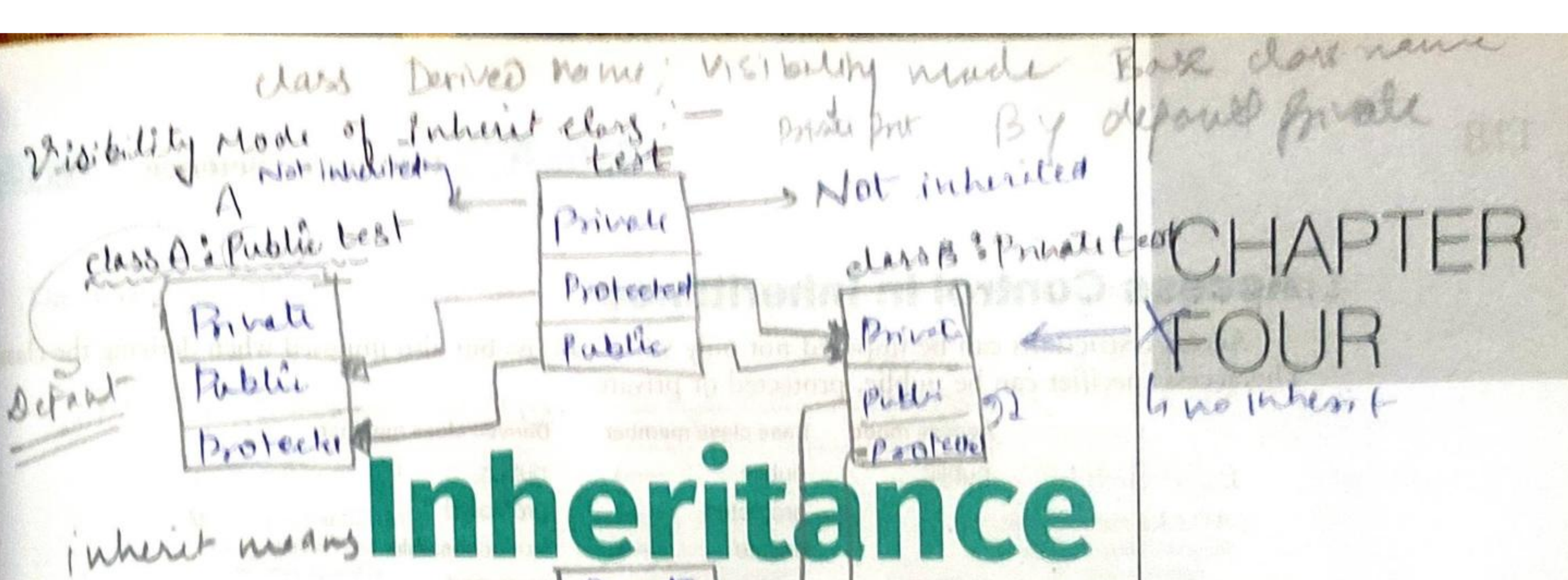
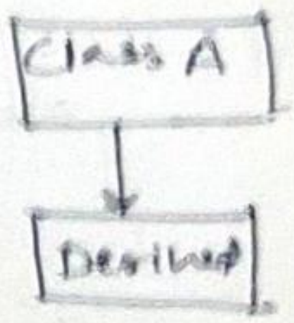
CHAPTER FOUR

Inheritance



Inheritance is a feature by which new classes are derived from the existing classes and can add embellishments and refinements of its own.

The original existing class is said to be the **base (or super or parent) class** and the newly made class, i.e. derived from the original existing class is called a **derived (or sub or child) class**. A class that is derived from base class will possess all the properties and behaviours of the base class apart from that, it also has its own unique properties. However, the derived class will not inherit constructors, destructors and friend functions and derived classes cannot remove any data members and member functions present in the base class.



Chapter Checklist

- Defining Derived Class
- Access Control in Inheritance
- Types of Inheritance
  - *Single inheritance*
  - *Multiple inheritance*
  - *Multilevel inheritance*
  - *Hierarchical inheritance*
  - *Hybrid inheritance*
- Constructors and Destructors in Derived Classes
- Parameterized Constructors in Derived Classes

Advantages of Inheritance

Advantages of inheritance are given below:

(i) **Reusability** Inheritance helps the code to be reused in many situations. The base class is defined and once it is compiled, it needs not to be reworked. Using the concept of inheritance, the programmer can create as many derived classes from the base class as needed, while adding specific features to each derived class as needed.

