CS-15
C++ and Object Oriented Programming

Unit – 3

Presented By : Dhruvita Savaliya

Operator Overloading and type conversion, Inheritance

- **TOPICS :**
- Concept of operator overloading
- Overloading unary and binary operators
- Overloading of operators using friend Function
- Manipulation of string using operators
- Rules for operator overloading
- Type conversions
- Comparison of different method of conversion
- Defining derived classes
- Types of inheritance (Single, Multiple, Multi-level,
- Hierarchical, Hybrid)
- Virtual base class & Abstract class
- Constructors in derived class
- Application of Constructor and Destructor in inheritance
- Containership, Inheritance V/s Containership

Operator Overloading :

- in C++, Operator overloading is a compile-time polymorphism.
- It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
- For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +.
- Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big integers, etc.
- In this case, we can define the behavior of the + operator to work with objects as well.
- This concept of defining operators to work with objects and structure variables is known as **operator overloading**.
- The advantage of Operators overloading is to perform different operations on the same operand.

- **Operator that *cannot* be overloaded are as follows:**
  - Scope operator (::)
  - Sizeof
  - member selector(.)
  - member pointer selector(*)
  - ternary operator(?:)
- **Operators that *can be* Overloaded in C++ : →**

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, —
Assignment	=, +=,*=, /=,-=, %=
Bitwise	& , | , << , >> , ~ , ^
De-referencing	(->)
Dynamic memory allocation, De-allocation	New, delete
Subscript	[ ]
Function call	()
Logical	&, | |, !
Relational	>, < , = =, <=, >=

- **Criteria/Rules to Define the Operator Function :**
- In the case of a **non-static member function**, the binary operator should have only one argument and the unary should not have an argument.
- In the case of a **friend function**, the binary operator should have only two arguments and the unary should have only one argument.
- Operators that cannot be overloaded are **.* :: ?:**
- Operators that cannot be overloaded when declaring that function as friend function are **= () [] ->**.
- The operator function must be either a non-static (member function), global free function or a friend function.

- **Syntax :**
  returnType **operator** op_symbol (arguments)
  {
  ... .
  }
- Here,
  returnType - the return type of the function
  operator - a special keyword
  symbol - the operator we want to overload (+, <, -, ++, etc.)
  arguments - the arguments passed to the function

Overloading unary and binary operators :

❖ **Overloading Unary Operator :**

- Let us consider overloading (-) unary operator.
- In the unary operator function, no arguments should be passed.
- It works only with one class object. It is the overloading of an operator operating on a single operand.

Example :

```
#include<iostream.h>
#include<conio.h>
class op_cls
{
   int a;
   public:
   op_cls(int x)
   {
        a=x;
   }
   void show()
   {
        cout<<"value of A
   :"<<a;
   }
   void operator  -()
   {
        a=-a;
        cout<<"hello"<<a;
   }
};
void main()
{
   clrscr();
   op_cls object(10);
   object.show();
   -object;
   object.show();
   getch();
}
```

❖ 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.

- **Example of Binary Operator overloading :**

```
class op_cls
{
  int a;
  public:
  op_cls( ){ };
  op_cls(int x)
  {
        a=x;
  }
  void show()
  {
        cout<<"\nvalue of A
  :"<<a;
  }
  op_cls operator -(op_cls
  obj_para)
  {
        op_cls ret_obj;
        ret_obj.a=a+obj_para.a;
        return ret_obj;
    }

};
void main()
{
  clrscr();
  op_cls obj1(10),obj2(29);
  obj1.show();
  op_cls obj3(0);

  obj3=obj1+obj2;
  //obj1 is calling object
  //obj2 is parameter object
  obj3.show();
  getch();
}
```

Overloading of operators 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.
- *We can not use friend function to overload some operators:*

1. = Assignment Operator
2. ( )Function Call Operator
3. [ ]Subscript or Array indexing operator
4. ->Class member access operator

- **Example of unary operator overloading using friend function**

```
class op_cls
{
  int a;
  public:
  op_cls(int x)
  {
        a=x;
  }
  void show()
  {
        cout<<"value of A :"<<a;
  }
  friend void operator -(op_cls
  &o);
};
void operator -(op_cls &o)
{
  o.a=-o.a;//it assign value of
  a : negative
}
void main()
{
  clrscr();
  op_cls object(10);
  object.show();
  -object;
  object.show();
  getch();
}
```

