operator + ( time);
};
time :: time (void)
{
```

```
  hr =0 ; min =0;
}
time :: time (int hh , int mm)
{
 hr = hh;
 min =mm;
}
time time ::  operator +  (time t2 )
{
  time temp;
  temp.hr = hr + t2.hr;
  temp.min = min + t2.min;
  if (temp.min >= 60 )
 {
   temp.hr = temp.hr + (temp.min / 60);
   temp.min = temp.min % 60;
  }
 return temp;
}
void time :: showdata(void)
{
cout << hr << ":" << min ;
}
void main ()
{
time t1(3,50) ,t2( 4,30) ;
time t3;  // default constructor is invoked
t3 =t1 +  t2 ;      // result is in t3
t3.showdata();
}
```

OUTPUT

8 **:** 20

Note the following features of operator function

1) It takes only one time object as an argument.
2) It is a member function of time class.
3) When we write the statement t3 =t1 + t2 the object which is on LHS of the + operator (i.e. t1) invokes the operator function and the object on the RHS of the + i.e. t2 is passed explicitly to the operator function. Here object c1 is invoking the operator function that is why its data members are accessed directly.

Assignment: write a program to add two distance objects represented in feet and inches.

- **Write a program to overload + operator to add two complex numbers.**

```
# include<iostream.h>
class complex
{
   int real,;
   float img;
public :
            complex ( ); // default constructor
            complex ( int , float);
            complex operator + ( complex );
            void showdata(void);
};
complex :: complex ()
{
  real =0 ;
  img = 0.0
 }
complex  :: complex ( int r , float i)
```

```
  {
        real = r;
        img =i;
  }
complex  complex :: operator ( complex c2)
{
 complex temp; // create a temporary object
 temp. real = real + c2.real ;
 temp. img = img + c2.img;        // result is in temp object
 return temp;                  // return temp object to the main prog.
}
void complex :: showdata (void)
{
 cout << real << "+" << "i" << img;
}
void main(void)
{
  complex c1(5,6)  ,c2 ( 3,4) ,c3;
  c3 =c1 + c2 ;   // c2 is explicitly passed , result will be in c3.
  C3. showdata();
}
```

OUTPUT
8 + i 10

FUNCTION OVERRIDING:

"When a base class and derived class define a function of same name , same number of arguments with same data types, the derived class function supersedes (take precedence over) the base class function. This mechanism is called function overriding."

- The base class function will be called only if the derived class does not redefine the function.
- Here the base class function is said to be overridden by the derived class function.

```
Class base
{
  public :
        void fun1 (void)
        {
              cout << "In base class fun1 "<<endl;
        }
        void fun2 (void)
        {
              cout << "In base class fun2 "<<endl;
```

```
        }
};
class derived : public base
{
  public :
        void fun1 (void)          // redefining fun1 ()
        {
                cout << "In derived class fun1 " <<endl;
        }
        void fun3 (void)          // redefining fun1 ()
        {
                cout << "In derived class fun3 " <<endl;
        }
};
void main (void)
{
derived d1;
d1.fun1();        // Invokes derived :: fun1()
d1.fun2 () ;   //Invokes base :: fun2()
d1.fun3();        // Invokes derived :: fun3()
}
```

OUTPUT

In derived class fun1

In base class fun2

In derived class fun3

- The rule is when the same function exist in both base and derived class the derived class function will be executed.

Function overloading	Function overriding
1)When more than one function exists with the same name but with different no. of arguments and / or different data types it is called function overloading.	1)The same function name with the same prototype exist in base class and derived class, it is called function overriding.
2) They can be static or non-static	2) They must be non-static members of classes
3) Overloaded functions can be friends	3) These functions cannot be friends
4) Constructor can be overloaded but destructors cannot be overloaded	4) Constructors & destructors cannot be override.
5) The function to be executed is determined by matching the no. and type of arguments passed to it.	5) When function is overridden always the derived class function is invoked.

Example:Class area
{
public :
void area (int) // overloaded function
{ }
void area (int ,int) // overloaded fun
{ }
};
	Example:Class base
	{
	void display() // overriden function
	{ }
	};
	class derived : public base
	{
	void display ()
	{ }
	};

RUN TIME POLYMORPHISM

VIRTUAL FUNCTIONS:

Ques : Explain the concept of virtual function.

Ques : What is virtual function . Why do we need virtual function?

- Virtual function is that which does not really exist but nevertheless appears real to some part of the program.
- We use virtual functions when we have number of objects of different classes and we want to put them together on a list and perform a particular operation using a single function call.
- When we use a pointer to base class to refer to the derived class object, always base class function is invoked, even if the pointer contains the address of derived class object. This is because; compiler ignores the contents of pointer and chooses the member function that matches the type of a pointer.
- Using virtual function mechanism we can force the compiler to look at the contents of the pointer rather than the type of the pointer at the time invoking the function.

- When the same function exist in both base and derived class, the base class function is made virtual using keyword virtual.
- When a function in a base class is made virtual, C++ determines which function to invoke at run time based on the type of object pointed by the base class pointer rather than the type of the pointer.
- Thus by making the base class pointer to point to different objects, we can execute different versions of the virtual functions.
- Runtime polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Syntax for declaring virtual function :

```
virtual return_type function_name(parameter if any)
{
---------
---------
}
```

Example :

