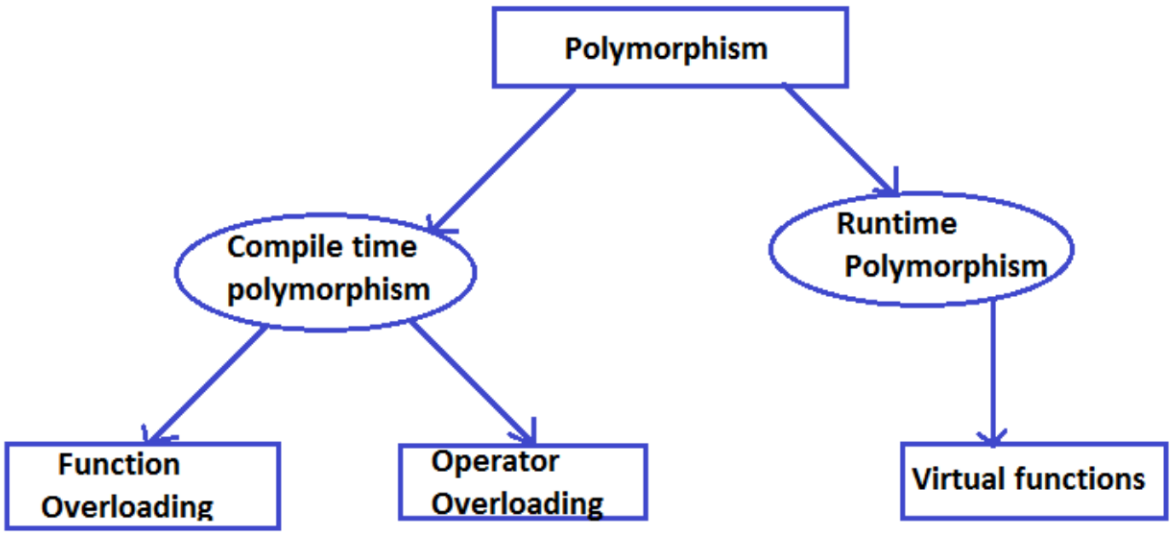
UNIT 4

POLYMORPHISM

Introduction:

- Polymorphism means 'One name-multiple forms'
- Greek word **"poly"** means many and **"morphos"** means form.
- C++ polymorphism means that a call to member function will cause a different function to be executed depending on the type of object invokes the functions.
- There are two types of polymorphism. They are compile time polymorphism and runtime polymorphism.



❖ Compile time Polymorphism

- Compile time polymorphism means that an object is bound to its function call at compile time.
- In this type of polymorphism Compiler is able to select the appropriate function for particular function call at compile time.
- This mechanism is also called early binding or static binding or static linking.
- Compile time polymorphism is achieved in two ways.
  - Function overloading
  - Operator overloading

❖ Runtime Polymorphism

- The selection of appropriate function is done dynamically at runtime. Thus it is not known which function will be invoked till an object actually makes a function call during program execution.
- This mechanism is also called late binding or dynamic binding.
- Runtime polymorphism can be achieved with the help of virtual functions.

Function Overloading

- The method of using same function name but with different parameter list along with different data type to perform the variety of different tasks is known as function overloading.
- The correct function is to be invoked is determined by checking the number and type of arguments but not function return type.

```
#include<iostream>
using namespace std;
class calcarea
{
public:
int area(int s)
{
        return (s*s);
}
int area(int l,int b)
{
        return(l*b);
}

float area(float r)
{
        return(3.14*r*r);
}
};

int main()
{
calcarea c1;
cout<<"Area of square="<<c1.area(5)<<endl;
cout<<"Area of Rectangle="<<c1.area(5,10)<<endl;
cout<<"Area of Circle="<<c1.area(2.5f)<<endl;
return 0;
}
```

Operator overloading

- The mechanism of adding special meaning to an operator is called operator overloading.
- It provides a flexibility for the creation of new definitions for most C++ operators.
- Using operator overloading we can give additional meaning to normal C++ operations such as (+,-,=,<=,+= etc.) when they are applied to user defined data types.

Things to be understand

- Usually Operations can perform only on basic data type. Example int a,b,c;
  c=a+b; But if we declare a class complex {} and complex c1,c2,c3 ;
  c3=c1+c2;is not possible because object is user defined data type.
- Operator overloading helps to define usuage of operator for user defined data type i.e. objects.
- After overloading operands used with operator are objects instead of basic data types.

Operators that cannot be overloaded

All operators are not overloaded. The operators that are not overloaded are:

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

General form of Operator function

The general form of operator function

```
return_type operator op(arglist)
{
function body   //task defined
}
OR
return_type classname::operator op(arglist)
{
function body     //task defined
}
```

Where,

- **return_type** is the type of value returned by the specified operation.
- **op** is the operator being overloaded.
- **Operator op** is function name, where **operator** is keyword.

Note:

- Operator function must be either member function (non-static) or friend function.
- Friend function will have only one argument for unary operators and two for binary operators.
- Member function has no arguments for unary operators and only one for binary operators.

  This is because the object used to invoke the member function is passed implicitly and therefore available for member function. This is not the case with friend functions.

Arguments may be passed either by value or by reference. Operator functions are declared in class using prototype as follows.

```
vector operator+(vector);       //vector addition
vector operator-();             //unary minus
friend vector operator+(vector,vector);       //vector addition
friend vector operator-(vector);              //unary minus
vector operator-(vector &a);                  //subtraction
int operator==(vector);                       //comparison
friend int operator==(vector,vector);         //comparison
```

vector is a datatype of class and may represent both magnitude and direction (as in physics and engineering) or a series of points called elements(as in mathematics).

The process of overloading involves the following steps:

1. Create a class that defines the data type that is used in the overloading function.
2. Declare the operator function **operator** op() in the public part of class. It may be either a member function or a **friend** function.
3. Define operator function to implement the required operations.

Operator overloaded function can be invoked using expression such as:

	In case of member function	**In case of friend function**
For unary operators	**op x or x op** **(eg.++x, or x++)** *x.operator op();*	**op x or x op** **(eg.++x, or x++)** *operator op(x)*
For binary operators	**x op y** **eg x+y** *x.operator op(y);*	**x op y** **eg x+y** *operator op(x,y)*

Note:

- Here **op** represents the operator being overloaded.
- **x** and **y** represents the object.

Overloading Unary Operators

Unary operators operates on single operand. Some of unary operators are

- Increment operator (**++**)
- Decrement operator (**--**)
- Unary minus operator(**-**)

Eg. **++a** ; Here a is only one operand.

