Module-3

Operator overloading: Operator overloading as member functions and using friend functions. Overloading of binary operators like +, -, *. Creating Prefix and Postfix forms of ++, -- Operators, Operator Overloading Restrictions, revision, Operator Overloading Using a Friend Function to Overload ++ or – –, Overloading ( ).

Inheritance: Base Class, Inheritance & protected members, protected base class inheritance, inheriting multiple base classes, Constructors, Destructors & Inheritance. Passing parameters to base Class Constructors. Granting access, Virtual base classes.

Friend Function and class

Friend Function:

- Definition: A non-member function that can access private members of a class known as friend function of a class.
- Syntax:

```
class class_name{
     friend data_type function_name(argument/s);
};
```

- In the above declaration, the friend function is preceded by the keyword `friend`. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword friend or scope resolution operator
- Example:

```
class Distance {
    private:
        int meter;
    public:
        Distance(): meter(0){ }
        friend int fun(Distance); //friend function
};

int fun(Distance d){
    //function definition
    d.meter=10;  //accessing private data from non-member function
    return d.meter;
}

int main(){
    Distance D;
    cout<<"Distance: "<<fun(D);
    return 0;
}
```

Output:

Distance: 10

Characteristics of a Friend function

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Friend Classes

- Similarly, like a friend function, a class can also be made a friend of another class using keyword friend. For example:
- When a class is made a friend class, all the member functions of that class becomes friend functions.
- General form:

```
class B;
class A
{
      // class B is a friend class of class A
   friend class B;
   ... .. ...
}
class B
{
   ... .. ...
}
```

- Example:

```
class myclass2;
class myclass1
{
    int i;
      public:
          myclass1(){
                    i=25;
           }
     friend myclass2;
};
class myclass2
{
     public:
            void myfun(myclass1 ob){

                      cout<<ob.i; }
     };
```

```
int main()
{
     myclass1 m;
     myclass2 n;
     n.myfun(m);
     return 0;
}
```

Output:

25

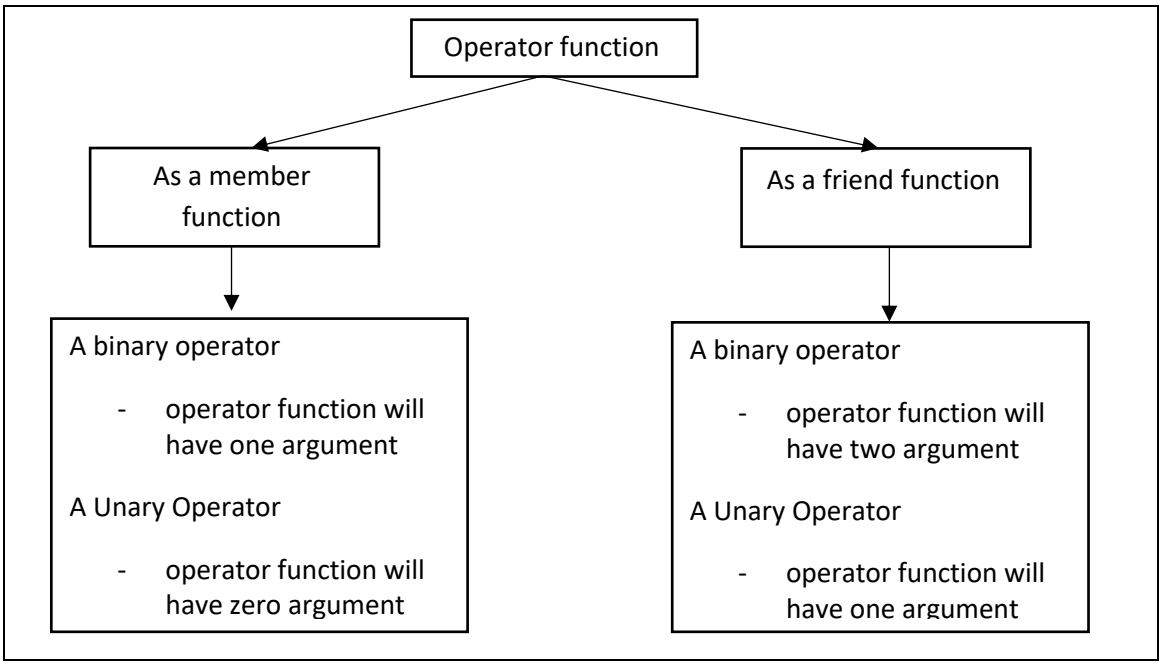
- Here, as `myclass2` is a friend of myclass1, all the private members of `myclass1` can be accessed from the member functions of `myclass2`.
- **Note** that friend classes are seldom used as it is against the basic rule of encapsulation i.e "only member function can access the private data". Hence friend classes are used when it is really essential.

