INHERITANCE

Questions & Answers:

1) Define Inheritance.

A) Inheritance allows us to reuse the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations of their parent class and may add new operations and new instance variables.

Or, we can also define as Inheritance is a mechanism of deriving a new class from an old class. Here we say new class as derived Class and an Old Class as Base Class.
Here Inheritance is the concept were new classes takes the attributes and Behavior of the pre-existing class that is we can say this pre-existing class as base class.
If you want to create a new class and most of its functionalities have defined in another already existing Class then instead of defining the same functionality in our new Class we can **reuse** these functionalities in our new Class using the concept of Inheritance.
Now we will see a sample example of implementing Inheritance

Example:

```
Class Base
 {
        Base Members;
        Base Methods;
 };

 Class Derived: public Base
 {
        Derived Members;
        Derived Methods;
 };
```

From the above example we have a Class named as “Base” and it contains its own data members (as Base Members) and Member Functions (as Base Methods) and another class named as “Derived” and it contains its own data members and member functions. As we see here the class Derived is **derived** from the base class (i.e. Old Class) Base using an operator ‘**:**’ (Colon).

Inheritance is also sometimes called **generalization**, because the **is-a** relationships represent a hierarchy between classes of objects. For instance, a

"fruit" is a generalization of "apple", "orange", "mango" and many others. One can consider fruit to be an abstraction of apple, orange, etc. Conversely, since apples are fruit (i.e., an apple **is-a** fruit), apples may naturally inherit all the properties common to all fruit, such as being a fleshy container for the seed of a plant.

2) How Inheritance is importance in Object Oriented Programming?

A) One of the important features in the Object Oriented Programming is **Reusability**, as we said above that it is always good way to reuse the already existing functionality rather than trying to create the same one again and again. By reusing the properties not only saves the time and money but also increases the reliability.
An advantage of inheritance is that modules with sufficiently similar interfaces can share a lot of code, reducing the complexity of the program.

The Benefits of Inheritance

- Subclasses provide specialized behaviors from the basis of common elements provided by the superclass. Through the use of inheritance, programmers can reuse the code in the superclass many times.
- Programmers can implement superclasses called abstract classes that define "generic" behaviors. The abstract superclass defines and may partially implement the behavior but much of the class is undefined and unimplemented. Other programmers fill in the details with specialized subclasses.

3) Define Base Class and Derive Class.

A) A Class is said to be a Base Class if it can derive one or more derived classes. It is a nice way to design a base class which includes all common or generic members and functions, So that the classes derived from this class can utilize all the functionalities in the base class including functionalities specific to this derived class.

For example we take vehicle; a vehicle can be a bike, car, and lorry. If you keep all work functionalities which is common for building a bike, car and lorry like vehicle attach Engine, Tyres and breaks this all properties also belongs to every kind of vehicle so instead of again defining all these properties in you specific vehicle class like car or lorry we can just derive your specific car class(for example) from vehicle class(in our example this class is a base class) so that now you just define only functionalities which is specific to the car class like Number of passengers(Capacity) , include AC or not etc.., Now you can define your specific features like

number of persons can seat in your car and you want to include an AC in your car or not, now you need not to again define the properties like attach Engine, Tyres, breaks etc.., because your class is derived from the class where already it has been defined these properties means these properties now belongs to our Car Class our duty is to just use it using our Car Class Object.

Till now we learn what kind of data should be declare in base class and the data in a class that is derived from the base class. Now we will see how we can create a new class from already existing class and also how we can use the properties of the parent class (or base class).

As explained in the above example, now take a class 'vehicle' with the following members:

```
Class vehicle

{

    Public:

    Void Aengine (int engine_displacement);

    Void Abreaks ();

};
```

Now this is a class which contains a common properties like, every vehicle will contain an engine but they differs only with engine displacement due to this reason we sending an integer variable 'engine_displacement' so that a particular vehicle have to attach a engine depends on the value send through the parameters of Aengine () Function.

Now we create a new class called Car, as we know Car is one kind of vehicle so why don't we use this above vehicle class because to build a Car we need to attach an separate engine with particular engine displacement and also we should place breaks to the Car. So, instead of again defining the methods for attaching engine and breaks we can reuse (Here Inheritance comes in to play) the methods define in our above vehicle class because for every kind of vehicle only one common method to attach an engine or breaks.

```
Class Car
{

    Public:
```

```
    Car_capacity (int capacity);
};
```

Here we declare a class but we didn't derive this class from vehicle class, to derive this new class Car from our vehicle class we use a Colon Operator (**:**) before we show how to derive car class from vehicle class you just understand the syntax of deriving a new class from old class.

Class derived_class_name : <Access_specifier> base_class_name
{

// derived class members.

};

As you see in the above syntax the class derived_class_name derives from the base_class_name using Colon operator but the visibility of the base_class functionalities in the derived_class depends on the Access_specifier it may be either private or public. Latter I specify how scope of the functionalities of base class depends on the Access_specifier you declares (either public or private).