Overloading unary minus Operator

Overloading unary minus using member function

```
#include<iostream>
using namespace std;
class space
{
private:
int x;
int y;
int z;
public:
void getdata(int a,int b,int c);
void display();
void operator -();
};
void space::getdata(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
void space::display()
{
 cout<<"x="<<x<<endl;
 cout<<"y="<<y<<endl;
 cout<<"z="<<z<<endl;
}
void space::operator -()
{
x=-x;
y=-y;
z=-z;
}
```

```
int main()
{
space s;
s.getdata(5,10,15);
cout<<"s:"<<endl;
s.display();
-s;
cout<<"-s:"<<endl;
s.display();
return 0;
}
```

Output:

```
s:
x=5
y=10
z=15
-s:
x=-5
y=-10
z=-15
```

Overloading unary minus using friend function

```
#include<iostream>
using namespace std;
class space
{
private:
int x;
int y;
int z;
public:
void getdata(int a,int b,int c);
void display();
friend void operator -(space &s);
};
void space::getdata(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
```

```
void space::display()
{
 cout<<"x="<<x<<endl;
 cout<<"y="<<y<<endl;
 cout<<"z="<<z<<endl;
}
void operator -(space &s)
{
s.x=-s.x;
s.y=-s.y;
s.z=-s.z;
}
int main()
{
space s;
s.getdata(5,10,15);
cout<<"s:"<<endl;
s.display();
-s;
cout<<"-s:"<<endl;
s.display();
return 0;
}
```

WAP to overload unary (-) minus operator so that the statement s2=-s1 can be returned when s1 and s2 are of type space that represent 3 dimensional coordinate system.
(using friend function)

```
#include<iostream>
using namespace std;
class space
{
private:
int x;
int y;
int z;
public:
void getdata(int a,int b,int c);
void display();
friend space operator -(space s);
};
```

```
void space::getdata(int a,int b,int c)
{
x=a;
y=b;
z=c;
}
void space::display()
{
 cout<<"x="<<x<<endl;
 cout<<"y="<<y<<endl;
 cout<<"z="<<z<<endl;
}
space operator -(space s)
{
space temp;
temp.x=-s.x;
temp.y=-s.y;
temp.z=-s.z;
return temp;
}

int main()
{
space s1,s2;
s1.getdata(5,10,15);
cout<<"s:"<<endl;
s1.display();
cout<<"-s:"<<endl;
s2=-s1;
s2.display();
return 0;
}
```

Write a simple program to overload (unary ++) operator. [PU:2016 spring]

OR

WAP to overload prefix increment (++) operator by returning value through object

```
#include <iostream>
using namespace std;
class counter
{
private:
int count ;
public:
void getdata (int x)
{
count=x ;
}
void showdata()
{
 cout<<"count="<<count<<endl;
}
counter operator ++();
};
counter counter::operator ++()
{
    counter temp;
    temp.count=++count;
    return temp;
}
int main()
{
counter c1,c2;
c1.getdata(3) ;
c1.showdata() ;
c2=++c1 ;
c1.showdata();
c2.showdata();
return 0;
}
```

Output

```
count=3
count =4
count =4
```

Unary operator overloading for postfix increment.

```
#include<iostream>
using namespace std;
class counter
{
private:
int count;
public:
void getdata (int x)
{
count=x;
}
void showdata( )
{
cout<<"count="<<count<<endl;
}
counter operator ++(int);
};
counter counter::operator ++(int)
{
counter temp;
temp.count=count++;
return temp;
}
int main( )
{
counter c1,c2;
c1.getdata(5);
c1.showdata();
c2=c1++;
c1.showdata();
c2.showdata();
return 0;
}
```

Output:

```
count =5
count =6
count =5
```

Write a program to generate Fibonacci series using operator overloading of (++) operator. Which type of overloading is it.[PU:2009 fall]

```
#include<iostream>
using namespace std;
class fibo
{
private:
int a,b,c;
public:
fibo()
{
a=-1;
b=1;
c=a+b;
}
void display()
{
cout<<c<<",";
}
void operator++()
{
 a=b;
 b=c;
 c=a+b;
}
};
int main()
{
        int n,i;
        fibo f;
        cout<<"Enter the number of terms"<<endl;
        cin>>n;
        for(i=1;i<=n;i++)
        {
        f.display();
        ++f;
        }
        return 0;
}
```

This is an example of unary operator overloading.

Binary Operator overloading

The operator which operates on two operands is known as binary operators.

For Example c=a+b where a and b are two operands.

Overloading + operator

Write a program to add two complex number using binary operator overloading.

[PU:2013 fall]

```
#include<iostream>
using namespace std;
class complex
{
private:
        int real,imag;
public:
        complex()
        {
        }
        complex(int r,int i)
        {
                real=r;
                imag=i;
        }
        void display()
        {
        cout<<real<<"+"<<imag<<"i"<<endl;
        }
        complex operator +(complex c2)
        {
                complex temp;
                temp.real=real+c2.real;
                temp.imag=imag+c2.imag;
                return temp;
        }
};
```

```
int main()
{
complex c1(2,3);
complex c2(2,4);
complex c3;
c3=c1+c2;
cout<<"c1=";
c1.display();
cout<<"c2=";
c2.display();
cout<<"c3=";
c3.display();
return 0;
}
```

WAP to Overload Binary operator (+) using friend function.

```
#include <iostream>
using namespace std;
class complex
{
private:
int real ;
int imag ;
public:
complex()
{
}
complex(int r, int i)
{
real=r;
imag=i;
}
void display()
{
cout<<real<< "+"<<imag<<"i"<<endl;
}
friend complex operator+ (complex,complex) ;
} ;
```

```
complex operator + (complex c1, complex c2)
{
complex temp ;
temp.real=c1.real+c2.real;
temp.imag=c1.imag+c2.imag;
return temp;
}
int main()
{
complex c1(5,3);
complex c2(3,4);
complex c3;
c3=c1+c2;
cout<<"c1=";
c1.display();
cout<<"c2=";
c2.display();
cout<<"c3=";
c3.display();
return 0;
}
```

Write a program to overload the arithmetic operators(+,-,*./)

```
#include<iostream>
using namespace std;
class Arithmetic
{
   private:
   float num;
   public:
   void getdata()
   {
      cout<<"Enter the number"<<endl;
      cin>>num;
    }
    void putdata()
    {
      cout<<num<<endl;
    }
    Arithmetic operator+(Arithmetic);
    Arithmetic operator*(Arithmetic);
    Arithmetic operator-(Arithmetic);
    Arithmetic operator/(Arithmetic);
};
```

```
Arithmetic Arithmetic::operator+(Arithmetic b)
{
   Arithmetic temp;
   temp.num=num+b.num;
   return temp;
}
Arithmetic Arithmetic::operator*(Arithmetic b)
{
   Arithmetic temp;
   temp.num=num*b.num;
   return temp;
}
Arithmetic Arithmetic::operator-(Arithmetic b)
{
   Arithmetic temp;
   temp.num=num-b.num;
   return temp;
}
Arithmetic Arithmetic::operator/(Arithmetic b)
{
   Arithmetic temp;
   temp.num=num/b.num;
   return temp;
}

int main()
{
  Arithmetic a,b,c;
  a.getdata();
  b.getdata();
  c=a+b;
  cout<<"Additon of two objects="<<endl;
  c.putdata();
  cout<<"Multiplication of two objects="<<endl;
  c=a*b;
  c.putdata();
  cout<<"Substraction of two objects="<<endl;
  c=a-b;
  c.putdata();
  cout<<"Division of two objects="<<endl;
  c=a/b;
  c.putdata();
  return 0;
}
```

