2/04/2023
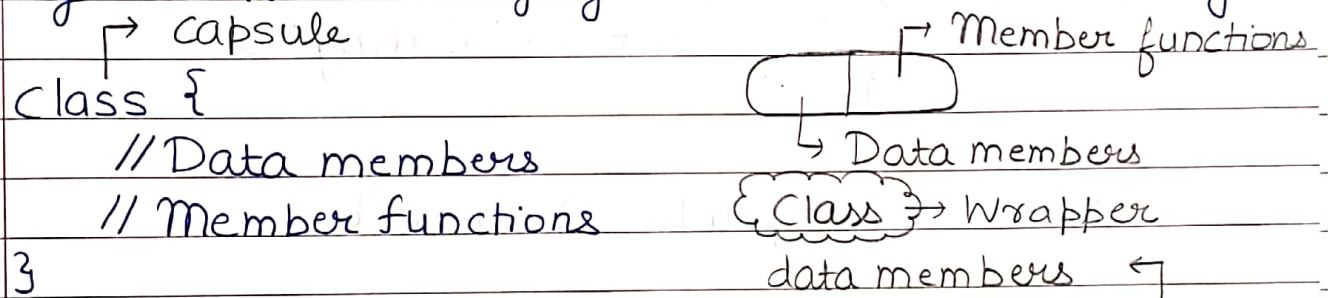
4 pillars of OOPs
We can say everything is abstraction. 4 pillars of OOPs are :
1) Encapsulation
2) Inheritance
3) Polymorphism
4) Abstraction → Implementation hiding

Encapsulation
Wrapping of data members & member functions in the capsule (class) is known as encapsulation. By this we are trying to achieve data hiding.
        ┌→ Capsule   ┌→ Member functions
Class {
    // Data members   ↳ Data members
    // Member functions   (Class)→ Wrapper
}   data members ↰
Perfect encapsulation is that in which everything is set as private. We can access these private elements via setter & getter.
By this we are increasing the security, like what type of data to show & what type of data not to show.

Note → Abstract = not going into much details

Ex → class Animal {
            public :
                int age ;
                void eat () {
                    cout << "Eating" ;
                }
        };

Inheritance

There will be a super/parent/base class and there will be subclass/child/derived class. Derived class will inherit the properties of base class. There can be multiple derived classes as well.

Syntax

class child : mode parent
         ↳ public, private, protected

Ex→
```
class Animal {
        public :
            int age ;
            void eat () {
                cout << "Eating" ;
            }
};
class Dog : public Animal {

};
main () {
    Dog dl ;
    dl.eat () ;
}
```

O/p→ Eating

The above code did not give an error as the data members & member functions gets inherited to the Dog class i.e. child or derived class.

Chart of inheritance

Base class access modifier	Type of inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	NA	NA	NA

NA → Not accessible

Ex→
```
class Animal {
        public :    → Base class access modifier
            int age ;
            void eat () { -- }
};                  ┌ Type of inheritance
class Dog : public Animal {

            int age ;    → public        ⎫ Inherited
            void eat () { -- } → public   ⎬ from base
};                                        ⎭ class
```

when
Hence, we were doing D1. eat(), we were not getting an error as eat function was in public mode.
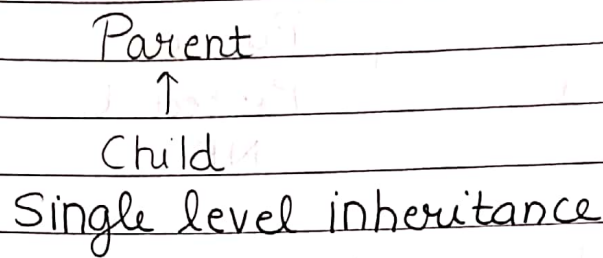
Protected mode (access modifier)
Accessible in derived class. This is combination of private + (only accessible in the derived class).
Private → not inherited
Protected → inherited

Types of inheritance

1) <u>Single Level inheritance</u> → There is one base class and only one child class.

<div align="center">

Parent

↑

Child

Single level inheritance

</div>

Ex→ class car {

};

class rapid : public car {

};

<div align="right">

Car

↑

Rapid

</div>

Note→ Inheritance is basically <mark>is - a relation</mark>. We can say rapid is a car.

       ↳derived   ↳ base class

        Class

2) <u>Multilevel inheritance</u>

<div align="center">

Parent

↑

Child

↑

GrandChild

⋮

</div>

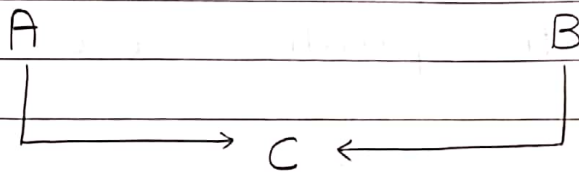There can be many levels.

Here also <mark>is-a relationship will be</mark> followed.

Ex→ class Fruit { --- };
class mango : public Fruit { --- };
class alphanso : public mango { --- };

Note→ Hence we can say alphanso is a mango & mango is a fruit.

