OOPS-I

#A class: is a template for an object (Blueprint for an object)
↳ also known as user defined data types.

Creating objects:

↳ 1) Statically: `Student S1;` → initializes all variables to garbage values.

↳ 2) Dynamically: `Student *S2 = new Student;` → internally, `S1.Student();` is called, (default constructor)

Access Member variables / functions:

↳ ~~0) Through ptr~~

↳ 1) Statically: `S1.age = 23;` `S1.rollNo = 1688;`

2) Dynamically: `(*S2).age = 19;` or `S2->rollNo = 2000;`

#Access Modifiers:

(default) ↳ Private: properties are visible only in the class.

↳ Public: properties are accessible everywhere.

→ Protected: (discussed later)

#this keyword:

↳ holds the address of the current object.

↳ 'this' is a pointer, available to every member function in the class.

---

1. Default constructor: ~~`S1 Student();`~~ `Student S1;` [Student()]
2. Copy Constructor: `Student S2(S1);` OR `Student S2 = S1;`
   ↳ creates a new object S2, with all the properties same as S1. (copies S1)
3) Copy assignment operator: (=)
   ↳ copy values of one object into another object, after their creation.

```
Student S1(10, 1000);
Student S2(20, 2000);
S2 = S1;
```
→ copy S1 into S2

4) **Destructor**: Deallocate the memory of our object.
- Same name as our class
- No return type
- No input arguments.

↳ ~Student(){ ... }

→ Called when the scope of the object variable is finishing (e.g., coming out of a function)
- called automatically: (i) function ends (ii) program ends (iii) Block containing local variable ends

OR when the delete operator is called.

~~#fraction class~~ | #Complex Number class:

```
class Complex {
 private:
    int real;
    int imaginary;
 public:
   Complex (int r, int i){
     real = r;
     imaginary = i;
   }
   void plus (Complex const &c2){
     real += c2.real;
     imaginary += c2.imaginary;
   }
   void multiply (Complex const &c2){
     int x = (real * c2.real) - (imaginary * c2.imaginary);
     int y = (real * c2.imaginary) + (imaginary * c2.real);
     real = x;
     imaginary = y;
   }
   void print(){
     cout << real << "+i" << imaginary;
   }
};
```

&c2 to create a reference variable to the actual variable (object) being passed in (To avoid the unnecessary copying)

const so that we don't accidentally change the value of the actual variable being passed. (Because this is a reference variable)

#Shallow copy:

When we pass in an array to a constructor, the whole array is not copied, but only the pointer to the 0th index.
→ Changes made to the main program can also reflect in an object.

for e.g.,

```
class Student{
    int age;
    char *name;
  public:
    Student(int age, char *name){
        this->age = age;
        this->name = name;          (Shallow copy)
      OR
        this->name = new char[strlen(name)+1];
        strcpy(this->name, name);   (Deep copy)
    }
};
```

→ copies just the pointer, and it points to the same space in memory.

→ copies the whole array to a new location in the memory.

#Deep copy: As understood above.

#Inbuilt Copy Constructor creates a shallow copy of variables.

So, ~~changing~~ If we have Student s1(18, "abcd");
and Student s2(s1); → shallow copy; so changing s1.name will also change s2.name, as they both point to the same memory location.

We need to create custom copy constructors for such cases:

```
Student(Student s){
    this->age = s.age;
    this->name = new char[strlen(s.name)+1];
    strcpy(this->name, s.name);
}
```

→ deep copy of name

→ Copy constructor called here, which uses this constructor, which uses the copy constructor... → ∞ loop reached

DON'T create a copy constructor LIKE THIS → XXX

To create a copy constructor without getting into an ∞ loop, pass the object by reference.

```
Student (Student const &s){
    this->age = s.age;
    //Deep copy:
    this->name = new char[strlen(s.name)+1];
    strcpy(this->name, s.name);
}
```

Initialization List:

→ We can not change the values of const variables.

→ Const variables MUST be assigned values at the time of declaration (otherwise, they will have a garbage value in them)

```
int a = 5;
int const b = 6;
```

allowed & valid:
```
int a;   → allocate 4 bytes of memory, which has garbage value.
a = 5;   → put 5 in that place
```
(4 bytes → block.)

