Chapter : 3

Operator Overloading And Type Conversion, Inheritance

Concept Of Operator Overloading :

- C++ introduces a feature known as operator overloading that enables you to give a special meaning to an operator
- In simple terms, we can change the meaning of an operator to apply it to objects
- For example, the + operator can be used to add two numbers of any types.
- We can overload + operator to apply it on the object of a class
- In C++, we can overload all the operators except the following :

These operators cannot be overloaded :

Sr No	Operator Symbol	Name
1	Sizeof	Size of operator
2	?:	Conditional Operator
3	::	Scope resolution operator
4	**.** And **.***	Member access operator

Cont…

- Remember, when you overload an operator, you cannot change its syntax i.e. if you overload binary operator +, it will remain binary operator and will perform operation on 2 operands
- Also, the basic meaning of the operator is not changed when you overload them
- It implies that its original meaning still remains same on normal operands

Overloading Unary And Binary Operators :

- Overloading Unary Operations :

Unary operators are the ones that take only one operand such as unary minus operator. The unary minus (-) operator simply changes the sign of the number,

```
a = 5;
b = -a;
```

Here, we have used unary minus operator with a to negative its value

But the basic meaning of unary minus operator which applies to the normal numbers. Now we will overload this :

Cont.…

- **Syntax :**

```
return_type operator operator_symbol(args)
{
        //code for overloading
}
```

- Here the operator is a keyword to specify that the function defines operator overloading . Symbol specifies the symbol of operator to be overloaded

Program : To Overload Unary – Operator

```
#include<iostream.h>
#include<conio.h>
Class number
{
        int a,b;
        public:
        void input(int x, int y)
        {
                a=x;
                b=y;
        }
        void show()
        {
                cout<<“a=“<<a;
                cout<<“b=“<<b;
        }
        void operator –();
};
Void number :: operator –()
{
        a=-a;
        b=-b;
}
```

Cont….

```
Void main()
{
        number n1;
        n1.input(10,-20);
        n1.show();
        cout<<“After overloading”;
        -n1;                    // - function call
        n1.show();
        getch();
}
```

Overloading Binary Operator :

- Binary operators perform operation on two operands
- The simplest binary operator is + to add two numbers
- We can overload + operator to add two objects
- The + operator will add respected member variables of class and return the resultant object
- Here, we have to pass an object to the operator function so that it can perform operation on members of both objects

Program :

```
#include<iostream.h>
#include<conio.h>
Class numbers
{
        float a,b;
        public:
        void input(int x, int y)
        {
                a=x;
                b=y;
        }
        void show()
        {
                cout<<“a=“<<a;
                cout<<“b=“<<b;
        }
        numbers operator + (numbers n1);
};
Numbers numbers :: operator + (numbers n1)
{
        numbers n;
        n.a=a+n1.a;
        n.b=b+n1.b;
        return n;
}
Void main()
{
        numbers n1;
        n1.input(10,20);
        n1.show();
        numbers n2;
        n2.input(100,200);
```

Cont…

```
        n2.show();
        numbers n3 = n1 + n2;
        cout<<“After adding n1 and n2”;
        n3.show();
        getch();
}
```

Cont…

- Here, you can see that the operator function + specifies return type numbers because it returns an object of class numbers.
- n3= n1 + n2;
- is something like the function call:
- n3 = n1 .**+ (n2);**
- Here, + is the operator function and n2 is passed as argument, the result is returned to n3 so the function should return an object of class numbers

Overloading Of Operators Using Friend Function :

- You can use friend function to overload operators in place of member function
- It makes the function more readable as it takes one argument to overload unary operator and two arguments to overload binary operators
- Without using friend function, the unary operator overloading function does not require any argument and binary operator overloading function needs only one argument
- Example is here,

Example to overload binary * operator :

```
#include<iostream.h>
#include<conio.h>
Class numbers
{
        float a,b;
        public:
        void get(int x, int y)
        {
                a=x;
                b=y;
        }
        void display()
        {
                cout<<“a=“<<a<<endl;
                cout<<“b=“<<b;
        }
        friend numbers operator *(numbers n1, numbers n2);
};

Numbers operator * (numbers n1,numbers n2)
{
        numbers n;
        n.a = n1.a * n2.a ;
```