3.1 Operator Overloading

- *Definition:* Making existing operators to operate on user defined data (i.e. class) is known as operator overloading.
- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- For example, if we overload a **+** operator for string concatenation, then, when string operands are provided, the concatenation operation will be performed and if numeric operands are provided, addition of numbers is performed as usual.
- We will overload the operators by creating operator functions. An operator function defines that the overloaded operator will perform relative to the class on which it works. This function is created using the keyword `operator.`

3.1.1 Operator overloading as member functions and using friend functions

- The operator functions may be member of the class or they be can be friend functions of the class possibility of an operator and its possible arguments are better understood in figure 3.1



3.1.1.1 Creating Member Operator Function

The general form of member operator function is:

```
return_type class_name  :: operator op(argument_list)
{
     // body of the function.
}
```

Usually, the return type will be the name of the class only, but it can be of some other type too.

- While overloading a unary operator (like --, ++, etc.) the argument list will be empty, and while overloading a binary operator (like +, - * etc.), there will be one argument.
- While calling an operator function, the first operand is automatically passed to the function through `this` pointer. So, for unary operator it is not necessary to pass the parameter explicitly.
- Hence, there will not be any argument in operator function. But a binary operator has two operands, of which, the first operand is passed through this pointer and the second operand has to be passed as an argument.

3.1.2 Overloading of binary operators like +, -, *

- Binary operators like +, - , * , / and % can be overloaded to extend their operations on the objects of a class. Since these operators requiring two operands, the operator function requires one parameter.

Program to overload + (binary) operator

```
class Complex
{
    private:
      int real;
      int imag;
    public:
       Complex(int r, int i){
             real=r;
             imag=i;
       }

       // Operator overloading
       Complex operator + (Complex c2)
       {
           Complex temp;
           temp.real = this->real + c2.real;
           temp.imag = this->imag + c2.imag;
           return temp;
       }
       void show()
       {
         cout << real << imag << "i";
       }
};
```

```
int main()
{
    Complex c1(5,10), c2(2,3), c3;

    c3 = c1 + c2;
    cout<<"the sum of c1 and c2 is:";
    c3.show();

    return 0;
}
```

```
Output:
The sum of c1 and c2
is: 7 13i
```

- In the above program, when the statement
  c3 = c1 + c2;
- is about to get executed, first there will be function call for **+** operator. The operand at the left side of + i.e. `c1` will be automatically passed to the operator+ function through this pointer. And the operand `c2` is passed as a parameter to the function. Thus, it clearly shows that, in case of binary operators it is that operand which is at the left side of the operator that will invoke the function call.
  That is internally, the statement
  `c3=c1+c2;`
  Is treated by the compiler as-
  `c3=c1.operator + (c2);`

- **Note:** in the above program within the operator functions, we have used this pointer in the statement such as
  `c3.real=this->real+c2.real;`
- But, as this pointer is obvious here, we can use directly as-
  `c3.real=real+c2.real;`

Program to overloading - (binary) operator

```
class Complex
{
    private:
      int real;
      int imag;
    public:
       Complex(int r, int i){
             real=r;
             imag=i;
       }

       // Operator overloading
       Complex operator - (Complex c2)
       {
           Complex temp;
           temp.real = this->real - c2.real;
           temp.imag = this->imag - c2.imag;
           return temp;
       }
       void show()
```

```
        {
          cout << real << imag << "i";
        }
};

int main()
{
    Complex c1(5,10), c2(2,3), c3;

    c3 = c1 - c2;
    cout<<"the difference of c1 and c2 is:";
    c3.show();

    return 0;
}
```

Output:
the difference of c1 and c2 is: 3 7i

Program to overloading * (binary) operator

```
class Complex
{
    private:
      int real;
      int imag;
    public:
       Complex(int r, int i){
             real=r;
             imag=i;
       }

       // Operator overloading
       Complex operator * (Complex c2)
       {
           Complex temp;
           temp.real = this->real * c2.real;
           temp.imag = this->imag * c2.imag;
           return temp;
       }
       void show()
       {
         cout << real << imag << "i";
       }
};

int main()
{
    Complex c1(5,10), c2(2,3), c3;

    c3 = c1 * c2;
    cout<<"the product of c1 and c2 is:";
    c3.show();

    return 0;
}
```