(ii) **Saves Time and Effort** The above concept of reusability, achieved by inheritance, saves the programmer's time and effort since, the main code written can be reused in various situations as needed.

(iii) **Data Hiding** Base class can decide to keep some data private so that it cannot be altered by the derived class.

(iv) **Transitivity** If a class *B* inherits properties of another class *A*, then all subclasses of class *B* will automatically inherit the properties of *A*. This is called transitive property.

Defining Derived Class

Whenever a derived class is defined, we have to specify its relationship with the base class. The general form of specifying a derived class is:

```
class derived_class : access_mode base_class1,access_mode base_class2,..
{
    ..............
    ..............    //Members of derived class
    ..............
};
```

where, derived_class is the name of the derived class and base_class1, base_class2, .. are the names of the base classes. The colon symbol (:) shows the relationship of a derived class to its base class. The access mode is a visibility mode that may be private, protected or public, which shows the accessibility of class. The default access mode is private.

Access Control in Inheritance

Access restrictions can be imposed not only within a class but also imposed when deriving the class. The access specifier can be public, protected or private.

Access mode	Base class member		Derived class member
public	public	⟶	public
	protected	⟶	protected
	private	⟶	Not accessible
protected	public	⟶	protected
	protected	⟶	protected
	private	⟶	Not accessible
private	public	⟶	private
	protected	⟶	private
	private	⟶	Not accessible

The public Access Specifier

In a public access specifier, all the private members of a base class remain private in the derived class, the protected members remain protected and the public members remain public.

The protected Access Specifier

In a protected access specifier, all the private members of a base class remain private in the derived class, the protected members remain protected, but all the public members of the base class become protected.

The private Access Specifier

In a private access specifier, all the private members of a base class remain private in the derived class and all the public and protected members in the base class become private. It depicts a composition relationship between the base and the derived classes, which means that the derived class contains only one instance of the base class. To illustrate the access control, declarations of a DERIVED class from a BASE class are shown below. In each of the three cases, the access specifier is different.

Access Specifiers in Inheritance

Case 1

```
class BASE
{
   private:
      int a,b;
   protected:
      int c,d;
   public:
      int e;
      int Fn();
};
class DERIVED
   :public BASE
{
   private:
      int p,q;
   protected:
      int r,s;
   public:
      int t;
      int D();
};
```

Case 2

```
class BASE
{
   private:
      int a,b;
   protected:
      int c,d;
   public:
      int e;
      int Fn();
};
class DERIVED
   : protected BASE
{
   private:
      int p,q;
   protected:
      int r,s;
   public:
      int t;
      int D();
};
```

Case 3

```
class BASE
{
   private:
      int a,b;
   protected:
      int c,d;
   public:
      int e;
      int Fn();
};
class DERIVED
   : private BASE
{
   private:
      int p,q;
   protected:
      int r,s;
   public:
      int t;
      int D();
};
```

The visual representation of the members of the DERIVED class is shown below:

	Private members	Protected members	Public members
Members of the DERIVED class in case1-public access specifier	p, q	c, d, r, s	e, Fn(), t, D()
Members of the DERIVED class in case2-protected access specifier	p, q	c, d, e, r, s, Fn()	t, D()
Members of the DERIVED class in case3-private access specifier	c, d, p q, e, Fn()	r, s	t, D()

In the code shown previously, e is a public member variable of the BASE class. As you can see from the visual representation of members, the member variable e has public access in the DERIVED class, if BASE class inherits publicly. It has a protected access in the DERIVED class if BASE class inherits protectedly. It has a private access in the DERIVED class if BASE class inherits privately.

