Object Oriented Programming

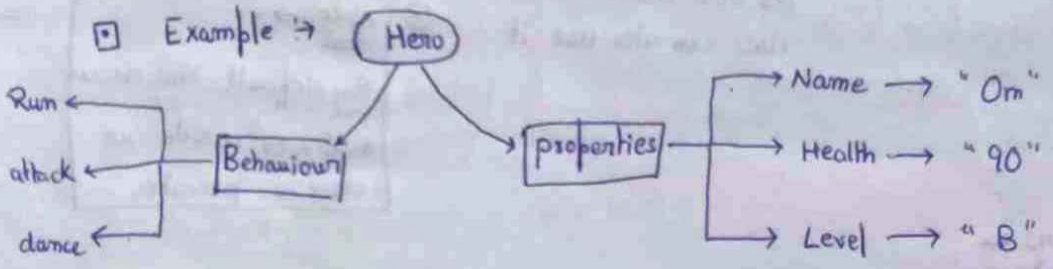
⊙ What is Object Oriented Programming?

ans/ ⊡ It is a programming technique in which the code revolves around an object.

⊡ Object → It is an entity comprising of two things ....
- State / Properties
- Behaviour

⊙ Note →
An object is a instance of class.

⊡ Example →



⊙ Class → A class is a user defined datatype.

Example →
```
class Hero {
    public :
        int health ;
        char level ;
        void print() {
            cout << "Rohit" << endl;
        }
};

int main () {
    Hero rohit;
    rohit . health = 70;
    rohit . level = 'A';
    cout << rohit . health << " , " << rohit . level ;
    rohit . print () ;
}
```

- int health; char level; → properties / data members.
- void print() { cout << "Rohit" << endl; } → behaviour / function
- Hero rohit; → object creation
- rohit . health = 70; rohit . level = 'A'; → object initialising
- cout << rohit . health << " , " << rohit . level ; → 70, A
- rohit . print () ; → Rohit

⊡ class → template / Blueprint → object → Real objects

⊙ Note →
In case of empty class, it has a memory size of 1 byte.

⊙ Note →
using dot (.) operator we initialize and get object values.

⊙ Access modifiers → It defines the accessibility of properties and behaviour that lie within a class.

→ public :→ The properties and behaviour that lie below public can be accesed both inside and outside the class.

→ private → The properties and behaviour below private can be accesed only inside class.

→ protected → The properties and behaviour below protected can be accesed only ~~out~~ inside its own class and its child class can also use it.

⊙ Note :→ By default the access modifier inside our class is private.

⊙ Getters and Setters :→

- It reads the private properties and behaviours.
- It make changes in private properties.

⊡ Snippets →

```
class Hero {
    private:
        int Health;
    public:
        void setHealth (int h, int password){
            if( password == 1234){
                Health = h;
            }
        }
        int getHealth (){
            return Health;
        }
};
int main (){
    Hero rohit;
    rohit.setHealth ( 90, 1234);
    cout << rohit.getHealth(); → (90)
}
```

For not loosing the privacy of ~~prote~~ private files, we can add password conditions in our setter while changing values.

⊙ Static and Dynamic allocation in OOPs ⇢

// Static Allocation

```
Hero a;
a.setHealth(80);
a.setLevel('B');
cout << a.level << "," << a.health;
```

// Dynamic Allocation

```
Hero *b = new Hero;
(*b).setHealth(80);
b->setLevel('B');
cout << (*b).Health << b->level;
```

`Hero *b = new Hero;` → object Creation

`(*b).setHealth(80);` and `b->setLevel('B');` → object initialising ⇢ 2 ways

⊙ Note ⇢

⊡ padding ⇢

Adding one or more empty bytes between memory addresses which are allocated for other data members to align data in memory location. This concept is called padding.

⊙ Constructor ⇢

⊡ Constructionor ⇢

- It gets invoke at time of object creation.
- It has no return type.
- Default constructors have no parameters even.
- It's name is same to our class name.

⊙ Note ⇢

when we create our own constructors in class then it over write s the default constructor.