Output:
the product of c1 and c2 is: 10 30i

3.1.3 Creating Prefix and Postfix forms of ++, -- operators

- Unary operators like increment (++) and (--) operators, unary minus (-) and unary (+) operator also can be overloaded. A unary operator requires only one operand and it is passed to the operator function through `this` pointer. So, the member operator for these operators contains *zero* arguments.

Program to overload unary(–) operator

```
class test{
  int x, y;
  public:
      test(int a, int b)
      {
          X=a;
      }
      test operator – () //unary – operator function
      {
         x=-x;
         return *this;
       }
      void display()
      {
         cout<<"x="<<x;
       }
};
int main(){
     test ob(5);
     cout<<"before invert";
     ob.display();
     ob = - ob;
     cout<<"before invert";
     ob.display();
}
```

```
Before invert
X =5
After invert
X=-5
```

Program to Prefix and Postfix of Increment and decrement operator overloading

```
class Check
{
  private:
    int i;
  public:
    Check() { i=0;  }

    // Return type is Check
    Check operator ++() //pre-increment operator
    {
       ++i;
    }
```

```
    Check operator ++(int) //post-increment operator
    {
       ++i;
    }
    Check operator --() //pre-decrement operator
    {
       --i;
    }
    Check operator --(int) //post-decrement operator
    {
       i--;
    }
    void Display()
    { cout << "i = " << i << endl; }
};

int main()
{
    Check obj;
    obj.Display(); //display's 0

    ++obj;              //calls pre-increment operator overloading
    obj.Display(); //display's 1

    obj++;              //calls post-increment operator overloading
    obj.Display(); //display's 2

    --obj;              //calls pre-decrement operator overloading
    obj.Display(); //display's 1

    obj--;             //calls post-decrement operator overloading
    obj.Display(); //display's 0

    return 0;
}
```

Output:
i =0;
i =1;
i =2;
i =1;
i =0;

3.1.4 Overloading Restrictions

- There are some restrictions to overload an operators viz.
  - We cannot alter the precedence of an operator.
  - We cannot change the number of operands that an operator takes.
  - We cannot overload **.** (dot), **::** (scope resolution), (**.***) and (**?**) operators.

3.15 Operator Overloading using friend function

- An Operator overloading functions for a class may be a member function of the class or a friend function for a class.
- As friend function is not a member function, it will not have `this` pointer associated with it. Therefore, for overloading a binary operator we need to specify two parameters for the function corresponding to two operands. And for unary operator, we must have one parameter.
- When overloading a binary operator using a friend function, the left operand is passed into the first parameter and the right operand is passed to the second parameter.

3.15.1 Binary Operator Overloading using friend function

```
class complex{
  public:
   int real, img;
   complex(){
      this->real=0;
      this->img->=0;
    }
    complex(int r, int c)
      {
         this->real=r;
         this->img=I;
      }
    friend complex operator + (complex c1, complex c2);
};
complex operator + (complex &c1, complex &c2)
{
    complex temp;
    temp.real=c1.real + c2.real;
    temp.img=c1.img + c2.img;
    return temp;
}
int main()
{
  complex c1(7,8);
  complex c2(6,5), c3;
  c3= c1 + c2;
  cout<<c3.real<<c3.img;
}
```

Unary operator overloading using friend function (prefix and postfix)

```
#include<iostream>
using namespace std;
class overload{
   int n1,n2;
   public :
   void setdata(int x, int y){
     n1=x;
```

```
      n2=y;
   }
   void show(){
     cout<<n1<<endl<<n2<<endl;
   }
   friend overload operator ++ (overload& ob, int x);//postfix++
   friend overload operator ++ (overload& ob); //prefix --
   friend overload operator -- (overload& ob,int x); //postfix--
   friend overload operator -- (overload& ob); //prefix--
};
overload operator ++ (overload &ob) //prefix operation ++
{
   ++ob.n1;
   ++ob.n2;
}
overload operator ++ (overload &ob, int x) //postfix operation ++
{
   ob.n1++;
   ob.n2++;
}
overload operator -- (overload& ob,int x) //postfix operation --
{
   ob.n1--;
   ob.n2--;
}
overload operator -- (overload& ob) //prefix operation --
{
   --ob.n1;
   --ob.n2;
}
int main(){
  overload ob1;
  ob1.setdata(100, 200);
  ob1.show();

  ob1++;    //call for postfix (++) overload function
  ob1.show();

  ++ob1;     //call for (--) prefix overload function
  ob1.show();

  ob1--;  //call for postfix (--) overload function
  ob1.show();

  --ob1;    //call for (--) prefix overload function
  ob1.show();
}
```