- **Example of Binary operator Overloading using friend function :**

```
class op_cls
{
    int a;
    public:
    op_cls(){};
    op_cls(int x)
    {
            a=x;
    }
    void show()
    {
            cout<<"\nvalue of A :"<<a;
    }
    friend op_cls operator +(op_cls , op_cls);
};
op_cls operator +(op_cls o1,op_cls o2)
{
    op_cls ret_obj;
    ret_obj.a=o1.a+o2.a;
    return ret_obj;
}
void main()
{
    clrscr();
    op_cls obj1(10),obj2(29);
    obj1.show();
    op_cls obj3(0);
    obj3=obj1+obj2;
    //obj1 is calling object & obj2 is parameter object
    obj3.show();
    getch();
}
```

Rules for Operator Overloading :

1. By overloading an operator, we cannot change the original meaning of an operator.
2. We cannot change the syntax of the original operator.
3. We cannot change the rule by overloading operator.
4. We can overload only existing operators.
5. We cannot define our own operator.
6. Some operators cannot be overloaded. **(.* :: ? :)**
7. Overloading unary operators using member function will not take any argument, but using friend function it will take one argument.
8. Overloading binary operators using member function will take one argument, but using friend function it will take two arguments.

9. 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

10. We can overload this operators using friend function :

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, —
Assignment	=, +=,*=, /=,-=, %=
Bitwise	& , | , << , >> , ~ , ^
Logical	&, | |, !
Relational	>, < , = =, <=, >=

Manipulation of string using operators :

- There are lots of limitations in string manipulation in C as well as in C++.
- Implementation of strings requires character arrays, pointers and string functions.
- C++ permits us to create our own definitions of operators that can be used to manipulate the strings very much similar to other built-in data types.
- ANSI C++ committee has added a new class called string to the C++ class library that supports all kinds of string manipulations.
- Strings can be defined as class objects which can be then manipulated like the built-in types.
- Since the strings vary in size, we use new to allocate memory for each string and a pointer variable to point to the string array

- **Example of string manipulation :**

```
#include<iostream.h>
#include<conio.h>
#include<string.h>
class op_cls
{
   public:
   char s1[20],s2[20];
   op_cls(){};
   op_cls(char x[],char y[])
   {
          strcpy(s1,x);
          strcpy(s2,y);
   }
   void show()
   {
          cout<<"\nString
   Concatenation :"<<s1<<endl;
   }
   operator +()
   {
          strcat(s1,s2);
          //it is very similar to
          //s1=s1+s2
          return 0;
   }

};
void main()
{
   clrscr();
   op_cls obj1("hello"," bca");
   +obj1;
   obj1.show();
   getch();
}
```

Type conversions :

- When the different types are *Mixed* in an expression, then automatic type conversion is applied on operands according to certain rules(Whether implicit or explicit).
- *Similarly in assignment operation, the type of data on the right hand of assignment operator then automatically converted to the type of variable on the left.*
- **Example :**

```
float a=12.34;
int b=a;//here value of b is 12
```

- In the above example, the value of a is transferred to b, but the ***fractional part will be truncates*** *the variable of b is integer.*
- The automatic type conversion is performed only when only primitive types are involved in an expression.
- When User Defined Types are involved in an expression then automatic type conversion is cannot be performed.

❖ Type conversions with the User Defined Types (Casting operator Functions):

- We can also applies the concept of conversion to class.
- This can be done by performing proper type conversion as following:

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

Basic Type To Class Type Conversion :

- When we create objects using the variable of primary/ primitive types then it is called basic to class type conversion.
- Generally we use single argument constructor to perform type conversion from basic type to class type.

Example for Basic to Class Type :

```
#include<iostream.h>
#include<conio.h>
class type
{
    int n;
    public:
    type(int n)
    {
            this->n=n;
    }
    void show()
    {
            cout<<"basic to class type"<<n;
    }
};
void main()
{
    clrscr();
    type t=10;//it will convert int to obj
    t.show();
    getch();
}
```

Class type to basic type conversion :

- When we assign an object to a primitive data type’s variable, then it is called as class type to basic conversion.
- To perform class type to basic type conversion we have to define the casting operator function.
- The casting operator function must be a member of the class.
- The casting operator function cannot have any return datatype and it does not take any parameter.
- **Syntax:**

```
operator datatype( )
{
        ….;
}
```

- **Example for class type to basic**

```
#include<iostream.h>
#include<conio.h>
class type
{
    int n;
    public:
    type(int n)
    {
            this->n=n;
    }
    operator int( )
    {
            return n;
    }
};
void main()
{
    clrscr();
    type t=10;//it will convert int to obj

    int b_dt=t;//it will convert obj to int
    cout<<endl<<b_dt;
    getch();
}
```

One class type to other class type :

- When we assign an object of a class into the object of another class then it is called as class to class conversion.
- The class to class conversion can be performed either by defining casting operator function in source class or using the constructor in the destination class.
- One class type can be converted to another class type in two ways:

