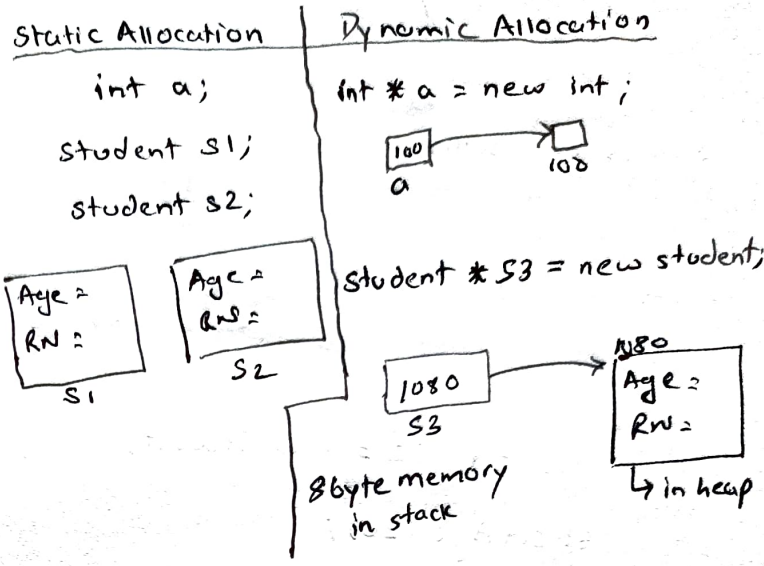
OOPS

Classes are user defined data types.

```
Class student {
    int rollNumber;
    int age;
};
```




```
int main ()
{
    student S1;
    student S2;
    student S3, S4, S5;
```

Creates object statically

```
    student *S6 = new student;
```

Creates object dynamically.

```
    S1.age = 24;
    S1.RollNumber = 101;
    S2.age = 30;
    (*S6).age = 23;
    (*S6).rollNumber = 104;
```

Both are same

```
    S6 -> age = 23;
    S6 -> roll Number = 104;
}
```

Access modifiers

- public
- private (by default)
- protected

```
S1.display();
(*S6).display();
S6 -> display();
```

Call display function.

Getters and Setters

```
int main()
{
    Student S1;
    Student* S2 = new Student;
    S1.setAge(20);
    S2->setAge(24);
    S1.display();
    S2->display();
}
```

```
class Student {
    private:
        int age;
    public:
        int rollNumber;
    void display() {
        cout << age << rollnumber;
    }
    int getAge() {
        return age;
    }
    void setAge(int a) {
        age = a;
    }
```

Using this function we are able to access → private member from main class.

Constructors

- Same name as class
- No return type
- No input arguments.

} Default

Default constructor →

```
Student()
{
}
```

Whenever a class is created, constructor is called internally. Assigning garbage values to variable.

```
Student (int r)
{
  rollNumber = r;
}
```

Parametrised Constructors

```
int main ()
{
  Student s1 (10);
  Student * s2 = new Student (11);
}
```

①
```
Student (int a)
{ };

Student (int a, int r)
{
};
```

```
int main ()
{
  student s1;
};
```

The above program throws an error. Since once an constructor is made, inbuilt constructor won't be ~~able to be~~ available to us. Hence, Student s1(), throws an error.

②
```
class student {
  int rollNumber;

  Student (int rollNumber)
  {
    rollNumber = rollNumber;
  }
};
```

Since the variable name and the variable of the constructor is same, there will be a problem.

The scope of the rollNumber variable will be within the constructor class. So the variable coming from main class will come and change itself to the same value, without affecting rollNumber variable of the class.

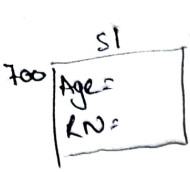
this keyword → stores the address of current ~~class~~ object.

pointer variable

So, Student (int rollNumber) {

this → rollNumber = rollNumber;

}



this → 700

(*this.rollNumber = rollNumber;) ← value passed by main

↓

variable of the object

Copy constructor →

Student S1 (10, 100);

Student S2;

S2.age = S1.age;

S2.rollNo. = S2.rollNo;

Student S2 (S1);

Both are same.

Copies the values of variables of S1 to S2.

Student *S3 = new Student (20, 2001);

Copying
- dynamic to static → Student S4 (*S3);
- dynamic to dynamic → Student *S5 = new student (*S3);
- static to dynamic → Student *S6 = new student (S1);

Copy Assignment Operator (=) →

Student S1 (10, 100);

Student S2 (20, 200);

S1 = S2;

Student *S3 = New student (30, 300);

*S3 = S1;

S2 = *S3;

Suppose we want to copy the value of S2 into S1 after they have been created.

We use copy assignment operator. Since copy constructor could not be called after initialization

Copy constructor → used to copy values of the time

copy assignment → used to copy values after the initialization.

Destructor →

```
~Student() {

}
```

objects in static memory will be deallocated automatically but not for objects in dynamic memory.

```
delete s3;   // Destructor called.
```

⊙ `Student s5;` `s5 = s4;` | `Student s5 = s4;` | `Student s5(s4);` copy constructor called

All of them are same.

