C++

Operator Overloading AND Type Conversion, Inheritance

Concept Of Operator Overloading

- C++ introduces a striking feature known as **Operator Overloading** that enables us to give a special meaning to an operator.
- In simple terms, using **Operator Overloading** 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 primitive types. We can overload the + operator to apply it on the objects of a class.

Concept Of Operator Overloading

- List Of Operator that cannot be overloaded

Operator Symbol	**Operator Name**
sizeof()	Size Of Operator
?:	Conditional Operator
::	Scope Resolution Operator
. and .*	Member Access Operators

Concept Of Operator Overloading

- Remember that when we overload on operator, we cannot change its syntax.
- If we overload binary operator **+**, it will remain binary operator and will perform operation on **two** operands.
- Here, the basic meaning of the operators is not changed when we overload them. It implies that its original meaning still remains same on normal operands.

Overloading Unary and Binary Operators:

- Overloading Unary Operators :
  - Unary operators are the once that take only one operand such as unary minus operator.
  - The unary minus (-) operator simply changes the sing of the number.
  - Example:

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

This unary operator will convert the positive value to negative value.

Overloading Unary and Binary Operators:

- Syntax :

```
Return_type operator Operator_Symbol(Args)
{
    //Code For Overloading
}
```

- Here, the *operator* is a keyword to specify that the function defines operator overloading.
- The operator symbol specifies the symbol of the operator you want to overload.
- We can pass arguments to the operator function if needed.

Overloading Unary and Binary Operators:

- Overloading Binary Operators :
  - Binary operators perform operation on two operands.
  - The simplest binary operator is + to add two numbers. We can overload + operator to add 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.

Overloading Binary Operator :

Using Friend Function :

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

Manipulation of String Using Operators :

- In C++, we can also use operator overloading to manipulate strings.
- For Example:
  - We can overload **+** operator to concatenate two string objects.
  - We can overload **==** operator to compare two string objects.

Rules for Operator Overloading :

- There are some rules that should be considered when using operator overloading.
  - By overloading an operator, we cannot change the original meaning of an operator.
  - We cannot change the syntax of the original operator. We cannot change the rule by overloading operator.
  - We can overload only existing operators. We cannot define our own operator.

Rules for Operator Overloading :

- Some operators cannot be overloaded.
- Overloading unary operators using member function will not take any argument, but using friend function it will take one argument.
- Overloading binary operators using member function will take one argument, but using friend function it will take two arguments.

Rules for Operator Overloading :

- We can not use friend function to overload some operators:

Operator Symbol	**Operator Name**
=.	Assignment Operator
( )	Function Call Operator
[ ]	Subscript or Array indexing operator
->	Class member access operator

Type Conversion :

- When we write an expression containing variables of different data types, type conversion is necessary (Whether implicit or explicit)
- Example :

```
float a=12.34;
int b=a;
```

- In the above example, the value of **a** is transferred to **b**, but the *fractional part will be truncated* as the variable of **b** is integer.

Type Conversion :

- We can also applies the concept of conversion to class.
- This can be done by performing proper type conversion as following:
  - Basic type to class type conversion
  - Class type to basic type conversion
  - One class to another class conversion
- Basic Type To Class Type Conversion
  - To understand this situation, consider a class Test and following statement:

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

Class type to basic type conversion

- In basic type to class type conversion, we create a constructor to perform the conversion.
- But to perform class to basic type conversion, we have to define a conversion function for the type we want to convert.

operator ***basic_type_name()***

{ *//Conversion statements…* }

- Here, the basic_type_name can be any basic data type such as int, float, double etc.

Class type to basic type conversion

- We can define the operator function for the type we want to convert into but the casting (conversion) operator function should meet following conditions:
  - The conversion function must be the member of class.
  - The function must not specify any return type.
  - It cannot take any argument.

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.
- In the following Example:
  - We will create two classes **shop1** and **shop2.** We will create a constructor to implement conversion of one class object to another class object.

Comparison of Different Method of Conversion :

- We learned all the 3 methods of type of conversions.
- We have to take care of some specific point which is mentioned in the following table.