Now I hope you understand how to implement an inheritance, so now we will implement for our example.

```
Class vehicle

{

    Public:

   Void Aengine (int engine_displacement);

  Void Abreaks ();

};

Class Car : public vehicle

{

  Public:

    Car_capacity (int capacity);
};
```

```
Void main()
{
   Car car_obj =new Car(); // Creating an Instance for Derived class Car

   car_obj.Car_capacity(5); // Calling the method defined in derived class

    car_obj. Aengine (80);   // Calling a method defined in Base Class

     car_obj.Abreaks ();        // using derived class object.


}
```

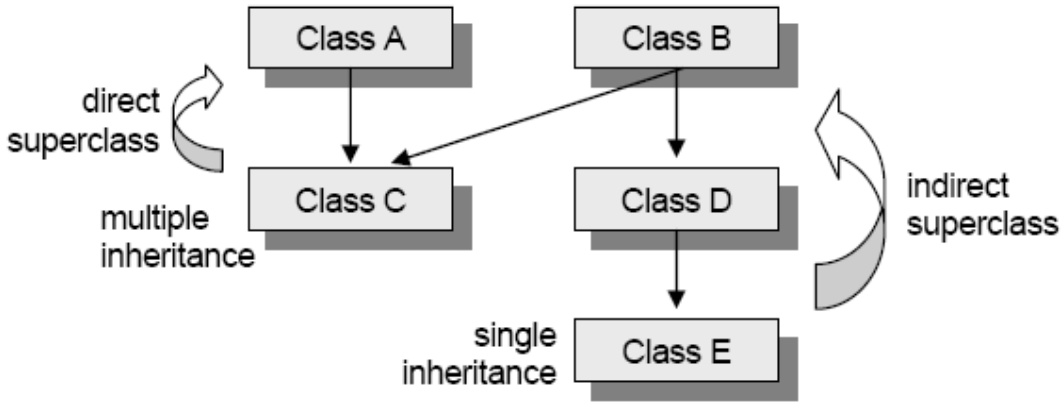
Another Important point should be remembered that, as every class is a subset to itself so every single class act itself as a Base Class. Means as in the first question it has been already explained that Inheritance supports **is-a relationship** so every class is-a base class to itself.

4) What are the different forms of Inheritance?

A) Before we go for the different forms of Inheritance, we need to know about Hierarchy in object oriented programming.

In object oriented programming, the mapped relationships of sub and super classes is known as a hierarchy. This can be visualized as an upside-down tree (or perhaps a pyramid), the top of which is known as the root.

Consider the following class hierarchy:



The terminology is defined below:

• The direct superclass of a subclass is that class from which it explicitly inherits within the class definition, **e.g. class C : public A** here **A** is a direct superclass of C.

• An indirect superclass of a subclass is a class which the superclass inherits from (directly or indirectly), **e.g. class E : public D; class D :public B** here B is an indirect superclass of E.

• Single inheritance entails that a class can only inherit directly from one Superclass.

• Multiple inheritance entails that a class can inherit directly from a number of superclasses.

A hierarchy can link entities either directly or indirectly, and either vertically or horizontally. The only direct links in a hierarchy, insofar as they are hierarchical, are to one's immediate superior or to one of one's subordinates, although a system that is largely hierarchical can also incorporate other organizational patterns. Indirect hierarchical links can extend "vertically" upwards or downwards via multiple links in the same direction. All parts of the hierarchy which are not vertically linked to one another can nevertheless be "horizontally" linked by traveling up the hierarchy to find a common direct or indirect superior, and then down again. This is akin to two co-workers, neither of whom is the other's boss, but both of whose chains of command will eventually meet.

In object oriented programming, we can define a Hierarchical relationship among set of classes and its sub classes in Six forms. They are

Single Inheritance

Multiple Inheritance

Multi-level Inheritance

Hierarchical Inheritance

Hybrid Inheritance

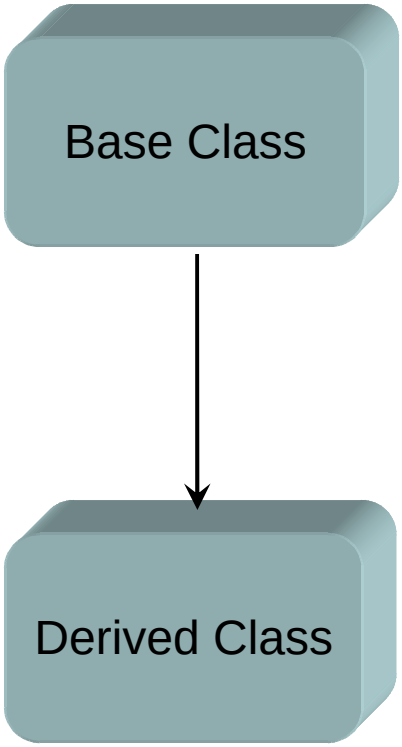
Multi-Path Inheritance

Each one will be explain in the following sessions.

5) Define Single Inheritance.

A) If the class hierarchy contains only two classes one is a base class and another is derived class then this form of class hierarchy is known as single Inheritance.

The class hierarchy shown below represents Single Inheritance.



The direction arrow represents “Inherits” Symbol.

Now we will see a sample example which represents single Inheritance.

```
# include<iostream.h>
# include<conio.h>
class A
```

```
{
 public:
        int a, b;
        void get ();
};

class B : public A
{
        int c;
 public:
        void cal ();
        void display ();
};

void A :: get ()
{

         Cout<<"enter the a, b value\n";
         Cin>>a>>b;
}
Void B :: cal()
{

        c = (a + b);
}
 Void B :: display()
 {

        Cout<<"the added value is: "<<c;
 }
 Void main ()
 {
  B b_obj = new b_obj();
  Clrscr ();
        b_obj.get ();
        b_obj.cal();
        b_obj.display ();
  getch ();
 }
```

In the above program class A known as Base class and class B as derived class, as we mentioned earlier in the class A we declare a method (void get ()) as we see the declaration of class B

class B: public A

In the above statement, we created a class B and inherited all the methods and variables of class A to class B in public mode, which means all the methods and variables of class A now belongs to class B and acts as public members in class B.