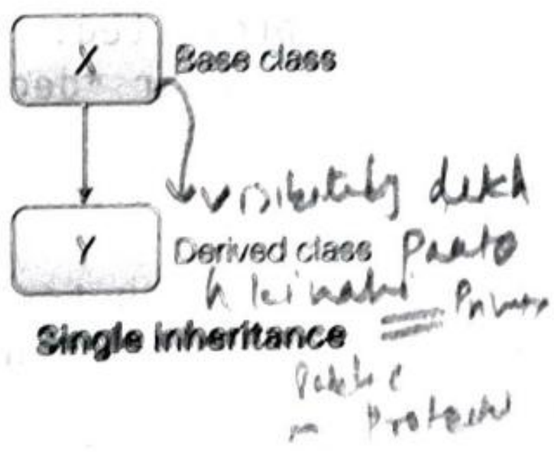
Note *Only public members of a class are accessibly by the object of that class. Private and protected members are not accessible.*

Types of Inheritance

There are five types of inheritance:

1. Single Inheritance

When a class is derived from a single base class, it is said to be single inheritance. It is also called single level inheritance. It can be thought of as a specialization of an existing class. The single inheritance diagram is shown in the figure. Here, class Y inherits the properties and behaviours of class X.



Single inheritance

Syntax

```
class derived_class : accessmode base_class
{
  //body of the derived class
};
```

Program 1. *To illustrate the use of single inheritance in which derived class (number2) inherits the base class (number1) in public mode.*

```
#include<iostream.h>
#include<conio.h>
class number1
{
  int a;
  public:
    void get_a()
    {
      cout<<"Enter the value for a : ";
      cin>>a;
    }
    void show_a()
    {
      cout<<"a is "<<a;
    }
};
class number2 : public number1
{
  private:
    int b;
  public:
    void get_b()
    {
      cout<<"Enter the value for b : ";
      cin>>b;
    }
    void show_b()
    {
      cout<<"\nb is "<<b;
    }
};
void main()
{
  number2 obj;
  obj.get_a();
  obj.get_b();
  obj.show_a();
  obj.show_b();
  getch();
}
```

Output

```
Enter the value for a : 8
Enter the value for b : 9
a is 8
b is 9
```

Here, the inheritance is made in the public access mode. In the main program, we create the object for derived class and access all public functions present in the base class.

The derived class number2 will have the following members:

From class number2

Public Functions	Private Variable
void get_b( ) void show_b( )	int b

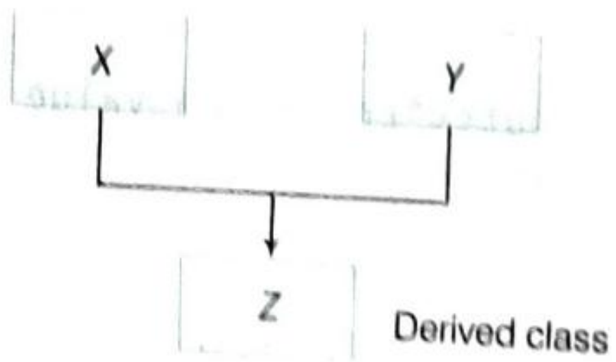
From class number1

Public Functions	Private Variable
void get_a( ) void show_a( )	None

2. Multiple Inheritance

C++ allows a class to get derived from multiple base classes. A derived class can directly inherit more than one base classes. In this situation, two or more base classes are inherited to create the derived class.

The multiple inheritance diagram is shown below:



Multiple inheritance

Here, the class Z inherits the properties and behaviours of both classes X and Y.

Syntax

```
class derived_class : accessmode base1,
accessmode base2,…, accessmode basen
{
   //body of the derived class
};
```

Program 2. ***To show the multiple inheritance.***

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class Expdet
{
   protected:
      char *name;
      int tot_exp;
   public:
      void expr()
      {
         cout<<"Enter your Name"<<endl;
         cin>>name;
         cout<<"Enter your Total Experience";
         cout<<endl;
         cin>>tot_exp;
      }
};
class Saldet
{
   protected:
      int salary;
   public:
      void Salary()
      {
         cout<<"Enter your Salary"<<endl;
         cin>>salary;
      }
};
class Edudet
{
   protected:
      char *degree;
   public:
      void eduqua()
      {
         cout<<"Enter your Degree"<<endl;
         cin>>degree;
      }
};
class Promotion : public Expdet,public
Saldet,public Edudet
{
   public:
      void promote()
      {
         if(tot_exp>10 && salary>=20000 &&
                  (strcmp(degree,"PG")==0))
            cout<<"PROMOTED FOR HIGHER GRADE";
         else
            cout<<"CANNOT BE PROMOTED";
      }
};
void main()
{
      Promotion obj;
      obj.expr();
      obj.Salary();
      obj.eduqua();
      obj.promote();
      getch();
}
```

Output