NOT allowed:
```
int const b;   → garbage value will be present, which won't be changed later.
b = 6;
```

→ Reference variables ALSO must be assigned values at the time of declaration (they need to point to some block)

```
int a = 5;
int &j = a;
```

a points to 5; also j points to the same block.

```
int &j;
j = a;
```
✗ Not allowed

→ If a member variable in a class is declared constant, then we have to create a custom constructor where that variable is assigned a value (otherwise, a garbage value will be assigned and error will be thrown)

~~class Student{~~

↳ we must do this with Initialization List.

Without & with initialization list:

```
class Student {
public:
    int age;
    const int rollNumber;
    Student (int r) {
        rollNumber = r;
    }
};
```

→ XXX without initialization list, will still throw an error (while doing rollNumber = r, 'rollNumber' has already been declared.

→ We can say memory is allocated to the members at this point. We have to initialize constant variables before this point.

```
Student (int r) : rollNumber (r) {
    //
}
```

→ with initialization list, correct way.

→ Non-const variables can also be initialized in the initialization list.

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

→ we don't need to use 'this' here, age (age) would be understood.

⇒ We will have to use initialization list if we have a reference member variable as well, for the same reason.

#Constant Objects:

⇒ like const variables, we can declare const objects, for e.g.

→ Fraction const f3; (Initialized with a garbage value in this case)

- → No value can be changed
- → No function can be accessed (compiler worries that we will change something, so, not even getter functions can be run).
- → Only const functions can be called.

#Const functions:

Those functions which don't change any property of the current object (i.e, changes no property of 'this' object)

⇒ So, we should mark our getter functions as const.

↳
```
int getNumerator() const {
    return numerator;
}
```

⇒ If we try to change a property in a const function, we will get an error. (for e.g., marking a setter function as const).

Static Properties:

↳ properties that are common to all the objects of a class (i.e., properties that belong to a class)

→ A separate copy is not created for each object of the class.

e.g.:

```
static int totalStudents;
```

→ We can't access a static property with <object>.property, since the property belongs to a class.

[This is allowed, but logically incorrect & not used. (No error will be thrown)]

→ To access a static property:

```
<className>::property
```

↳ scope resolution operator.

→ for e.g. →
```
cout << Student::totalStudents;
```

→ We have to initialize our static properties OUTSIDE the class, (because we cannot initialize them in a constructor, since a constructor belongs to objects not classes).

↳
```
int Student::totalStudents = 0;
```

[We can do s1.totalStudents = 20; and then access using s2.totalStudents (will give 20) but this is logically incorrect and should not be used]

⇒ Like static variables, we can have ~~data~~ static functions as well. (for e.g., getters and setters for static variables)

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

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

★ Static functions can only access static properties (functions and variables both, but has to be static).

★ Static functions don't have access to this keyword.

- ('this' is a pointer to the function invoking object's address, and there is no object in a static functions case.)

example: Student class:

```
class Student{
    static int totalStudents;        // private.
    public:
    int rollNumber;
    int age;
    Student(){
        totalStudents++;
    }
    int getRollNumber(){             // (Non-static function)
        return rollNumber;
    }
    static int getTotalStudents(){
        return totalStudents;
    }
};
int Student::totalStudents = 0;      // initialization
```

```
# int main() →
Student s1, s2, s3;
// create 3 students
cout<< Student::getTotalStudents();
```
↳ prints 3.

Operator Overloading

⇒ Extend the functionality of a pre-existing operator, such that it will work for user defined classes also.

`f1 = f2 + f3` → objects of fraction class

- first operand goes into 'this'
- second operand goes as an argument.

Binary operators, +, *, ==

`f1 = f2.add(f3);` ⟷ similar ⟷ `f1 = f2+f3;`

How to overload?

↳ Create a function, with the name 'operator' immediately followed by the operator we want to overload. (No space)

for e.g.,

```
Fraction operator+(Fraction const &f2){
    // Code
    return fnew;
}
```

↳ this should be a const function, so ideally, (doesn't change any property of 'this')

~~Fraction~~

for e.g.,

```
Fraction operator*(Fraction const &f2) const {
    int num = numerator * f2.numerator;
    int den = denominator * f2.denominator;
    Fraction fnew(num, den);
    fnew.simplify();
    return fnew;
}
```

↳ Now, `f3 = f2 * f1` can be done properly.

PTO

#Overloading the ++ Operator: (unary operator, only one operand)

```
++f1;
f2 = ++f1;
f3 = ++(++f1);
and ++(++f1);
```

→ pre-increment
→ post-increment

operand passed into 'this'

→ nesting not allowed ((i++)++ X)

↑ Take care of all these cases.