WAP to overload two binary operator '+' and '-' so that statement
c3=c1+c2;
c3=c1-c2; exists
OR
Write a program to find the sum and difference of any two complex number by overloading '+' and '-' operator .[PU:2019 fall]

```
#include<iostream>
using namespace std;
class complex
{
private:
 int real,imag;
public:
complex()
{
}
complex(int r,int i)
{
real=r;
imag=i;
}
void display()
{
cout<<real<<"+i"<<imag<<endl;
}
friend complex operator +(complex c1,complex c2);
friend complex operator -(complex c1,complex c2);
};
complex operator +(complex c1,complex c2)
{
complex temp;
temp.real=c1.real+c2.real;
temp.imag=c1.imag+c2.imag;
return temp;
}
complex operator -(complex c1,complex c2)
{
complex temp;
temp.real=c1.real-c2.real;
temp.imag=c1.imag-c2.imag;
return temp;
}
```

```
int main()
{
complex c1(15,10);
complex c2(10,5);
complex c3;
c3=c1+c2;
cout<<"After overloading + operator"<<endl;
c3.display();
c3=c1-c2;
cout<<"After overloading - operator"<<endl;
c3.display();
return 0;
}
```

Write a class Time with three integer attributes hour, minute and second. Include the following responsibilities in class

i. **Default and parameterized constructor**
ii. **Display method to display time in hour, minute and second format.**
iii. **Appropriate function overload to realize addition of two time objects with '+' operator[PU:2013 spring]**

```
#include<iostream>
#include <conio.h>
using namespace std;
class Time
{
   private:
   int hour,minute,second;
   public:
   Time()
   {
          hour=2,
          minute=45;
          second=55;
   }
   Time(int h,int m,int s)
   {
   hour=h;
   minute=m;
   second=s;
  }
```

```
    void display()
    {
        cout<< hour<< ":"<<minute<<":"<< second<<endl;
    }
    Time operator+(Time);
};

Time Time::operator+(Time t1)
{
    Time temp;
    temp.second=second+t1.second;
    temp.minute=temp.second/60;
    temp.second=temp.second%60;
    temp.minute=temp.minute+minute+t1.minute;
    temp.hour=temp.minute/60;
    temp.minute=temp.minute%60;
    temp.hour=hour+temp.hour+t1.hour;
    return temp;
}

int main()
{
    Time t1;
    Time t2(2,35,10);
    Time t3;
    t3 = t1 + t2;
    cout<<"T1=";
    t1.display();
    cout << "T2=";
    t2.display();
    cout << "T3=";
    t3.display();
    return 0;
}
```

Alternative solution:

```
#include<iostream>
using namespace std;
#include <conio.h>
class Time
{
   private:
   int hour,minute,second;
   public:
   Time()
   {
      hour=2;
      minute=45;
      second=55;
   }
   Time(int h,int m,int s)
   {
   hour=h;
   minute=m;
   second=s;
  }
   void display()
   {
      cout<< hour<< ":"<<minute<<":"<<second<<endl;
   }
   friend Time operator+(Time,Time);
};

Time operator+(Time t1,Time t2)
{
   Time temp;
   temp.second=t1.second+t2.second;
   temp.minute=temp.second/60;
   temp.second=temp.second%60;
   temp.minute=temp.minute+t1.minute+t2.minute;
   temp.hour=temp.minute/60;
   temp.minute=temp.minute%60;
   temp.hour=temp.hour+t1.hour+t2.hour;
   return temp;
}
```

```
int main()
{
   Time t1;
   Time t2(2,35,15);
   Time t3;
   t3 = t1 + t2;
   cout<<"T1=";
   t1.display();
   cout <<"T2=";
   t2.display();
   cout <<"T3=";
   t3.display();
   return 0;
}
```

Write a program to overload '+=' operator to add distance of two objects.

```
#include<iostream>
using namespace std;
class Height
{
private:
int feet;
int inches;
public:
Height(int f, float i)
{
feet=f;
inches=i;
}
void display()
{
cout<< feet <<"feet and "<<inches<<"inches"<<endl;
}
void operator +=(Height h)
{
feet+=h.feet;
inches+= h.inches;
if(inches>=12)
{
inches=inches-12;
feet++;
}
}
};
```

```
int main()
{
Height h1(5,9);
Height h2(10,5);
cout <<"first height=";
h1.display();
cout <<"second height=";
h2.display();
h1 += h2;
cout <<"After addition h1=";
h1.display();
return 0;
}
```

WAP to concatenate two strings by overloading '+' operator.

```
#include<iostream>
#include<string.h>
using namespace std;
class stringc
{
private:
char str[50];
public:
stringc()
{

}
stringc(char s[])
{
strcpy(str,s);
}
void display()
{
cout<<"String is:"<<str<<endl;
}
stringc operator +(stringc s2)
{
stringc s3;
strcpy(s3.str,str);
strcat(s3.str,s2.str);
return s3;
}
};
```

```
int main()
{
stringc s1("pradip");
stringc s2("paudel");
stringc s3;
s1.display();
s2.display();
s3=s1+s2;
s3.display();
return 0;
}
```

Write a program to overload == operator

```
#include<iostream>
using namespace std;
class Time
{
  private:
   int hours, minutes, seconds;
   public:
   Time()
   {
      hours=0;
      minutes=0;
      seconds=0;
   }
   Time(int h, int m, int s)
   {
      hours=h;
      minutes=m;
      seconds=s;
   }
 friend int operator==(Time t1, Time t2);
};
int operator==(Time t1, Time t2)
{
   if ( t1.hours==t2.hours&&t1.minutes==t2.minutes&&t1.seconds==t2.seconds )
   {
          return 1;
          }
          else
          {
                  return 0;
          }
}
```

```
int main()
{
   Time t1(3,15,45);
   Time t2(4,15,45);
   if(t1 == t2)
   {
      cout << "Both the time values are equal";
   }
   else
   {
      cout << "Both the time values are not equal";
   }
   return 0;
}
```

WAP to overload > operator.

```
#include<iostream>
using namespace std;
class Maximum
{
private:
int x;
public:
Maximum(int a)
{
x=a;
}

int operator >(Maximum m2)
{
        if(x>m2.x)
        {
          return 1;
        }
        else
        {
          return 0;
        }
}
```

```
void display()
{
        cout<<"Maximum value="<<x<<endl;
}
};
int main()
{
Maximum m1(55);
Maximum m2(10);
if(m1>m2)
{
m1.display();
}
else
{
m2.display();
}
return 0;
}
```

Function overriding

If we inherit a class into the derived class and provide a definition for one of the base class function again inside the derived class, then that function is said to be overridden, and this mechanism is called Function Overriding.

