Polymorphism

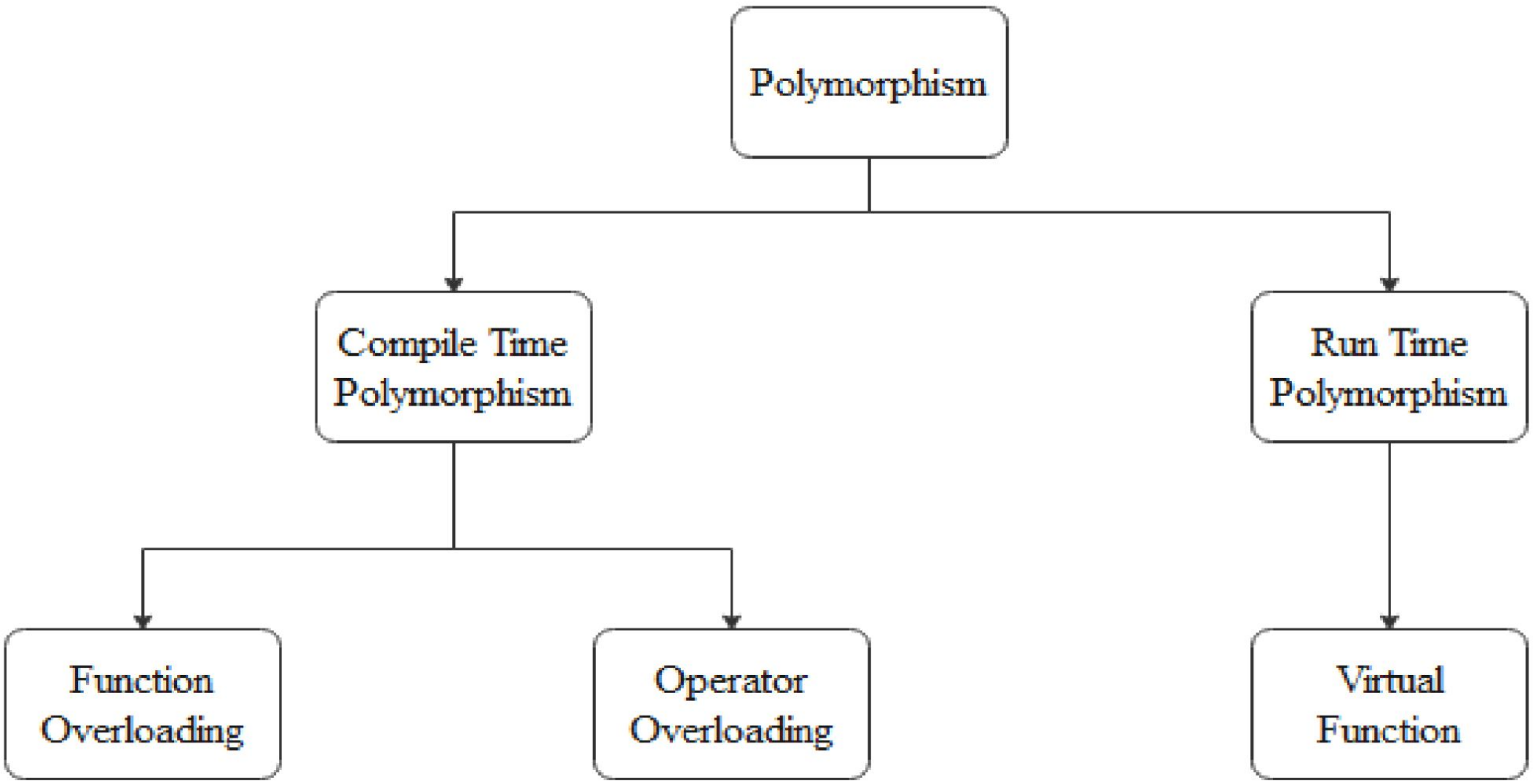
Chapter 5

Introduction

Polymorphism:

- “One name multiple forms”
- Polymorphic in Greek: poly means many and morphos means forms

In programming languages, a polymorphic object is any entity such as a variable or function argument that is permitted to hold values of different types during the course of execution


Polymorphism
Compile Time Polymorphism
Run Time Polymorphism
Function Overloading
Operator Overloading
Virtual Function

Compile time polymorphism

- Compile time polymorphism refers to binding of function on the basis of their signature (name, type, sequence of parameters)
- Also called early binding as the calls are already bound to the proper type of functions during the compilation of the program
- Overloaded functions and operators support Compile Time Polymorphism

Example :

- void calculateArea (int);
- void calculateArea(int, int);
- When the function calculateArea () is invoked, the passed argument determine which one is to be executed
- This resolution takes place at compile time.