⇒ A function's return is first received by the system in a temporary buffer, then that buffer value is placed in the desired location. [Watch operator overloading 2]

⇒ To take care of the ++(++f1) case, we must return our value by reference.
(The same block of memory is returned, no buffer memory is used, so the second ++ also works fine).

#Pre-increment ++ ++f:

```
Fraction & operator++ (){
    numerator = numerator + denominator;
    simplify();
    return (*this);
}
```

#Post-increment ++ : (f++)

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

int: To signify post-increment

#Overloading the += operator: i=5, j=3; (i+=j)+=j;

→ i = 11
→ j = 3

→ binary operator.

this → i += j; → operand

```
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);
    numerator = num;
    denominator = lcm;
    simplify();
    return (*this);
}
```

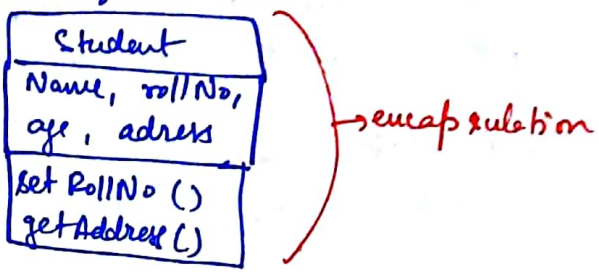
OOPS-3

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

#Encapsulation:

↳ to "club" together, data members and functions related to a particular entity.

→ done using classes. e.g. →



#Abstraction → uses access control specifics.

↳ hiding the unnecessary details (of implementation)

⇒ we use the gear stick to drive, but don't know the mechanism of the gear box in our car, and don't need to know.

STL sort() function: We just know that our array will be sorted, we don't know and don't care about which algorithm is actually used.

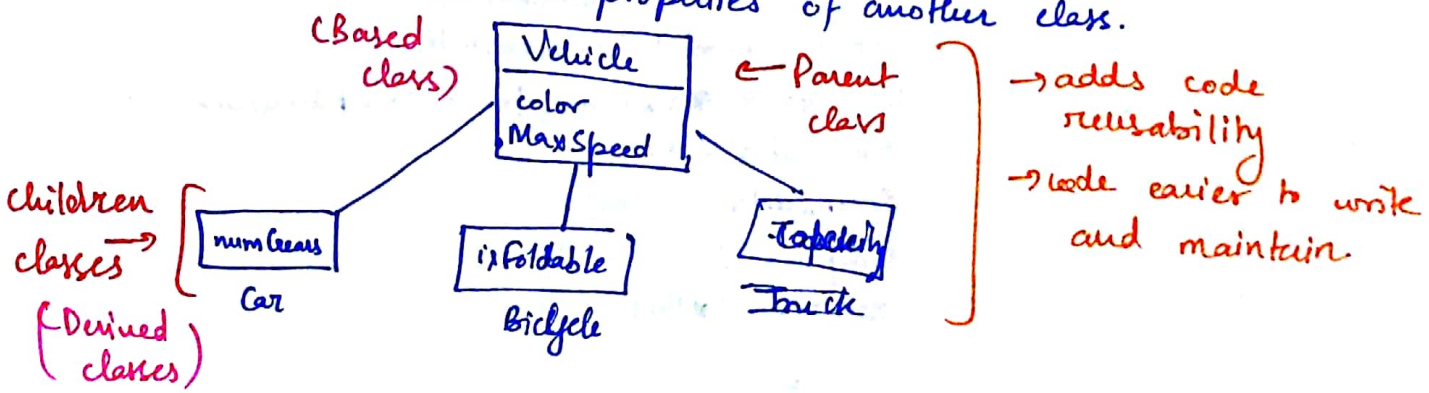
Need for abstraction:

→ updation / changes: should not make a difference in the output expected by the end user. for e.g., we won't realise if the sort() function changes its sorting algorithm one day, to improve time complexity.

→ Avoid errors: Random/ unauthorized users should not be able to access and change the code, possibly introducing errors.

#Inheritance:

↳ one class can inherit the properties of another class.



Access Modifiers:

- **private** → properties accessible only in the class itself.
- **public** → properties of the class accessible from anywhere.
- **protected** → properties of the class accessible from outside the class, but only by the child classes of that class.