Requirements for Overriding a Function

- Inheritance should be there. Function overriding cannot be done within a class. For this we require a derived class and a base class.
- Function that is redefined must have exactly the same signature in both base and derived class, that means same name, same return type and same number and type of parameter list.

```
#include<iostream>
using namespace std;

class A
{
public:
void display()
{
cout<<"This is Base class"<<endl;
}
};
class B : public A
{
public:
void display()
{
cout<<"This is Derived class"<<endl;
}
};
int main()
{
B b;
b.display();
return 0;
}
```

output:

```
This is Derived class
```

Here, the function display() is overridden and If the function is invoked from an object of the derived class, so the function in the derived is executed.

Object pointer

The pointer pointing to objects are referred to as object pointer.

Declaration

```
class_name *object_pointer_name;

Eg. student  *ptr;
```

Here, ptr is an object pointer of student class type has been declared. where ,student is already defined class.

Initialization

```
object_pointer_name=&object;

Eg. ptr=&st;
```

Here, ptr is an object pointer of student class type and st is an object of class student.

Note: When accessing members of class using object pointer the arrow operator **(->)** is used instead to dot operator.

Example:

```
#include<iostream>
using namespace std;
class student
{
private:
char name[20];
int roll;
public:
void getdata()
{
cout<<"Enter student Name"<<endl;
cin>>name;
cout<<"Enter student Rollno"<<endl;
cin>>roll;
}

void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Rollno:"<<roll<<endl;
}
};
```

```
int main()
{
student st;
student *ptr;
ptr=&st;
ptr->getdata();
ptr->display();
return 0;
}
```

this pointer

- **this** pointer stores the address of current calling object. For example, consider an object obj calling one of its member function say method() as obj.method(). Then, this pointer will hold the address of object obj inside the member function method().
- **this** pointer is automatically passed to a member function when it is called. The pointer t**his** acts as an implicit argument to all member function.
- **this** pointers are not accessible for static member functions.
- **this** pointers are not modifiable.

Application:

1. this pointer can be used to refer current class instance variable. Another application of this pointer is distinguishing data members from local variables of member functions if they have same name.

Example:

```
#include<iostream>
using namespace std;
class Employee
{
private:
int eid;
float salary;
public:
        Employee(int eid,float salary)
        {
                this->eid=eid;
                this->salary=salary;
        }
```

```
        void display()
        {
        cout<<"Employee ID="<<eid<<endl;
        cout<<"Salary="<<salary<<endl;
        }
};

int main()
{
Employee e1(101,25452.55);
Employee e2(102,54485.25);
e1.display();
e2.display();
return 0;
}
```

2. One important application of the pointer this is to return the objects it points to.
   For example, the statement
   return *this;
   inside a member function will return the object that invoked the function.

Example:

```
#include<iostream>
#include<string.h>
using namespace std;
class person
{
char name[20];
float age;
public:
person()
{
}
person (char n[],float a)
{
strcpy(name,n);
age=a;
}
```

```
Person& greater(person &x)
{
        if(x.age>=age)
        {
                return x;
        }
        else
        {
                return *this;
        }
}
void display()
{
cout<<"Name:"<<name<<endl;
cout<<"Age:"<<age<<endl;
}
};

int main()
{
person p1("Ram",52);
person p2("Hari",24);
person p3;
p3=p1.greater(p2);
cout<<"Elder person is:"<<endl;
p3.display();
return 0;
}
```

Pointer to derived class

Can you derive a pointer from a base class? Explain with suitable example.

Yes, we can derive a pointer from a base class. Pointers to object of base class are type compactible with pointer to objects of the derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if **B** is a base class and **D** is the derived class from **B**, then a pointer declared as a pointer to **B** can also be a pointer to **D**. consider the following declarations.

```
B *bptr;        //pointer to class B type variable
B b;            //base object
D d;            //derived object
bptr=&b;        //bptr points to object b

We can make bptr to point to the object d as follows:

bptr=&d;        //bptr points to object d


This is perfectly valid with C++ because d is an object derived from the class B.
```

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. we can access only those members which are inherited from **B** and not the members that originally belong to **D**. We may have to use another pointer declared as pointer to derived type.

Program:

```
#include<iostream>
using namespace std;
class B
{
   public:
   int x;
   void display()
   {
    cout<<"x="<<x<<endl;
   }
};
class D:public B
{
   public:
   int y;
   void display()
   {
      cout<<"x="<<x<<endl;
      cout<<"y="<<y<<endl;
   }
};
```

```
int main ()
{
   B b1;
   B *bptr;                //base pointer
   bptr=&b1;               //base address
   bptr->x = 10;           //access B via base pointer
   cout<<"bptr points to base object"<<endl;
   bptr->display();

   D d1;
   bptr=&d1;   //address of derived object
   bptr->x=10; //access D via base pointer
   //bptr->y = 20;  wont work
   cout<<"bptr now points to derived object"<<endl;
   bptr->display(); //access to base class function

   D *dptr;      //derived type pointer
   dptr=&d1;
   dptr->y=20;
   cout<<"dptr is a derived type pointer"<<endl;
   dptr->display();


   cout<<"using ((D*)bptr)"<<endl;
   ((D*)bptr)->y=30;
    ((D*)bptr)->display();
    return 0;
}
```

The statements,

```
dptr->display();
((D*)bptr)->display();          //cast bptr to D type
```

Displays the contents of **derived** object.

This shows that, although a base pointer cannot directly access the members defined by a derived class. But it can be made to point any number of derived objects, so that we can access all the members of derived class.

Virtual Functions

- ✓ Virtual function is declared by using a keyword 'virtual' preceding the normal declaration of a function.
- ✓ It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- ✓ 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.
- ✓ There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects.
- ✓ Thus by making the base pointer to point to different objects we can execute the different version of virtual function.

Necessity of virtual function

When base class pointer contains the address of the derived class object, always executes the base class function. Here, the compiler simply ignores the contents of the (base) pointer and chooses the member function that matches the type of the pointer.
This issue can only be resolved by using the 'virtual' function. When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer. In this way runtime polymorphism can also achieved.

```
#include<iostream>
using namespace std;
class B
{
public:
virtual void show()
{
cout<<"Show base"<<endl;
}
};
class D:public B
{
public:
void show()
{
cout<<"Show derived"<<endl;
}
};
```

```
int main()
{
B b1;
D d1;
B *bptr;
cout<<"bptr points to base"<<endl;
bptr=&b1;
bptr->show();          //calls base class function
cout<<"bptr points to derived"<<endl;
bptr=&d1;
bptr->show();          //calls derived class function
return 0;
}
```

Rules of Virtual Function

- The virtual functions must be members of some 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, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor.
- While a base pointer can point to any of the derived object, the reverse is not true. That is to say. We cannot use a pointer to derived class to access an object of base type.