Fraction Class

```
class Fraction {
  private:
    int num;
    int denom;
  public:
    Fraction(int num, int denom) {
      this → num = num;
      this → denom = denom;
    }
    void print() {
      cout << this → num << "/" << denom << endl;
    }
    void add(Fraction f)
    {
      num = (num * f.denom)
            +
            (denom * f.num);
      denom = denom * f.denom;
      simplify();
    } // updates the value of f1
};
```

f1 = f1 + f2

```
void simplify()
{
    int gcd=1;
    int j = min(num, denom);
    for(int i=1; i<=j; i++)
        if(num%i==0 && num%j==0)
            gcd=i;
    num = num/gcd;
    denom = denom/gcd;
}
```

(b)

```
void multiply(Fraction const &f)
{
    numerator = numerator * f.numerator;
    denominator = denominator * f.denominator;
    simplify();
}
```

→ by passing the reference, we have saved space and also the unnecessary creation of a new space for a new object. Since we can pass the reference to the object, there is a possibility of modifying it. Hence it is passed as const. so that any change through f is not allowed.

Shallow and Deep Copy

```
class Student {
    int age;
    char *name;

public:
    Student (int age, char *name) {
        this -> age = age;
        this -> name = name;
    }

    void display () {
        cout << name << " " << age << endl;
    }
};

int main()
{
    char name[] = "abcd";
    Student s1(20, name);
    s1.display();

    name[3] = 'e';
    Student s2 (24, name);
    s2.display();

    s1.display();
}
```

→ Shallow Copy

Deep Copy

```
this -> name = new char[strlen(name)+1]
strcpy(this -> name, name);
```

Creates a new char array and copy the contents into it.

// 20 abcd

// 24 abce

// 20 abce

Since the address was passed only in place of entire array, any changes done is reflected in the copy of both the objects. This is called shallow copy, should be avoided.

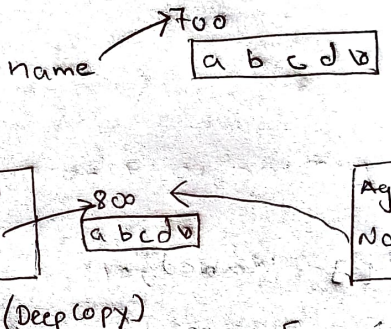
Copy Constructor

```
int main()
{
  char name[] = "abcd";
  Student s1(20, name);
  Student s2(s1);
  s2.name[0] = 'x';
  s1.display();
  s2.display();
}
```

```
class student {
  student(int age, char *name)
  {
    // this->age = age;
    // this->name = name;
    // Deep copy
    this->name = new char
                 [strlen(name)+1];
    strcpy(this->name, name);
  }
```

SHALLOW COPY

Output →
abcd
xbcd
xbcd



∴ inbuilt copy constructor is shallow copy

```
// Copy constructor
Student(Student s)   // Improv. needed)
{
  this->age = s.age;
  // this->name = s.name;   // Shallow copy
  // Deep copy
  this->name = new char[strlen(s.name)+1];
  strcpy(this->name, s.name);
}
```

We should use this

Student (Student s)

Student s = main.s1 // So this is copy constructor.

But since we have made our own copy constructor, we have lost access to the inbuit one.

So, (student s) will call

Student (student s), so it will be an infinite loop.

Hence, to avoid this. We need to receive the argument as reference, so that copy constructor is not used.

Hence, student (student &s) // more improv needed.

{

.

.

}

For example → int i = 5
int &j = i;

j++;
i++;
} Both are allowed

So, s.age--;
↳ this changes the value of age in main function to avoid this const func is used.

Student (Student const &s) → No changes allowed. Only for reading

{

.

.

}

Output ⇒ abcd 20
abcd 20
xbcd 20

Initialisation list

```
Class Student {

public:

    int age;

    const int rollNumber;

Student (int r) : rollNumber (r)
{

}
```

```
int main()
{
    Student S1;        → this will throw a error at this line
    S1.age = 20;
    S1.rollNumber = 101;
```

```
Student S1 (101);
S1.age = 20;
```

const int rollNumber = r.

```
Student (int r, int a) : rollNumber (r), age (a) {}
{
}
```

// Both of them are valid, for const or normal variables.

if we want to set constant variables of all the objects, we use initialisation list.

```
Student (int r, int age) : rollnumber (r), age (age)
{
}
```

This is valid, no need of this → age (age);

for reference variables →

```
Class Student {

public:

    int age;

    const int rollNumber;

    int &n;

Student (int r, int age) : rollNumber (r), age (age),
                           n (this → age).
{

}

};
```

Constant Functions

```
int main()
{
    fraction f1(10, 2);
    fraction f2(15, 4);
    fraction const f3;
    cout << f3.getNumerator() << " " << f3.getDenominator() << endl;
    f3.setNumerator(10);
}
```

Through constant objects you can only call constant functions

Throws error here

constant func^{n}s → which doesn't change any property of current object.