Function Overloading:

1. Write a program to calculate volume of rectangular box, sphere and cylinder. Use calculateVolume() function to calculate the result. Use concept of function overloading.

Runtime polymorphism

- A function exhibits runtime polymorphism/ dynamic polymorphism if it exists in various forms and the resolution to different function calls are made during execution time
- Runtime polymorphism is achieved using virtual function

Deferred Methods

- A deferred method (sometimes called abstract method and in C++ called pure virtual method) is a virtual function without a body
- The virtual function of base class are not used, it is just a place holder and all useful activity is defined as part of the code provided by the child classes

Pure polymorphism

- In inheritance where classes have hierarchical relationship where base class have derived classes, all of objects of derived class can be pointed at by a base class pointer.
- By accessing the virtual function through base pointer, C++ selects the appropriate function definition at runtime. This is a form of polymorphism called pure polymorphism.

Explain with an example program implementing the concept of function overriding using virtual function.

Compile time polymorphism using operator overloading

- Operator overloading is one of the many exciting features of C++ language.
- C++ makes the user-defined data types behave in much the same way as the built in types.
- For example, C++ permits us to add two variables of user-defined types with the same syntax that is applied to the basic types.
- This means that C++ has the ability to provide the operator with a special meaning for a data type.
- The mechanism of giving such special meanings to an operator is known as operator overloading.

Operator overloading

a= b+c; works only with basic data types like int and float.

• using operator overloading, we can make this statement legal for user-defined data type also (like object).

• The use of + operator for two objects, means giving special meaning to + operator.

• The statement like C3.add_complex(C1,C2) can be converted into statement like C3=C1+C2 using operator overloading.

Rules on operator overloading

1. Only existing operators can be overloaded, new operators cannot be created

2. The precedence and associativity of operators can’t be changed while overloading

3. The overloaded operators follow the syntax rules of the original predefined operators.

4. The overloaded operators must have at least one user-defined type operand

Defining operator overloading

- An operator overloading is defined with the help of a special function, called operator function, which describes the task to be done by the operator.
- syntax for operator function is: return_type operator op(argument list); where,
- operator is keyword
- op is any existing valid operator to be overloaded.
    - operator op is function name.

Defining operator overloading

- Syntax for defining operator function:

```
return_type className :: operator existing_operator( argument/s) {
        //task of operator
        }
```

- Operator function may be either member function or friend function

this pointer

- this pointer holds the address of current calling object when calling a member function

objectA. functionX();

Here this pointer holds the address of objectA

- Whenever a non static member function is called, this pointer is passed as hidden argument implicitly

Program to use this pointer

```
#include<iostream>
class Complex
{
int real,img;
public:
void getAddress()
{
cout<<"The address of object using this
pointer"<<this;
}
};
```

```
int main()
{
Complex c1,c2;
cout<<"For c1"<<endl;
c1.getAddress();
cout<<endl<<"Address of object
directly is "<<&c1<<endl;
cout<<endl<<endl<<"For
c2"<<endl;
c2.getAddress();
cout<<endl<<"Address of object
directly is "<<&c2<<endl;
}
```

```cpp
#include<iostream>
using namespace std;
class Complex
{
int real,img;
public:
Complex( int real, int img)
{
this->real=real;
this->img=img;
}
void display()
{
cout<<"Real part : "<<this->real<<endl
<<"Imaginary part : "<<this->img<<endl;
}
};
```

```cpp
int main() {
	Complex obj(10,20);
	obj.display();

}
```