```
Enter your Name
John
Enter your Total Experience
25
Enter your Salary
23000
Enter your Degree
PG
PROMOTED FOR HIGHER GRADE
```

Here, the class Promotion is inherited from the class Expdet, class Saldet and class Edudet. Thus, the derived class Promotion will have the following members:

From class Expdet

Public Function	Protected Variable
void expr( )	name, total_exp

From class Saldet

Public Function	Protected Variable
void Salary( )	salary

From class Edudet

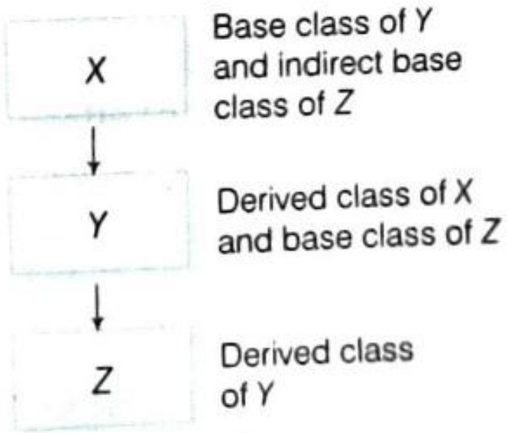
Public Function	Protected Variable
void eduqua( )	degree

Note *A base class cannot appear in the base list of a derived class more than once:*

```
class Base
{
   :
   :
};
class Derived : public Base, private
   Base//Error:repetition not allowed.
{
   :
   :
};
```

3. Multilevel Inheritance

In multilevel inheritance, a derived class can be used as a base class for another derived class, creating a multilevel class hierarchy. In this case, the original base class is said to be an indirect base class of the second derived class. The multilevel inheritance diagram is shown in the figure.



Multilevel inheritance

The transitive nature of inheritance is reflected by this type of inheritance. If a class *Y* inherits properties of class *X* then all the derived classes of *Y* will automatically inherit the properties of *X*, hence indirectly the class *X* is the base class of *Z* also. We can say that, class *Y* is a direct base class of class *Z*, class *X* is a direct base class of class *Y* and class *X* is an indirect base class of class *Z*.

Syntax

```
class derived1 : accessmode baseclass
{
   //body of the derived1 class
};
class derived2 : accessmode derived1
{
   //body of the derived2 class
};
:
:
class derivedn : accessmode derivedn-1
{
   //body of the derivedn class
};
```

Thus, this scheme can be extended to any *level* (i.e. generalisation to specialisation).

Program 3. *To show the multilevel inheritance.*

```
#include<iostream.h>
#include<conio.h>
class person
{
   protected:
      int age;
      char *name;
   public:
      void get1();
};
class emp : public person
{
   protected:
      int basic,hra;
   public:
      void get2();
};
class manager : public emp
{
   protected:
      int deptcode;
   public:
      void get3();
      void display();
};
void person :: get1()
{
   cout<<"Enter your Age\n";
   cin>>age;
   cout<<"Enter your Name\n";
   cin>>name;
}
```

```
void emp :: get2()
{
   cout<<"Enter your Basic and  HRA\n";
   cin>>basic>>hra;
}
void manager :: get3()
{
   cout<<"Enter your DeptCode\n";
   cin>>deptcode;
}
void manager :: display()
{
   cout<<"Name is "<<name;
   cout<<"\nAge is "<<age;
   cout<<"\nBasic and HRA "<<basic<<""<<hra;
   cout<<"\nDeptCode is "<<deptcode;
}
void main()
{
   manager obj;
   obj.get1();
   obj.get2();
   obj.get3();
   obj.display();
   getch();
}
```

Output