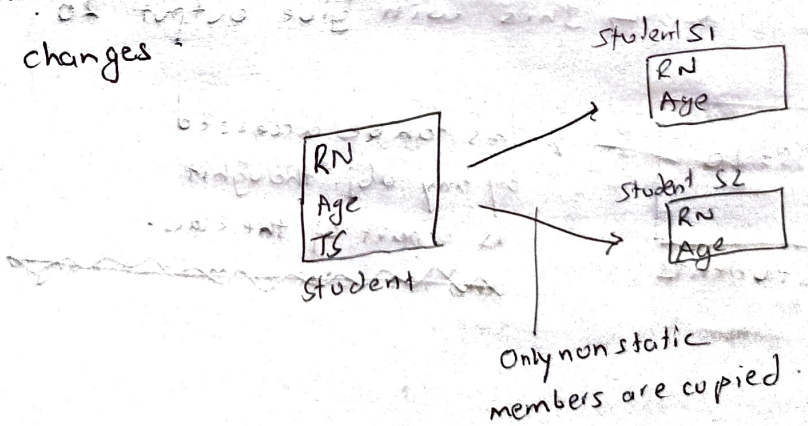
```
int getNumerator const
{
    return numerator;
}
```

```
int getDenominator const
{
    return Denominator;
}
```

Static Members

the properties that aren't different for every object, but they belong to the class. That is, irrespective of the objects created its value never changes.

```
Class Student {
public:
    int rollNumber;
    int age;
    static int totalStudents;
};
```



how to access static members →

① cout << student :: totalstudents;

scope resolution op^r.

The static members are initialized outside the class as →

② int student :: totalStudents = 0;

- int: type of data member to be written again
- student: class name
- totalStudents: name of data member.

If we write →

student S1;

S1.totalstudents;

This wont throw error, but it is logically incorrect.

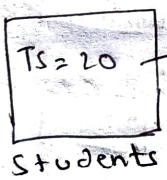
and,

S1. totalStudents = 20;

student S2;

cout << S2. totalStudents;

↳ This will give output '20'.



This can be accessed by any obj. thoughit is apart of the class.

~~and its value remains unchanged~~

③

```
Student()
{
    totalstudents ++;
}
```

```
int main()
{
    student S1;
    student S2, S3, S4, S5;
    cout << student :: totalstudents;
}
```

Output = 5

~~Even fun~~

Even functions can be made static. These functions won't belong seperately to every object rather they will be belonging to the class.

```
static int getTotalStudents() {
    return totalStudents;
}

cout << Student :: getTotalStudents();
```

- Static functions can use only static members.
- Static function don't have 'this' ~~key~~ access. Since they don't seperately belong to objects.

Operator Overloading

Suppose we want, f3 = f1 + f2;

To add two fractions and return it to new obj.

```
Fraction operator+ (Fraction const &f2) {
    int lcm = denominator * f2.denominator;
    int x = lcm / denominator;
    int y = lcm / f2.denominator;
    int num = x * numerator + (y * f2.numerator);

    Fraction fNew(num, lcm);
    fNew.Simplify();
    return fNew;
}
```

```
int main()
{
    Fraction f1(10, 4);
    Fraction f2(15, 2);

    Fraction f3 = f1 + f2;

    f3.print();
}
```

Now, `fraction f3 = f1 + f2;`

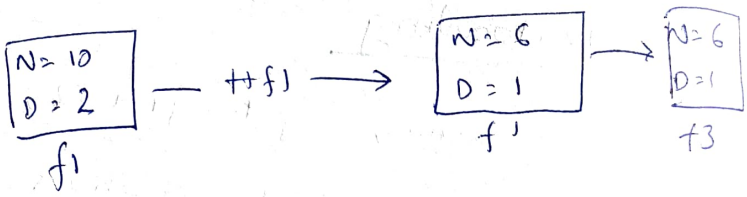
this will be passed to the function as 'this' argument. To use members of f1. we use this or simply the name

this is passed as argument

→ The operand before operator is passed as 'this' and after operator is passed as argument.

Pre-increment Operator →

We want `fraction f3 = ++f1;`



So, first modify f1, then copy to f3.

```
Fraction operator++ (){
num = num + denom;
simplify ();
return *this;
}
```

this contains the address of f1.
*this returns its contents

```
int main ()
{
  fraction f1(10,2);
  fraction f3 = ++f1;
}
```

```
int main ()
{
  fraction f1(10,2);
  f1.print();
  ++(++f1);
  f1.print();   // prints 6
}
```

```
int main()
{
  fraction f1(10,2)
  fraction f3 = ++(++f1);
  f3.print();   // prints 7
  f1.print();   // prints 6.
}
```

So, why the disperancy, ideally both should have printed 7.

(*) Why ??

```
fraction operator++ (){
  return *this
}
```

(++f1) → 700

700: N = 10, D = 2 (f1)

return this modified →

N = 6, D = 1

Now since this is not recieved and the next (++) is called. Hence the system stores it in a temporary buffer.

++(++f1)

800: N = 6, D = 1 (temp. buffer)

and on this temp. buffer the next (++) is called

(++) is called

800: N = 7, D = 1

f3 = ++(++f1)

and then this is copied to f3.