Cont.…

```
n.b= n1.b * n2.b;
Return n;
}

Void main()
{
        numbers n1;
        n1.get(5,7);
        n1.display();
        numbers n2;
        n2.get(2,3);
        n2.display();
        numbers n3 = n1 * n2;
        //n3=n1.*(n2);
        cout<<“After multiplying n1 and n2”;
        n3.display();
        getch();
}

n.a=10
n.b=21
```

Manipulation Of String Using Operators :

- In C++, you can also use operator overloading to manipulate strings
- For example, you can overload + operator to concatenate two string objects or you can overload == operator to compare two string objects
- In this example, we will overload + operator to concatenate two strings objects and == operator to compare two strings

Example :

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
Class string1
{
        char *s;
        int length;
        public:
        string1()
        {
                length=0;
                s=0;
        }
        string1(char *a)
        {
                length=strlen(a);
                s=new char[length+1];
                strcpy(s,a);
        }
        void display()
        {
                cout<<“String=“<<s;
        }
        friend void operator ==(string1 s1, string1 s2);
        friend void operator +(string1 s1, string1 s2);
};
```

Cont…

```
Void operator ==(string1 s1, string1 s2)
{
        if(strcmp(s1.s,s2.s)==0)
        {
                cout<<“same string”;
        }
        else
        {
                cout<<“not same”;
        }
}
```

```
String1 operator +(string1 s1, string1 s2)
{
        string1 s3;
        s3.s=new char[s1.length+s2.length+1];
        strcpy(s3.s, s1.s);
        strcat(s3.s, s2.s);
        return s3;
}
```

Cont…

```
Void main()
{
        string1 s1(“Leeza”);
        string1 s2(“Patel”);
        string1 s3(“Leeza”);
        cout<<“comparing s1,
s2”;
        s1==s2;
        cout<<“comparing s1,
s3”;
        s1==s3;
        string1 s4;
        s4=s1+s2;
        s4.display();
        getch();
}
```

Cont.…

- Here, we have used 2 constructors for creating string objects
- The == operator just compares 2 strings and prints message whether they are same or not
- The + operator creates a new array of character and allocates memory required by it

Rules for operator overloading :

- There are some rules that should be considered when using operator overloading:

1. By overloading an operator, you cannot change the original meaning of an operator. For ex, you cannot overload * operator to add 2 numbers instead of multiplying them
2. You cannot change the syntax of the original operators. For ex, a-b subtracts b from a. you cannot change this rule by overloading them
3. You can overload only existing operators. You cannot define your own operator
4. Some operators cannot be overloaded as discussed above

Cont…

5. Overloading unary operators using member function will not take any argument, but using friend function it will take one argument
6. Overloading binary operators using member function will take one argument, but using friend function, it will take 2 arguments
7. You cannot use friend function to overload following operators

Sr no	Operator symbol	Name
1	=.	Assignment operator
2	()	Function call operator
3	[]	Array indexing or subscript
4	->	Class member access op

Type Conversion :

- When we write an expression containing variables of different data types, type conversion is must. (implicit or explicit)
- For example, float a=12.34; int b=a;
- Here the value of a is transferred to b, but the fractional part will be truncated as the variable b is of type integer.
- This also applies to class objects.
- test t1,t2,t3;
- t1=t2+t3;
- This is also valid as the t1,t2,t3 are the objects of same class and provided proper operator overloading

Cont…

- But what, if there are objects of different class or combination of objects and variable data type variable?
- This can be done by performing proper type conversion method which can be done as follows:

1. Basic type to class type conversion
2. Class type to basic type conversion
3. One class to another class conversion

1. Basic type to class type conversion :

- Consider a class test and the statement

```
test t1;
int a;
t1=a; //int to class type conversion
```

- Here is an example :

Program:

```
#include<iostream.h>
#include<conio.h>
Class time
{
        int hours,minutes;
        public:
        time()
        {
        hours=minutes=0;
        }
        time(int t)
        {
        hours=t/60;
        minutes=t%60;
        }
        void display()
        {
        cout<<“hours =“<<hours;
        cout<<“minutes=“<<minutes;
        }
};

Void main()
{
        int t;
        cout<<“Enter time in minutes”;
        cin>>t;
        time t1=t;              //time t1(t);
        t1.display();
        getch();
}
```

2. Class to Basic type conversion :

- In basic type to class type conversion, we created a constructor to perform the conversion
- But to perform class to basic type conversion, you have to define a conversion function for the type you want to convert.
- General form is :

```
operator basic_type_nm()
{
        //code
}
```

Cont…

- Following condition should meet:

1. The conversion function must be the member of class
2. The function must not specify any return type
3. It cannot take any arguments

Example :

```
#include<iostream.h>
#include<conio.h>
Class product
{
        int qty;
        float price;
        public:
        product(int q,float p)
        {
                qty=q;
                price=p;
        }
        void display()
        {
        cout<<“quantity is”<<qty;
        cout<<“Price is”<<price;
        }
        operator float()
        {
        return qty*price;
        }
};
```

Cont.…

```
Void main()
{
        product p(10,25);
        p.display();
        float amount;
        amount=p;
        cout<<“total amount=“<<amount;
        getch();
}
```

3. One Class to another class conversion :

- We may need to apply one class to another class conversion in some cases where in an expression there are objects of different classes.
- For example,

  obj1=obj2;
- Here, obj1 and obj2 are objects of different classes
- For this, we will create 2 classes, shop1 and shop2
- Also create constructor to implement conversion of one class object to another class object

Example :

```
#include<iostream.h>
#include<conio.h>
Class shop1
{
        int code,qty;
        float price;
        public:
        shop1(int c,int q,float p)
        {
        code=c;
        qty=q;
        price=p;
        }
int getcode()
{
return code;
}
int getqty()
{
return qty;
}
float getprice()
{
return price;
}
```

Cont…

```
        void display()
        {
        cout<<“shop1 details”;
        cout<<“item code”<<code;
        cout<<“quantity”<<qty;
        cout<<“price”<<price;
        }
};
```

```
Class shop2
{
        int code;
        float amount;
        public:
        shop2(int c, float a)
        {
        code=c;
        amount=a;
        }
```

Cont….

```
        void display()
        {
        cout<<“shop2 details”;
        cout<<“item code”<<code;
        cout<<“total amount”<<amount;
        }
        shop2(shop1 s1)
        {
        code=s1.getcode();
        amount=s1.getqty()*s1.getprice()
;
        }
};
```

```
Void main()
{
        shop1 s1(111,5,24.5);
        shop2 s2=s1; //shop2 s2(s1);
        s1.display();
        s2.display();
        getch();
}
```

Comparison of different methods of conversion :

Conversion Type	Point to remember	Remarks
Basic to class type	**Constructor:** Test(int a)	Basic type as argument of constructor Int a; Test t=a;

Class type to basic	**Casting operator function**	Operator function of specific basic type
		Operator int()
		{
		}

One class to another class	**Constructor(copy constructor)**
	Obj2=obj1;
		The source class' object as argument of the constructor
		Class2(class1 c1)
		{
		}

Defining Derived Class :

- It is the concept under Inheritance. Inheritance provides benefits of code reusability and hierarchical classification
- In C++, a class can use some or all properties of another class using inheritance
- The class which is being inherited is known as **Base Class** and the class that inherits the base class is known as **Derived Class**
- To define a derived class from a base class is known as inheriting a class from a base class
- By deriving a class the class acquire some or all properties of the base class

Cont…

- Syntax :

```
class derived_class_name : [private/public] base_class_name
{
    //code
};
```

- Here : symbol specifies the inheritance means it specifies that the class at the left side of the symbol is derived class and the class at the right side is the base class
- The base class can be derived privately or publically.

Cont…

- If you do not specify any visibility, the class is derived privately. Means the private members of the base class remain private and the public members also become private in the derived class
- Example,