Overloading operators using unary operator

- The operator function defined as member function of a class has no arguments for unary operators
- The member function takes one less argument because member function is called using object of the class such that the calling object acts as implicit argument
- The operator that operates only one operand to perform its operation is called unary operators.
    - Unary plus(+)
    - Unary minus (-)
    - Increment operator(++)
    - Decrement operator(--)

Operator function as member dunction

- Syntax for declaration:

```
return_type operator unary_operator()
{
    task of unary_operator
}
```

- The operator function is invoked using,

```
unary_operator   object_of_class;
object_of_class   unary_operator;
```

which is equivalent to:

```
object_of_class . operator unary_operator( );
```

Example Program to overload unary minus operator:

```
#include<iostream>
using namespace std;
class Complex
{
int real,img;
public:
Complex( int r, int i)
{
real=r;
img=i;
}
void operator - ()
{
real = - real;
img = - img;
}
void display()
{
cout<<"Real part : "<<real<<endl
<<"Imaginary part : "<<img<<endl;
}
};
int main()
{
Complex obj(10,20);
obj.display();
-obj ; //obj.operator - ();
cout<<endl<<"Complex number after
negation is :"<<endl;
obj.display();
}
```

Overloading Unary operator using friend function:

- The operator function defined as friend function will have one argument
- Friend function is called without object of the class thus all arguments have to be passes explicitly

Example Program to overload unary minus operator and perform negation operation on Complex type

```
#include<iostream>
using namespace std;
class Complex
{
int real,img;
public:
Complex( int r, int i)
{
real=r;
img=i;
}
void display()
{
cout<<"Real part : "<<real<<endl
<<"Imaginary part : "<<img<<endl;
}
friend void operator -(Complex C);
};
```

```
void operator - (Complex C)
{
C.real = - C.real;
C.img = - C.img;
}
int main()
{
Complex obj(10,20);
obj.display();
-obj ; //operator - (obj);
cout<<endl<<"Complex number after negation is
:"<<endl;
obj.display();
}
```

Practice

1. Define a class Distance with data members kilometer and meter. Increase the distance by one using increment operator. Write a program to overload increment operator(++) in prefix notation.

2. Write a program to overload unary - operator and the operator function returns object such that

   complexOne = -complexTwo;

   where complexOne and complexTwo are complex numbers

   a. Using operator function as member function

   b. Using operator function as friend function

Overloading as member function

```
#include <iostream>
using namespace std;
class Complex {
 private:
  double real;
  double imag;
 public:
  Complex(double r, double i) {
      real = r;
      imag=i;
  }

 // Overload the unary minus operator
  Complex operator-() {
    return Complex(-real, -imag);
  }

  void print() {
    cout << real << " + " << imag << "i" <<
endl;
  }
};

int main() {
  Complex c1(1, -2);
  Complex c2 = -c1;  // Negate c1 using the
overloaded unary minus operator
  c2.print();  // Output: -1 + -2i

  return 0;
}
```

Overloading as friend function

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

class Complex {
 private:
  double real;
  double imag;

 public:
  Complex(double r, double i) {
      real = r;
      imag=i;
  }

  // Overload the unary minus operator
  friend Complex operator-(Complex);
void print() {
    cout << real << " + " << imag << "i" <<
endl;
  }
};
Complex operator-(Complex c) {
    return Complex(-c.real, -c.imag);
  }
int main() {
  Complex c1(1, 2);
  Complex c2 = -c1;  // Negate c1 using the
overloaded unary minus operator
  c2.print();  // Output: -1 + -2i

  return 0;
}
```

Overloading binary operator

Binary operator:

Operator that requires two operands for its operations

- Example:
- binary plus (+) • Binary minus (-) • Multiplication (*) • Division(/)
- Equality operator(==) • Greater than(>) etc

Overloading binary operator: using member function:

- The operator function defined as member function of a class takes one argument for binary operators
- Syntax for defining operator function as member function