⊙ Snippets →

```
class Hero {
    int health;

    // Default Constructor
    Hero() {
        cout << "Rohit << endl;
    }

    // Parameterised Constructor
    Hero (int health) {
        this -> health = health;
    }
};

int main () {
    Hero rohit1 (10);
    Hero * rohit2 = new Hero (11);
    return 0;
}
```

⊙ Note →
We can build multiple parameterised constructor but they should have different number or type of parameters
↓
Also called constructor overloading.

this Keyword
↳ It is same as pointer and stores memory address of the current object

Sending values to parameterised constructor statically and dynamically.

⊡ Copy Constructor → ⚠ It is present as default inside our class.

```
Ex → Hero rohit1 (10)
     Hero rohit2 ( rohit1);
```

All properties of rohit1, is copied in rohit2.

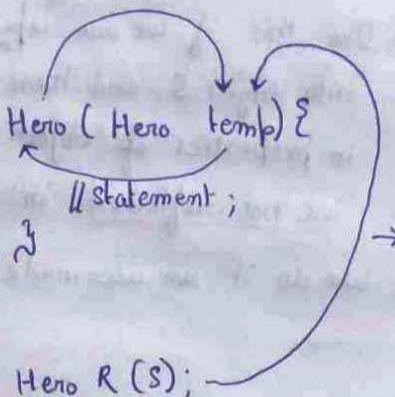
⚠ Creat your own copy constructor →

In copy constructor we do pass by reference.

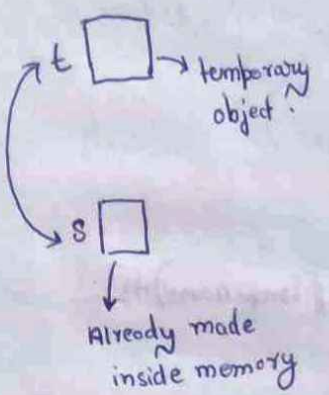
```
Hero ( Hero & temp) {
    this -> health = temp. health;
}
```

⊙ Note :→

If we pass copy constructor as pass by value instead of pass by reference, then



→ so what happens is there becomes a infinite loop between copy constructor and temporary object.



⊙ Shallow and Deep Copy :→

⊡ Shallow Copy :→ In shallow copy we acces the same memory. Example - we have an object R whose name property is pointer (char *name), storing the value as "Rohit". And when we do shallow copy of object R & in object S. Then name property of object S become Rohit. Now, if we change the name property of object R, that change is also reflected in object S, as we are accesing the same memory.

⊙ Note :→

Our default copy constructor does shallow copy.

⊡ Deep Copy → △ Here we dont access the same memory, rather we make new memory here and make changes into it.

△ Due this if we are copying object R into object S, and then making changes in properties of object R, then changes are not reflected in object S.

△ We do it in usermade copy constructor.

⊡ Example :→

```
Hero (Hero & temp) {
    char * ch = new char [strlen(temp.name)+1];
    strcpy (ch, temp.name);
    this → name = ch;
}

Hero S(R);
```

Here new memory is created, and the values are copied in new memory and then new memory is copied to our object S.

⊙ Copy Assignment Operator → ⊡ If two objects are already created then, we can simply copy one object to another using this operator ("=").

⊡ Example → A=B ; Here all properties of object B is copied in object A.

⊙ Destructor → ⊡ To deallocate our memory

destructor is used.

⊡ has no return type, parameter and name is equal to the class name.

⊡ Example →

```
~Hero () {
    cout << "Destructor called";
}

int main () {
    Hero a;
    Hero *b = new Hero();
    delete b;
}
```

For static destructor is called automatically.

For dynamic destructor is called manually by us.

⊙ Static Keyword →

- It creates a property / data member which only belongs to class.
- It is object Independent, i.e., we donot need object to access it

⊡ Example →

```
class Hero {
    static int timetocomplete;
};
int Hero :: timetocomplete = 5;

int main () {
    cout << Hero:: timetocomplete ; → 5
}
```

Static property / data member

Scope Resolution operator.

initialising values

Accessing static properties.

⊡ Static functions →

- no need of object creation, it is a function of class only.
- this keyword is not used here as it is a pointer to a object. if there is no object, no requirement of it.
- It can only access static property/ data members.