1. using casting operator
2. using constructor

- **//1st way of class to class type using casting operator function**

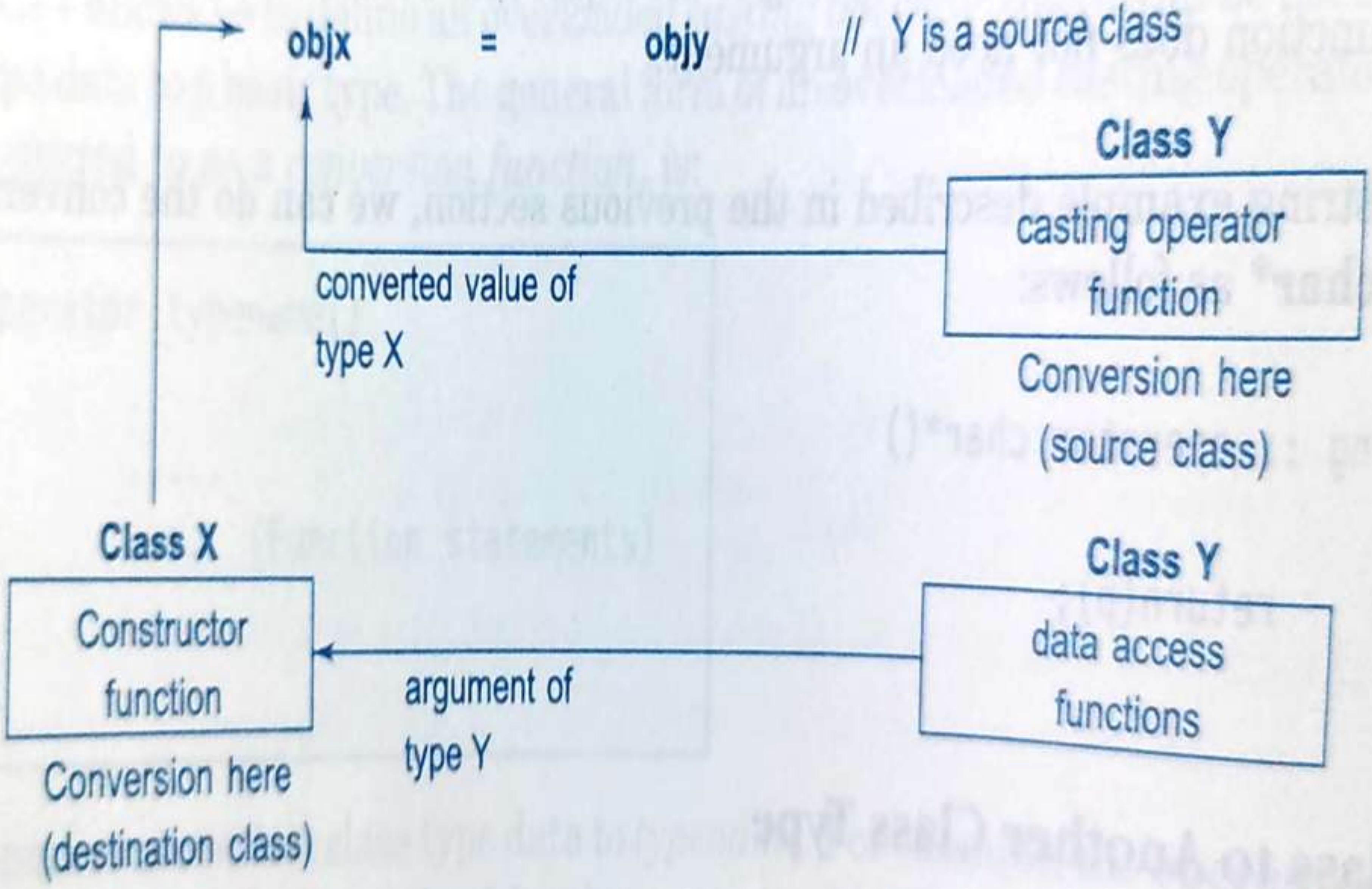
```
class syb
{
    int b;
    public:
    syb(int x)
    {       b=x;        }
    void display()
    {
            cout<<endl<<"Display :"<<b;
    }
};
class sya
{
    int a;
    public:
    sya(int n)
    {
            a=n;
    }
    void show()
    {
            cout<<endl<<"show :"<<a;
    }
    //destination casting operator function
    operator syb()
    {
            syb tmp(a);
            return tmp;
    }
};
void main()
{
    clrscr();
    sya a_obj(101);
    syb b_obj=a_obj;
    //source : sya to destination : syb suing
    casting operator function
    a_obj.show();
    b_obj.display();
    getch();
}
```