```
Enter your Age
25
Enter your Name
Harish
Enter your Basic and HRA
25000 5000
Enter your DeptCode
1001
Name is Harish
Age is 25
Basic and HRA 25000 5000
DeptCode is 1001
```

The class manager is inherited from the class emp which, emp class is also inherited from class person. Thus, the class emp and manager will possess the following features:

From class emp

Public Functions	Protected Variables
void get1() (from person class) void get2()	name, age (from person class), basic, hra

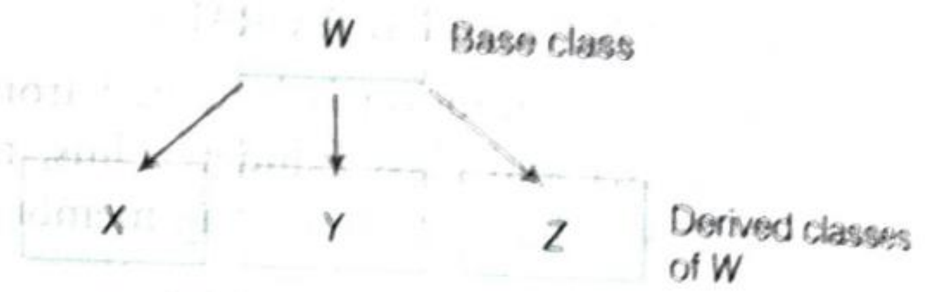
From class manager

Public Functions	Protected Variables
void get1() (from person class) void get2() (from emp class) void get3() void display()	name, age (from person class), basic,hra (from emp class) deptcode

4. Hierarchical Inheritance

When many derived classes inherited from a single base class, it is known as hierarchical inheritance. This type of inheritance is helpful in class hierarchies. In this inheritance, the base class will include the features that are common to all the sub classes.

The hierarchical inheritance diagram is given below:



Hierarchical inheritance

Here, class *X*, *Y* and *Z* inherits the properties and behaviours of class *W*.

Syntax

```
class derived1 : accessmode baseclass
{
   //body of the derived1 class
};
class derived2 : accessmode baseclass
{
   //body of the derived2 class
};
:
:
class derivedn : accessmode baseclass
{
   //body of the derivedn class
};
```

In hierarchical inheritance, it is necessary to create objects for all the derived classes at the lower level because each derived class will have its own unique feature.

Program 4. *To show the hierarchical inheritance:*

```
#include<iostream.h>
#include<conio.h>
class A
{
   protected:
      int x,y;
   public:
      void get()
      {
         cout<<"\nEnter two values\n";
         cin>>x>>y;
      }
};
```

```
class B : public A
{
   private:
      int m;
   public:
      void add()
      {
         m=x+y;
         cout<<"The Sum is "<<m;
      }
};
class C : public A
{
   private:
      int n;
   public:
      void mul()
      {
         n=x*y;
         cout<<"The Product is "<<n;
      }
};
class D : public A
{
   private:
      float l;
   public:
      void division()
      {
         l=x/y;
         cout<<"The Quotient is "<<l;
      }
};
void main()
{
   B obj1;
   C obj2;
   D obj3;
   obj1.get();
   obj1.add();
   obj2.get();
   obj2.mul();
   obj3.get();
   obj3.division();
   getch();
}
```

Output