Example →

```
class Hero {
        static int health;
static int random() {
        return health;
}; }
int main() {
        cout << Hero :: random() << endl;
        return 0;
}
```

⊙ Encapsulation → Information / Data Hiding → For Security Reasons

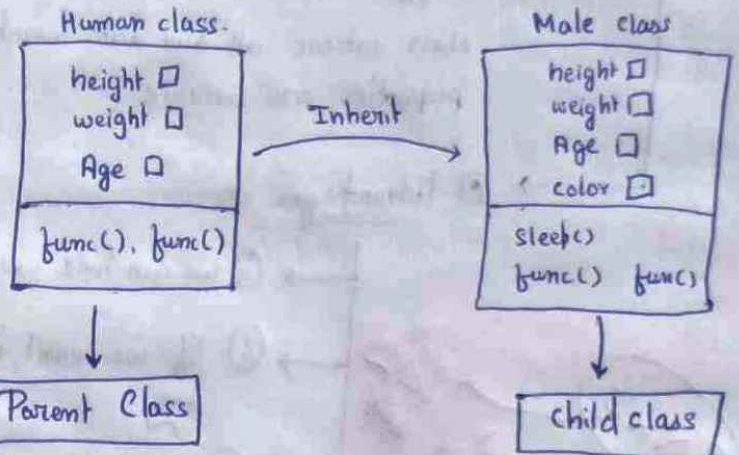
- ⊡ It is basically wrapping up data members and functions inside a single entity, ie., class.
- ⊡ Fully encapsulated class is a class where all our data members / properties are private.
- ⊡ Advantages →
  - (i) we can hide our data → security increases.
  - (ii) If we want, we can make our class read only.
  - (iii) Code Reusability.
  - (iv) It helps in unit testing, due to which things become manageble.

⊡ Example →

```
class Student {
private:
    string name;
    int roll;
public:
    int getAge () {
        return this → age;
    }
};
int main () {
    student first;
    first . getAge ();
}
```

Inheritance :→

- Basically a mechanism by which we can derive a class from another class for a hierchy of classes that share a same properties and behaviour.
- Example :→



It can inherit all properties and all behaviour of our parent class

Code :→

```
class Human {
public:
    int height;
    int weight;
    int age;
    int getage() {
        return this → age;
    }
    void set age (int n){
        this → age = n;
    }
};
class Male : public Human {
public:
    string color;
    void sleep() {
        cout << "Sleeping";
    }
};
```

Parent → Male class / Super Class

mode of inherit

parent class

→ child class / sub class

```
int main (){
    Male rohit;
    cout << rohit.height;
    cout << rohit.weight;
    rohit.setage(40);
    rohit.sleep();
    return 0;
}
```

⊙ Super Class → Access modifier of property. → mode of inheritance → Sub class

1. (i) public + public → public.
   (ii) public + protected → protected.
   (iii) public + private → private.

2. (i) protected + public → protected
   (ii) protected + protected → protected
   (iii) protected + private → private.

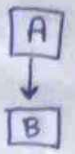
3. (i) private + public → private
   (ii) private + protected → private
   (iii) private + private → private.

⊡ Types of Inheritance →

- (i) Single
- (ii) Multi-Level.
- (iii) Multiple.
- (iv) Hybrid
- (v) Hierarical.

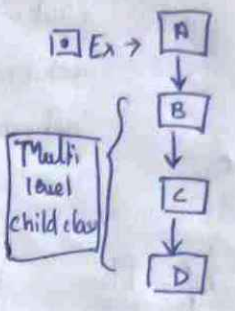
① Single Inheritence → ⊡ Our Super/Parent class can have only 1 child / ~~base~~ Sub class.

⊡ Ex →



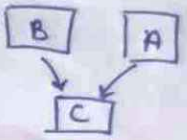
② Multilevel Inheritence → ⊡ If a Super class have multi levels of chid class then, it is called as Multilevel Inheritence.

⊡ Ex →



③ Multiple Inheritence → ⊡ If child class has multible parent classes, then it is said to be Multible Inheritence.