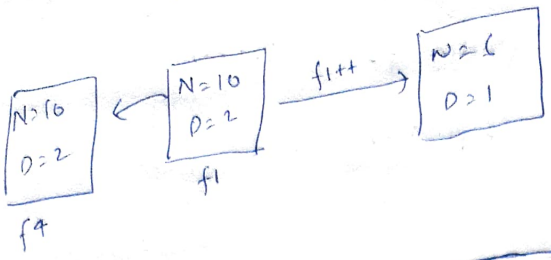
That is why f3 changed to 7 and f1 is still 6.

(·) And in order, to change both f1 & f3 so as they both share the same value. We need to pass the reference of f1, so that the reference is never created.

```
Fraction & operator++ () {
```

Post-increment operator

Fraction f4 = f1++;



This is how differentiate post increment operator from pre increment

```
Fraction operator++ (int) {
    Fraction fNew (numerator, denominator)
    numerator = numerator + denominator;
    Simplify (); fNew.Simplify ();
    return fNew;
}
```

```
int main ()
{
Fraction f1(10,2);
Fraction f3 = f1++;
f1.print ();
f3.print ();
}
```

```
int i = 5;
(i++)++; → error
cout << i
```

[Nesting is not allowed in case of post-increment operator.]

+= operator

Nesting is allowed.

» f1 += f2
» f1 = f1+f2;

```
void operator+= (Fraction const &f2)
{
    int num = (numerator * f2.denominator) + (f2.numerator * denominator);
    int denom = denominator * f2.denominator;

    numerator = num;
    denominator = denom;
    simplify ();
}
```

Can't do nesting in this one

To allow nesting →

```
Fraction & operator += (Fraction const &f2)
{
    .
    .
    .
    return *this;
}
```

Discussed Earlier

Dynamic Array Class

```
class DynamicArray {
    int *data;
    int nextIndex;
    int capacity;          // current total size

public:  DynamicArray() {
            data = new int[5];
            nextIndex = 0;
            capacity = 5;          // initial size
         }

DynamicArray(DynamicArray const &d) {
    // this -> data = d.data;          // shallow copy
    this -> data = new int[d.capacity];
    for(int i=0; i<d.nextIndex; i++)
        this -> data[i] = d.data[i];

    this -> nextIndex = d.nextIndex;
    this -> capacity = d.capacity;
}
```

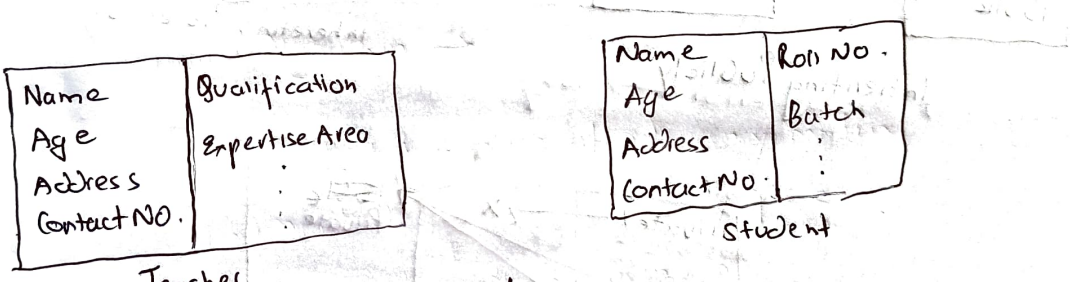
```
void add (int element)
{
    if (nextElement == capacity)
        {
            int *newData = new int [2 * capacity];
            for (int i=0 ; i< capacity ; i++)
                newdata [i] = data[i];
        }
        delete [] data;
        data = newdata;
        capacity *= 2 ;
    }
    data [nextIndex] = element;
    nextIndex ++ ;
}
```

Abstraction and Encapsulation

- Encapsulation is the clubbing / combining of datamembers and functions
- Can be done using classes.

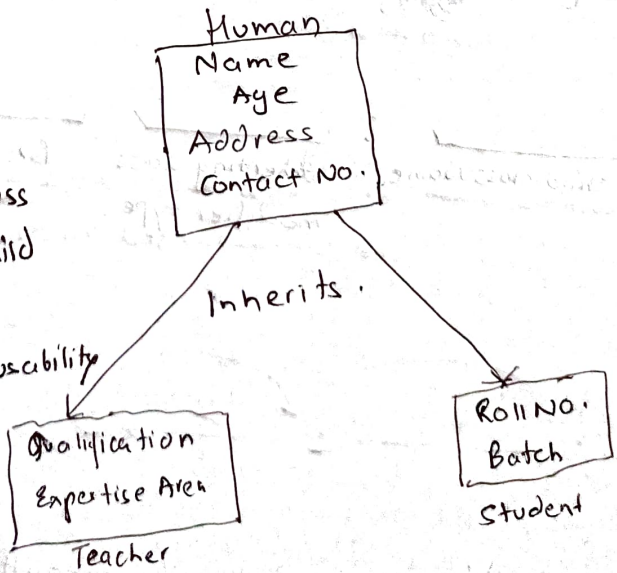
- Abstraction is the hiding of the data from the outside world.
- Examples :- inbuilt func^n, STLs like stack, queue, hashmaps, PQs.
- In classes, defining any property as private, hides them from outside world. (Abstraction).
- if any hidden property is changed/updated it should not effect the user usage of the property.
- So that user doesn't change the properties and cause the code to break