- **//2nd way of class to class type using constructor and member function**

```
#include<iostream.h>
#include<conio.h>
class sya
{
    int a;
    public:

    void show()
    {
            cout<<endl<<"show :"<<a;
    }
    void assign()
    {
            return a;
    }
};
class syb
{
    int b;
    public:

    syb(sya o_a)
    {
            b=o_a.assign();
    }
    int display()
    {
            cout<<endl<<"Display :"<<b;
    }
};
void main()
{
    clrscr();
    sya a_obj(101);
    syb b_obj=a_obj;
    //it is same as b_obj(a_obj);//class type to
    class type
    a_obj.show();
    b_obj.display();
    getch();
}
```


objx = objy // Y is a source class
Class Y
casting operator function
Conversion here (source class)
converted value of type X
Class X
Constructor function
Conversion here (destination class)
argument of type Y
Class Y
data access functions

Comparison of Different Method of Conversion :

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;

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;
	
	Casting Operator Function
		The source class' object as argument of the constructor
		class2(class1 c1)
		{ }
		operator destination_class()
		{ }

Introduction of Inheritance :

- Reusability is very important features of OOPs.
- In C++ we can reuse a class and add additional features to it.
- Reusing classes saves time and memory.
- Reusing already tested and debugged class will saves a lot of effort of developing and debugging the same thing again.

What is Inheritance ?

- The capability of a class to derive properties and characteristics from another class is called 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.**
- **derived class** (child) - the class that inherits from another class (derived/child/sub class)
- **base class** (parent) - the existing class being inherited from. (base/parent/super class)
- To inherit from a class, use the : symbol.
- The idea of inheritance implements the 'is-a' relationship.
- The inheritance relationship enables a derived class to inherit features from its base class. Furthermore, the derived class can add new features of its own. Therefore, rather than create completely new classes from scratch, you can take advantage of inheritance and reduce software complexity.

- **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.
- **Syntax :**

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

- **class:** keyword to create a new class
- **derived_class_name**: name of the new class, which will inherit the base class
- **access-specifier**: Specifies the access mode which can be either of private, public or protected. If neither is specified, private is taken as default.
- **base-class-name**: name of the base class, Other class who shares their property to child class.

- 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.
- **Note**: A derived class *doesn't* inherit access to *private* data members.
- **Note :**However, it does inherit a full parent object, which contains any private members which that class declares.
- 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 ABC : private XYZ {…}        // private derivation
class ABC : public XYZ {…}         // public derivation
class ABC : protected XYZ {…}      // protected derivation
class ABC: XYZ {…}                 // private derivation by default
```

• 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.**

Access Modifier	Private	Protected	Public
Within same class	YES	YES	YES
In derived class	NO	YES	YES
In classes other than derived class	NO	NO	YES

- **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.

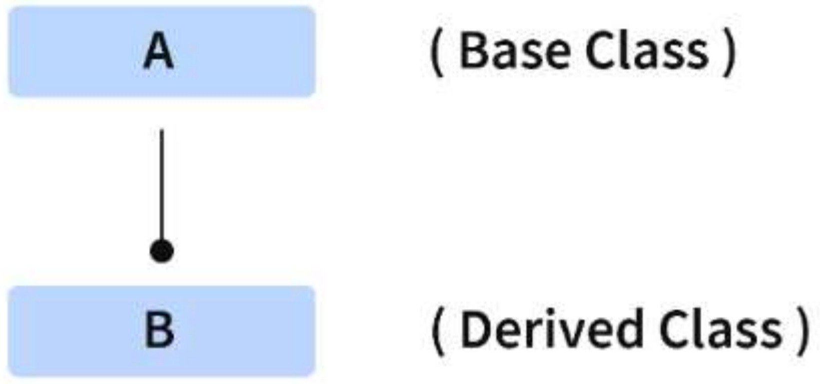
- **Types Of Inheritance :**
- There are total five types of inheritance in C++ as listed below:

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

Single Inheritance :

- In single inheritance, there is only one base class and one derived class.
- **Syntax :**