Output

```
100
200

101
201

102
202

101
201

100
200
```

3.2 Inheritance

Definition: The capability of a class to derive the properties and characteristic of another class is known as Inheritance

- Inheritance is one of the OOP mechanism by which the reusability of the code is achieved. Here the properties of one class can be acquired by other class.
- The class being inherited is known as base class and the class that inherits from some other class is known as derived class.
- Along with the properties of base class, a derived class can contain its own properties to show its individuality.
- General form of class inhertitance.

```
class d_class_name : access_specifier b_class_name {
            //body of the derived class
  };
```

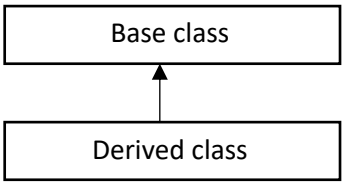
Here, d_class_name is the name of derived class
b_class_name is the name of base class

3.2.1 Types of inheritance

- There are various types of inheritance viz.
  - Single inheritance
  - Multiple inheritance
  - Multi-level inheritance
  - Hierarchical inheritance
  - Hybrid inheritance

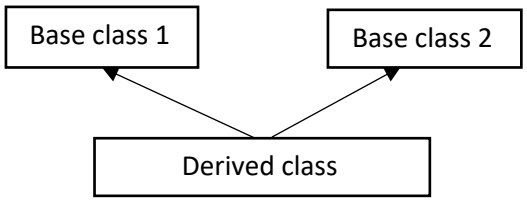
Single Inheritance

- When a class is derived from only one parent class, then it is called as single inheritance.
- The derived class can have properties of its derived class as well as its own properties.



Multiple inheritance

- When a particular class has been derived from more than one base class, then it is known as multiple inheritance.



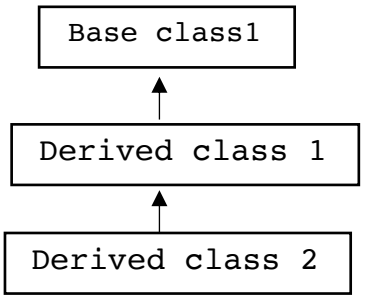
- The general form for deriving a class from more than one class can be given as.

```
class d_class_name :access_specifier b1, access_specifier b2{
            //body of the derived class
  };
```

- It is to be noted that several base classes such as b1, b2 etc. are separated by commas. Each base class can have its own access specifier.

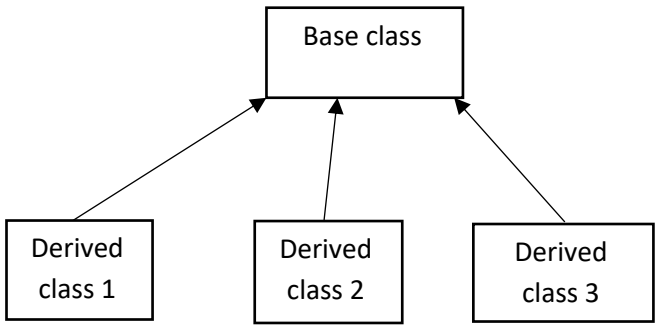
Multi-level inheritance

- A class is derived from one parent class, which is in turn derived from another parent class is known as multilevel inheritance.