Inheritance

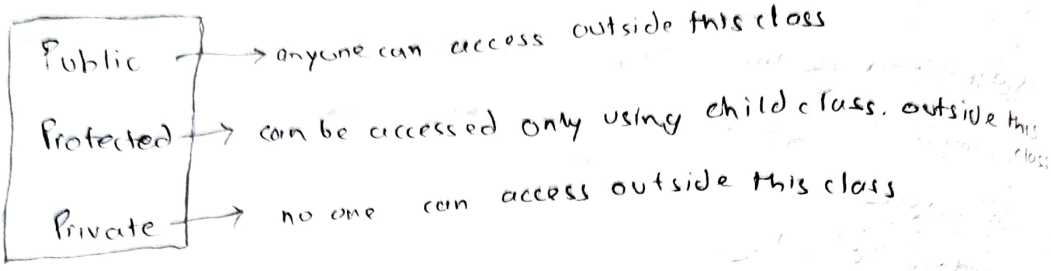


→ Instead of mentioning some properties again and again. We store them in a seperate class and inherit them to child classes.

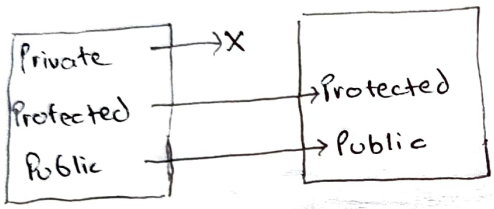
→ It increases code reusability



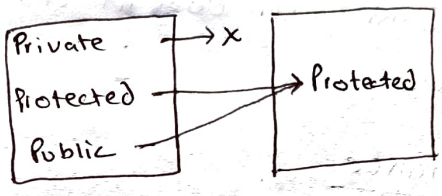
Protected Access Modifier → if any property is defined as protected that means this property is accessible outside the class only to those, who inherits them



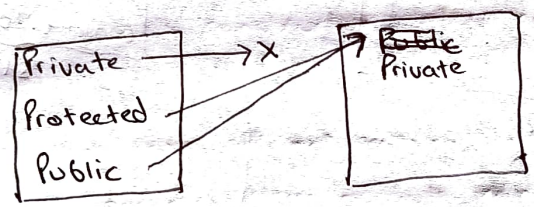
Inheritance Ways →



Inheriting Publicly



Inhereting Protectedly



Inheriting Privately.

class _____ : _____ _____
child class Name — Inherting access modifier type — Parent class Name

{

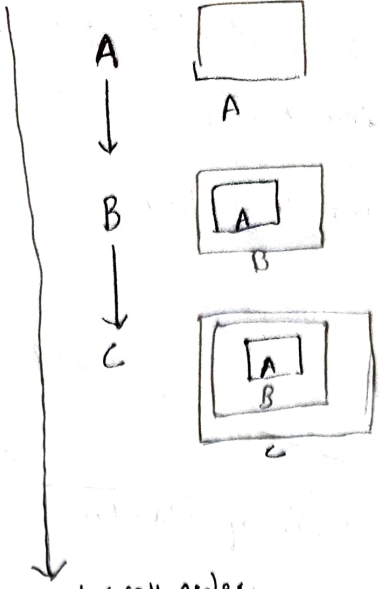
}

⊙ if no acess modifier is used, then by default inheritance will be of type private.

A a; → A()

B b; → A()
B()

C c; → A()
B()
C()



Destructor call order

constructor call order

⊙ When the base child constructor is called, it will implicity call the parent class constructor (default constructor)

```
class Vehicle
{

}
```

```
class car : public Vehicle {

car () {

}

};
```

← Before entering here Vehicle class constructor is automatically called

Internally it happens through initialisation list. We dont need to write this. It happens onits own.

```
car () : Vehicle () {

}
```

→ Implicit calling is only for default constructor.

⊙ But if the parent class has parametrized constructor, we need to write the initialisation list in derived class, otherwise we get error.

```
Vehicle
{

Vehicle (int n)
{
}

};
```

```
class car : public Vehicle
{

car () : Vehicle (5)
{
}

};
```

Initialisation list

```
int main ()
{
car c;
}
```

Ex

```
Vehicle
{
  Vehicle (int n)
  {
  }
}
```

```
Car : public Vehicle
{
  Car (int n) : Vehicle (n)
  {
  }
};
```

```
int main
{
  Car c(5);        → This works now
  HondaCity h;     ✗
  HondaCity h(10);
```

```
HondaCity : public Car
{
  HondaCity() : Car()
  {
  }
};
```

```
HondaCity : Public Car
{
  HondaCity (int y) : Car(y)
  {
  }
};
```

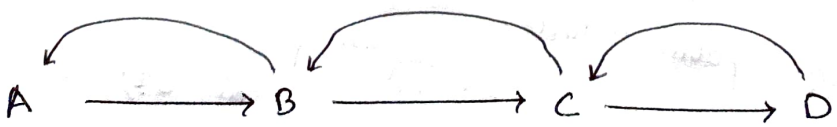
Type of constructor of parent class to be called

HondaCity h; throws error → why

- → h goes to its constructor, before calling self it calls its parent
- → Now since we are not calling and the call is implicit, it calls parent's class default constructor. Hence it throws error, since now Car class has parametrised Constructor.