⊡ Example →



⊡ code :

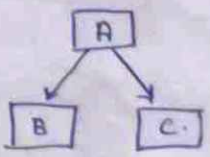
```
class A {
    // statement ;
}

class B {
    // statement ;
}

class C : public A, public B {
    // statement ;
}
```

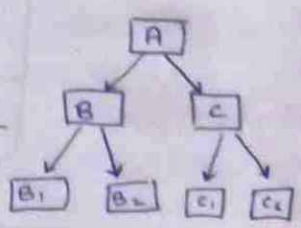
④ Hierarical Inheritance → ⊡ If a parent class have more than 1 child class, then it is cocid to Hierarical inheritance.

⊡ Example →



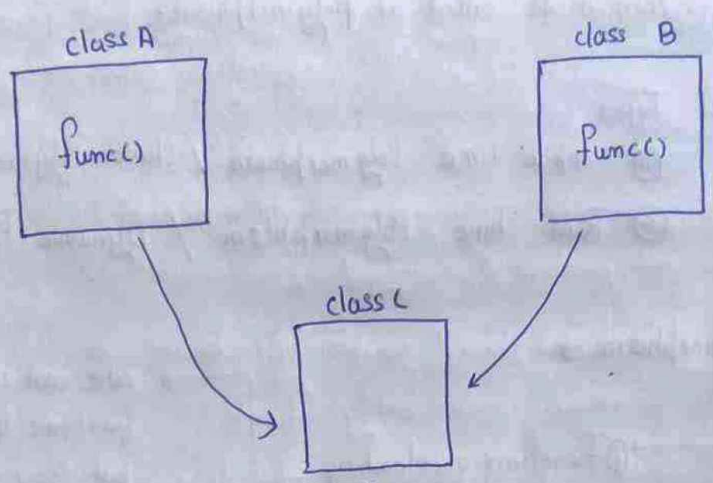
⑤ Hybrid Inheritance → ⊡ Combination of more than 1 above types of Inheritance is called as Hybrid Inheritance

⊡ Example →



← combination of multi level multiple and Heirarical Inheritance.

⊙ Inheritance Ambiguity →



Now, if we make an object using class C. and try to acces function func(), now there would be a problem as both of our parent has this function func(), So, this problem is called as Inheritence Ambiguity.

⇓

It is resolved using scope resolution operator. (::).

⊡ Code →

```
class A {
    public:
    void func () {
        cout << "Rohit";
    }
};
class B {
    public:
    void func () {
        cout << "Rohit";
    }
};
class C : public B public A {
};
int main () {
    C obj;
    obj. A:: func ();
}
```

→ Rohit

⊙ Polymorphism → ⊡ If something exists in multiple forms then it is called as polymorphism.

Poly → many
morphism → forms.

⊡ Types ....
① Compile time Polymorphism / Static Polymorphism
② Run time Polymorphism / Dynamic Polymorphism

⊡ Compile time Polymorphism :→

→ (i) Function overloading → we can make multiple functions having same name, but their parameters should be different in number or datatypes.

→ (ii) Operator overloading:
→ Except (::, .*, ., ?:), we can overload all operators, and make them do our own custom operation.

→ Example:

```
Class B {
    public:
        int a;
        void operator + (B &obj){
            int val1 = this -> a;
            int val2 = obj.a;
            cout << val2 - val1;
        }
        void operator () (){
            cout << "Dog";
        }
};
int main() {
    B obj1, obj2;
    obj1.a = 4;
    obj2.a = 7;
```

⊡ Snippets for operator overloading.

```
obj1 , obj2 ;  ⟶ (3)
obj1 ();       ⟶ (Dog)
return 0;
}
```

Notes ⇒

For binary operators:

a + b

a → current object.

b → input argument

Run time Polymorphism ⇒

→ Method overiding.

→ Rules

1. Name of method should be same in both parent and child class.
2. ~~&~~ Parameter of methods should also be same in both
3. It can be done in Inheritance only.