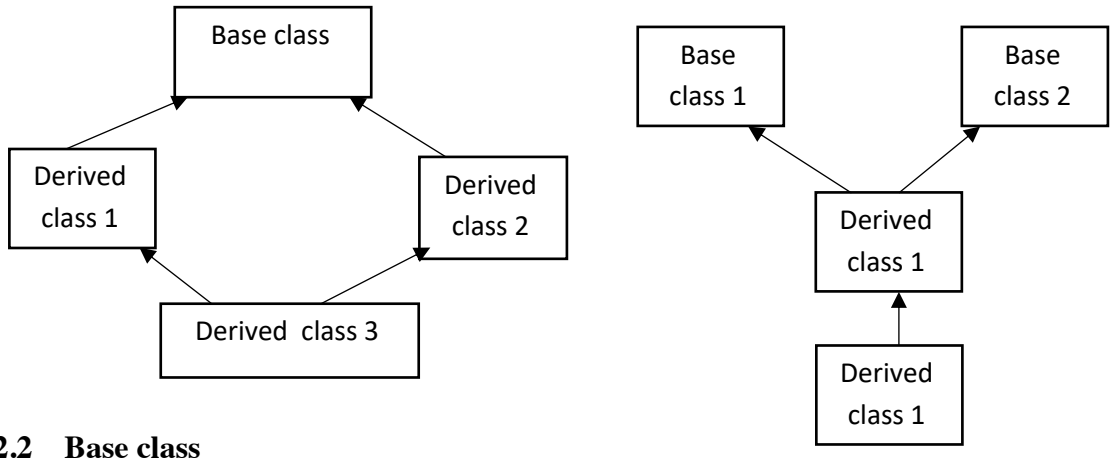
Hierarchical inheritance

- A class may become a parent for more than on class. Such a situation is known as hierarchical inheritance.
- Each of derived classes may have properties of base class and their own individual properties. Hence, every such derived class is different from its siblings.



Hybrid inheritance

- The combination of above types of inheritance is known as hybrid inheritance. For example, assume that there are two classes b1 and b2. Now a class d1 is derived from both b1 and b2 by applying multiple inheritance. Then a class d2 is derived from the class d1.



3.2.2 Base class

- The access specifier used for the base class in the declaration of derived class will take control over the accessibility of members of base class inside the derived class.
- The members of a base class can be *private, protected, or public*. Similarly, the access specifier that can be provided for a base class while deriving a child class also can be any one of these keywords. Thus there will be nine possibilities are summarized in the table.

Table 3.1 Behavior of base class members in derived class

Base class members	**Inheritance access specifier for base class**		
	Public	**Private**	**Protected**
Public	Retained as **Public** in derived class	Converted as **Private** in derived class	Converted as protected in **derived** class
Private	Not accessible in derived class	Not accessible in derived class	Not accessible in derived class
Protected	Retained as **protected** in derived class	Converted as **Private** in derived class	Retained as **protected** in derived class

- It is observed from the table that if the base class member is private then they are not available inside the derived class, irrespective of the type of inheritance specifier.
- **Note**: If no access specifier is provided for base class during inheritance, by default, it is private.

```
class base
{
      public:
            int x;
      protected:
            int y;
      private:
            int z;
};
```

```
class publicDerived: public base
{
      // x is public
      // y is protected
      // z is not accessible from publicDerived
};

class protectedDerived: protected base
{
      // x is protected
      // y is protected
      // z is not accessible from protectedDerived
};


class privateDerived: private base
{
      // x is private
      // y is private
      // z is not accessible from privateDerived
}
```

- base has three member variables: x, y and z which are public, protected and private member respectively.
- `publicDerived` inherits variables x and y as public and protected. z is not inherited as it is a private member variable of base.
- `protectedDerived` inherits variables x and y. Both variables become protected. z is not inherited
- If we derive a class derivedFromProtectedDerived from `protectedDerived`, variables x and y are also inherited to the derived class.
- `privateDerived` inherits variables x and y. Both variables become private. z is not inherited
  If we derive a class derivedFromPrivateDerived from `privateDerived`, variables x and y are not inherited because they are private variables of privateDerived.

3.2.3 Inheritance & Protected Members

- When a single class exists in a program, the keywords private, public and protected behave the same.
- That is, both private and protected members of a class are not accessible outside the class. Whereas, during inheritance, these keywords do behave differently. It is observed from the above table 3.1 that protected members of a base class may be available to derived class, but private members are not at all available.
- The usage of protected members ensures security as well as code reusability.
- Protected members are secured from unwanted manipulation by non-class members and at same time, they are made available to child classes to maintain generalization-specialization nature of inheritance.

```
class B{
      protected:
            int I,j;
      public:
            void setb(int a, int b){
                  I=a;
                  J=b;
            }
      void dispalyb(){
            Cout<<i<<"\t"<<j<<endl;
      }
};
class D1 : public B{
            int k;
      public:
            void setd(){
                  k= i * j;
            }
      void dispalyd(){
            cout<<k<<endl;}};
      }
};
class D2 :public D1
{
      int m,n;
      public:
            void setd2(){
                  m = i+j;
            }
            void dispalyd2(){
                  cout<<m<<endl;
            }
};

int main(){
      D1 ob1;
      D2 ob2;
      Ob1.set(5,6);  //display 5 ,6
      Ob1.displayb();
      Ob1.setd1();
      Ob1.displayd1();    //display 30
      Ob2.setb(3,4);
      Ob2.setd1();
      Ob2.setd2();
      Ob2.displayd2();     //display 7
}
```