HondaCity h(10) works → Since now we at each step are calling the parameterised constructor.

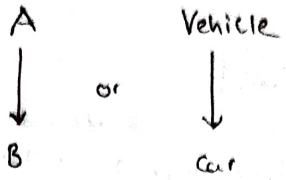
①



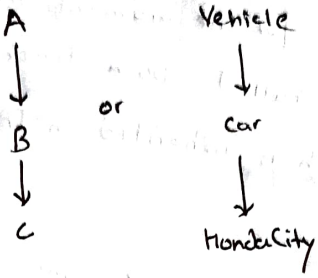
A child class can only calls its parent class

Types of Inheritance →

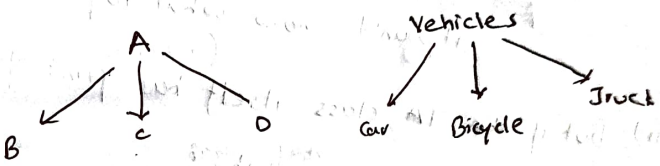
(i) Single Inheritance :



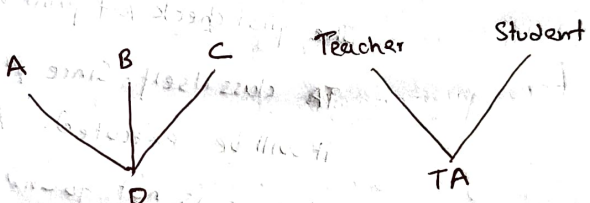
(ii) Multilevel Inheritance :



(iii) Heirarchial Inheritance :



(iv) Multiple Inheritance :

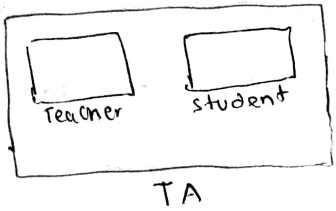



```
class Teacher
{
public: string Name
    void print()
    { cout <<"teacher";
    }
};
```

```
class Student
{public: string Name;
  void print()
  { cout << "student";
  }
};
```

```
class TA: public Teacher, public Student
{
public:
    void print()
    { cout << "TA";
    }
};
```

```
int main()
{
  TA a;
  cout << a.print();
  a.Student :: print();
  a.Teacher :: print();
  a.Student :: name = "abc";
}
```



(i) ~~Suppose~~ Since the teacher class is written at the first, its constructor will be called before the student class.

(ii) Suppose, TA class for the moment doe's nt have a print class and a.print() is called. Both teacher and student have the print() funcn, which is inherited in TA. So which print will be called?

↳ Compiler throws a error. Since they have a matching name it won't know what to pick.

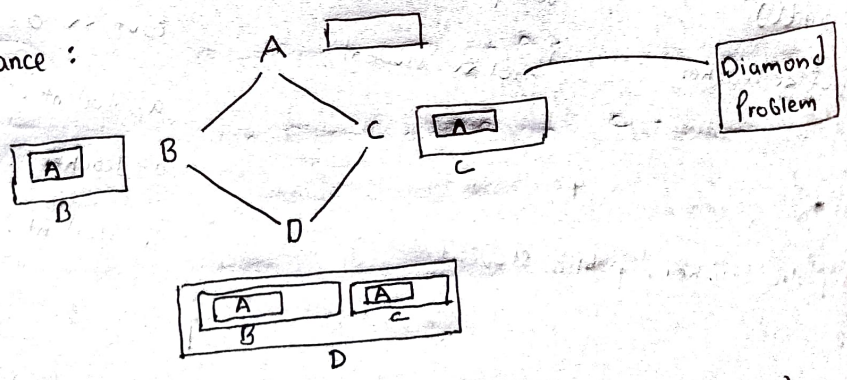
(iii) But if the TA class itself has print funcn. And a.print() is executed "TA" will be print. Why?

↓

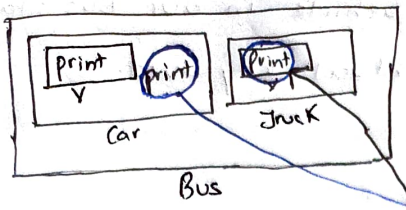
The first check of print will be done in the TA class itself. Since print is found there it will be executed. And in the case of (ii) if print is not found, then it will look into all the parent classes.

(iv) Same logic is also true for any matching data member names.