```
class derived : base
{
        //code
};
```

Here if private keyword is omitted it will have the same effect as it is default visibility modifier

Cont.…

- If the visibility mode is public, then the private members of base class remain private and the public members remain public in the derived class
- class derived : public base

```
{
    //code
};
```

Types Of Inheritance :

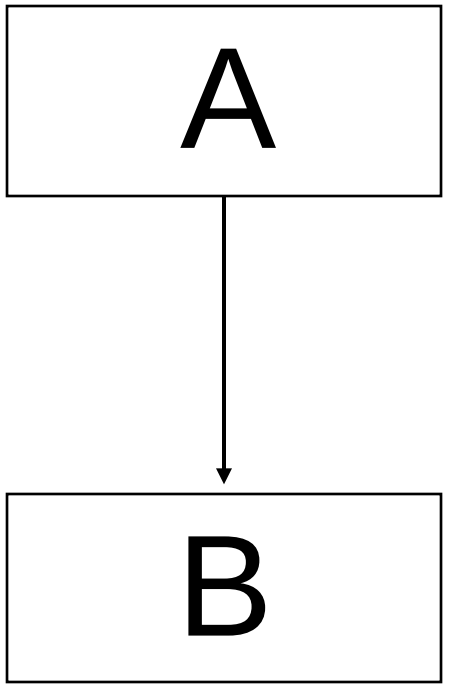
- There are total 5 types of Inheritance in C++ as listed below :

1. Single Inheritance
2. Hierarchical Inheritance
3. Multiple Inheritance
4. Multi-level Inheritance
5. Hybrid Inheritance

1. Single Inheritance

- In Single Inheritance, there is only one base class and one derived class. The general structure of a single inheritance is shown here :

```
Class Base
{
        //code
};
Class Derived : public Base
{
        //code
};
```



Program :

```
#include<iostream.h>
#include<conio.h>
Class base
{
        public:
        int x;
        void getdata()
        {
        cout<<“Enter value of x”;
        cin>>x;
        }
};
```

```
Class derived : public base
{
        int y;
        public:
        void get()
        {
        cout<<“Enter value of y”;
        cin>>y;
        }
        void product()
        {
        cout<<“product is”<<x*y;
        }
};
```

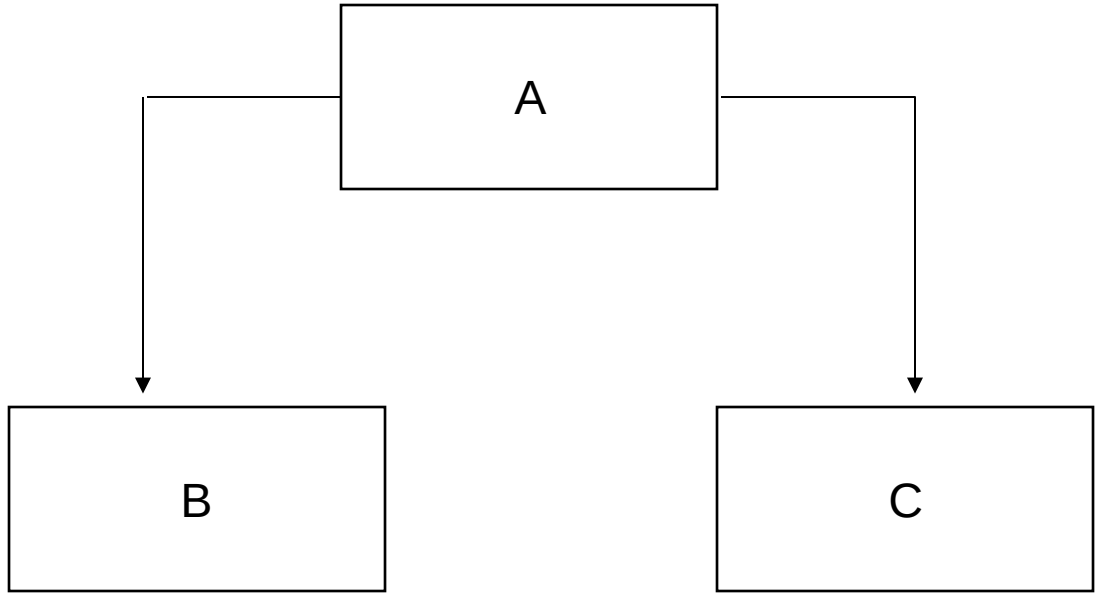
Cont..