Example ⇒

```
class Animal {
    public:
    void speak() {
        cout << "speak";
    }
};
class Dog : public Animal {
    public:
    void speak() {
        cout << "Bark";
    }
};
int main() {
    Dog obj;
    obj.speak();
    return 0;
}
```

→ (Bark)

Here as child class has its own function speak so, produces its own output. If it won't had its own ~~output~~ function then, it had used speak() of it parent class.

⊙ Abstraction → Hides underlying high complexities from user.

→ Implementation hiding. (Basically hiding background/unnecessary details/complexities from user).

→ Advantages →

C++
- (i) Access specifiers
- (ii) header files

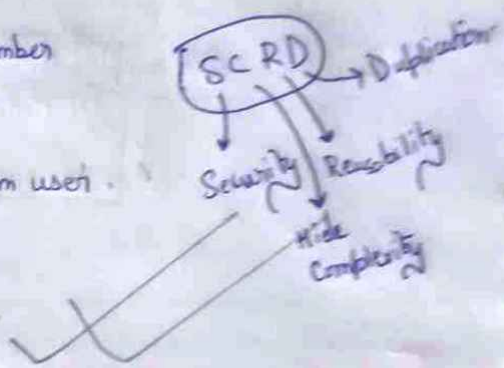
Java
- (i) Interfaces
- (ii) Abstract classes

(i) Only we can make change to data member and function (security ↑es.)

(ii) hide background complexities from user.

(iii) Increase reusability of code.

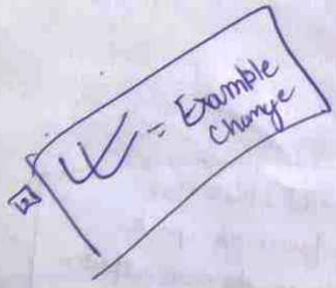
(iv) Avoids ✗ duplication of code

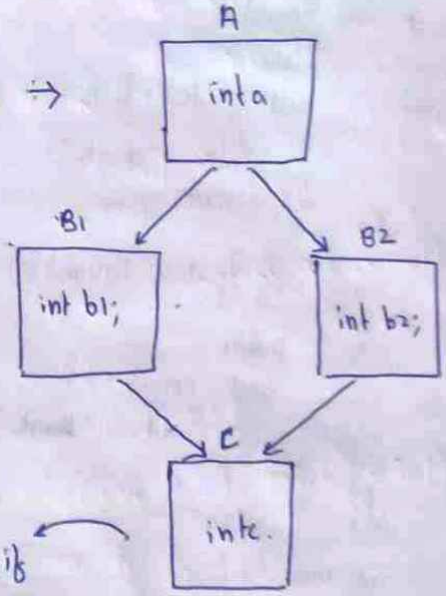


⊙ Diamond Problem in inheritence →

→ Happens in Hybrid Inheritence.