- In the program the variables `i` and `j` are protected members of `B`. Here, we have multi-level inheritance. The class `D1` is publicly derived from `B`, and in turn, `D2` is publicly derived from `D1`. Hence, the protected members of topmost base class are available to both `D1` and `D2`.
- Note that, the availability of protected members to the subsequent derived classes can go up to any number of levels, if proper inheritance-access-specifier is maintained. That is, if the class derived as either public or protected in every level, then base class

protected members are available till the last level. But if at any level, the class is derived as private, then the protected members of base class will immediately be converted as private and hence will not be available for successive levels of inheritance.(refer 3rd row of table 3.1)

3.3.4 Protected Base class Inheritance

- When inheriting a class from some base class, we can also use the use the access specifier as protected. By doing this, all the public and protected members of the base class will become protected members of the derived class (refer table 3.1).They cannot be accessed outside the derived class. For example.
- In the below program the class D is derived from B as protected. Hence, the protected members of B will remain as protected in D, whereas, the public members (`setb( )` and `display()` functions) of B are converted as protected in D. Hence, inside main() function, an attempt made to invoke `setb( )` and `displayb( )` functions generates error.

```
class B{
    int i;

    protected:
        int j,k;

    public:
    void setb(int a, int b){
        j=a;
        k=b;
    }

    void displayb(){
        cout<<j<<k;
    }
};
class D :protected B{
    int p;

    public:
    void setd(){
        setb(10,7);
        p=j*k;

    }

    void displayd(){
        cout<<p;
        displayb();
    }
};
int main()
{
    D ob;
```

Output:

70 10 7

```
   //ob.setb(3,4); // results in error
     ob.setd();

   // ob.displayb(); // results in error
      ob.displayd();
    return 0;
}
```

3.3.5 Inheriting Multiple base classes

- Making use of the code written in more than one class is known as multiple inheritance. We may have more than one independent class written for different purposes. Now, if we want a class which needs to acquire properties of many independent classes, then we can use multiple inheritance.

```
class A {
     public:
          int a = 5;
     A() {
   cout<< "Constructor for class A; }
};

class B {
    public:
           int b = 10;
    B(){
     cout <<"Constructor for class B"; }
  };

class C: public A, public B {
        public:
               int c = 20;
         C(){
           cout <<"Constructor for class C";
           cout<<"Class C inherits from class A and class B";
}
};

int main() {
     C obj;
     cout<<"a = "<< obj.a <<"b = "<< obj.b <<endl<<"c = "<<
obj.c <<endl;
return 0;
 }
```

Output:

Constructor for class A
Constructor for class B
Constructor for class C
Class C inherits from class A and class B
a = 5
b = 10
c = 20

3.3.6 Constructors, Destructors and inheritance

- Constructors are generally used for initialization of members during object creation. Whereas, the destructors are used for de-allocation of resources held by the object when object is about to die.

Execution of constructor and Destructor

- When an object of a derived class is created, the constructor of all the classes starting from the *topmost* base class will be invoked. Similarly, when the object is about to go out of the scope, the destructor are called in the reverse order of inheritance hierarchy.

```
class B
{
  public:
   B( ){
    cout<<"Constructing B";
   }
   ~B( ){
    cout<<"Destructing B";
   }
};
class D1 : public B
{
  public:
   D1( ){
     cout<<"Constructing D1";
   }
   ~D1( ){
     cout<<"Destructing D1";
   }
};
class D2 : public D1
{
  public:
   D2( ){
     cout<<"Constructing D";
   }
   ~D2( ){
     cout<<"Destructing D2";
   }
};
int main()
{
  D2 ob;
  cout<<"hello";
}
Output:
Constructing B
Constructing D1
Constructing D2
Hello
Destructing B
```

```
Destructing D1
Destructing D2
```