```
Void main()
{
        derived d;
        d.getdata();
        d.get();
        d.product();
        getch();
}
```

2. Hierarchical Inheritance :

- When in a program, there is only one base class and several classes from the single base class, it is known as hierarchical inheritance.

```
Class base
{
        //code
};
Class derived1 : public base
{
        //code
};
Class derived2 : public base
{
        //code
};
```



Program :

```
#include<iostream.h>
#include<conio.h>
Class A
{
        public:
        int x,y;
        void getdata()
        {
        cout<<“Enter x and y”;
        cin>>x>>y;
        }
};
Class B : public A
{
        public:
        void product()
        {
        cout<<“Product is”<<x*y;
        }
};
Class C : public A
{
        public:
        void sum()
        {
        cout<<“Sum is”<<x+y;
        }
};
```

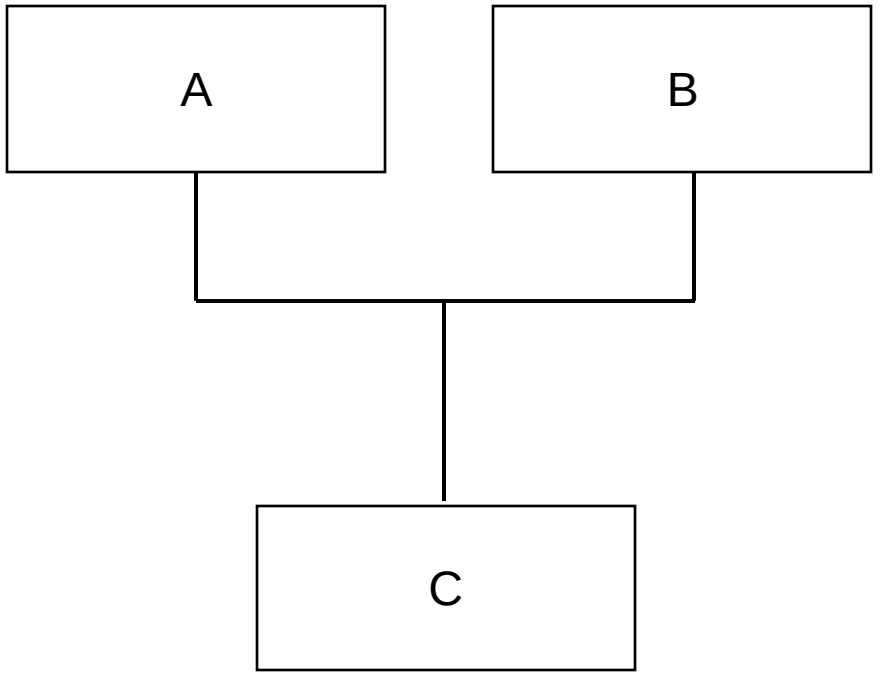
Cont..

```
Void main()
{
        B objb;
        C objc;
        objb.getdata();
        objb.product();
        objc.getdata();
        objc.sum();
        getch();
}
```

3. Multiple Inheritance :

- When in a program, a class derives from more than one base class, it is known as multiple inheritance.
- Syntax:

```
class base1
{
        //code
};
class base2
{
        //code
};
class derived : public base1,public base2
{
        //code
};
```



Example :

```
#include<iostream.h>
#include<conio.h>
class A
{
        public:
        int a;
        A()
        {
        a=5;
        cout<<“Constructor of A”;
        }
};
```