Note:

- Colon (**:**) Symbol is used for inheriting a new class from its parent class.
- The most important to note that we should mention the visibility mode i.e. in which mode (either private or public) the members of its base class should act in its derived class.

6) Define Multiple Inheritance.

A) Multiple inheritance refers to a feature in which a class can inherit behaviors and features from more than one superclass. This contrasts with **single inheritance**, where a class may inherit from only one superclass.

Only few Object Oriented Programming Languages supports Multiple Inhritance because of some Ambiguities arise in multiple inheritance.

These ambiguities in multiple inheritance are explained in next coming sessions.

Languages that mostly support multiple inheritance are: Eiffel, C++, Python, Perl, and CLOS.

Java and C# do not allow multiple inheritance; this results in no ambiguity. However, Java and C# allow classes to inherit from multiple interfaces.

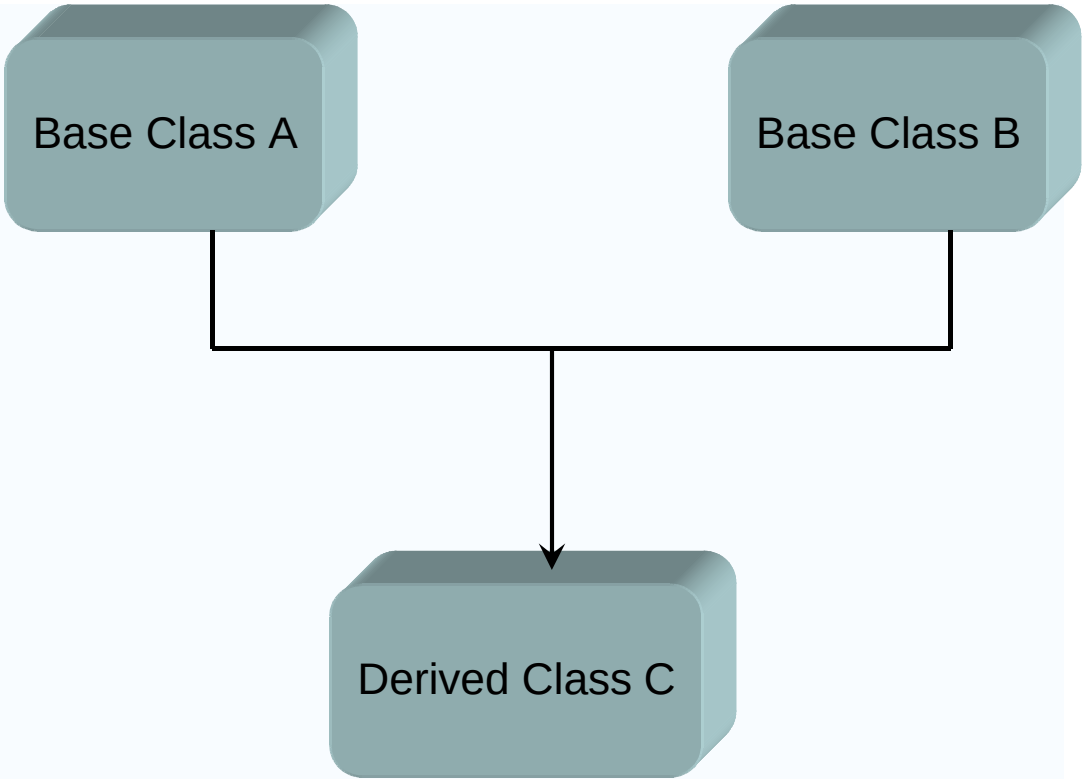
Syntax**:**

```
Class base_class_name1
{
        // List of members
};

Class base_class_name2
{
```

```
        // List of members
};

Class derived: <access specifier> base_class_name1, <access
specifier> base_class_name2
{
        // List of members
};
```

From the above syntax we can clearly understand about multiple inheritance.

The class hierarchy shown below represents Multiple Inheritance.



Now we will see a sample example.

```
# include<iostream.h>
```

```
# include<conio.h>
# include<math.h>
Class A
{
 Public:
        int a, b, c;
 void get ()
 {
        cout <<"enter the value of a, b\n";
        cin>>a>>b;
 }
 void cal ()
 {
        c=(a+b)*(a-b);
        Cout<<"added value of c is: "<<c;
        Cout<<"\n";
 }
};
Class B
{
 public:

   Void calc (int a, int b)
   {
        int c;
        c = pow (a, 3) + pow (b, 3);
        Cout<<"cubed value of c is: "<<c;
        Cout<<"\n";
   }
};
class C: public A, public B
{
  public:

  void calculate ()
  {
      float c = (a / b);
        Cout <<"divided value of c is: "<<c;
  }

};
void main ()
{
        C p;
 Clrscr ();
        p.get ();
```

```
        p.cal ();
        p.calc (4, 2);
        p.calculate ();
    Getch ();
}
```

```
OUT PUT
=========

enter the values of a, b
2 4
added value of c is       :-12
cubed value of c is: 72
divided value of c is: 2
```

In the above example program the functions of class A i.e. Get () & cal () and functions of class B i.e. Calc (int a, int b) are inherited means these functions are declared in their respective classes but now these functions representing as it belongs to class C, by the statement as mentioned above the compiler represents all the functions of these three functions in only one class i.e.. Class C because in the main function, all the functions are invoked using object of class C, So a compiler reads only class C from which it invokes all the functions.