```
Enter two values
12 6
The Sum is 18
Enter two values
12 6
The Product is 72
Enter two values
12 6
The Quotient is 2
```

Here, the class *B*, class *C* and class *D* are inherited from the class *A*. In the main program, the objects were created for all the subclasses to access their member functions. Thus, the class *B*, *C* and *D* will possess the following features:

From class B

Public Functions	Protected Variables	Private Variable
void get() (from class A) void add()	x, y (from class A)	m

From class C

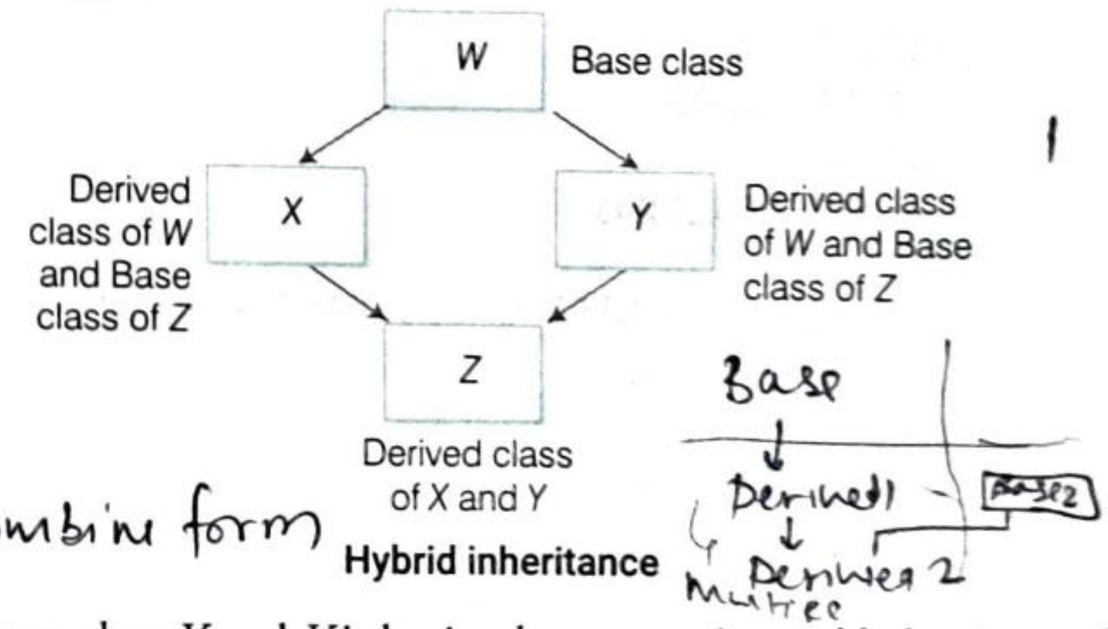
Public Functions	Protected Variables	Private Variable
void get() (from class A) void mul()	x, y (from class A)	n

From class D

Public Functions	Protected Variables	Private Variable
void get() (from class A) void division()	x, y (from class A)	l

5. Hybrid Inheritance

A hybrid inheritance is the combination of all the inheritance discussed earlier. The hybrid inheritance diagram is shown below:



Here, class *X* and *Y* inherits the properties and behaviours of class *W* and class *Z* inherits the properties and behaviours of class *W*, *X* and *Y*.

Syntax

```
class derived1 : accessmode baseclass
{
   //body of the derived1 class
};
:
class derivedn : accessmode baseclass
{
   //body of the derivedn class
};
```

```
class derivedn+1 : accessmode derived1,....,
accessmode derivedn
 {
    //body of the derivedn+1 class
 };
```

Program 5. *To calculate net pay for an employee, apart from the normal pay, we may also need to get additional pay by working overtime, thus the net pay will be the sum of the normal pay and overtime pay using hybrid inheritance.*

```
#include<iostream.h>
#include<conio.h>
class emp_det
{
    protected:
       int emp_id;
    public:
       void get_id()
       {
          cout<<"ENTER EMPID"<<endl;
          cin>>emp_id;
       }
       void disp_id()
       {
          cout<<"EMPID IS "<<emp_id<<endl;
       }
};
class norm_pay : public virtual emp_det
{
    protected:
       float bpay;
    public:
       void get_bpay()
       {
          cout<<"ENTER BASIC PAY"<<endl;
          cin>>bpay;
       }
       void disp_bpay()
       {
          cout<<"BASIC PAY IS "<<bpay<<endl;
       }
};
class add_pay : public virtual emp_det
{
    protected:
       int hrs, rp, ap;
    public:
       void get_addpay()
       {
          cout<<"ENTER OVERTIME HOURS"<<endl;
          cin>>hrs;
          cout<<"ENTER HOW MUCH RUPEES PER HOUR";
          cout<<endl;
          cin>>rp;
       }
       void disp_addpay()
       {
          ap=hrs*rp;
          cout<<"TOTAL OVERTIME HOURS ";
          cout<<hrs<<endl;
          cout<<"ADDITIONAL PAY ";
          cout<<ap<<endl;
       }
};
class net_pay : public norm_pay,public
                                    add_pay
{
    int netpay;
    public:
       void display()
       {
          netpay=ap+bpay;
          cout<<"NET PAY IS "<<netpay;
       }
};
void main()
{
       net_pay obj;
       obj.get_id();
       obj.get_bpay();
       obj.get_addpay();
       obj.disp_id();
       obj.disp_bpay();
       obj.disp_addpay();
       obj.display();
       getch();
}
```

Output