```
class Base_cls
{
        //Base Class Definition
};
class Derived_cls:Base_cls
{
        //Derived Class Definition
};
```



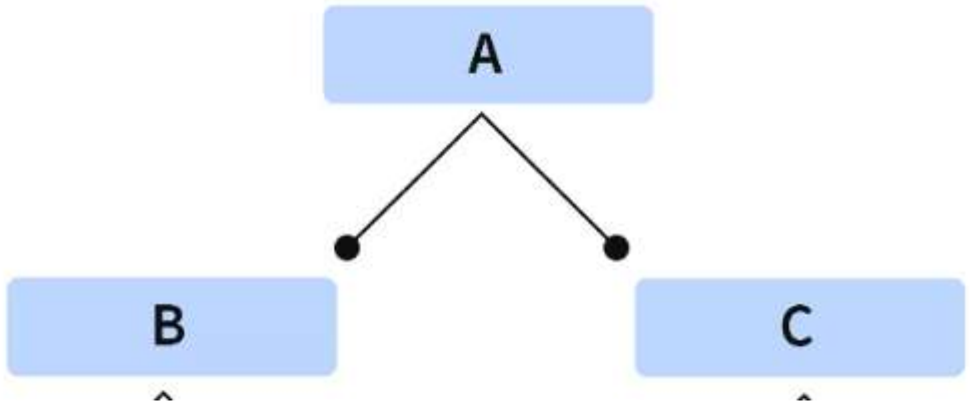
- We have to create a base class and the base class properties are derived by a derived class.
- Note that the derived class can not access the private properties of its base class.

- **Example of single inheritance**

```
class A
{
    int x;
    protected:
    int y;
    public:
    int z;
    void show()
    {
            cout<<"\nX:"<<x<<"\nY:"<<y<<"\nZ:
    "<<z;
    }
    A()
    {
            x=0;//a is private
    }
};
class B:public A
{
    public:
    void assign()
    {
            y=20;//protected
            z=30;//public
            //x is not access here because it is private
    }
};
void main()
{
    clrscr();
    B o_b;
    o_b.assign();
    o_b.show();
    getch();
}
```

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.



- **Syntax :**

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

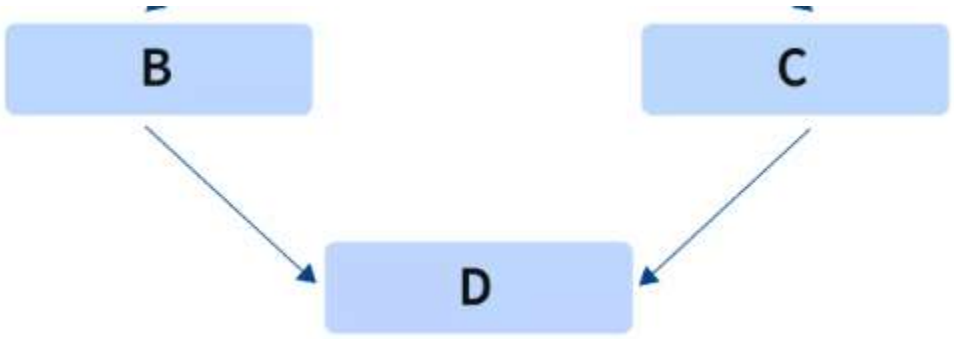
```
//hirarchical inheritance
class A
{
    int x;
    protected:
    int y;
    public:
    int z;
    void pr()
    {
            cout<<"\nX:"<<x;
    }
    A()
    {
            x=0;//a is private
    }
};
class B:public A
{
    public:
    void assign()
    {
            y=37;//protected
            z=35;//public
    //x is not access here bcz it is
    private
}
void show()
{
cout<<"\n\nB Class:\n";
pr();//calling for print private
    member
cout<<"\nY:"<<y<<"\nZ:"<<z;
}
};
class C: public A
{
    public:
void assign()
{
    y=20;//protected of A class
    z=30;//public
    //x is not access here bcz it is
    private
}
void show()
{
    cout<<"\n\nC Class:\n";
    pr();//calling for print
    private member

cout<<"\nY:"<<y<<"\nZ:"<<z;
    }
};
void main()
{
    clrscr();
    B o_b;
    C o_c;
    o_b.assign();
    o_b.show();
    o_c.assign();
    o_c.show();
    getch();
}
```

Multiple Inheritance :

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

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



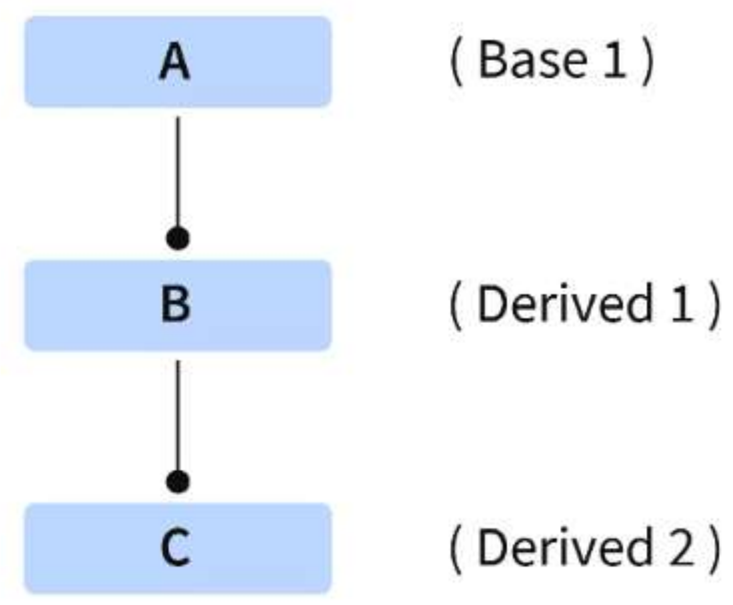
Example of multiple inheritance