```
Class B
{
        public:
        int b=10;
        B()
        {
        cout<<“Constructor of B”;
        }
};
Class C : public B, public A
{
```

Cont…

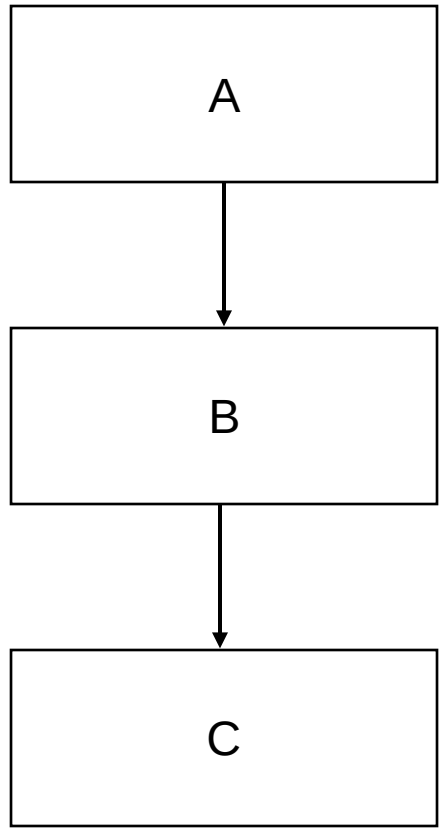
```
        public:
        int c=20;
        C()
        {
        cout<<“Constructor of c”;
        }
};
```

```
Void main()
{
        C obj;
        cout<<“a=“<<obj.a;
        cout<<“b=“<<obj.b;
        cout<<“c=“<<obj.c;
        getch();
}
```

4. Multi-level Inheritance :

- When a class derives from another derived class, it is known as multi-level inheritance.
- Syntax:

```
Class base
{
        //code
};
Class derived1 : public base
{
        //code
};
Class derived2 : public derived1
{
        //code
};
```



Example :

```
#include<iostream.h>
#include<conio.h>
Class A
{
        public:
        void dispA()
        {
        cout<<"display of A";
        }
};
Class B : public A
{
        public:
        void dispB()
        {
        cout<<"display of B";
        }
};
Class C : public B
{
        public:
        void dispC()
        {
        cout<<"display of C";
        }
};
```

Cont…

```
Void main()
{
        C obj;
        obj.dispA();
        obj.dispB();
        obj.dispC();
        getch();
}
```

5. Hybrid Inheritance :

- Hybrid inheritance is the combination of two different types of inheritance.
- For example, there is a combination of multiple and hierarchical inheritance is known as hybrid inheritance.
- Syntax:

```
class base
{
        //code
};
class derived1: public base
{
        //code
};
class derived2 : public base
{
        //code
};

class derived3 :public derived1, public derived2
{
        //code
};
```

Example :

```
#include<iostream.h>
#include<conio.h>
Class A
{
        public:
        int x;
};
Class B : public A
{
        public:
        B()
        {
                x=10;
        }
};
{
Class C
        public:
        int y;
        C()
        {
                y=4;
        }
};
Class D :public B, public C
{
        public:
        void sum()
        {
                cout<<“sum”<<x+y;
        }
};
```

Cont…