Note:

In programming, when a new class is derived from two or more base classes Then after Colon Symbol all base class names with its respective visibility Modes should be separated by Comma (**,**) Symbol as represented below.

Class derived: <access specifier> base_class_name1,

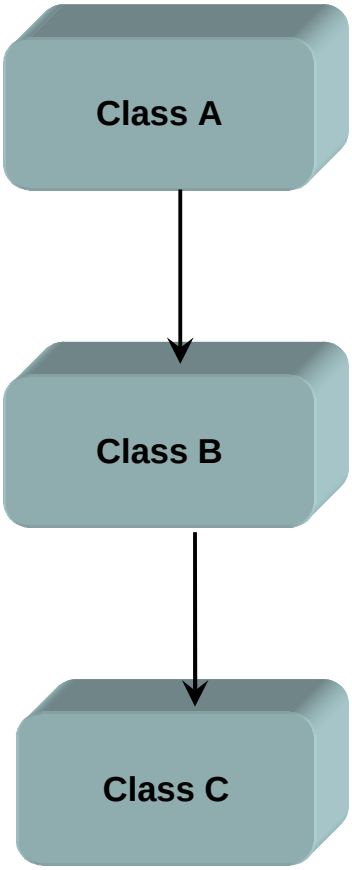
<access specifier> base_class_name2

7) Define Multi-Level Inheritance.

A) When you define more than two levels of inheritance (in the form of a chain of classes), it would be generally referred to as multi-level inheritance. In the case of multi-level inheritance, all the members of all super classes would be automatically available within the sub class.

The class hierarchy shown below represents Multi-Level Inheritance. Here I shown only three level chain of classes but we can build multiple chain of classes according to the requirement.

In the below diagram I also given explanation for each level of classes.



- **Base Class To Both classes B and C but it is Direct Base Class to B and Indirect Base Class To Class C.**

- **Direct Base Class To Class C and Derived from Class A.**

- **Derived from Class B, But it also act as An Indirect Derived Class of A.**

Now we see how multi-Level inheritance should implement through this syntax

Syntax:

```
Class base_class_name1
{
                    // List of members
};

Class derived_base_name2 : <access specifier> base_class_name1
{
                    // List of members
 };

Class derived : <access specifier> derived_base_name2
{
                    // List of members
```

```
};
```

As you seen the syntax above clearly shows how to implement the multi-Level inheritance through programming.

Now we will implement multi-level inheritance through this sample Example.

Example:

Assume that the test results of a batch of students are stored in three different classes. Class A stores the roll number, class B stores the marks obtained in two subjects and class C contains the total marks obtained in the test. The class C can inherit the details of the marks obtained in the test and the roll number of students through multi-level inheritance.

```
# include<iostream.h>
# include<conio.h>
class A
{

 Public:
 Int roll no;
 void getrollno (int r)
 {
  roll no=r;
 }
 void putrollno ()
 {
  cout <<"student roll number: "<<roll no<<"\n";
 }
};
 class B: public A
 {

   Public:
   Int math, cs;
   Void getmak (int x, int y)
   {
   Math=x;
   Cs =y;
   }
   void putmak ()
   {
```

```
   Cout<<"marks in maths :"<< math<<"\n";
   Cout<<"marks in comp science: "<<cs<<"\n";
  }
};
 Class C: public B
 {
  Public:
  int tot;
  Void get tot ()
  {
   tot = math+cs;
   }
  Void put tot ()
  {
   Cout<<"total marks secured: "<<tot;
  }
 };


 Void main ()
 {
  C p;
  P.getrollno (626);
  P.getmak (116,282);
  p.gettot ();
  Clrscr ();
  P.putrollno ();
  P.putmak ();
  P.puttot ();
  Getch ();
 }
```

In the above example we have seen how all three chains of classes are related to each other and performing a complete task where each class perform partial work, combining these partial works to get a complete work.

Hence, any subclass object can be treated as an instance of the corresponding superclass object. This stands to reason, as a subclass can do everything a superclass object can do. The ability to treat a subclass object as an instance of the superclass is a key principle of OOP.

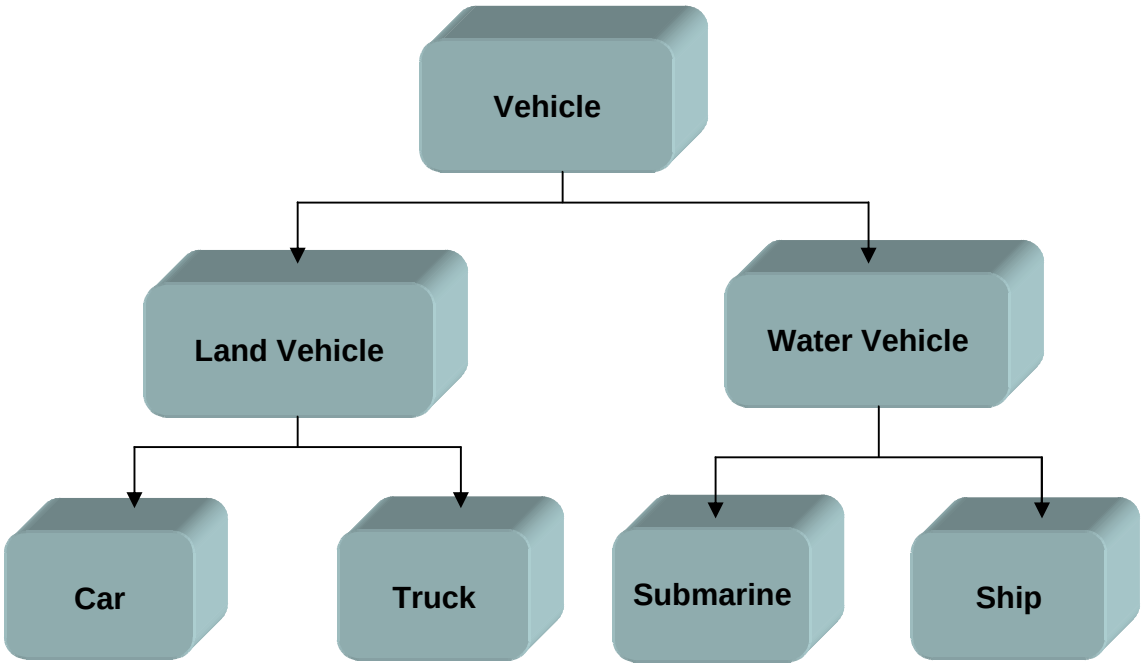
8) Define Hierarchical Inheritance.