```
#include<iostream.h>
#include<conio.h>
class A
{
   protected:
   int x;
   public:
   int y;
};
class B
{
   protected:
   int z;
};
class C: public A,public B
{
   public:
   void show()
   {
   cout<<"\n\nC Class:\n";
   cout<<"\nX:"<<x<<"\nY:"<<y
   <<"\nZ:"<<z;
   }
   void assign()
   {
         x=10;//protected of A class
         y=20;//public of A class
         z=30;//protected of B class
   }
};
void main()
{
   clrscr();
   C o_c;
   o_c.assign();
   o_c.show();
   getch();
}
```

Multi - level Inheritance :

- In multilevel inheritance we can create more levels of inheritance.
- We can derive from a derived class.
- It is a technique that ensures that only one copy of the superclasses or base class member variables is inherited by the second-level derivatives that are grandchild.
- **Syntax :**

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

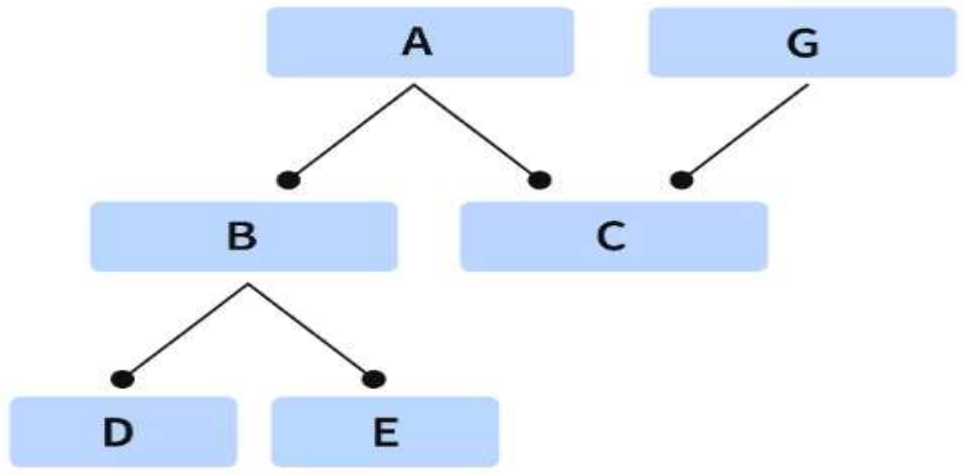


- **Example of multi-level inheritance**

```
class A
{
    int x;
    protected:
    int y;
    public:
    int z;
    void pr()
    {
            cout<<"\nX:"<<x;
    }
    A()
    {
            x=0;//a is private
    }
};
class B:public A
{
    public:
    void show()
    {
            pr();//calling for print private member
            cout<<"\nY:"<<y<<"\nZ:"<<z;
    }
};
class C: public B
{
    public:
    void assign()
    {
            y=20;//protected of A class
            z=30;//public of A class
            //x is not access here bcz it is private
    }
};
void main()
{
    clrscr();
    C o_c;
    o_c.assign();
    o_c.show();
    getch();
}
```

Hybrid Inheritance :

- Hybrid inheritance is combination of two different types of inheritance.
- In hybrid inheritance, we can combine two or more types of inheritance.
- Generally it is combination of multi-level inheritance and hierarchical inheritance.



Syntax :