1. **What is virtual function? When and how to we make function virtual? Explain with suitable example.[PU:2010 spring]**

A virtual function is a member function declared in base class with keyword virtual and uses a single pointer to base class pointer to refer to object of different classes.

When we use the same function name in both 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 is use at runtime 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 object of different versions of the virtual functions**.**

When virtual functions are used, a program that appears to be calling a function of one class may in reality be calling a function of a different class. Furthermore, when we use virtual functions, different functions can be executed by the same function call. The information regarding which function to invoke is determined at run time.

Program:

```
#include<iostream>
using namespace std;
class B
{
public:
virtual void show()
{
cout<<"Show base"<<endl;
}
};
class D:public B
{
public:
void show()
{
cout<<"Show derived"<<endl;
}
};

int main()
{
B b1;
D d1;
B *bptr;
cout<<"bptr points to base"<<endl;
bptr=&b1;
bptr->show();           //calls base class function
cout<<"bptr points to derived"<<endl;
bptr=&d1;
bptr->show();           //calls derived class function
return 0;
}
```

2. **What happens when a base and derived classes have same functions with same name and these are accessed using pointers with and without using virtual functions.**

```
#include<iostream>
using namespace std;
class B
{
public:
void display()
{
cout<<"Display base"<<endl;
}
virtual void show()
{
cout<<"Show base"<<endl;
}
};
class D:public B
{
public:
void display()
{
cout<<"Display derived"<<endl;
}
void show()
{
cout<<"Show derived"<<endl;
}
};
int main()
{
B b1;
D d1;
B *bptr;
cout<<"bptr points to base"<<endl;
bptr=&b1;
bptr->display();        //calls base class function
bptr->show();           //calls base class function
cout<<"bptr points to derived"<<endl;
bptr=&d1;
bptr->display();        //calls base class function
bptr->show();           //calls derived class function
return 0;
}
```

Output:

```
bptr points to the base
Display base
Show base
bptr points to derived
Display base
Show derived
```

Note:

When bptr is made to point the object d (ie. bptr=&d1),
the statement
bptr->display();
Calls only the function associated with the B. (ie.B::display())

This is because compiler actually ignores the content of the pointer bptr and chooses a member function that matches the type of the pointer.
whereas the statement
bptr->show();
calls the derived version of show().This is because function show() has been made virtual in Base class.

This is because first the base pointer has address of the base class object, then its content is changed to contain the address of the derived object.

3. How can you achieve runtime polymorphism in C++? Discuss with suitable example.

We should use virtual functions and pointers to objects to achieve run time polymorphism. For this, we use functions having same name, same number of parameters, and similar type of parameters in both base and derived classes. The function in the base class is declared as virtual using the keyword virtual.

A virtual function uses a single pointer to base class pointer to refer to all the derived objects. When a function in the base class is made virtual, C++ determines which function to use at run time based on the type of object pointed by the base class pointer, rather than the type of the pointer.

```
#include<iostream>
using namespace std;
class base
{
public:
virtual void show()
{
cout<<"Base Class Show function"<<endl;
}
};
class derived:public base
{
public:
void show()
{
cout<<"Derived Class Show function "<<endl;
}
};
int main()
{
base b1,*bptr;
derived d1;
bptr=&b1;
bptr->show();
bptr=&d1;
bptr->show();
return 0;
}
```

Output:

```
Base Class Show function
Derived Class Show function
```

A bookshop sells both books and video tapes. Create an abstract class known as media that stores the title and price of a publication. Create two child classes one for storing the number of pages in a book and another for storing the playing time of a tape. A function display is used in all the classes to display the class contents. Create necessary constructors in the child classes to store the information. In the main display the information regarding the book and tape using the base pointer (an object pointer of the class media)

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

```
using namespace std;
class media
{
protected:
char title[20];
float price;
public:
media(char t[], float p)
{
strcpy(title,t);
price = p;
}
virtual void display() = 0;
};
class book : public media
{
int pages;
public:
book(char t[], float p, int pag):media(t,p)
{
pages = pag;
}
void display()
{
cout<<"Title:"<<title<<endl;
cout<<"Price:"<<price<<endl;
cout<<"Pages:"<<pages<<endl;
}
};
class tape:public media
{
int time;
public:
tape(char t[], float p, int tm):media(t,p)
{
time = tm;
}
void display( )
{
cout <<"Title:"<<title<<endl;
cout <<"Price:"<<price<<endl;
cout <<"play time(mins):"<<time<<endl;
}
};
```

```
int main()
{
media *m[2];
book b("OOP",550.25,350);
tape t("computing concepts",255.6,55);
m[0] = &b;
m[1] = &t;
cout<<"Book  details:"<<endl;
m[0]->display();
cout<<"Tape details:"<<endl;
m[1]->display();
return 0;
}
```

Pure virtual function (Deferred methods/Abstract methods)

- A virtual function will become pure virtual function when we append "=0" at the end of declaration of virtual function.
  Example: virtual void display() =0;
- Pure virtual functions are also known as “do-nothing “functions.
- A pure virtual function is a function declared in a base class that has no definition (implementation/body).
- It serves only as a placeholder.
- The child classes are allowed to inherit them.
- In this situation, the compiler requires each derived class to either define the function or re-declare it as a pure virtual function. Otherwise the compilation error will occur.

Program:

```
#include<iostream>
using namespace std;
class Book
{
public:
virtual void display()=0;
};
```

```
class Math:public Book
{
public:
void display()
{
 cout<<"We are studying Math "<<endl;
}
};
class OOP:public Book
{
        public:
        void display()
        {
                cout<<"We are studying OOP"<<endl;
        }
};
int main()
{
 Book *bptr;
 Math m;
 OOP o;
 bptr=&m;
 bptr->display();
 bptr=&o;
 bptr->display();
 return 0;
}
```

Abstract class

- A class having at least one pure virtual function is called an abstract class.
- The object of abstract classes cannot be created.
- Pointers to abstract class can be created for selecting the proper virtual function.
- An abstract class is designed to act only as base class .It is a design concept in program development and provides a base upon which program is built.
- Sometimes implementation of all function cannot be provided in a base class because we don't know the implementation. For example, let Shape be a base class. We cannot provide implementation of function draw() in Shape, but we know every derived class must have implementation of draw().In this case concept of abstract class is used.