→ Example →





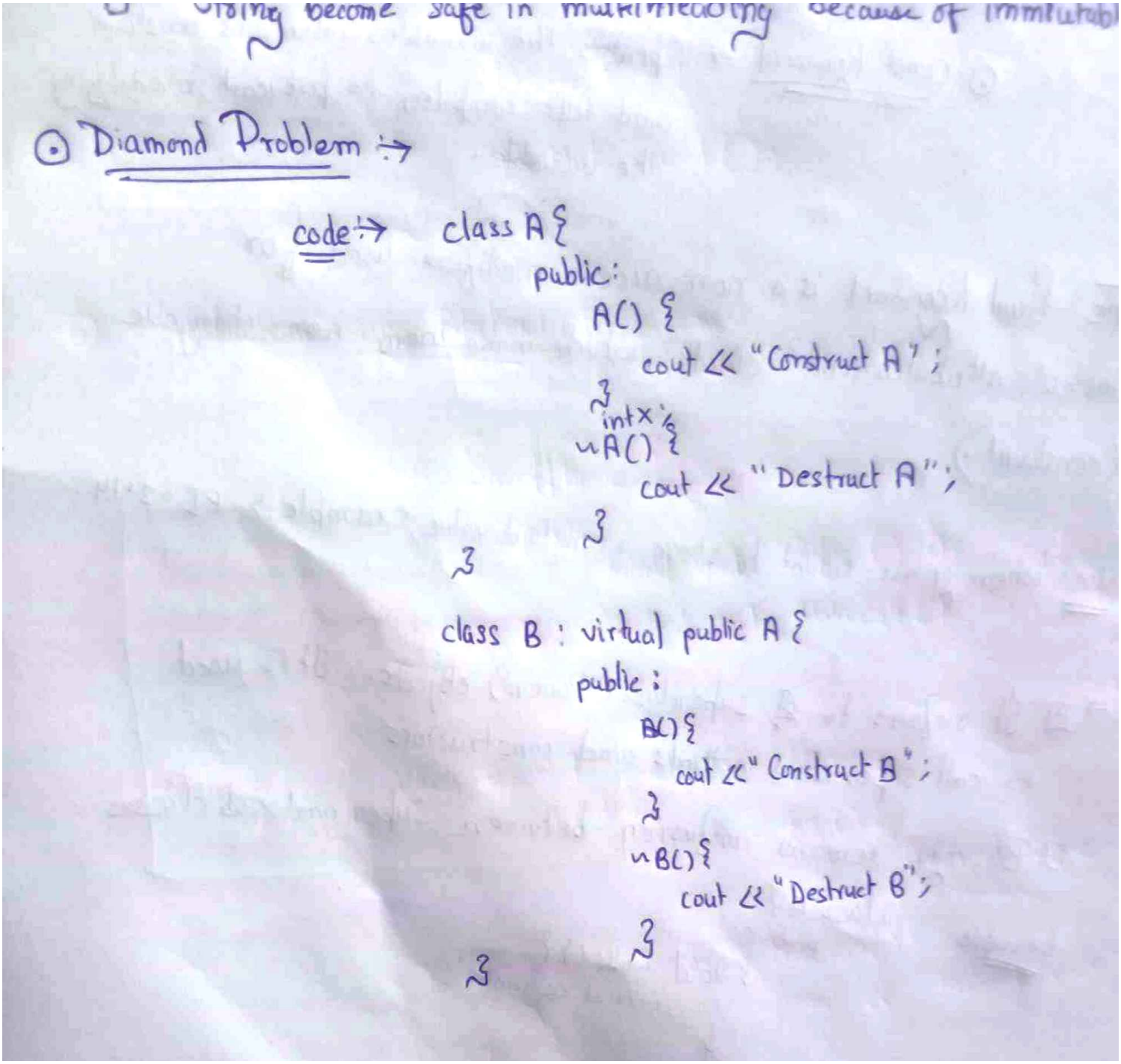
Now here, if we make an object of class C, then it would have 5 variables (a, b1, a, b2, c) instead of 4 variables (a, b1, b2, c)

This leads to a ambiguous error as we are accessing variable a 2 time.

This concept is called as Diamond Problem.

⊙ Minimising Diamond →

(i) Only by using virtual functions.



String become safe in multithreading because of immutab

⊙ Diamond Problem :→

code :→

```
class A {
    public:
        A() {
            cout << "Construct A";
        }
        int x;
        ~A() {
            cout << "Destruct A";
        }
};

class B : virtual public A {
    public:
        B() {
            cout << "Construct B";
        }
        ~B() {
            cout << "Destruct B";
        }
};
```

```
class C : virtual public A {
    public:
        C() {
            cout << " Construct C";
        }
        ~C() {
            cout << " Destruct C";
        }
};

class D : public B, public C {
    public:
        D() {
            cout << " construct D";
        }
        ~D() {
            cout << " Destruct D";
        }
};

int main() {
    D obj1;
    obj1.x = 10;
}
```

output :→

```
construct A
construct B
construct C.
construct D.
Destruct D
Destruct C
Destruct B.
Destruct A.
```

→ And called twice is ensured by virtual keyword. and this inheritence is called virtual inheritence.

```
    int b2;
    a=5;
};
class C: public B2, public B1 {
public:
    int c;
    cout << a;
};
```

have 4 variables only (a, b1, be, c)

⊙ Pure virtual function :→ (Abstract Classes)

⊡ Abstract classes → △ Classes whose objects cannot be made are called as abstract classes.

△ In C++, we create abstract classes using pure virtual functions.