A) Any complex task we can solve by designing the problem in some hierarchical way. Similarly, by Hierarchical designing of a program we can solve any complex task in Object Oriented Programming.

Another Interesting application of inheritance is to use it as a support to the hierarchical design of a program.

In C++, A problem can be easily solved by converting the problem into class hierarchies. In this class hierarchy the top most class also called as root includes all the features that are common to its subclasses and these subclasses can be constructed by inheriting features of its base class, in turn these subclasses can act as a base class for the lower level classes and so on.

As an example shown in the below diagram,



The attributes and operations below are for **vehicle class,** which the **SubClass** inherits.

Attributes: registration Number, seat number, engine power, capacity.

Operations: started (), stopped (), reversed ().

Here the subclasses of vehicle class are land vehicle and water vehicle, now because of these two classes inherited from vehicle class so the above mentioned attributes and operations also belongs to these land vehicle and water vehicle classes and also it contains attributes and operations which are specific

to their classes i.e. Extra operation in the land Vehicle works only on Land and in water vehicle works only in water.

Car and truck will move on land so these classes can inherits from land vehicle class So, these two classes contains all the attributes and operations of its parent class (land vehicle) as well as it also contains its grand parents (vehicle class) attributes and operations which contains all the common features of vehicle. similarly for submarine and ship subclasses which moves in the water so these classes can inherits from water vehicle class So, these two classes contains all the attributes and operations of its parent class (water vehicle) as well as it also contains its grand parents (vehicle class).

Now we will implement a sample example which demonstrates the hierarchical Inheritance,

```
# include<iostream.h>
# include<conio.h>
Class A
{

Public:
        Int a, b, c;
 Void get ()
 {
        Cout<<"enter the values of a, b\n";
        Cin>>a>>b;
 }
 };
 Class B: public A
 {
   Public:
     Void add ()
 {
        c= a + b;
 }
 void put add ()
 {
        Cout<<"the add result: "<<c<<"\n";
 }

 };
 Class C: public A
 {
   Public:
   void div ()
```

```
  {
        c=a/b;
  }
  void getdiv ()
  {
        cout<<"divided value: "<<c<<"\n";
  }
  };
void main ()
{
        C p;
        B e;
 Clrscr ();
        e.get ();
        e.add ();
        e.putadd ();
        p.div ();
        p.getdiv ();
 Getch ();
}
```

In the above example to invoke their specific functions we created two separate objects for both classes C and B.

I hope now you understand the how class hierarchy can help to solve any task. But in real-world instead of classes we use objects.

These objects will have two limitations:

1. One object may not occupy two different places at the same time.
2. Two objects may not occupy the same place at the same time.

Hierarchical classification (object oriented structure) works for representations of real-world objects because objects are objects. That is, they can only be in one place at a time and two cannot occupy the same place at the same time. We can therefore place an object within a hierarchical class representation of all the objects we know about and, if we put it in the right spot, we can be relatively sure we've reached the global (absolute) minima - the most correct place for that object within the collection of all objects we know about.

As we add objects, we may not get it right on the first try, or even on the thousandth try, but eventually each object will wind its way down to its correct place in relation to all the other objects.

We can be relatively sure that there is exactly one right way to fit all the objects in correct relationship to each other. As new objects are discovered they will displace our existing objects and move them to various different places within the hierarchy. But the structure will always ensure we find the one and only one location where a given object and no other object should rest within the hierarchy of objects.