```
return_type operator binaryOperator(object_of_class) {
        body of function
        }
```

Addition of two complex number

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

class Complex
{
int real,img;
public:
Complex()
{
}
Complex (int real, int img)
{
this->real=real;
this->img=img;
}
Complex operator + (Complex c)
{
Complex cobj;
cobj.real=real+c.real;
cobj.img=img+c.img;
return cobj;
}
void display()
{
cout<<endl<<"Real part is
"<<real<<endl<<"Imaginary part is "<<img;
}
};
```

```
int main()
{
Complex C3,C1(10,20),C2(30,40);
C3=C1+C2; //C3=C1.operator + (C2); /*left operand of
binary operator is used as calling object to call operator
function
//and right argument is passed as argument to the
function*/
cout<<"For C1"<<endl;
C1.display();
cout<<endl<<endl<<endl<<"For C2"<<endl;
C2.display();
cout<<endl<<endl<<endl<<"For C3"<<endl;
C3.display();

}
```

Overloading binary operator: using friend function:

The operator function defined as friend function of a class takes two argument for binary operators

Friend function is called without object of the class thus all arguments have to be passes explicitly

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

class Complex
{
int real,img;
public:
Complex()
{
}
Complex (int real, int img)
{
this->real=real;
this->img=img;
}
friend Complex operator + (Complex c1,
Complex c2);

void display()
{
cout<<endl<<"Real part is
"<<real<<endl<<"Imaginary part is
"<<img;
}
};
Complex operator +(Complex c1,
Complex c2){

Complex cobj;
cobj.real=c1.real+c2.real;
cobj.img=c1.img+c2.img;
return cobj;

}
```

```
int main()
{
Complex C3,C1(10,20),C2(30,40);
C3=C1+C2; //C3=C1.operator + (C2); /*left operand of
binary operator is used as calling object to call operator
function
//and right argument is passed as argument to the
function*/
cout<<"For C1"<<endl;
C1.display();
cout<<endl<<endl<<endl<<"For C2"<<endl;
C2.display();
cout<<endl<<endl<<endl<<"For C3"<<endl;
C3.display();

}
```

Practical :

1. Write a program to define a class Time with data members hours, minutes and seconds. Overload binary plus operator(+) to add two times.
    a. Using operator function as member function
    b. Using operator function as friend function
2. Write a program to overload binary plus operator(+) to concatenate two strings.
    a. Using operator function as member function
    b. Using operator function as friend function
3. Write a program to overload equality operator (==) to check if two strings are identical or not.
    a. Using operator function as member function
    b. Using operator function as friend function

4. Define a class Distance with data members kilometer and meter. Add 5 to the distance object by overloading binary plus operator.

5. Write a program to overload multiplication operator (*) to multiply each element of 3*3 matrix by 9.

6. Define a class Complex with data members real and img. Write a program to overload binary (+) operator to add two Complex numbers, binary minus (–) operator to subtract two Complex number and multiplication operator (*) to multiply two Complex numbers

Data Type Conversion:

1. Conversion from basic type to another basic type

2. Conversion from class type to basic type

3. Conversion from basic type to class type

4. Conversion from one class type to another class type

1. Conversion of Basic type to another basic type:

Program to illustrate use of implicit conversion from float to int

```
#include<iostream>
using namespace std;
int main()
{
	float a=100.5657;
	int b;
	b=a;
	cout<<b;

}
```

1. Conversion of Basic type to another basic type:

Program to illustrate use of explicit conversion from float to int

```
#include<iostream>
using namespace std;
int main()
{
int a=100;
float b=float(a)/3;
cout<<b;


}
```

2. Conversion from basic type to user-defined type:

- Constructor can be used to convert basic data type to user defined data type
- Distance d; // user defined type

int meter = 50; // basic type

d=meter; //error

Program to convert duration in seconds into object of a class Time which has second, minute and hour.