Example:

```
#include<iostream>
using namespace std;
class shape
{
public:
virtual void draw()=0;
};

class square:public shape
{
public:
void draw()
{
        cout<<"Implementing method to draw square"<<endl;
}
};
class circle:public shape
{
public:
void draw()
{
        cout<<"Implementing method to draw circle"<<endl;
}
};

int main()
{
        shape *bptr;
        square s;
        circle c;
        bptr=&s;
        bptr->draw();
        bptr=&c;
        bptr->draw();
        return 0;
}
```

Virtual Destructors

A constructors cannot be virtual due to the following reasons.

- To create an object of the constructor of the object class must be of the same type as class. But it is not possible with virtually implemented constructor.
- At the time of calling constructor, the virtual table would not have been created to resolve any virtual function calls.

Whereas destructors can be virtual.

Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Let's first see what happens when we do not have a virtual Base class destructor.

```
class Base
{
   public:
   ~Base()
   {
      cout << "Base  class destructor"<<endl;
   }
};

class Derived:public Base
{
   public:
   ~Derived()
   {
      cout<< "Derived class  destructor"<<endl;
   }
};


int main()
{
   Base* ptr = new Derived;    //Base class pointer points to derived class object
   delete ptr;
}
```

Output:

```
Base  class Destructor
```

In the above example, delete ptr will only call the Base class destructor, which is undesirable because, then the object of Derived class remains un-destructed, because its destructor is never called. Which leads to memory leak situation.
To make sure that the derived class destructor is mandatory called, we make base class destructor as virtual

```
class Base
{
   public:
   virtual ~Base()
   {
      cout << "Base  class destructor"<<endl;
   }
};
```

Output:.

```
Derived class Destructor
Base class Destructor
```

When we have Virtual destructor inside the base class, then first Derived class's destructor is called and then Base class's destructor is called, which is the desired behavior.

Type conversion:

- Type conversion is converting one type of data to another type.
- Compiler automatically converts basic to another basic data type (for eg int to float, float to int etc.) by applying type conversion rule provided by the compiler.
- The type of data to the right of the assignment operator (=) is automatically converted to the type of variable on the left.

For example:

```
int m;
float x=3.14159
m=x;
```

Value stored in m is 3 because, here the fractional part is truncated.
Compiler does not support automatic type conversion for user defined data type. Therefore, we must design conversion routines for the type conversion of user defined data type.
There are three possible type conversions are:

- Conversion from basic type to class type
- Conversion from class type to basic type
- Conversion from one class type to another class type

Conversion from basic to class type

The constructor is used for the type conversion takes the single argument which type is to be converted.

Example1:

```
#include<iostream>
using namespace std;
class time
{
int hours;
int minutes;
public:
time()
{
}
time(int t)
{
hours=t/60;
minutes=t%60;
}
void display()
{
cout<<"Hours="<<hours<<endl;
cout<<"Minutes="<<minutes<<endl;
}
};
int main()
{
int duration=65;
time t1;
t1=duration;
t1.display();
return 0;
}
```

After conversion, the hours members of t1 contains the value of 1 and minutes member contains the value of 5, denoting 1 hours and 5 minutes.

Example 2:

Write a Program to read the height of in meter and convert it into feet and inches using suitable type conversion methods.

```
#include <iostream>
using namespace std;
class Height
{
private:
int feet;
float inches ;
public:
Height()
{
}
Height(float m)
{
float f=3.28083*m ;
feet=int(f) ;
inches=12*(f-feet) ;
}
void display()
{
cout<<feet<<"feet and "<<inches<<"inches"<<endl;
}
};

int main()
{
float meter;
Height h1;
cout<<"Enter the height of person in meter"<<endl;
cin>>meter;
h1=meter;
cout<<"Height of person in feet and inches"<<endl;
h1.display();
return 0;
}
```

Make a class called memory with member data to represent bytes, kilobytes, and megabytes. in your program, you should be able to use statements like m1=1087665;
where m1 is an object of class memory and 1087665 is an integer representing some arbitrary number of bytes. Your program should display memory in a standard format like: 1 megabytes 38 kilobytes 177 bytes.

```
#include<iostream>
using namespace std;
class memory
{
int mb;
int kb;
int byte;
public:
memory()
{ }
memory(long int m)
{
int rem;
mb=m/(1024*1024);
rem=m%(1024*1024);
kb=rem/1024;
byte=rem%1024;
}
void display()
{
cout<<mb<<"megabytes"<<endl;
cout<<kb<<"kilobytes"<<endl;
cout<<byte<<"bytes"<<endl;
}
};
int main()
{
memory m1;
long int m=1087665;
m1=m;
m1.display();
return 0;
}
```

Class to Basic type

C++ allows us to define an overloaded casting operator that could be used to convert a class data type to basic type. The general form of an overloaded casting operator function, usually referred to as an conversion function is:

```
operator typename()
{
. . . . . .
//function statements
. . . . . . . .
}
```

The function converts a class type to ***typename***. For example,the **operator int()** converts an class object to type **int**, **operator float()** converts the class type object to **float** and so on.

The casting operator function should satisfy the following conditions.

- It must be a class member.
- It must not specify return type.
- It must not have any arguments.

Example 1:

```
#include<iostream>
using namespace std;
class Item
{
private:
float price;
int quantity;
public:
Item(float p,int q)
{
        price=p;
        quantity=q;
}

void display()
{
cout<<"Price of items="<<price<<endl;
cout<<"Quantity of items="<<quantity<<endl;
}
}
operator float()
{
 return (price*quantity);
}
};
```

```
int main()
{
Item i1(255.5,10);
float total;
total=i1;
i1.display();
cout<<"Total amount="<<total<<endl;
return 0;
}
```

Example 2:

Write a program to convert feet and inches into meter by using suitable type conversion method.

```
#include <iostream>
using namespace std;
class Height
{
private:
int feet;
float inches;
public:
Height(int f, float i)
{
 feet=f;
 inches=i;
}
void display()
{
 cout<<feet<< "feet and"<<inches<<"inches"<<endl;
}
operator float()
{
float f=inches/12 ;
f=f+feet ;
return(f/3.28083);
}
};
```

```
int main( )
{
Height h1(5,3.6) ;
float m=h1;
cout<<"Height in feet and inches:"<<endl;
h1.display();
cout<<"Height in meter="<<m<<endl;
return 0;
}
```