```
#include<iostream.h>
#include<conio.h>
class base
{
public:
void display()
{
cout<<"Display Base";
}
virtual void show()
{
cout<<"Show Base";
}
}
class derived:public base
{
public:
void display()
{
cout<<"Display Derived";
}
void show()
{
cout<<"Show Derived";
}
};
void main()
{
base b1;
derived d1;
base=*bptr;
bptr=&b1;
bptr->display();
bptr->show();
bptr=&d1;
bptr->display();
bptr->show();
getch();
}
```

```
Class shape
{
 public : virtual void draw()
            { }
};
class circle : public shape
{
            public :
```

```
            void draw ()
            {
             // code to draw a circle
            }
};
class box :public shape
{
   public: void draw()
            {
             // assume code to draw a box
            }
};
class rectangle :public shape
{
 public: void draw()
            {
             // assume code to draw a rectangle
            }
};
void main()
{
shape *p;
circle c;
p =&c ;
c -> draw() ;      // invokes circle :: draw();
rectangle r;
p = & r;
p -> draw();       // invokes rectangle :: draw()
box b;
p =&b;
p-> draw();        // Invokes box :: draw()
}
```

RULES FOR DECLARING VIRTUAL FUNCTIONS:

Ques : state any four rules to implement dynamic binding using virtual function.

Ans :

1. The virtual function must be a member of some class. The base class function is made virtual.
2. A virtual function must be defined in a base class, even though it may not be used.
3. The prototype of the base class version of a virtual function and all the derived class versions must be identical. If two functions with same name but with different data types exist then C++. Considers it as overloaded functions and the virtual function mechanism is ignored.
4. A base class pointer can point to any derived class object but the reverse is not true.
5. If a virtual function is defined in the base class it need not be necessarily redefined in the derived class. In such cases the base class version is invoked.
6. They cannot be static members.
7. When a class containing a virtual function is inherited the derived class redefines the virtual function to fit its own needs.
8. Virtual function implement “one interface multiple methods”
9. It is capable of supporting run-time.
10. When a base pointer to derived object that contains a virtual function, c++ determines which version of that function to call based upon the type of object pointed by the pointer and this is determined at run time.
11. Thus, when different objects are pointed to, different versions of functions are executed. Same effect applies to base class reference.

PROGRAM BASED ON VIRTUAL FUNCTIONS:

Create a class called shape. Use this class to store two double values that could be used to compute the area of a figure. Derive two classes’ triangle and rectangle from the base class shape. Add to the base class , a member function getdata() to initialize values and another function display_area() to compute and display area of figure.. Make display_area() function of shape class virtual.

```
# include <iostream.h>
class shape
```

```
{         private : double a,b;
public:  void getdata()
          {
           cout << "Enter  value of a and b";
           cin >> a>.b;
          }
          virtual void display_area()  // virtual function
          {
           }
};
class triangle: public shape
{
public:void display_area()
          {
           float area;
           area = 0.5 * a*b;
           cout <<"Area of triangle : "<<area;
          }
};
class rectangle: public shape
{
public:void display_area()
          {
           float area;
           area =   a * b;
           cout <<"Area of rectangle : "<<area;
          }
};
void main();
{
shape *p;
rectangle r1;
p = &r1;
cout << "Enter data for rectangle :"<<endl;
p->getdata();
p->display_area();
triangle t1;
cout << "Enter data for triangle :"<<endl;
p = &t1;
p->getdata();
p->display_area();
}
```

OUTPUT:

Enter data for rectangle:
Enter value of a and b: 2 3
Area of rectangle : 6
Enter data for triangle:
Enter value of a and b: 12 4
Area of rectangle : 24

A pure virtual function

So far, all of the virtual functions we have written have a body (a definition). However, C++ allows you to create a special kind of virtual function called a **pure virtual function** (or **abstract function**) that has no body at all! A pure virtual function simply acts as a placeholder that is meant to be redefined by derived classes.To create a pure virtual function, rather than define a body for the function, we simply assign the function the value 0.

A **pure virtual function** is a function that has ***the notation "= 0"*** in the declaration of that function. Here is a simple example of what a pure virtual function in C++ would look like:

Simple Example of a pure virtual function in C++

```
class SomeClass {
public:
   virtual void pure_virtual( ) = 0;  // a pure virtual function
   // note that there is no function body
};
```

When we add a pure virtual function to our class, we are effectively saying, "it is up to the derived classes to implement this function".Using a pure virtual function has two main consequences: First, any class with one or more pure virtual functions becomes an **abstract base class**, which means that it cannot be instantiated. Second, any derived class must define a body for this function, or that derived class will be considered an abstract base class as well.

```
// pure virtual members can be called from the abstract base class
#include <iostream>
class Polygon
{  protected:     int width, height;
  public:
    void set_values (int a, int b)
      { width=a; height=b; }
    virtual int area() =0;
    void printarea()
      { cout << this->area() << '\n'; }
};
class Rectangle: public Polygon
{  public:
    int area (void)
      { return (width * height); }
};
class Triangle: public Polygon
{  public:
    int area (void)
      { return (width * height / 2); }
};
int main ()
 {  Rectangle rect;
  Triangle trgl;
  Polygon * p1 = &rect;
  Polygon * p2 = &trgl;
  p1->set_values (4,5);
  p2->set_values (4,5);
  p1->printarea();
  p2->printarea();
  return 0;
}
```