(v) Hybrid Inheritance :



Since the D class has two copies of A class inherited the compiler will have ambiguety issues



Bus b;

b. print (); ← if the bus doesnt have its own print() func^n this will throw error. Since two print func^n are there causing ambiguity.

b. print () → ⊗ error

b. car :: print () → this will print

this will call car's ecall own print first. Since it is present that gets print

b. truck :: print() → this will print

It searches for truck's own print func^n. Since it is not there it searches in its derived class. and hence, vehicle print() gets printed.

```
class Vehicle
{
  Vehicle ()
  {
    cout << "Vehicle Constructor";
  }
};
```

```
class Car : public Vehicle
{
  Car ()
  {
    cout << "Car constructor";
  }
};
```

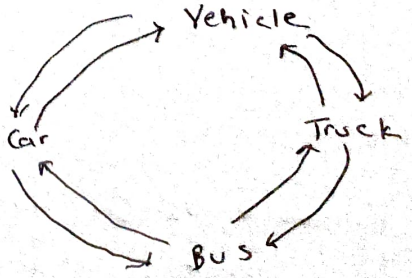
```
class Truck : public Vehicle
  {
    Truck ()
    {
      cout <<"Truck constructor";
    }
  }
```

```
class Bus: public Car, public Truck
  {
    Bus ()
    {
      cout << "Bus constructor";
    }
  };
```

```
int main ()
{
  Bus b;
}
```

Output

Vehicle constructor
car constructor
Vehicle constructor
Truck constructor
Bus constructor



Now we still have two copies of vehicle in our Bus class.

How can we avoid that. Using virtual keyword.

```
class Vehicle
{

};
```

```
Class Car: virtual Vehicle
{

};
```

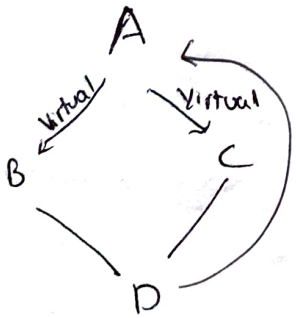
```
class Truck: virtual Vehicle
{

};
```

```
class Bus: public Car, public Truck
{

};
```

Output →

```
Vehicle Constructor
Car Constructor
Truck Constructor
Bus constructor
```

```
int main()
{
Bus b;
}
```



D → directly calls on A.

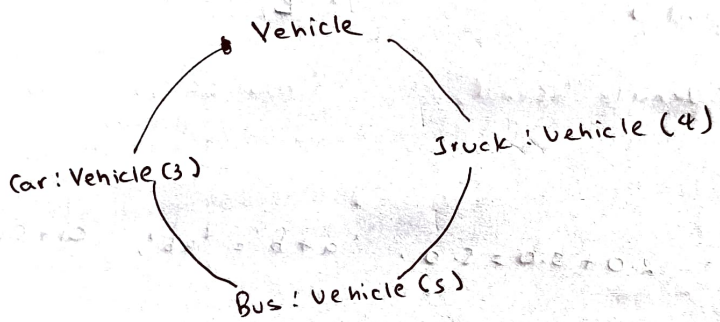
then D calls on B and C.

```
Vehicle
{
  Vehicle (int z)
  {
    cout << "Vehicle z";
  }
};
```

```
Car : public Vehicle
{
  Car : Vehicle (3)
  {
  }
```

```
Truck : public Vehicle
{
  Truck () : Vehicle (4)
  {
  }
};
```

```
Bus : public Car, public Truck
{
  Bus () : Vehicle (5)
  {
  }
};
```



Output → Vehicle parametrized constructor 5
Car constructor
Truck "
Bus "

(*) In virtual inheritance, exception is there. Bus makes direct call on its grandparent. Hence '5' gets printed rather than '3' ~~or~~ '4'. Hence calls the constructors accordingly keeping in mind that Bus will make a direct call on Vehicle, which if virtual was, not then it would have only called on car and truck.

Polymorphism : Compile Time

Polymorphism → means many forms

At compile time only, we decide how our code is gonna behave.

Achieved by →
- (i) function overloading
- (ii) Operator overloading
- (iii) Method / Function overriding.

Ex

```
int test (int a, int b)
{

}
int test (int a)
{
}
int test ()
{
}
int main ()
{

}
```

(i) ← Same func^n name, but difference in which they are used.

int test () ← cant be used in a single program together. → void test ()

[Since compiler can't differentiate on the basis of return type]

(ii) → 2 + 3 = 5 , 2.0 + 3.0 = 5.0 , 'a' + 'b' = 'ab' , c1 + c2 =

Same operator
Diff. behaviour

[Polymorphism]

(iii)

A
↓
B

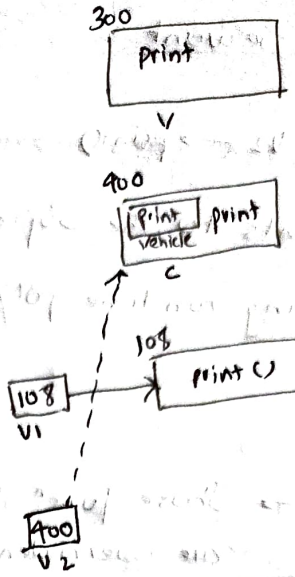
```
class vehicle
{
    void print()
    {
        cout << vehicle;
    }
};
```

```
class car : public vehicle
{
    void print()
    {
        cout << car;
    }
};
```

```
int main()
{
    vehicle v; car c;
    v.print(); // vehicle
    c.print(); // car
```