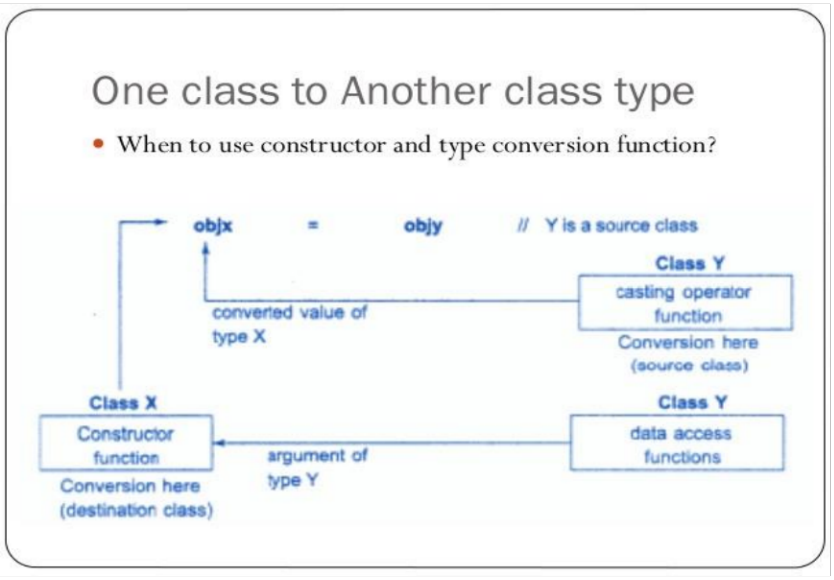
One class to another class type

Define type conversion. Explain with conversion from one class type to another class type. **[PU: 2016 fall]**

We can convert one class type data to another class type.
Example:
objX=objY; //objects of different classes
ObjX is an object of class **X** and **objY** is an object of class **Y**. The class **Y** type data is converted to the class **X** type data and the converted value is assigned to the **objX**. Since the conversion takes place from class **Y** to class **X**, **Y** is known as source class and **X** is known as destination class.
Such conversions between objects of different classes can be carried out by either a constructor or a conversion function.

Note:

- Conversion form a class to any other type (or any other class) should make use of casting operator function in the source class.
- On the other hand to perform the conversion from any other type /class type constructor should be used in destination class.



Conversion Routine in source class *(conversion function in source class)*

Conversion from polar to rectangle (using conversion routine in polar)

```
#include<iostream>
#include<math.h>
using namespace std;
class Rectangle
{
private:
float xco;
float yco;
public:
Rectangle()
{
        xco=0.0;
        yco=0.0;
}

Rectangle(float x,float y)
{
        xco=x;
        yco=y;
}
void display()
{
cout<<"("<<xco<<","<<yco<<")"<<endl;
}
};

class Polar
{
private:
        float radius;
        float angle;
public:
        Polar()
        {
         radius=0.0;
         angle=0.0;
        }
```

```
        Polar(float r,float a)
        {
        radius=r;
        angle=a;
        }
        void display()
        {
        cout<<"("<<radius<<","<<angle<<")"<<endl;
        }
        operator Rectangle()
        {
         float x=radius*cos(angle);
         float y=radius*sin(angle);
         return Rectangle(x,y);
        }
};
int main()
{
Polar p(10.0,0.758539);
Rectangle r;
r=p;
cout<<"Polar coordinates=";
p.display();
cout<<"Rectangular coordinates=";
r.display();
return 0;
}
```

Conversion from Rectangle to Polar (using conversion routine in rectangle)

```
#include<iostream>
#include<math.h>
using namespace std;
class Polar
{
private:
        float radius;
        float angle;
public:
        Polar()
        {
                radius=0.0;
                angle=0.0;
        }
```

```
        Polar(float r,float a)
        {
        radius=r;
        angle=a;
        }
        void display()
        {
        cout<<"("<<radius<<","<<angle<<")"<<endl;
        }
};

class Rectangle
{
private:
float xco;
float yco;
public:
Rectangle()
{
        xco=0.0;
        yco=0.0;
}
Rectangle(float x,float y)
{
        xco=x;
        yco=y;
}
void display()
{
cout<<"("<<xco<<","<<yco<<")"<<endl;
}
operator Polar()
{
float a=atan(yco/xco);
float r=sqrt(xco*xco+yco*yco);
return Polar(r,a);
}
};
```

```
int main()
{
Rectangle r(7.07107,7.07107);
Polar p;
p=r;
cout<<"Rectangular coordinates=";
r.display();
cout<<"Polar coordinates=";
p.display();
return 0;
}
```

Conversion Routine in destination class

Conversion from polar to rectangle (using conversion routine in Rectangle)

```
#include <iostream>
#include <math.h>
using namespace std;
class Polar
{
private:
float radius;
float angle;
public:
Polar()
{
radius=0.0;
angle=0.0;
}
Polar(float r, float a)
{
radius=r;
angle=a;
}
void display()
{
cout<<"("<<radius<<","<<angle<<")"<<endl;
}
float getr()
{
return radius;
}
```

```
float geta()
{
return angle;
}
};

class Rectangle
{
private:
float xco;
float yco;
public:
Rectangle()
{
xco=0.0;
yco=0.0;
}
void display()
{
cout<<"("<<xco<<","<<yco<<")";
}

Rectangle(Polar p)
{
float r=p.getr();
float a=p.geta();
xco=r*cos(a);
yco=r*sin(a);
}
};

int main()
{
Polar p(10.0,0.785398);
Rectangle r;
r=p;
cout<<"Polar coordinates=";
p.display();
cout<<"Rectangular coordinates=:";
r.display();
return 0;
}
```

Conversion from rectangle to polar (using conversion routine in polar)

```
#include<iostream>
#include<math.h>
using namespace std;
class Rectangle
{
private:
float xco;
float yco;
public:
Rectangle()
{
xco=0.0;
yco=0.0;
}
Rectangle(float x,float y)
{
        xco=x;
        yco=y;
}
void display()
{
cout<<"("<<xco<<","<<yco<<")"<<endl;
}
float getx()
{
 return xco;
}
float gety()
{
return yco;
}
};
```

```
class Polar
{
private:
        float radius;
        float angle;
public:
        Polar()
        {
        radius=0.0;
        angle=0.0;  }
        void display()
        {
        cout<<"("<<radius<<","<<angle<<")"<<endl;
        }

        Polar(Rectangle r)
        {
                float x=r.getx();
                float y=r.gety();
                angle=atan(y/x);
                radius=sqrt(x*x+y*y);
        }
};
int main()
{
Rectangle r(7.07107,7.07107);
Polar p;
p=r;
cout<<"Rectangular coordinates=";
r.display();
cout<<"Polar coordinates=";
p.display();
return 0;
}
```

Previous Old Question from this chapter

Theory

Compile time and runtime polymorphism

1) What is polymorphism? Differentiate between compile time and runtime polymorphism with program in each.
2) How can polymorphism be achieved during compile time and during runtime? Explain with examples in C++.**[PU:2013 spring][PU:2019 spring]**
3) What are the advantages of using runtime polymorphism over compile time polymorphism. [PU:2014 spring]
4) Explain importance of polymorphism with a suitable example.[PU:2009 fall]
5) What is the difference between compile time and runtime polymorphism? Which is best and why?
6) What is compile time and runtime polymorphism? How can you achieve runtime polymorphism in C++?