Conversion Type	Point to Remember	Remarks

Basic to class type	**Constructor**
	Example:
	Test(int a)
		Basic Type as argument of the constructor.
		Example:
		int a;
		Test t=a;

Comparison of Different Method of Conversion :

Conversion Type	Point to Remember	Remarks

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

One class to another class	**Constructor**
	Example:
	Obj2=Obj1;
		The source class' object as argument of the constructor
		class2(class1 c1)
		{
		}

Introduction :

- Inheritance is a way by which we can get benefit of reusability.
- In C++, a class can use some or all properties of another class using inheritance.
- The class which is being inherited is also known as the **base class** and the class that inherits the base class is known as the **derived class.**
- **Defining Derived Classes :**
  - 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 can acquire some or all properties of the base class.
- Follow is the syntax to define a derived class

```
class Derived_Class_Name:[private|public]
                         Base_Class_Name
{
        //Derived class definition
};
```

- Here the **:** 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 either privately or publically.
- 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.
- There fore all the members of the base class will be inaccessible by the object of the derived class.
- Example :

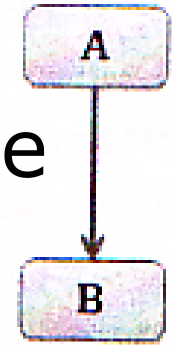
```
class Derived_class:private Base_Class
{         //class definition           };
```

- Here if private keyword is omitted it will have the same effect as it is the default visibility modifier.
- 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.
- Therefore only the public members of the base class can be accessed by the object of the derived class.
- Example :

```
class Derived_Class:public Base_Class
{                   };
```

Types Of Inheritance :

- There are total five types of inheritance in C++ as listed below:
  - Single Inheritance
  - Hierarchical Inheritance
  - Multiple Inheritance
  - Multi-level Inheritance
  - Hybrid Inheritance
- Single Inheritance
  - In single inheritance, there is only one base class and one derived class.



Single Inheritance

Types Of Inheritance : Single Inheritance

```
class Base
{          //Base Class Definition
};
class Derived:[public | private] Base
{          //Derived Class Definition
};
```

- We have to create a base class and the base class properties are derived by a derived class.
- Note that the derived class can not access the private properties of its base class.

Visibility Of Modifiers :

- We had learn public and private modifiers.
- C++ support one more useful visibility modifier that has more visibility than **private** and less visibility than public members.

Modifier / Access	Private	Protected	Public
Within same class	Yes	Yes	Yes
In derived class	No	Yes	Yes
In classes other than derived class	No	No	Yes

Visibility Of Modifiers :

- Private :
  - The private members can be accessed within the same class only.
- Protected :
  - The members declared as protected can be accessed within the same class well as they can be accessed by the ***members of its derived class also.***
- Public :
  - The public members can be accessed from anywhere in the program. They can be accessed within the same class, derived class and from other classes also.

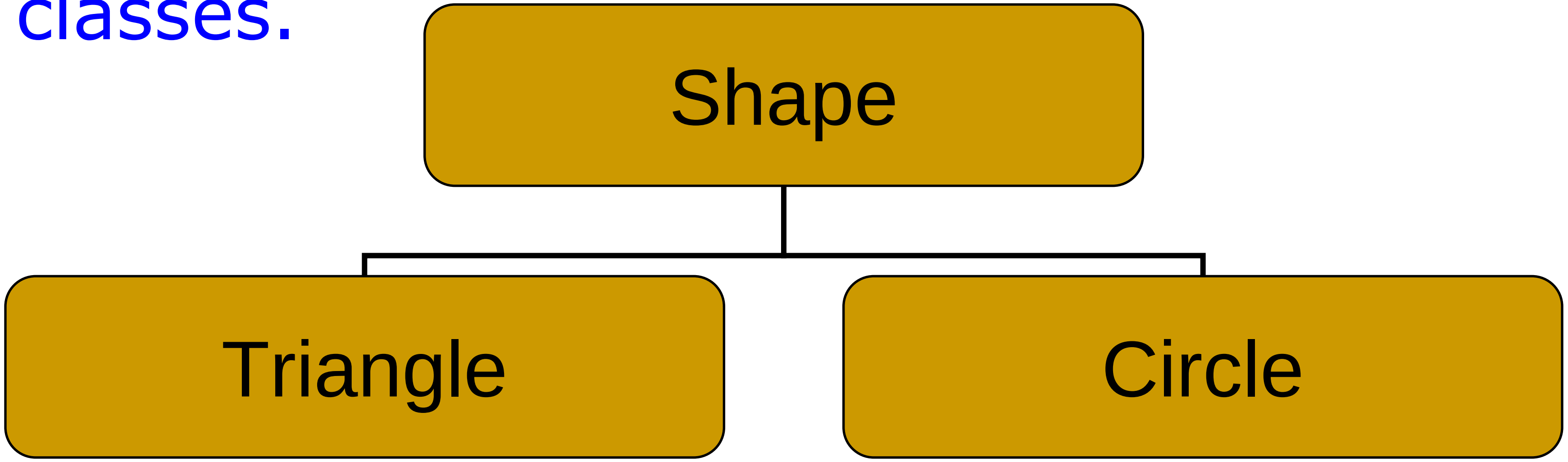
Types Of Inheritance: Hierarchical Inheritance