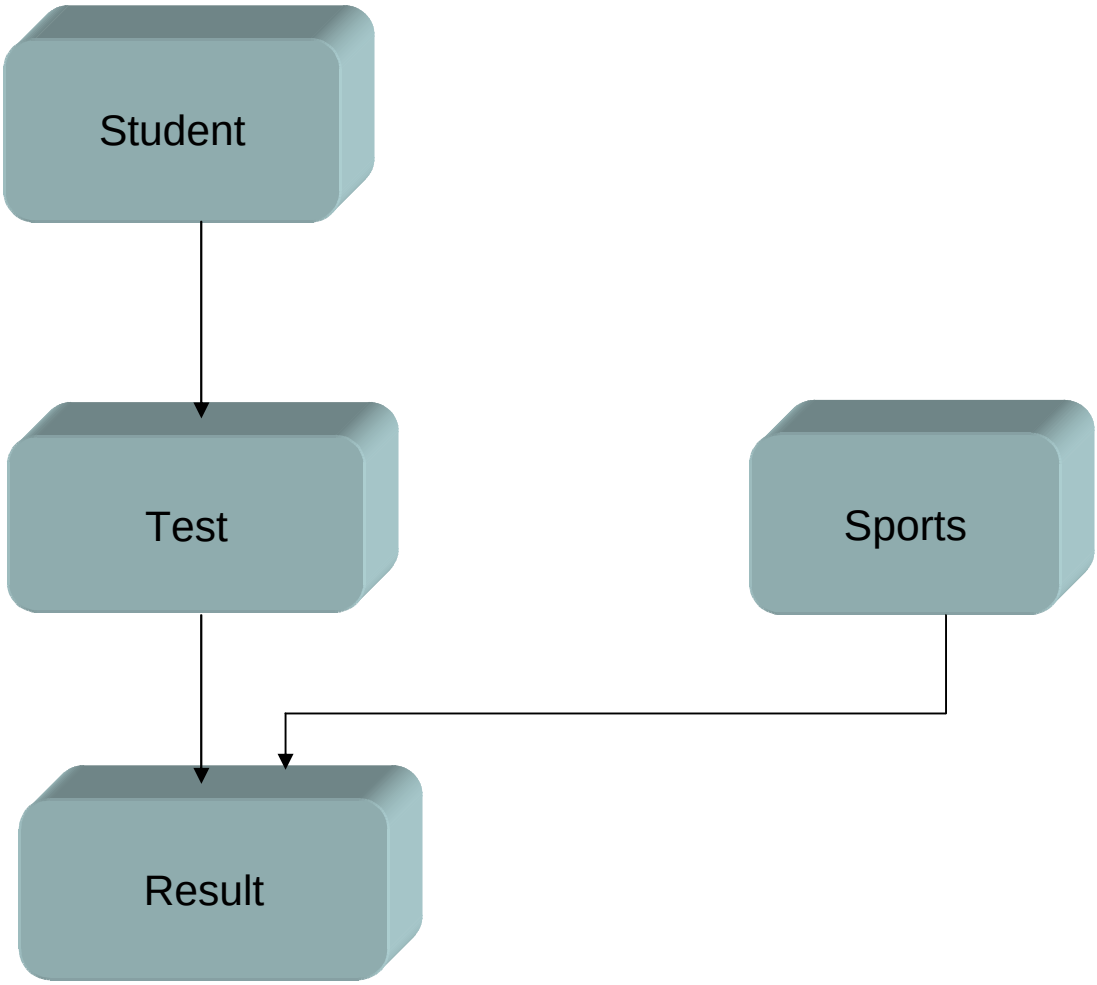
This is because the limitations of nodes in our object oriented hierarchical classification system perfectly match the limitations of real world objects. That is, they may only be in one place at any one time, and no two may occupy the same place at the same time.

For this reason, programs that simulate and represent our real world with all its objects are prime candidates for Object Oriented Design (e.g. desktops, shelves, cabinets, file-folders, and paper documents).

9) Define Hybrid Inheritance.

A) Some times there could be situations where we need to use two or more types of inheritance to design a program.

For example, consider the case of processing the student results discussed in multi-level instance example. Assume that we have to give weightage for sports before finalizing the results. The weightage for sports is stored in a separate class called sports. The new inheritance relationship between the various classes would be as shown in below diagram



As I said hybrid inheritance is a mix of two or more types of inheritance, in our example now we mixed Multi-Level and Multiple Inheritance.

In the above class hierarchy observe the chain of classes student, test and result represents multi-level inheritance and class hierarchy between classes Test, sports and result represents Multiple Inheritance where result class derived from two classes i.e. test and sports classes.

Now the below example program illustrates the implementation of both multiple and multi-level inheritance.

```
# include<iostream.h>
# include<conio.h>


class Student
{

Public:
 Int roll no;
 void getrollno (int r)
 {
  roll no= r;
 }
 void putrollno ()
 {
  cout <<"student roll number:"<<roll no<<"\n";
 }
};

 class test : public Student
 {

   Public:
   Int math, cs;
   Void getmak (int x, int y)
   {
   Math=x;
   Cs =y;
   }
   void putmak ()
   {
    Cout<<"marks in maths:"<<math<<"\n";
    Cout<<"marks in comp science:"<<cs<<"\n";
   }
 };

      Class sports
      {

         Float score;

      Public:

      Void get_score (float s)
      {
         Score = s;
```

```
}
Void put_score ()
{
 Cout<<"" sports wt: "<<score<<"\n";

}

};

Class result : public test, public sports
{
   Float total;

Public:
     Void display()
      {
           Total= math + cs;
           Putrollno();
           Putmak();
           Put_score();
           Cout<<"Total Score "<<total<<"\n";
      }
};

Void main()
{
   Result std;
    Std . getrollno(020);
     Std . getmak(55, 65);
     Std . get_score(10.0);
     Getch();
}
```

OUTPUT

Student roll number : 020

Marks in math : 55

Marks in comp. science : 65

Sports wt : 10.0

Total Score : 120.0

10) Define virtual Inheritance.