Operator overloading

7) How can you use operator overloading in C++? Give syntax
8) What is polymorphism? How operator overloading is used to support polymorphism. Explain it by overloading '+' operator to concatenate two strings**.[PU:2017 fall]**
9) What do you mean by operator overloading? How do your overload the + operator in main program (c3=c1+c2).so that c3 can store a complex number obtained by adding c2 and c2.**[PU: 2006 spring]**

Type conversion

10) What is type casting?**[PU:2018 fall][PU:2019 fall]**
11) Define type conversion .Explain with example conversion from one class type to another class type.**[PU:2016 fall]**

Pointer to derived class

12) Can you derive a pointer from base class? Explain with suitable example.**[PU:2014 spring]**

Virtual function/pure virtual function/abstract class

13) What is virtual function? When do we make a function virtual and when we make function pure virtual? Explain with suitable example.**[PU:2010 spring]**
14) When do you use virtual function? How does it provides runtime polymorphism. Explain with suitable example**.[PU:2016 fall]**

15) Discuss the role of virtual function in c++ to cause dynamic polymorphism. Show with example the how it is different from the compile time polymorphism.[PU:2006 spring]
16) What is virtual function? When do we make a function virtual? Explain with suitable example**.[PU: 2014 spring]**

Pure virtual function

17) Explain deferred methods.
18) What is pure virtual function?
19) How can you define pure virtual function in C++? The pure virtual function do nothing but it is defined in base class why?[**PU:2015 spring]**

Abstract class

20) What is an abstract class? How does it differ from virtual base class? Explain .[**PU:2006 spring]**

This pointer

21) Why is 'this' Pointer is widely used than object pointer? Write a program to implement pure polymorphism.**[PU:2019 fall]**
22) Define role of this pointer and pure abstract class in object oriented programming to create multiple object with suitable program.[**PU:2015 spring]**

Function overriding

23) When base class and derived class have same function name ,what happens when the derived class object class the functions? Differentiate overloading and overriding.**[PU:2017 fall]**
24) How does overloading differ from overriding. Explain.
25) Describe overriding. How do you differentiate function overloading from function overriding. Explain with suitable example**.[PU:2016 spring]**

Write a short notes on:

- Virtual function vs friend function[PU:2016 spring]
- Virtual function vs pure virtual function[PU:2016 spring]
- Deferred methods[PU:2006 spring]
- Virtual functions[2015 fall]
- Operator overloading [PU:2016 fall]
- Overriding [PU:2018 fall]
- Virtual Destructor[PU:2013 fall]
- This pointer[PU:2020 fall]

Program

Function overloading

1. Write a program finding area of square, rectangle, triangle. Use function overloading technique.

Operator overloading

2. Write a simple program to overload unary ++ operator.[PU: 2016 spring]
3. Write a program to generate Fibonacci series using operator overloading of ++ operator. Which type of overloading is it**.[PU:2009 fall]**
4. Write a program to add two complex number using operator overloading. **[PU:2010 spring]**

   OR

   Write a program to add two complex number using binary operator overloading**.[PU:2013 fall]**

   OR

   WAP to add two complex numbers. Your program should have three objects. Each object contains two attributes(i.e. real and imaginary part) Now add each attribute and save them into third object separately. Use the concept of '+' operator overloading.

5. WAP to overload multiplication operator showing (*) showing the multiplication of two objects.**[PU:2020 fall]**
6. Write a program with class fibo to realize the following code snippet. **[PU: 2014 fall]**

```
fibo f=1;
for (i=1;i<=10;i++)
{
++f;
f.display();
}
```

   [Hint: overload ++ operator and conversion technique.]
7. Design a soccer player class that includes three ineger fields:a player's jersey number,number of goals,number of assists and necessary constructors to initialize the data members.Overload the > operator (Greater than) .One player is considered greater than another if the sum of goals plus assists is greater than that of others. Create an array of 11 soccer players, then use the overloaded > operator to find the greater total of goal plus assists**.[PU:2015 fall]**

8. Write a program to implement vector addition using operator overloading
   i. Using Friend Function
   ii. Without using Friend Function(using member function)

Type conversion

9. Make a class called memory with member data to represent bytes, kilobytes and megabytes .Read the value of memory in bytes from the user as basic types and display the result in user defined memory type. Like for m (basic type) = 108766, your program should display as: 1 megabyte 38 kilobytes 177 bytes. [Hint: Use basic to user defined (basic-to-class) conversion method.
10. Make a class called memory with member data to represent bytes, kilobytes and megabytes .Create an object of memory and initialize data members and then convert them into number of bytes using suitable conversion method

Hint: use class to basic type conversion method

```
Operator long int()
{
        return (mb*1024*1024+kb*1024+byte)
}
```

11. Write a program to create a class age with attributes YY, MM, DD and convert the object of class age to basic data type int days.
12. Write a program that converts object that represents 24 hrs times to 12 hrs times and vice versa.
13. Write a complete program to convert the polar coordinates into rectangular coordinates.(hint: polar-coordinates(radius, angle) and rectangular co-ordinates (x,y) where x=r*cos(angle) and y=r*sin(angle))**[PU:2017spring]**

    OR

    WAP to convert object of polar class into the object of rectangle class by using type conversion technique.**[PU:2019 spring]**
14. Write a program to read a height of person in feet and inches and convert it into meter using user defined to class type conversion method.1 meter=3.28084 feet,1 feet=12 inch**[PU:2018 fall]**

15. Define two classes named 'Polar' and 'Rectangle' to represent points in polar and rectangle systems. Use conversion routines to convert from one system to another system.**[PU:2005 fall][PU:2014 fall]**

    OR

    Create a class Rectangle with xco and yco as data members and use appropriate function to initialize then and display them. Now create another class polar with radius and angle as data members and member functions to initialize them and display the data. Now use conversion function in source class to convert rectangular object to polar object and vice-versa.

Virtual function

16. A bookshop sells both books and video tapes. Create an abstract class known as media that stores the title and price of a publication. Create two child classes one for storing the number of pages in a book and another for storing the playing time of a tape. A function display is used in all the classes to display the class contents .Create necessary constructors in the child classes to store the information. In the main display the information regarding the book and tape using the base pointer (an object pointer of the class media).

17. Create a base class student. Use the class to store name, dob, rollno and includes the member function getdata(),discount().Derive two classes PG and UG from student. make dispresult() as virtual function in the derived class to suit the requirement.**[PU:2013 spring]**

Abstract class

18. Create a abstract class shape with two members base and height, a member function for initialization and a pure virtual function to compute area().Derive two specific classes, Triangle and Rectangle which override the function area ().use these classes in main function and display the area of triangle and rectangle**.[PU:2009 spring]**