⊙ At compile time we only decide which print to call. (c.print) / (v.print) Hence it is an example of compile time polymorphism

```
int main()
{
 Vehicle v;
  Car c;
  v.print();
  c.print();

  Vehicle *v1 = new Vehicle;
   Vehicle * v2;
   v2 = &c;
```



v2 → print();

↓ prints vehicle

v1 → print()

↓ prints vehicle

}

(*) Parent (Base class) class pointer can point to derived class address. But not vice versa

(*) When something as v2 = &c is written, through v2, we can only acess properties that is in Vehicle class. Only base class properties can be accessed not child class

This is compile time polymorphism. Since at compile time we decide that what func^n/property to be used.

Polymorphism : Run Time

We decide at run time, that how we want a particular section of our code to behave.

```
vehicle * v2 = new Car;
v2 → print();
```

→ print "Vehicle

→ if print func^n is absent in vehicle class → error.

But, what if we want

v2 ⟶ print() ⟶ to print the car class func^n.

Since, v2 points to the object of car class.

We do that using run time polymorphism. It can be implemented using virtual class

Virtual func^n ⟶ Those func^n that are present in base class, that are overriden in derived class (of the same name)

```
class Vehicle
{
 public:
   string color;

  virtual void print ()
   {
    cout << " Vehicle " << endl;
   }
};
```

```
class Car : public Vehicle
{
 public:
   int numGears

  void print ()
   {
    cout << "car" << endl;
   }
```

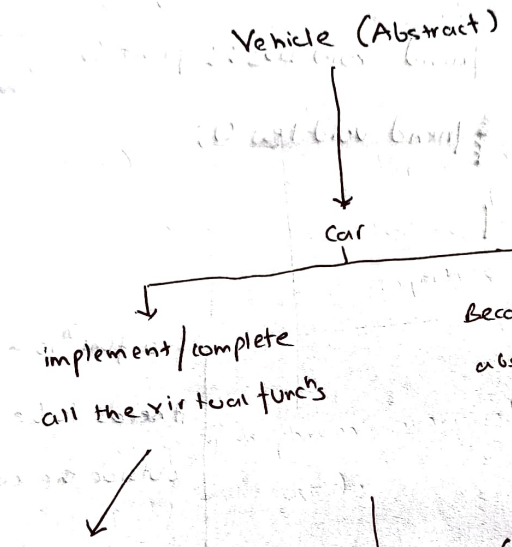
```
int main ()
{
 Vehicle * v2 = new Car;

 v2 -> print (); // Now this print "car". Since we used virtual
                 Now if we remove the print func$^n$ from car class
                 this will print "vehicle".
}
```

Pure Virtual funcⁿ → a funcⁿ that has no definition.

Example → virtual void print() = 0;

→ Any class that contains even a single a pure virtual funcⁿ, becomes abstract class.

→ We cannot create a object of abstract class. Because whenever object is created all the properties are initialized. But since virtual funcⁿ has no definition it cannot be initialized. Hence we get error if we try to create the object.




```
class Vehicle
{
  virtual void print() = 0;
};
class car : public Vehicle
{
  void print()
  {
    cout << "car";
  }
};
```

⊕ Can create the object of car class
⊕ Cannot " " " " vehicle class

```
class Vehicle
{
  virtual void print() = 0;
};
class Car : public Vehicle
{
  int numTyres;
};
```

⊕ Car becomes abstract class
⊕ Car & Vehicle both its object can't be created.

Friend Functions and Classes

A friend function can access the private and protected members of the class

```
class Bus {

public:
    void print();

};
```

Here we just specify that Bus has a print func^n

```
class Truck
{
private: int n;

protected:
    int y;

public:
    int t;

friend void Bus :: print();

friend void test();

};
```

Bus should be declared before Truck so that compiler knows that is a class called Bus.

```
void Bus :: print() {

    Truck t;

    t.n = 10;

    t.y = 20;

    cout << t.n << " " << t.y << endl;

}
```

if a func^n is a friend it needs to be defined outside the class

```
int main()
{
    Bus b;

    b.print();

    test();

}
```

→ With this now we are able to access the private & protected properties of truck class.

```
void test ()
{ Truck t;
  t.n = 10;
  t.y = 20;
  cout << t.n << "  " << t.y;
}
```

→ This is a global function not part of any class. It will work since no 'this' also has the access of private & protected members of truck class.

* friend functions does not have the access to 'this' pointer since they are not a member of any class.
* We can put friend func^n declaration under any type of access modifier. It wont have any effect.

```
class Bus {

  func1;
  func2;
  func3;

}
```

```
class Truck
{
  friend class Bus;
}
```

friend class

* Suppose we want all func^n of Bus class to have the access to all properties of truck class. We can individually mention or we can make the whole Bus class a friend of truck class.
* The friendship is one way

Truck ⟶ Bus

→ Bus is a friend of Truck
→ Bus can acess prop. of truck class
→ But truck can't access private/protected properties of Bus class.