```
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
};
```

- **Example of hybrid inheritance**

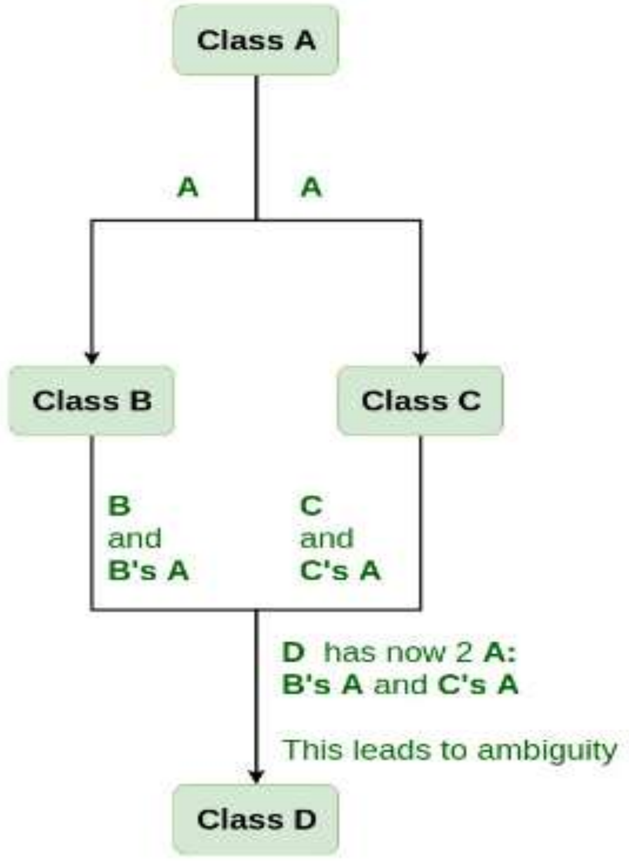
```
class A
{
     protected:
     int x;
     public:
     int y;
};
class B
{
     protected:
     int z;
};
class C:public A
{
     void data()
     {
               x=78;
               y=98;
     }
     void print()
     {
               cout<<"\n\nC Class\n";
               cout<<"\nX:"<<x<<"\nY:"<<y;
     }
};
class D: public A,public B
{
     public:
     void show()
     {
               cout<<"\n\nD Class:\n";
               cout<<"\nX:"<<x<<"\nY:"<<y<<"\nZ:"<<z;
     }
     void assign()
     {
               x=10;//protected of A class
               y=20;//public of A class
               z=30;//protected of B class
     }
};
void main()
{
     clrscr();
     C o_c;
     D o_d;
     o_c.data();
     o_c.print();
     o_d.assign();
     o_d.show();
     getch();
}
```

Virtual Base Class :

- An ambiguity can arise when several paths exist to a class from the same base class. This means that a child class could have duplicate sets of members inherited from a single base class.
- C++ solves this issue by introducing a virtual base class. When a class is made virtual, necessary care is taken so that the duplication is avoided regardless of the number of paths that exist to the child class.
- When two or more objects are derived from a common base class, we can prevent multiple copies of the base class being present in an object derived from those objects by declaring the base class as virtual when it is being inherited.
- Such a base class is known as virtual base class. This can be achieved by preceding the base class name with the word virtual.
- A class can declare as virtual by using virtual keyword.
- The virtual keyword can be used either before or after the visibility modifier.

- **Syntax :**
  class <derived_class_name> : virtual <visibility_mode> <base_class_name>
  or
  class <derived_class_name> : <visibility_mode> virtual <base_class_name>

The ambiguity is arise in this situation : →



- **Example of Virtual base class :**

```
#include<iostream.h>
#include<conio.h>
class A
{
   public:
   int x;
};
class B:virtual public A
{
   public:
   int y;
};
class C:public virtual A
{
   public:
   int z;
};
class D:public B,public C
{
   public:
   void show()
   {

         cout<<"\nX:"<<x<<"\nY:"<<
   y<<"\nZ:"<<z;
   }
};
void main()
{
   clrscr();
   D o_d;
   o_d.x=1;
   o_d.y=2;
   o_d.z=3;
   o_d.show();
   getch();
}
```

- **Note: virtual** can be written before or after the **public**.
- Now only one copy of data/function member will be copied to class **C** and class **B** and class **A** becomes the virtual base class.
- **Need for Virtual Base Class in C++**
- To prevent the error and let the compiler work efficiently, we've to use a virtual base class when multiple inheritances occur.
- It saves space and avoids ambiguity.
- When a class is specified as a virtual base class, it prevents duplication of its data members.
- Only one copy of its data members is shared by all the base classes that use the virtual base class.
- If a virtual base class is not used, all the derived classes will get duplicated data members.
- In this case, the compiler cannot decide which one to execute.

Abstract Class :

- A class which is define only for purpose of base class & not use to create the object is known as 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.
- ***Abstract class is always parent class / base class.***

- **Abstract Class Characteristics :**
- The Abstract class type cannot be instantiated, but pointers and references to it can be generated.
- In addition to normal functions and variables, an abstract class may have a pure virtual function.
- Abstract classes are mostly used for up casting, allowing derived classes to access their interface.
- All pure virtues must be implemented by classes that inherit from an Abstract Class.

Syntax of abstract class:

```
Class classname //abstract class
{
    //data members
    public:
    //pure virtual function
    virtual return_type fun_name()=0;
    /* Other members */
};
```

❖ **What is Pure Virtual Function ?**

- A function which has no body / definition.
- It starts with the virtual keyword and ends with euqal to zero(=0).
- We cannot create a object of that class.
- It must override in child class.
- If we don't override the pure virtual function in derived class, then the derived class also become a abstract class.
- We can't change the signature of virtual function in derived class.

- **Example of abstract class :**