```
Void main()
{
        d obj;
        obj.sum();
        getch();
}
```

Visibility Modifiers :

- C++ supports 3 useful modifies as follows:

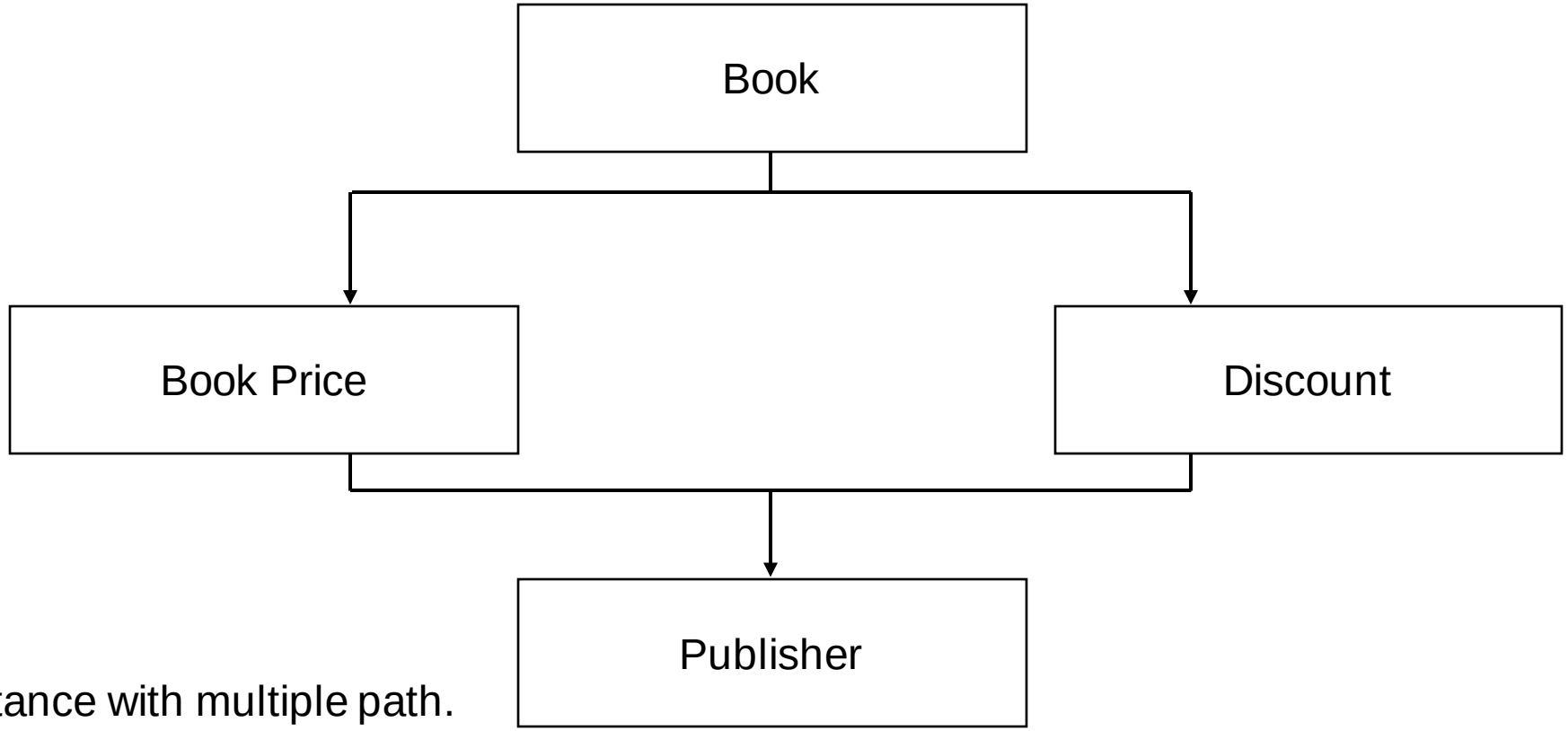
1. Private
2. Protected
3. Public

Access	**Private**	**Protected**	**Public**
Within the same class	Yes	Yes	Yes
In derived class	No	Yes	Yes
In classes other than derived class	No	No	yes

Cont…

- Private : The private members can be accessed within the same class only
- Protected : The protected members can be accessed within the same class as well as they can be accessed by the members of its derived class too. But cannot be accessed by the other class that does not inherit them
- Public : The public members can be accessed from anywhere in the program. Within the class, derived class and from other class also.

Virtual Base Class :



Hybrid inheritance with multiple path.

Cont…

- In hybrid inheritance, we combined multilevel and multiple inheritance.
- If we try to implement the above inheritance, there might be some problems because all the members of book class will be inherited in publisher class twice, one via bookprice and other via discout class.
- So properties of book class in publisher class will create ambiguity and will result in error.
- The solution to this problem is virtual base class.
- We can remove ambiguity by declaring the base class of publisher i.e. bookprice and discount as virtual base class.

Cont.…

- It ensures that the properties of the book class will be inherited only once.
- A class can be declared as virtual by using the virtual keyword.
- Syntax :