```
ENTER EMPID
1101
ENTER BASIC PAY
10000
ENTER OVERTIME HOURS
10
ENTER HOW MUCH RUPEES PER HOUR
50
EMPID IS 1101
BASIC PAY IS 10000
TOTAL OVERTIME HOURS 10
ADDITIONAL PAY 500
NET PAY IS 10500
```

Here, the class norm_pay and add_pay are inherited from the class emp_det and the class net_pay is inherited from the class norm_pay and add_pay.

Thus, the class norm_pay, add_pay and net_pay will possess the following features:

From class norm_pay

Public Functions	Protected Variables
void get_id() (from class emp_det) void disp_id() (from class emp_det) void get_bpay() void disp_bpay()	emp_id (from class emp_det) bpay

From class add_pay

Public Functions	Protected Variables
void get_id() (from class emp_det) void disp_id() (from class emp_det) void get_addpay() void disp_addpay()	emp_id (from class emp_det) hrs, rp, ap

From class net_pay

Public Functions	Protected Variables	Private Variable
void get_id() (from class emp_det) void disp_id() (from class emp_det) void get_bpay() (from class norm_pay) void disp_bpay() (from class norm_pay) void get_addpay() (from class add_pay) void disp_addpay() (from class add_pay) void display()	emp_id (from class emp_det) bpay (from class norm_pay) hrs, rp, ap (from class add_pay)	netpay

Note *The duplication of inherited members due to multiple paths can be avoided by making the common base class (ancestor class) as virtual base class instead of, declaring the direct or intermediate classes. When a class is defined as virtual base class, C++ takes necessary care to see that only one copy of that class is inherited, regardless of how many inheritance paths exist between the virtual base class and a derived class.*

Constructors and Destructors in Derived Classes

When a base class and a derived class both have constructor and destructor functions, the constructor functions are executed in order of inheritance and the destructor functions are executed in reverse order of derivation. That is, the base constructor is executed before the constructor of the derived class and the destructor of the derived class is executed before the base class destructor.

Program 6. ***To illustrate the order in which the constructors and destructors get executed in an inheritance.***

```
#include<iostream.h>
#include<conio.h>
class A
{
   public:
      A()
      {
         cout<<"INSIDE A\n";
      }
      ~A()
      {
         cout<<"EXIT A\n";
      }
};
class B : public A
{
   public:
      B()
      {
         cout<<"INSIDE B\n";
      }
      ~B()
      {
         cout<<"EXIT B\n";
      }
};
class C : public B
{
   public:
      C()
      {
         cout<<"INSIDE C\n";
      }
      ~C()
      {
         cout<<"EXIT C\n";
      }
};
void main()
{
   C obj;
   getch();
}
```

Output

```
INSIDE A
INSIDE B
INSIDE C
EXIT C
EXIT B
EXIT A
```

Parameterized Constructors in Derived Classes

C++ supports a unique way of initialising class objects when a hierarchy of classes are created using inheritance. Constructors in an inheritance program can also be written using the syntax given below:

```
derived_class_name(arguments withdatatype):
base_class_name(arguments without datatype)
{
   //Body of the derived class constructor
}
```

Program 7. ***To illustrate the use of parameterized constructors in an inheritance.***

```
#include<iostream.h>
#include<conio.h>
class A
{
   int a;
   public:
      A(int x)
      {
         a=x;
         cout<<"INSIDE A\n";
         cout<<"a is "<<a<<endl;
      }
      ~A()
      {
         cout<<"EXIT A\n";
      }
};
class B
{
   int b;
   public:
      B(int y)
      {
         b=y;
         cout<<"INSIDE B\n";
         cout<<"b is "<<b<<endl;
      }
      ~B()
      {
         cout<<"EXIT B\n";
      }
};
class C : public A,public B
{
   int m,n;
   public:
      C(int p,int q,int r,int s) : A(p),B(q*2)
      //passing arguments to base class
      //constructor
      {
         m=r;
         n=s;
         cout<<"INSIDE C\n";
         cout<<"m is "<<m<<"\nn is "<<n<<endl;
      }
      ~C()
      {
         cout<<"EXIT C\n";
      }
};
void main()
{
   C obj(10,20,30,40);
   getch();
}
```

Output

```
INSIDE A
a is 10
INSIDE B
b is 40
INSIDE C
m is 30
n is 40
EXIT C
EXIT B
EXIT A
```

In this program, the object for the class C is created. In the constructor of class C, we have initialised the values for the member variables m and n. Similarly, we have also initialised the values of the base class constructors. The statement A(p) will initialise the value of 'a' in the base class A and the statement B(q*2) will initialise the value of 'b' to q multiplied by 2 in the base class B.