```
#include<iostream>
using namespace std;
class Time
{
int minute,second,hour;
public:
Time()
{
minute=0;
second=0;
hour=0;
}
```

```
Time(int sec)
{
hour=sec/3600;
minute=(sec%3600)/60;
second=(sec%3600)%60;
}
void display()
{
cout<<hour<<":"<<minute<<":"<<second<<en
dl;
}
};
```

```
int main()
{
Time T; //Time T=3700 or Time T(3700)
int duration=3700;
T=duration; // T=Time(duration);
T.display();

}
```

2. Conversion from basic type to user-defined type:

- Overloaded assignment operator (=) can also be used to convert basic type to class type

Program to convert duration in seconds into object of a class Time which has second, minute and hour. {basic type to class type conversion by overloading assignment ‘=‘ operator}

```
#include<iostream>
using namespace std;
class Time
{
int minute,second,hour;
public:
Time()
{
minute=0;
second=0;
hour=0;
}
void operator = (int sec)
{
hour=sec/3600;
minute=(sec%3600)/60;
second=(sec%3600)%60;
}
void display()
{
cout<<hour<<":"<<minute<<":"<<second<<endl;
}
};
```

```
int main()
{
Time T; //Time T=3700 or Time T(3700)
int duration=3700;
T=duration; // T=Time(duration);
T.display();

}
```

Question

Practical:

Q1. Write a program to convert meters in value into object of a class Distance which has feet and inch as members 1 meter = 3.28034 feet

```
#include<iostream>
using namespace std;
class Distance
{
int feet;
int inches;
public:
Distance()
{
feet=0;
inches=0;
}
void operator = (int m)
{
float inch,f= m*3.28034;
feet =(int) f;
inch=(f-feet)*12;
inches = (int)inch;
}
void display()
{
cout<<feet<<" feet "<<inches<<" inches"<<endl;
}
};
int main()
{
Distance D;
int meter=3;
D=meter;
D.display();

}
```

3. Conversion from user defined type to basic type

- We define casting operator to convert class type to basic type
- We use casting operator function also called as conversion function
- Syntax to define an overloaded casting operator function

```
operator type_name() {
        function body return(data);
}
```

- operator float () converts class type to float
- operator int() converts class type to int

Note:

1. The operator function must be a member of the class

2. The operator function must not specify a return type even though it return the value

3. The operator function must not have any arguments

4. Being a member function, it is invoked by an object and values of that object are used inside the function. Hence no argument is passed to conversion function

Example Program to convert Time type to seconds, which is int type

```
#include<iostream>
using namespace std;
class Time
{
int minute, second, hour;
public:
Time()
{
minute=0;
second=0;
hour=0;
}
Time(int h,int m,int s)
{
minute=m;
second=s;
hour=h;
}
operator int ()
{
int duration;
duration=hour*3600.+minute*60+second;
return duration;
}
```

```
void display()
{
cout<<hour<<":"<<minute<<":"<<second<<endl;
}
};
 int main()
{
Time T(1,1,40); //Time T=3700 or Time T(3700)
int duration;
duration=T; //duration=T.operator int();
T.display();
cout<<"The duration in seconds is "<<duration;

}
```

Practical :

Q1.Define a class Distance expressed in feet and inches. Write a program to convert distance into single integer value meter. 1 meter = 3.28034 feet

```
#include<iostream>
using namespace std;
class Distance
{
int feet, inches;
public:
Distance()
{
feet=0;
inches=0;
}
Distance(int f,int i)
{
feet=f;
inches=i;
}
operator int ()
{
int meter;
meter=(float)feet/0.30485+(float)(inches/3.658185);
return meter;
}
void display()
{
cout<<feet<<" feet "<<inches<<" inches "<<endl;
}
};
```

```
 int main()
{
Distance D(5,7); //Time T=3700 or Time T(3700)
int meter;
meter=D; //duration=T.operator int();
D.display();
cout<<"The distance in meter is  "<<meter;

}
```

4. Conversion from One class type to another class type

objectOfDestination = objectOfSource

- It can be performed using either
    - One argument constructor
    - Conversion function

4.1 : Conversion Routine in Destination class

- When the conversion routine is in destination class, we use one argument constructor.

Example Program : class to class conversion using example of Dollar to Rupees conversion.