Example → `Virtual void print () = 0;`

⊙ Note →

If our classes have atleast 1 virtual function, then they are known as Abstract classes.

△ To access properties of abstract classes we can create child class of it and access its properties.

△ If we are creating a child class of an abstract class, then we need to define the pure virtual function of abstract class there. Else child class also becomes a abstract class.

⊙ Abstract classes example →

```
class Instrument {
    public:
        virtual void MakeSound() = 0;
};

class Piano : public Instrument {
    public:
        void MakeSound() {
            cout << "Piano Playing";
        }
};

int main() {
    Piano P;
    P.MakeSound();
}
```

abstract class.

pure virtual function

In child class we defined pure virtual function so as to avoid it from getting/becoming an Abstract class.

⊙ Abstraction using Abstract classes →

```
class Smartphone {
    public:
        virtual void Selfie() = 0;
        virtual void MakeCall() = 0;
};

class Android : public smartphone {
    public:
        void Selfie() {
            cout << "Android Selfie";
        }
        void MakeCall() {
            cout << "Android Call";
        }
};
```

user will only know this functionality not the underlying complexity.

```
class iphone : public smartphone {
    public:
        void selfie() {
            cout << " Iphone Selfie";
        }
        void MakeCall() {
            cout << " Iphone Call";
        }
};

int main() {
    Smartphone* S1 = new Smartphone();
    S1 -> Selfie();
    S1 -> MakeCall();
}
```

~~Header files~~ →

output →

Iphone Selfie
Iphone Call

⊙ Interfaces :→

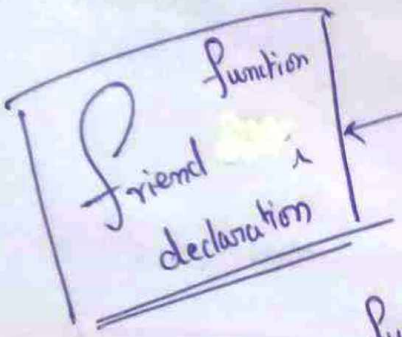
- ⊡ Interfaces are just like a class, which only contains abstract methods (pure virtual functions.), and we cannot create objects of it.
- ⊡ In Java, we achieve it using keyword `interface` implementation. But, our C++ does not have any implementations of interface. {We achieve it using abstract classes.}.

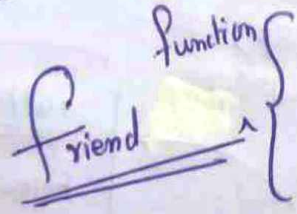
⊙ friend functions example →

Code →

```
class Rectangle {
    private:
        int length;
        int breadth;
        int area;
    public:
        void setRectangle (int l, int b) {
            length = l;
            breadth = b;
            area = l * b;
        }
        friend void PrintArea (Rectangle);
};

void PrintArea (Rectangle R1) {
    cout << "Area: " << R1.area << endl;
}
```





```
int main(){
    Rectangle R1;
    R1.setRectangle(10,20);
    PrintArea(R1);
}
```

output :→
Area: 200

⊙ Friend class :→

code :→

```
class Rectangle {
private:
    int length;
    int breadth;
    int area;
public:
    void setRectangle(int l, int b) {
        length = l;
        breadth = b;
        area = l*b;
    }
    friend class ChotaRec; }→ // friend class declaration.
};
```

Note :→
Basically the idea behind friend class or a friend global function is that they can access the private or protected, data variables or functions from the class which is has a friend relation with them.

friend class

```
class ChotaRec{
public:
    void PrintArea(Rectangle R1){
        cout << "Area: " << R1.area;
    }
};
```

```
int main(){
    Rectangle R1;
    R1.setRectangle(20,20);
    ChotaRec r1;
    r1.PrintArea(R1);
}
```

output :→
Area: 400

⊙ Smart Pointers →

- A smart pointer is a wrapper class over a normal pointer with an operator like * and → overloaded.
- They deallocate memory automatically.
- Types of smart pointers ....
  - (i) unique_ptr
  - (ii) shared_ptr
  - (iii) weak_ptr

  → For using this smart pointers we must import memory module.

```
#include <memory>
```

(1) Unique_ptr →

- Creation →

```
unique_ptr<int> unPtr1 = make_unique<int>(25);
```

- unique_ptr cannot be shared, i.e. two unique pointers cannot store same memory address.
- We can move the ownership of one unique_ptr to another unique_ptr.