For eg:, If a class 'Vehicle' has a protected property 'y', then Bicycle, Car and Truck classes (which inherit the Vehicle class) can access the property y, but otherwise we can't access 'y' outside the class Vehicle.

e.g.:

```
class Vehicle {
  private:
    int maxSpeed;
  protected:
    int numTyres;
  public:
    string color;
};
```

→ class car:

```
class car: access-specifier Vehicle{
  ...
};
```

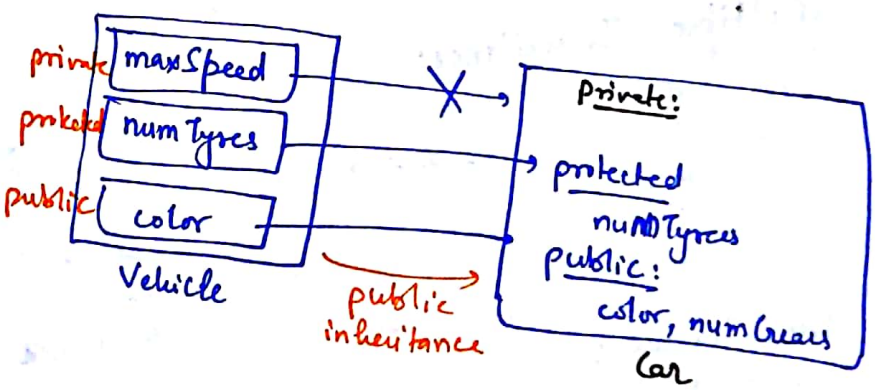
name of class to be inherited

specifies how we will be able to access the properties of it's parent class (private by default)

Public Inheritance:

```
class car: public Vehicle{
  public:
    int numGears;
};
```

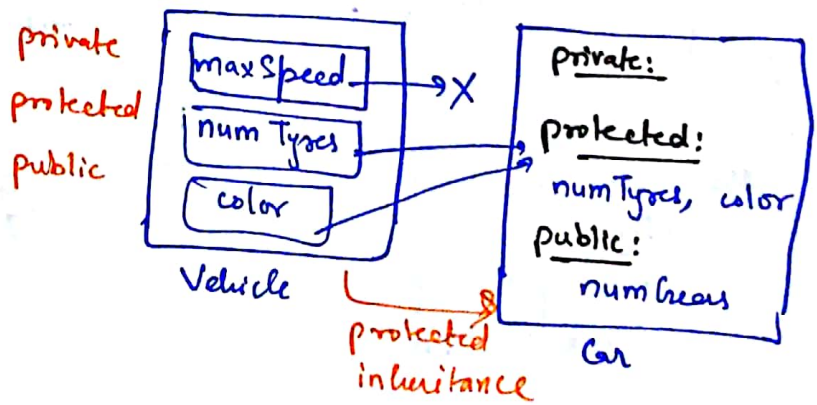
private → ×
protected → protected
public → public



Protected Inheritance:

```
class car: protected Vehicle{
  public:
    int numGears;
};
```

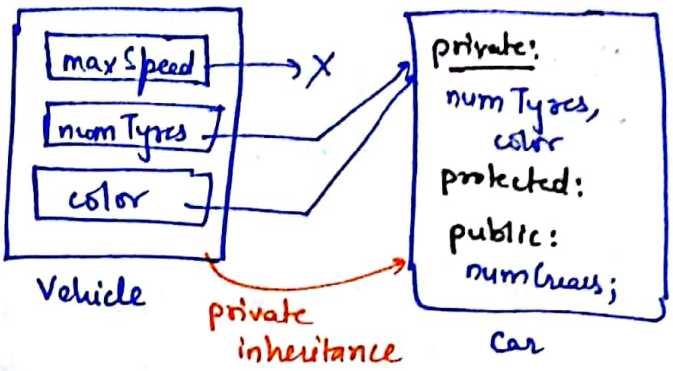
private → ×
protected → protected
public → protected



Private Inheritance:

```
class car : private Vehicle {
  public:
    int ~~numTypes~~ numGears;
};
```

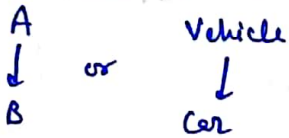
private
protected
public



private → X
protected → private
public → private

Types of Inheritance:

(i) Single Inheritance:



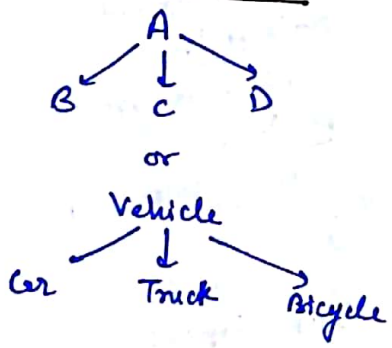
(ii) Multi-level Inheritance:

A → B → C

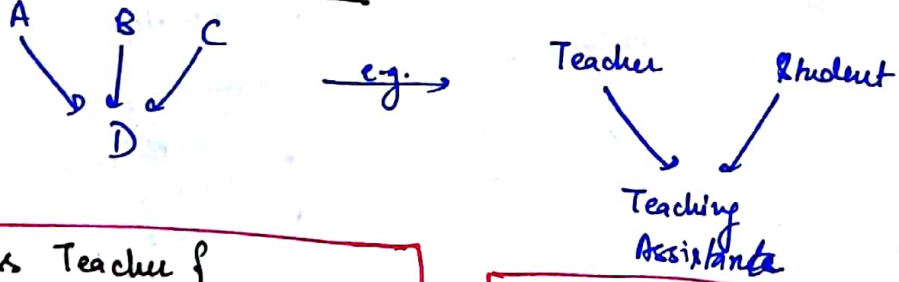
or

Vehicle → Car → HondaCity

(iii) Hierarchical Inheritance:



(iv) Multiple Inheritance:




```
class Teacher {
  public:
    string name, age;
    void print() {
      cout << "Teacher";
    }
};
```

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

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

(i) If the constructor of TA class is called, internally, the constructor of Teacher class will be called first, then the constructor of the Student class, as in the order they were mentioned during inheritance.

(ii) Let's say we have an object "Saksham" of class TA,

(a) If the TA class has its own print function:

Saksham.print() ⇒ will print "TA", as implicitly obvious.

We will have to use the scope resolution operator to explicitly call the print ~~fun~~ methods of Teacher and Student class.

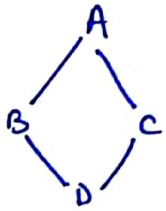
- Saksham.Student::print() ⇒ prints "Student"
- Saksham.Teacher::print() ⇒ prints "Teacher"

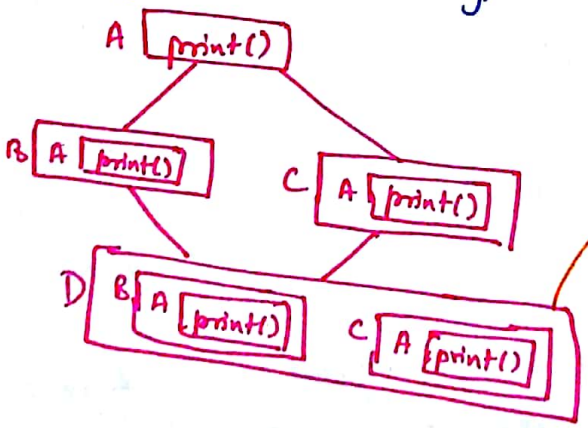
(b) If the TA class doesn't have a print function:

Since the TA class inherits the print method from both its parent classes, we will HAVE TO use the scope resolution operator to clear any ambiguity while calling the print method on an object of class TA.

(v) Hybrid Inheritance:

- ↳ More than one type of inheritance types used.





← Diamond problem

→ class D has 2 instances of print() function through A. How do we resolve this?

D d;

d.print() → error

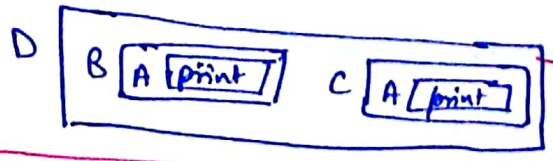
d.B::print() → } A's print called

d.C::print() → }

If B class has a print() function of its own, then:

d.B::print() → B's print() called

d.C::print() → A's print() called



→ we are getting 2 copies of properties of A in the class D. To avoid this, use the 'virtual keyword' for classes B and C.

```
class B: virtual public A{
...
};
class C: virtual public A{
...
};
```

→ A's constructor also called once only, using when using 'virtual' keyword.

⇒ This case is also called Virtual Inheritance.

#Polymorphism: means "many forms"

↳ set of code behaves different in different contexts.

(i) Compile Time: Decided during compile time

(ii) Run Time: Decided during run time

Compile Time ~~Poly~~ Polymorphism:

↳ Function overloading & operator overloading

↳ functions with same name but different type or number of arguments. (not differentiated by return type)

↳ Method / function Overriding:

Lets say we have 2 classes A and B, where B inherits from A, then, if B redefines a method inherited from A, it is called method overriding.