```
#include<iostream>
using namespace std;
class Dollar
{
double dol;
public:
Dollar()
{
dol=0;
}
Dollar(int d)
{
dol=d;
}
```

```
void displayDollar()
{
cout<<endl<<"$"<<dol<<endl;
}
double getDollar()
{
return dol;
}
};
```

```
class Rupees {
double rs;
public:
Rupees() {
rs=0;
}
Rupees(Dollar dollar) {
rs=129.311*(dollar.getDollar());
}
void displayRupees() {
cout<<"Rs."<<rs<<endl
;
}
};
```

```
int main() {
Dollar d(154);
Rupees r;
r=d; //r=Rupees(d);
r.displayRupees();
d.displayDollar();

}
```

Practical:

Q1. Define a class to represent rectangular co-ordinates and another class to represent polar co-ordinates. Convert polar co-ordinates into rectangular coordinate value using conversion routine in destination class. note: Polar co-ordinate has radius and angle Rectangular co-ordinates has x-co-ordinate and y-co-ordinate x-co-ordinate=radius *cos(angle) y-co-ordinate=radius *sin(angle)

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

class Polar
{
float radius;
float angle;
public:
Polar()
{
radius=0;
angle=0;
}
Polar(float r, float a){
    radius = r;
    angle = a;
}
```

```
float getradius(){
    return radius;
}
float getangle(){
    return angle;
}


void display()
{
cout<<radius<<" cos "<<angle<<" + "<<radius<<" sin
"<< angle <<endl;
}
};
```

```
class Rectangle
{
int x;
int y;
public:
Rectangle()
{
x=0;
y=0;
}
Rectangle(Polar p){
    x=p.getradius()*cos(p.getangle());
    y=p.getradius()*sin(p.getangle());
}
void display()
{
cout<<x<<" i "<<" + "<<y<<" j "<<endl;
}
};
int main()
{
Polar p(5,30);
Rectangle r;
r=p;
r.display();
p.display();

}
```

2. Define a class to represent rectangular co-ordinates and another class to represent polar co-ordinates. Convert rectangular co-ordinates into polar coordinate value using conversion routine in destination class.

Note: Polar co-ordinate has (radius, angle)

Rectangular co-ordinates has (x-coordinate and y-co-ordinate)

radius = sqrt(x-coordinate^2+y-coordinate^2)

angle = tan^-1 (y-coordinate / x-coordinate)

4.2 : Conversion routine in Source class:

• If the source class handles the conversion activity, it is performed using a conversion function syntax for conversion function is:

```
operator destination_className() {
    //body of function
    }
```

```
#include<iostream>
using namespace std;
class Rupees
{
double rs;
public:
Rupees()
{
rs=0;
}
Rupees(double r)
{
rs=r;
}
void displayRupees()
{
cout<<"Rs"<<rs<<endl;
}
};
```

```
class Dollar
{
double dol;
public:
Dollar()
{
dol=0;
}
Dollar(double d)
{
dol=d;
}
void displayDollar()
{
cout<<endl<<"$"<<dol<<endl;
}
double getDollar()
{
return dol;
}
```