- The base class will not have any knowledge about the derived class. Any initialization or action to be taken in the derived class is not known to base class. So, the base class constructor is called first so that all the initializations of base class members are done before. Then the derived class constructor is called.
- Similarly, while destructing, derived class destructor is called destructor is called first release the resource of derived class, and later, base class destructor is called.
- The same rule applies for multiple inheritance also. That is, the constructor gets called in the order of their occurrence. For example, if we have derived class such as.

```
class D : public b1, public b2{
        //….
}
```

- then, the constructor of b1 is called first, then the constructor of b2 and finally the constructor of d gets called for the object of the class d. Suppose the order of b1 and b2 changes in the above declaration, then the order of execution of their constructor also changes.

3.3.7 Passing parameters to base Class Constructors.

- Base class or derived class constructors may have arguments. These arguments may need to be initialized during object creation.
- If a derived class requires initialization, then its constructors is called just like any normal parametrized constructor.
- However, if the base class constructor with arguments need to be called, a different syntax to be used. In this case, the base class parameters are passed through the derived class constructors only.
- The general form :

```
D_constructor(arg_list) : b1 (arg_list), b2(arg_list),…
{
      body of constructor
}
```

Example:

```
class B1
{
  protected:
     int x;
  public:
   B1(int x ){
    a=x;
    cout<<“Constructing B1”;
```

```
    }
    ~B1( ){
     cout<<"Destructing B1";
    }
};

class B2
{
  protected:
     int k;
  public:
   B2(int x ){
      k=x;
     cout<<"Constructing D1";
    }
    ~B2( ){
      cout<<"Destructing B2"; }
};

class D : public B1,public B2
{
  int j;
  public:
   D2( int x, int y, int z): B1(y), B2(z){
        j=x;
     cout<<"Constructing D";
    }
    ~D2( ){
      cout<<"Destructing D2";
    }
    void display()
     {  cout<<a<<j<<k; }
 };

int main()
{
  D ob(10,15,20);
  ob.display();
}
```

```
output
Constructing B1
Constructing B2
Constructing D
10   15    20
Destructing D
Destructing B2
Destructing B1
```

3.3.8 Granting access

- It has already been discussed that when a base class is inherited as private, all public and protected members of that class become private members of the derived class.
- In some situations, some members of the base class may need to be in their original form. That is why we may require that the access specifier of some members in the derived class must be same as that in base class.
- That is possible by giving access declaration to such members in the derived class the general form:

```
base_class_name : : member name;
```

Example:

```
class B {
  int i;
   public:
     int j, k;
     void set(int x){
         i=x;
      }
    int get()
     { return i;  }
};
class D : private B
{
  public:
    B :: j; //make j public again
    B :: set; //make set public
    B :: get; //make get public
    int a;

    void display(){
        cout<<a<<j;
     }
};
```

```
int main(){

  D ob;
  ob.j=20;
  ob.a=33;


     ob.set(23);
      ob.display();
      cout<<ob.get();

}

Output:

      33     20   23
```

- In this program, I is private member of class B and j, k set( ), get( ) are the public members of class B. When a derived class is created using this base class with keyword private, then all the public members of the base will become private inside the derived class. The members j, k, set() and get() cannot be accessed outside the derived class. But, using the scope resolution operator, the public members of B are re-declared inside D to retain their original status and are accessed in main().
- Note: access declaration can be used to restore the status of public and protected members only.

3.3.9 Virtual base classes

- Suppose we are dealing with multiple inheritance (rather hybrid inheritance), i.e. more than one class is inherited from the same base class. For e.g. Consider the situation shown in the below figure 3.2. The classes `B` and `C` are derived from base class `A`. Now each of this `B` and `C` would have inherited some of the members of `A`. The class `D` is derived from both `B` and `C`.
- To overcome this problem, the keyword virtual is used while deriving a class from the base class.
- Below program shows how to use virtual keyword.