Example →

```
unique_ptr<int> unPtr1 = make_unique<int>(25);
unique_ptr<int> unPtr2 = move(unPtr1);
cout << *unPtr1;   → Error => as after removing ownership this pointer points to Null
cout << *unPtr2;   → 25
```

⚠ Example 2 →

```
class Dog {
public:
    Dog() {
        cout << "Constructor invoked";
    }
    ~Dog() {
        cout << "Destructor invoked";
    }
};
int main() {
    {
        unique_ptr<Dog> unPtr1 = make_unique<Dog>();
    }
}
```

Object made using unique_ptr

Output :→

Constructor invoked
Destructor invoked.

(2) Shared_ptr →

- We can share the shared pointer, i.e ~~two~~ multiple shared pointers can have same memory address.
- A shared pointer has the count of all of the owners with whom it shares its memory address. We can access the count by use_count().

Example →

```
int main() {
    {
        shared_ptr<Dog> P1 = make_shared<Dog>();
        shared_ptr<Dog> P2 = P1;
        cout << P1.use_count();
    }
    cout << P1.use_count();
}
```

Output →

constructor invoked
2
destructor invoked
0

Remember → In shared pointer when the scope of all shared_ptr gets over, then only destructor invokes.

③ **Weak_ptr** →

↳ Weak_ptr does not keeps the object alive, whereas as a ~~the~~ shared_ptr keeps the object alive. (It means that reference count/use_count() does not gets increased even if we shared ownership of shared_ptr to weak_ptr).

⊡ Example →

```
int main(){
    weak_ptr<int> W1;
    {
        shared_ptr<int> S1 = make_shared<int>(25);
        W1 = S1
        cout << *W1;
    }
    cout << *W1;
}
```

→ 25

→ Error

As weak_ptr has a weak hold over the reference, so when the scope of shared_ptr ends which has a strong hold on reference, the weak_ptr too loses the reference.

⊙ Singleton Design Pattern →

- Singleton class / Singleton design pattern is basically a software design principle that is used to restrict the instantiation of a class to one object.
- Uses → Singleton classes are used in
  - (i) logging
  - (ii) drivers object
  - (iii) caching → process of storing data in cache.
  - (iv) thread pool.
    - → manage collection of threading available to perform specific task.

(LDCT)

- Example →

```
class Singleton {
    static Singleton * S1;
    int data;
    Singleton () {
        data = 0;
    }
public:
    static Singleton * getS1 () {
        if (!S1) {
            S1 = new Singleton;
            return S1;
        }
    }
    int getData () {
        return this -> data;
```

```
void setData (int data) {
    this -> data = data;
}
};

Singleton * Singleton :: s_ = 0;
```

Initialize pointer to zero so that it can be initialized in first call to getInstance.

```
int main () {
    Singleton * S = S-> getS_();
    cout << s. getData(); ---> (0)
    S-> setData (100);
    cout << S. getData(); ---> (100)
}
```

Final + Super keyword include

⊙ Const Keyword → Specifies the variables values is constant and tells compiler to prevent modifying the variable.

⊙ final Keyword → ⊡ The final keyword is a non-access modifier used for classes, attributes and methods, which make them non-changeble (constant)

⊡ Used when you want to store a constant value, example → PI = 3.14.

⊙ Super Keyword → ⊡ It refers to superclass (Parent) objects. It is used to call super class methods and constructors.

⊡ Basically, removes confusion between Super and sub classes.

Example

```
class A {
public:
    void walk () {
        cout << "walk";
    }
};
class B: public A {
public:
    Super.walk();
};
```

⊙ Why are Strings immutable in Java?

ans:- ⊡ Because of security, synchronization, concurrency, caching, and class loading. (SSCCC). ~~The reason of making~~

⊡ String pool cannot be possible if string was not immutable.

⊡ String become safe in multithreading because of immutableness.