3) Multiple inheritance

```
A                          B
 |                          |
 └──────→ C ←───────────────┘
```
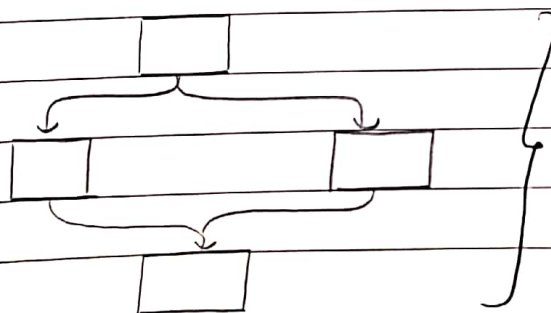
There are two base classes and single child or derived class.

Ex→ class A { ---- };
class B { --- };
class C : public A, public B { --- };

There is a problem in the above inheritance, that what if there are same items in A & B class & hence compiler can get confused here. This is known as **diamond problem** & we can solve it by **scope resolution operator.**
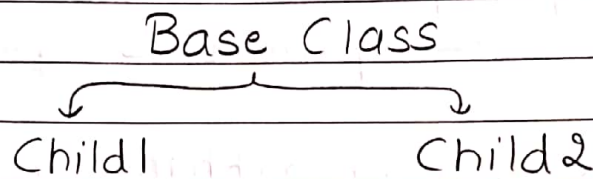
C obj;   → using var_name of class B
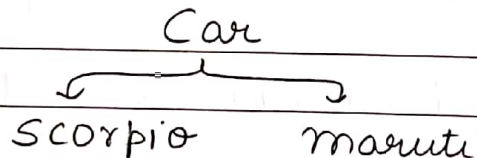cout << obj . B :: var_name;

Diamond
problem

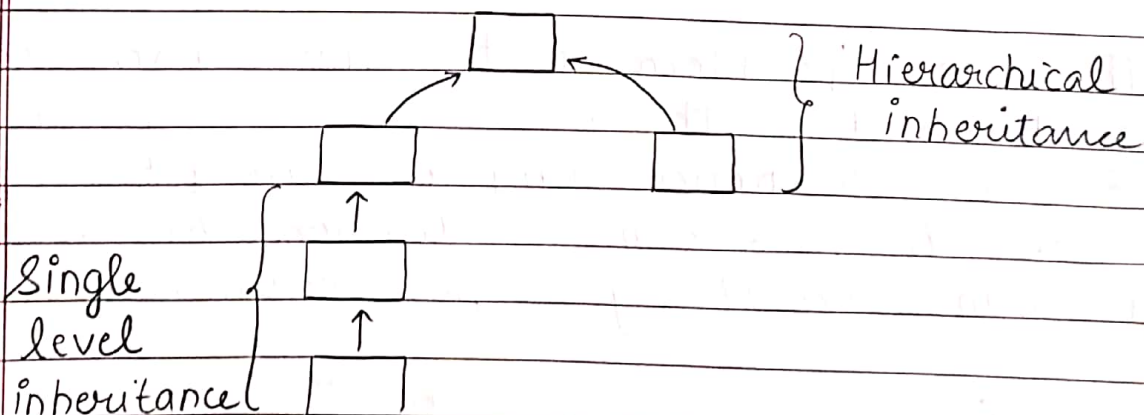## 4) Hierarchical Inheritance

Base Class

Child1   Child2

Single base class, multiple derived classes.

Ex → class car { --- };
class scorpio : public car { -- };
class maruti : public car { -- };

Car

scorpio   maruti

Scorpio is a car and maruti is a car.

5) Hybrid inheritance → It is the mixture of above studied inheritances.

Hierarchical
inheritance

Single
level
inheritance

Polymorphism → Important for interviews
Poly → many
morph → forms
This means existing in many forms.

There are 2 types of polymorphism namely compile time & run time polymorphism.

1) Compile time polymorphism
It is based on operator overloading & function overloading.

* Function overloading

```
class Maths {
    public:
        int sum (int a, int b) {
            return a+b;
        }
        int sum (int a, int b, int c) {
            return a+b+c;
        }
        int sum (int a, float b) {
            return a+b;
        }
};

main () {
    Maths obj;
    cout << obj.sum (2,3);       // 1st func
    cout << obj.sum (2,3,5);     // 2nd func
    cout << obj.sum (2,5.6f);    // 3rd func
}
```

In function overloading, return type of function should be same.

* Operator overloading
For example using + operator, we can print the difference also & this can be done via

Operator overloading.
Syntax   → operator symbol
return _ type   operator ⊕ ( ) {
    - - - -
}   → current obj

                            ⓐ + ⓑ → input parameter
Ex →   ↳ member function

class Param {
        public :
                int val;
                void operator + (Param & obj2) {
                        int value1 = this + val;
                        int value2 = Obj2 · val;
                        cout << (value1 - value2);
                }
};

main () {
        Param obj1 , obj2;
        obj1·val = 7;
        obj2· val = 2;

        Obj1 + obj2 ;   → 5
}

Note → Obj1   ⇒ current object
        +   ⇒ member function
        obj2   ⇒ input parameter

This is basically operator overloading as
+ operator is existing in many forms such
as addition of numbers, objects, concatenation
etc.