```
#include<iostream.h>
#include<conio.h>
class A
{
   public:
   virtual void show()=0;
//it is a pure virtual function
//do nothing function
};
class B:public A
{
   void show()
{
   cout<<"This is abstract class";
}
};
void main()
{
   clrscr();
   B b;
   b.show();//overied method
   getch();
}
   //A a;
   //we can't create an object of
   abstract class
   //because it has a pure virtual
   function
   //a.show();
```

```
//using pointer variable of class
   A

#include<iostream.h>
#include<conio.h>
class A
{
   public:
   virtual void show()=0;
};
class B:public A
{
   public:
   void show()
   {
         cout<<"abstract class";
   }
};

void main()
{
   clrscr();
   A *p;
   B obj_b;

   p=&obj_b;
   //we can access all the member
   of class A

   p->show();
   getch();
}
```

Constructors in derived class :

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.
- If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke.
- i.e the order of invocation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.
- 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

- class B : public A{ }
  - Here first calling of constructor is A and then B.
- class C:public B, public A
  - Here first calling of constructor is B and then A and then C.
- class C:public B, **abstract** public A
  - Here first calling of constructor is A and then B and then C.
  - Because A is abstract class.

- **Example of parameterized constructors in derived class :** *//multiple inheritance*

```
#include<iostream.h>
#include<conio.h>
class A
{
   public:
   A(int p)
   {
         cout<<"\nClass A :\n"<<p;
   }
};
class B
{
   public:
   B(float p1)
   {
         cout<<"\n\nClass B:\n"<<p1;
   }
};
class C: public A,public B
{
   public:
   C(int x,float y):A(x),B(y)
   {
         cout<<"\n\nClass
   C\n"<<x<<endl<<y;
   }
};
void main()
{
   clrscr();
   C o_c(10,5.5);
   getch();
}
```

Application of Constructor and Destructor in inheritance :

- 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.

Order of Inheritance

$$\begin{tikzcd}
\text{Class C} \arrow[d] & \text{(Base Class 2)} \\
\text{Class B} \arrow[d] & \text{(Base Class 1)} \\
\text{Class A} & \text{(Derived Class)}
\end{tikzcd}$$

Order of Constructor Call

1.	C()	(Class C's Constructor)
2.	B()	(Class B's Constructor)
3.	A()	(Class A's Constructor)

Order of Destructor Call

1.	~A()	(Class A's Destructor)
2.	~B()	(Class B's Destructor)
3.	~C()	(Class C's Destructor)

1. In inheritance, the order of constructors calling is: from *child* class to *parent* class (*child* -> *parent*).
2. In inheritance, the order of constructors execution is: from *parent* class to *child* class (*parent* -> *class*).
3. In inheritance, the order of destructors calling is: from *child* class to *parent* class (*child* -> *parent*).
4. In inheritance, the order of destructors execution is: from *child* class to *parent* class (*child* -> *parent*).

Containership :

- We can create an object of one class into another and that object will be a member of the class.
- This type of relationship between classes is known as containership or has_a relationship as one class contain the object of another class.
- And the class which contains the object and members of another class in this kind of relationship is called a container class.
- The object that is part of another object is called contained object, whereas object that contains another object as its part or attribute is called container object.

Syntax :

```
// Class that is to be contained
class first {
    .
    .
};

// Container class
class second {

    // creating object of first
    first f;
    .
    .
};
```

- **Example :**

```
class first {
public:
   void showf()
   {
         cout << "Hello from first
   class\n";
   }
};

// Container class
class second {

   // Create object of the first-
   class
   first f;
public:
   // Define Constructor
   second()
   {
         // Call function of first-
   class
         f.showf();
   }
};

void main()
{
   // Create object of second
   class
   second s;
}
```

Difference Between Inheritance & Containership :

Inheritance	**Containership**
It enables a class to inherit data and functions from a base class	It enables a class to contain objects of different classes as its data member.
The derived class may override the functionality of the base class.	The container class can't override the functionality of the contained class.
The derived class may add data or functions to the base class.	The container class can't add anything to the contained class.
Inheritance represents a "**is-a**" relationship.	Containership represents a "**has-a**" relationship.

Assignment Questions :

1. Explain operator overloading rules with example.
2. List-out the operators which are not overloaded in normal overloading and also list opt those who are not overloading in friend function.
3. What is inheritance ? Write down name of all inheritance and give definitions of all types of inheritance.
4. Explain multi-level inheritance in detail.
5. Explain abstract class.
6. Explain virtual class.
7. Explain type conversion.
8. What is containership ? Explain with the difference between inheritance and containership.