A. (Base)
↳ print()

B. (Derived)
↳ print() → method overriding.

A base class pointer can point to derived class's objects as well. (But vice-versa is not allowed)

Case: Let say we have a Vehicle class and a car class that inherits from vehicle class, and both have their own print() methods. Now, we have a pointer *V which is pointing to a

car object. Then V→print() will call the Vehicle class's print method.

> # A base class's pointer can only access properties of the base class (present in the base class) even if the pointer is pointing to an object of a derived class.

Run-Time Polymorphism:

In the last example, if we want `V→print()` to call the car class's print() method, then we have to use virtual functions. This will be run-time polymorphism.

#Virtual functions:

↳ declared within the base class and overridden by the derived class.

→ If the derived class doesn't override the virtual function, the base class's ~~use~~ function is used. (Need not be overridden)

→ Virtual functions cannot be static and cannot be friend functions of another class. Virtual Constructors also not allowed.

- ```
  virtual void print(){
    ...
  }
  ```
  → basically, tells the compiler to (check) achieve polymorphism during run time.
  → If while running, the base class pointer is pointing to a base class object, then the base class's version of the function is called. Else, if the base class pointer is pointing to a derived class object, then the overridden function is called (of the derived class).

#Pure Virtual functions: No definition in the function.

```
virtual void print() = 0;
```

→ If any class contains any pure virtual function, then it becomes an abstract class.

```

Abstract class:

↳ We cannot create an object of an abstract class.

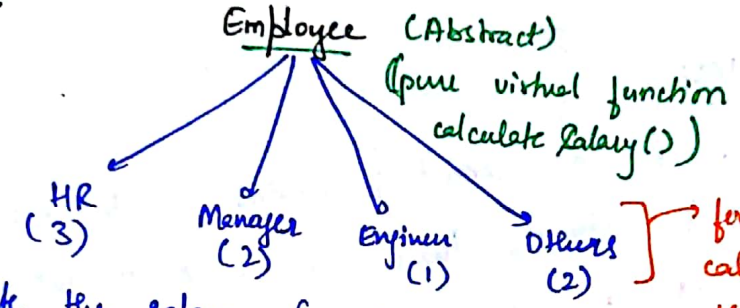
Let say we have,

Vehicle (Abstract)

↓

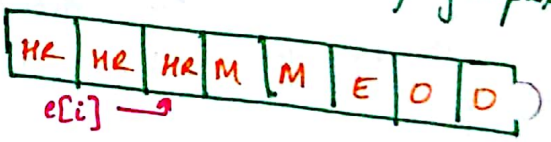
Car → has 2 options → (i) implement all the virtual functions.
(ii) become an abstract class itself.

Use case:



To calculate the salary of all of the employees, create an array of 'Employee' type (pointer will be employee type), add these employees, and call calculateSalary(), the function respective to each derived class will be called, because of runtime polymorphism.

```
Employee *e = new Employee[8];
for(int i=0; i<8; i++){
    e[i] -> calculateSalary();
}
```



Friend functions:

↳ can access the private and protected members of the class that declares a function as it's friend.

→ can be: (i) a method of another class
(ii) a global function.

→ don't have access to the 'this' keyword.

→ must be declared before the class ~~that~~ ~~de~~ whose friend that function is, and implemented after that class. ~~Its~~

Friend class: like friend functions, can access private and protected members of the class that declares it as it's friend.

Must be implemented before the class ~~that~~ whose friend it is.

Friend functions & class example:

```
class Scooty {
    public:
        void ~~public:~~ print();
};
class Bus {
    public:
        void print();
};
void test();
class Truck {
    private:
        int x;
    friend void Bus::print();
    friend void test();
    friend class Scooty;
};
void Scooty::print(){
    Truck t;
    t.x = 10;
    cout << t.x;
}
void test(){
    // Same as above
}
void Bus::print(){
    // Same as above
}
```

→ Whole class declared as friend. Any function that uses the properties of class Truck has to be declared here, and implemented after Truck class.

→ print() of Bus declared friend, so it must be declared here and implemented later.

→ Apparently needed to be declared before Truck class, but code works fine without pre-declaring too.

→ Implementations, using the private property 'x' of class Truck.

→ Friendship is not mutual. If B is friend of A, then A doesn't automatically becomes B's friend.

→ Friendship is not inherited.