```
class grandparent
{
};
class parent1 : virtual public grandparent
{
};
class parent2 : virtual public grandparent
{
};
class child : public parent1, public parent2
```

Example :

```
#include<iostream.h>
#include<conio.h>
Class book
{
        protected:
        int b_id;
        char name[10];
        public:
        void getbook()
        {
        cout<<“Enter book id”;
        cin>>b_id;
        couy<<“Enter book name”
        cin>>name;
        }
        void showbook()
        {
        cout<<“Book id”<<b_id;
        cout<<“Book name”<<name;
        }
};
Class bookprice : public virtual book
{
        protected:
        double price;
        public:
        void getprice()
        {
        cout<<“Enter book price”;
        cin>>price;
        }
```

Cont…

```
        void showprice()
        {
        cout<<“Book price”<<price;
        }
};
Class disount: virtual public book
{
        protected:
        double d;
        public:
        void getdiscount()
        {
        cout<<“Enter discount %”;
        cin>>d;
        }
```

```
        void showdiscount()
        {
        cout<<“discount”<<d<<“%”;
        }
};
Class publisher: public bookprice, public discount
{
        double fp;
        char pub[20];
        public:
        void getdetail()
        {
        getbook();
        getprice();
        getdiscount();
        cout<<“Enter publisher name”;
        cin>>pub;       }
```

Cont….

```
Void showdetail()
{
        showbook();
        showprice();
        showdiscount();
        fp=price-(price*d/100);
        cout<<“Final price”<<fp;
        cout<<“Book
publisher”<<pub;
}
};
```

```
Void main()
{
        publisher p;
        p.getdetail();
        p.showdetail();
        getch();
}
```

Abstract Class :

- An Abstract class, as its name implies, is a class which is not fully defined.
- Generally its objects are not created
- It is defined so that it can be inherited by its derived classes
- It just provides a base for its derived class
- The base class book in the above example can be called abstract class as we haven’t created its object

Constructors in Derived Class :

- In case of inheritance, if your base class contains a constructor with no arguments, the derived class does not need a constructor.
- But if, the base class contains a constructor with arguments then the derived class must have a constructor with argument
- If both the base and derived class have constructors, the base class constructor is executed first and then derived class constructor is executed
- In multiple inheritance, the constructor called in the order of the base class written

Cont…

- Example

Class derived: public base1, public base2

{

};

Here, the constructor of base1 is called first as it appears first

Class derived: public base2, public base1

{

};

Here, the constructor of base2 is called first.

Cont.…

- In case of virtual base class, the virtual class constructor is called first.

Class derived : public base1, virtual public base2

{

};

In multilevel inheritance, the constructors are executed in order of inheritance. The grandparent class constructor first, then the parent class and finally the child class constructor is executed

Example :

```
#include<iostream.h>
#include<conio.h>
Class grandparent
{
        public:
        grandparent()
        {
        cout<<“Grandparent”;
        }
};
Class parent: public grandparent
{
        public:
parent()
        {
        cout<<“Parent”:
        }
};
Class child :public parent
{
        public:
        child()
        {
        cout<<“child class”;
        }
};
```

Cont…

```
Void main()
{
        child c;
        getch();
}
```

Applications of Constructor and Destructor in Inheritance :

- Constructors and destructors play very important role in initialization of objects
- Similarly, they are important in inheritance also
- The main benefit of using constructor in inheritance is that the constructors of base class can be derived in derived class easily
- So the reusability concept is applied to the constructors and destructors also.
- It means that the derived class can use the constructors of base class and do not need to initialize the members again in derived class

Containership :

- When in a class, objects of other classes are created as member variables, the objects of that class will contain also the object created as member.
- This type of relationship is known as Containership.

```
Class a
{
};
```

```
Class B
{
        int a;
        a obja;
        public:
        B(int a, a obj): obja(obj)
        {
        }
};
```