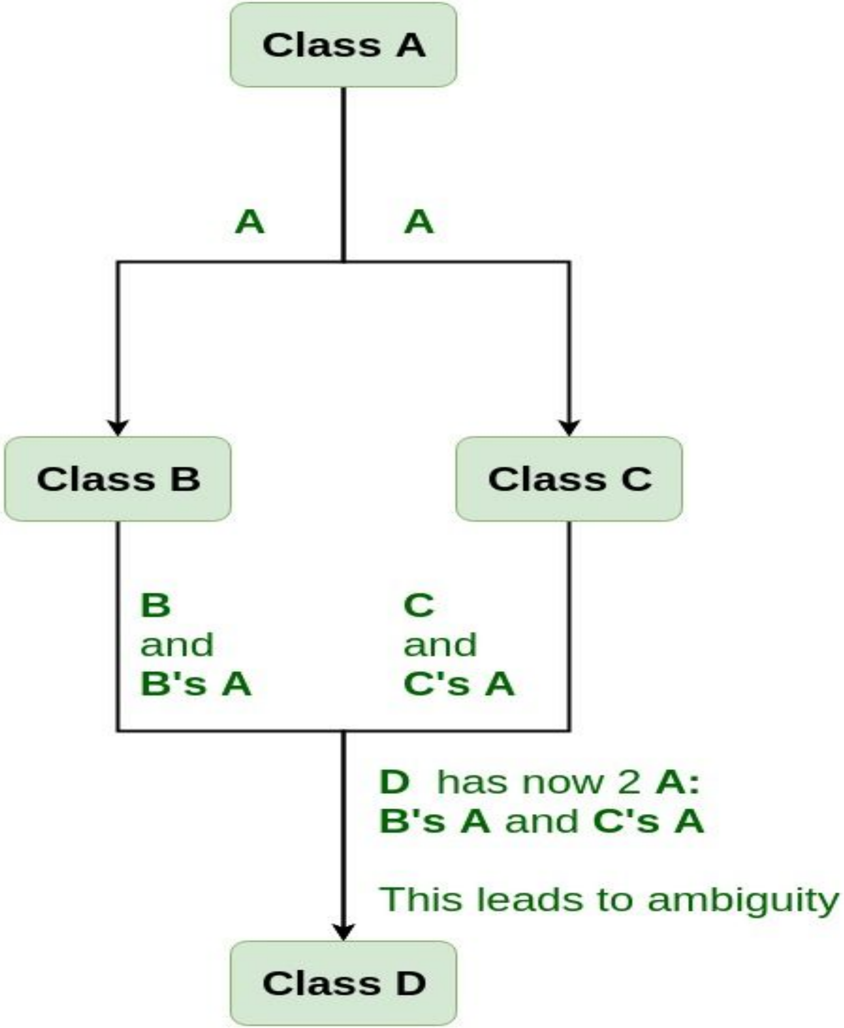
```
operator Rupees()
{
double rupees;
rupees=dol*120.11;
Rupees r(rupees);
return r;
}
};
int main()
{
Dollar d(154);
Rupees r;
r=d; //r=d.operator Rupees();
r.displayRupees();
d.displayDollar();

}
```

Virtual Base class in C++

Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Need for Virtual Base Classes: Consider the situation where we have one class **A** . This class **A** is inherited by two other classes **B** and **C**. Both these class are inherited into another in a new class **D** as shown in figure below.


Class A
A
A
Class B
Class C
B
and
B's A
C
and
C's A
D has now 2 A:
B's A and C's A
This leads to ambiguity
Class D

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

class A {
public:
    void show()
    {
        cout << "Hello from Base Class: \n";
    }
};

class B : public A {
};

class C : public A {
};

class D : public B, public C {
};
```

```cpp
int main()
{
    D object;
    object.show();
}
```

Removing the ambiguity using virtual base class

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

class A {
public:
    void show()
    {
        cout << "Hello from the baseclass A \n";
    }
};

class B : public virtual A {
};

class C : public virtual A {
};

class D : public B, public C {
};

int main()
{
    D object;
    object.show();
}
```

```
#include <iostream>

using namespace std;
class Person{
    protected:
    string name;
    int code;
    public:
    Person(string n,int c){
        name=n;
        code=c;
    }
    void display(){
        cout<<"Name: "<<name<<endl;
        cout<<"Code: "<<code<<endl;
    }
};
```

```
class Account: public Person{
    protected:
    int salary;
    public:
    Account(string name, int code, int s)
:Person(name,code){
        salary=s;
    }
    void display(){
        Person::display();
        cout<<"salary: "<<salary<<endl;
    }
};
```

```
class Admin: public Person{
    protected:
    int nos;
    public:
    Admin(string name, int code, int
n):Person(name,code){
        nos=n;
    }
    void display(){
        Person::display();
        cout<<"No. of years of
experience: "<<nos<<endl;
    }
};
```

```
class Record:public Account, public Admin{
    int record_no;
    public:
    Record(string name, int code, int s,int n, int
r):Account(name,code,s),Admin(name,code,n){
        record_no=r;
    }
    void display(){
        Account::display();
        cout<<"Record No.: "<<record_no<<endl;
    }
    void display1(){
        Admin::display();
        cout<<"Record No.: "<<record_no<<endl;
    }
};
```

```
int main()
{
    cout<<"Data of account"<<endl;
    Record r("Manil",101,20000,5,232);
    r.display();
    cout<<endl<<endl;
    cout<<"Data of admin"<<endl;
    Record r1("Vaidhya",102,5000,2,221);
    r1.display1();

    return 0;
}
```

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).