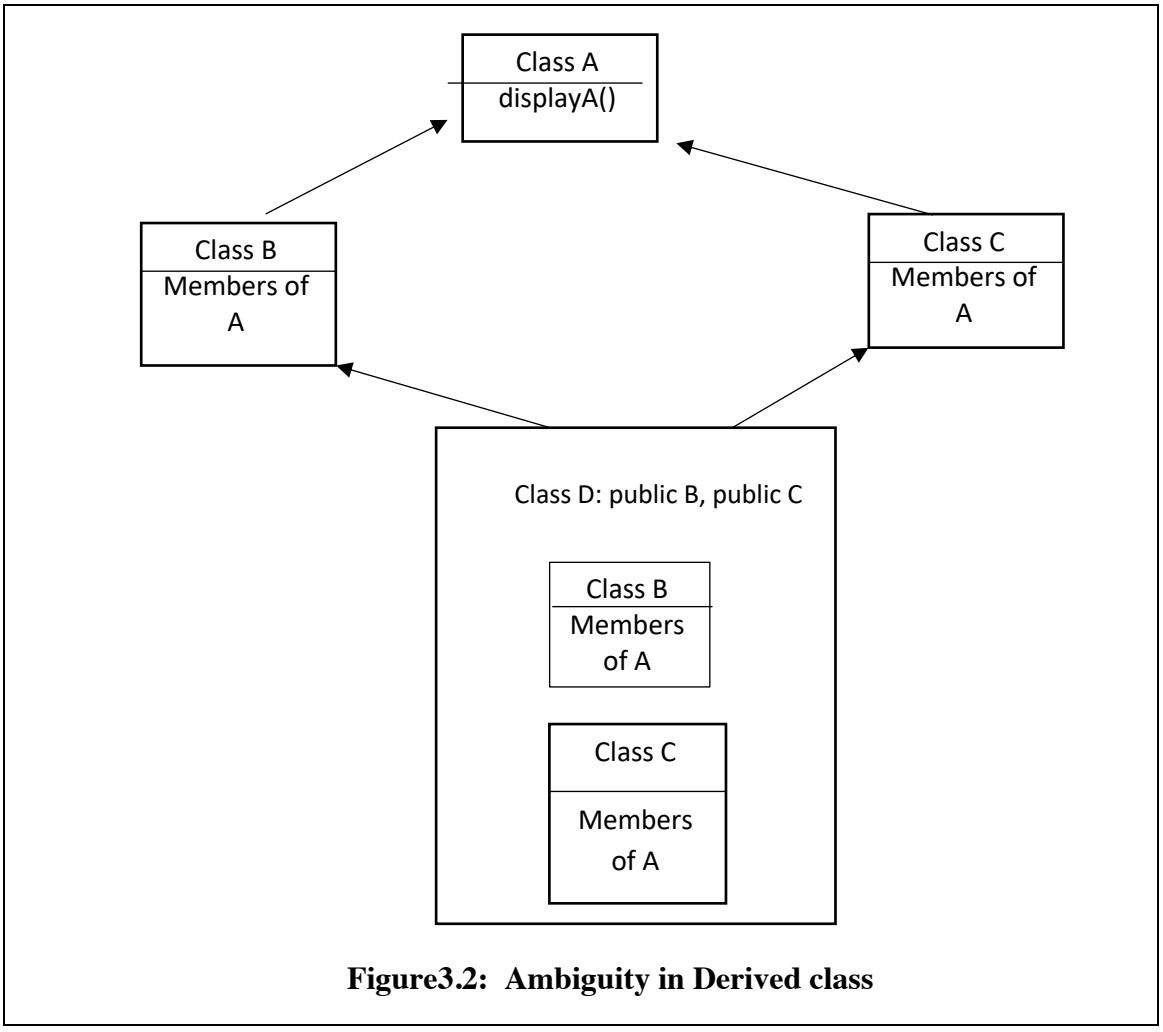


Figure3.2: Ambiguity in Derived class

Program to demonstrate usage of virtual keyword

```
class A {
   public:
   void display(){
       cout<<"In base class A";
   }
};

class B : virtual public A {

};

class C : virtual public A {

};

class D : public B, public C {

};
```

```
Output:
In base class A
```

```
int main() {
   D obj;
   obj.display();
   return 0;
}
```

- The keyword *virtual* used in declarations of B and C force them to share single copy base class so that the ambiguity is eliminated. Now the class D contains only one copy of ***display().*** When we refer the method ***display()*** through the object obj of the class D, without any doubt the compiler allows us to do so. Here the classes B and C are known as virtual base classes of the class D.

Note:

1. Without using the virtual base classes, it is possible to resolve the ambiguity during multiple inheritance. For example, in the above program, instead of declaring both B and C as virtual, in the main() function, we can use the following statement.

```
Ob.B :: display();
      Or
Obj.C : : display();
```