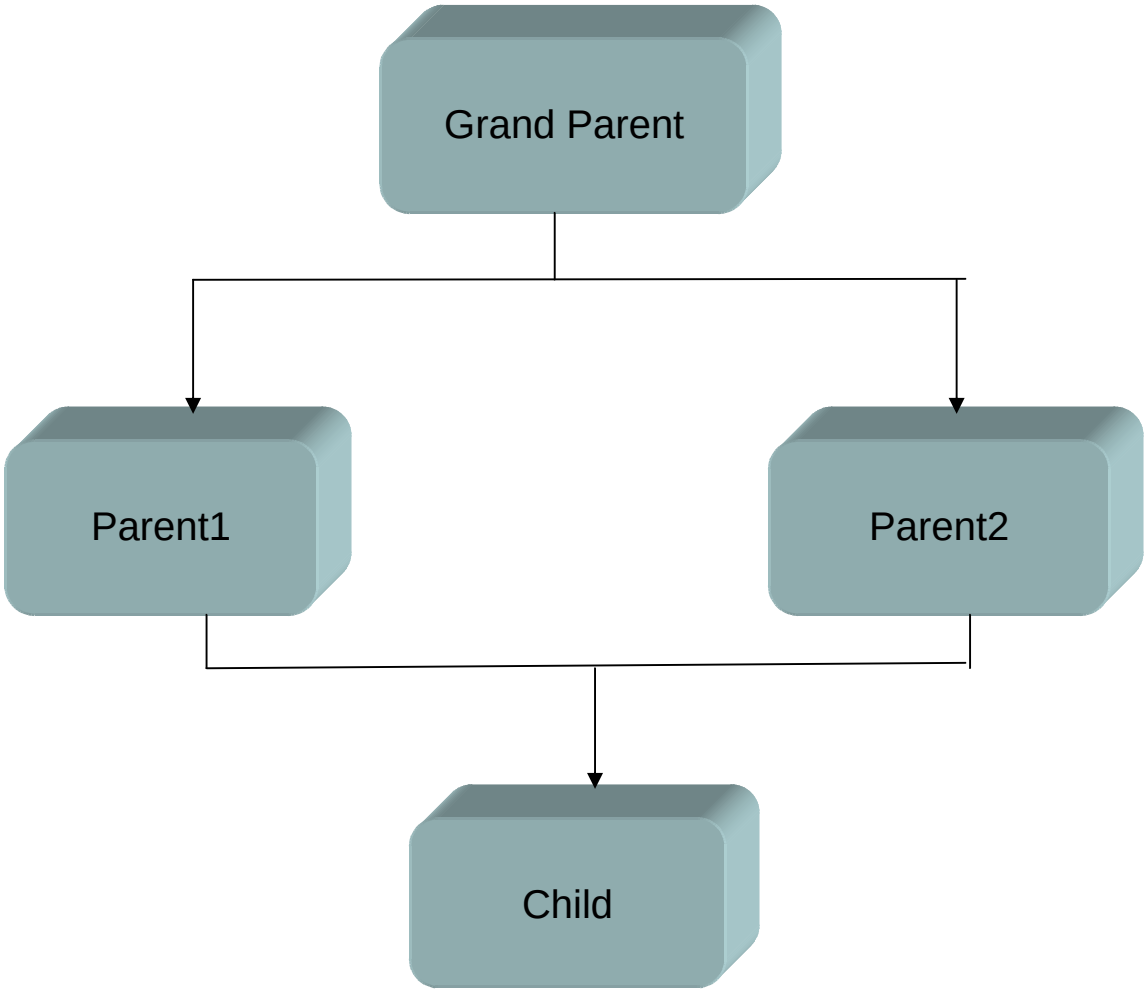
(Or)

Explain what drawback does Multiple Inheritance contains and Solution to this problem.

(Or)

Explain Diamond problem and its solution.

A) To explain virtual inheritance, consider the situation where we need to apply all three kinds of inheritance namely, Multiple, Multi-Level, and Hierarchical inheritance. This situation is illustrated in below diagram.



Where in this diagram child has two direct parent classes parent1 and parent2 in turn these two classes parent1 and parent2 has a common base class grandparent.

Here the problem is child class inherits the attributes and operations of grandparent class via two separate paths one through parent1 class and another through parent2 class, this means child class would have **duplicate** set of

attributes and operations of grandparent class. This grandparent class some times referred as indirect base class to child class.

These duplicate sets of attributes and operations of grandparent class in child class from multiple paths can be avoided by making the grandparent class as **virtual base class** while declaring as shown below:

```
Class A
{
   // List of operations

};

Class B : virtual public A
{
    // List of operations

};

Class C : public virtual A
{
   // List of operations

};

Class D : public B, public C
{
    // List of operations

};
```

Here in class D only one copy of attributes and operations of class A will be inherited.

The keywords virtual and public can be used in either order as shown in above example.

Now I hope you understand how virtual inheritance work and where. So, I will explain what diamond problem is briefly and its solution based on a sample example.

Virtual inheritance is a kind of inheritance that solves some of the problems caused by multiple inheritance (particularly the "diamond problem") by clarifying ambiguity over which ancestor class members to use.

Diamond problem:

Consider the following class hierarchy.

```
class Animal
 {
    virtual void Eat();
};

class Mammal : public Animal
 {
 Public:
  virtual Color GetHairColor ();
};
class WingedAnimal : public Animal
{
 Public:
  virtual void Flap ();
};



// A bat is a winged mammal
class Bat : public Mammal, public WingedAnimal
{

};
```

But how does a Bat Eat()? As declared above, a call to Bat.Eat() is ambiguous. One would have to call either Bat.WingedAnimal**::** Eat () or Bat.Mammal**::**Eat(). The problem is that semantics of conventional multiple inheritance do not model reality. In a sense, an Animal is only an Animal once; a Bat is a Mammal and a WingedAnimal, but the Animalness of a Bat's Mammalness is the same Animalness as that of its WingedAnimalness.