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

```
class Base
   { //Base class definition  };
class Derived1:[Public|Private] Base
   { //Derived class1 definition   };
class Derived2:[Public|Private] Base
   { //Derived class2 definition   };
```

Hierarchical Inheritance

- In hierarchical inheritance, we can create hierarchical classification.
- We can create multiple classes from a base class.
- The new classes will acquire the properties of the base class and in addition they can have their own properties in the new classes.



Hierarchical Inheritance : **Structure of HI**

```
class Base
{    //Base Class  definition
};
class Derived1:[public|private] Base
{    //Derived Class1 definition
};
class Derived2:[public|private] Base
{    //Derived Class2 definition
};
```

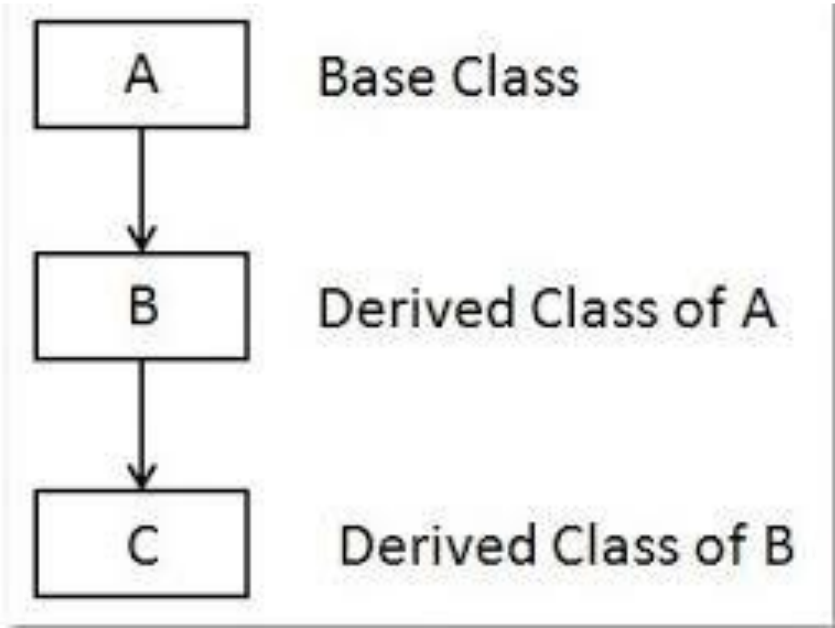
Multiple Inheritance

- A class derives from more then one base classes, then it is known as **multiple inheritance**.
- Structure of Multiple Inheritance.

```
class Base1
{   //Base Class 1 definition
};
class Base2
{   //Base Class 2 definition
};
class Derived:[public|private]Base1,
                          [public|private]Base2
{   //Derived class definition
};
```

Multilevel Inheritance

- In multilevel inheritance we can create more levels of inheritance.
- We can derive from a derived class.
- For Example :
  - Consider a class book having member variables book_id and name.
  - We can derive this class to add book price and further derive it to add other details.