This situation is sometimes referred to as **diamond inheritance** and is a problem that virtual inheritance works, in part, to solve.

Solution to this problem:

We can redeclare our classes as follows:

```
// Two classes virtually inheriting Animal:

class Mammal : public virtual Animal
{
 public:
  virtual Color GetHairColor();
};
class WingedAnimal : public virtual Animal
 {
 public:
  virtual void Flap();
};

// A bat is still a winged mammal
class Bat : public Mammal, public WingedAnimal
{

};
```

Now the Animal portion of Bat**::**WingedAnimal is the same Animal as the one used by Bat**::**Mammal, which is to say that a Bat has only one Animal in its representation and so a call to Bat**::**Eat () is unambiguous.

11) Define how we Access Members with different Visibility Modifiers

A) As we know in C++, there are three access modifiers namely, public, private and protected.

The private members of the base class cannot inherit therefore it is not available for the derived class directly. But indirectly we can access the private members of base class, how? The base class methods can access to the private members, the derived class reference it indirectly using its parents class methods.

The protected members of the base class can accessible with in its class and any class immediately derived from it. It cannot be accessed by any methods that are outside of these two classes i.e. methods do not belongs to these two classes.

A class can now use all three visibility modifiers as shown below:

```
Class A
{
   Private :

         // members visible only to this class member functions
```

```
Protected:

        // members visible to this class member functions and also for its derived
        // class member functions.

Public:

        // visible to all functions in the program.


};
```

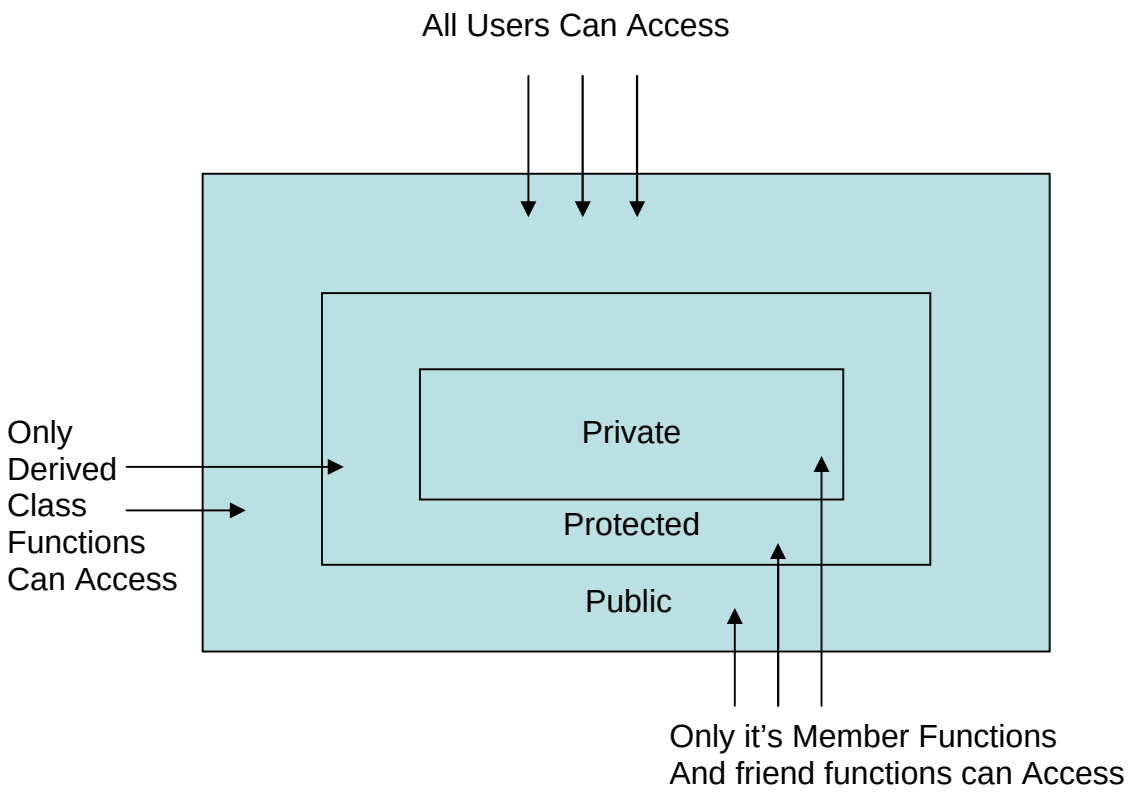
When a protected member is inherited in public mode, it becomes protected in its derived class therefore this members can accessible by member functions of the derived class.

When a protected member is inherited in private mode, it becomes private in its derived class therefore these members can accessible only by this class member functions, it is not available for further inheritance since private members cannot inherited.

The following table illustrates how visibility of base class members will undergo modifications in all the three kinds of derivation.

Base Class visibility	**Derived Class Visibility**		
	Public Derivation	**Private Derivation**	**Protected Derivation**
Private	Not Inherited	Not Inherited	Not Inherited
Protected	Protected	Private	Protected
public	Public	Private	Protected

This can also depict in diagrammatical form:


All Users Can Access
Private
Protected
Public
Only
Derived
Class
Functions
Can Access
Only it's Member Functions
And friend functions can Access