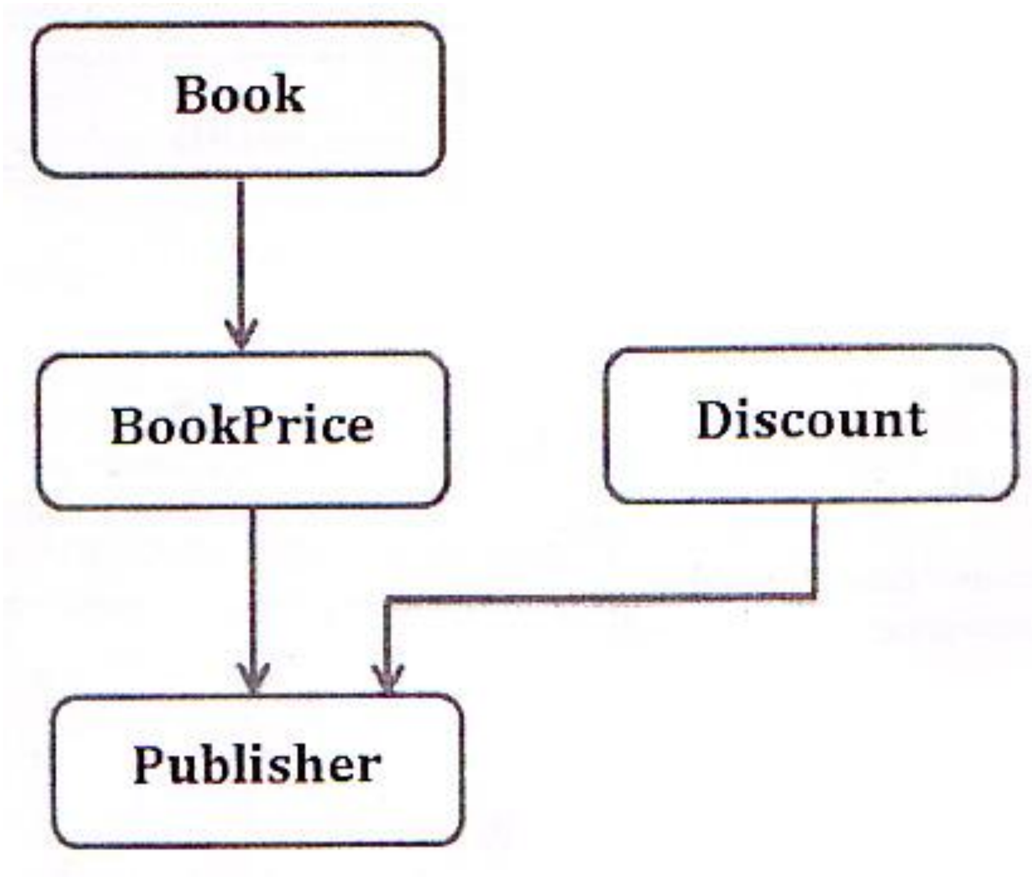


Multilevel Inheritance : **Structure of MI**

```
class Base
{     //Base Class 1 definition
};
class Derived1:[public|private]Base
{     //Derived1 definition
};
class Derived2:[public|private]Dervied1
{     //Derived2 class definition
};
```

Hybrid Inheritance

- Hybrid inheritance is combination of two different types of inheritance.
- In hybrid inheritance, we can combine two or more types of inheritance.
- For example,



Hybrid Inheritance : **Structure of HI**

```
class Base
{     //Base Class definition
};
class Derived1:[public|private]base
{     //Derived class1 definition
};
class Derived2:[public|private]base
{     //Derived class2 definition
};
class Derived3:[public|private]Derived1,
                    [public|private]Derived2
{     //Derived class3 definition
};
```

Virtual Base Class & Abstract Class

- Virtual Base Class:
  - A can be declared as virtual by using the virtual keyword.
  - The virtual keyword can be used either before or after the visibility modifier.
- Syntax :

```
class Grandparent
{
};
class Parent1:virtual public  Grandparent
{
};
```

Virtual Base Class

```
class Parent2:public virtual Grandparent
{
};
class Child:public Parent1, public Parent2
{
};
```

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 classes.

Constructors in Derived Class :

- In case of inheritance, if our 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 constructor is executed.

- In multiple inheritance, the constructors are called in the order of the base class written.

Applications of Constructor and Destructor

- Constructors and destructors play very important role in initialization of objects.
- Similarly constructors and destructors are very important in inheritance.
- 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, Inheritance v/s Containership

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

Containership, Inheritance v/s Containership

- **Inheritance V/S Containership :**
  - Creating object that contain another object is different as compared to creating normal objects.
  - In the containership when the object is created, first the member objects are created using their constructors and then the normal members are initialized.
  - While in inheritance when the object is created, the base class constructors are called first and then the derived class constructors are called based on the type of inheritance.