Object-Oriented programming -

↳ Entity → state / properties
→ behaviours

OOPs is a style of programming that uses objects to model real-world things like data and behaviours.

Example - In a game - Hero & villian

Tekken

Name - Paul, Health - 70, level - A

↳ properties

behaviours

attack() defense()

dance()

↳ behaviours

class - user-defined data type

int a;
string str;
char ch

} like a, str & ch are variable of data type - integer, string & character type.

Hero → Ramesh

variable

↳ data type

Hero

A class is a user-defined data-type that acts like blueprint to create objects that share similar properties (data) and behaviours (functions).

For Example - Define an Animal class like name, age and species, and behaviours - eat(), sleep() and make Sound().

```
class Animal {
    Public:
    string species;
    int age;
    int name;
    // Member functions
    void eat() {
        // eat something
    }
    void sleep() {
        // sleep for hrs
    }
    void makeSound() {
        // make sound
    }
};
```

properties

class Hero:-

int health

Hero h1:-

Health =

Object - h1

Memory released if for h1

Garbage value

In Empty class where nothing is define in properties then given 1 bit of memory size to keep track on identification

// to access the properties using dot (.) operator.

Object – An object is a real, usable instance of a class that has specific properties and behaviours.

For Example – Animal, is just an idea or blueprint, but a cat is a real object based on that class. So, classes are concepts and object are the actual things created from those concept

```
class Animal {
    public:
        String species;
        int age;
        int name;
        void eat () { }
        void sleep () { }
        void makeSound () { }
};
```

```
int main () {
    Animal cat;
}
```

Animal → Class, cat → object

Access Modifiers – Access Specifiers help in hiding data from outside the class and are a key part of OOPS.

- which part of your code can access certain method or variable.
- which parts cannot directly access them.

Type – ① Public ② Private (default) ③ Protected

1. Public – Can be accessed from anywhere in the program
2. Private – Can be accessible only within the class
3. Protected – Can be accessible within the class and derived class.

Getter & Setter

Getter – As accessors (as they access the value), are the Public member functions that are used to fetch private member's values. The getter starts with the word 'get' followed by the variable name.

Setter – As mutators (as they update the value), are also the public member functions that set the value of private member variable. the setter starts with the word 'set' followed by the variable name.

Example -

```
#include <iostream>
using namespace std;
class Employee {
  private:
    int salary;
  public:
    // Setter
    void setSalary (int s) { salary = s; }
    // Getter
    void getSalary () { return salary; }
};
int main () {
    Employee myobj;   myobj.setSalary (10000);
    cout << " Salary is : " << myobj.getSalary ();
    return 0;
}
```

⊛ Dynamic allocation -

Ex -

```
Hero *b = new Hero;
```

(Pointer Creation)

```
b -> setHealth (79);   b -> setLevel ('B');
```

(setter, setter)

```
cout << (b) -> getHealth () << endl;
cout << (b) -> getLevel () << endl;
```

(getter)

Constructor — constructor are special methods that are automatically called whenever an object of class are created.

Example -

```
#include <iostream>
using namespace std;
class A {
  public:
    A () {
        cout << "constructor called" << endl;
    }
};
int main () {
    A obj;
    return 0;
}
```

constructor called

Types of constructor –

① Default constructor ② Parameterized constructor ③ Copy constructor ④ Move constructor

1. Default constructor – Is automatically generated by the compiler if the programmer does not define one. This constructor doesn't take any argument as it is parameter less and initializes object members using default values. Also called as zero-argument constructor.

Example –

```
#include <iostream>
using namespace std;
class A {
    public:
};

int main () {
    A a; //static
    A *b = new A; //dynamic
    return 0;
}
```

2. Parameterized constructor – Allows us to pass arguments to constructors. Typically, these arguments help initialize an object's members.

Example –

```
#include <iostream>
using namespace std;
class A {
    public:
        int val;
        A (int x) {
            val = x;
        }
};
int main () {
    A a (10);
    cout << a.val;
    return 0;
}
```

→ 10

```
#include <iostream>
using namespace std;
class Hero {
    public:
        int health; char level;
        Hero (int health) {
            this -> health = health;
        }
};
int main () {
    Hero b (10);
    cout << b.health << endl;
    return 0;
}
```

this → Hero () one

define health

→ 10

3. Copy constructor – Is a member function that initializes an object using another object of the same class. Copy constructor takes a reference to an object of the same class as an argument.

Example –

```
#include <iostream>
using namespace std;
class A {
    public:
        int val;
    // parameterized constructor
    A (int x) {
        val = x;
    }
    // copy constructor
    A (A &a) {
        val = a.val;
    }
};
int main () {
    A a1 (20);
    // creating another object from a1
    A a2 (a1);
    cout << a2.val;
    return 0;
}
```

→ `A (A &a)` : pass by reference not pass by value

→ `val = a.val;` : copy the a value into the (int val);

→ `A a1 (20);` : define the val

→ `A a2 (a1);` : copy to the a1 value to the a2

→ `return 0;` : 20

4. Move constructor – It recent add to the family of constructors in C++. It like a copy constructor that constructs the object from the already existing objects., but instead of copying the object into the new memory. → Copying the object into the new memory.

Syntax –

```
ClassName (ClassName && obj) {
    // body the constructor
}
```

Example -

```
#include <iostream>
using namespace std;
class Myclass {
    private:
        int b;
    public:
        Myclass (int &&a) : b(move (a)) {
            cout << " Move constructor ! " << endl;
        }
        void display () {
            cout << b << endl;
        }
};
int main () {
    int a = 4;
    Myclass obj1 (move(a)); // Move constructor !
    obj1.display (); // 4
    return 0;
}
```

→ rvalue reference (int &&a)

→ moves the value a to b.

Shallow and Deep Copy

Shallow Copy - Stores the references of object to the original memory address.

Deep copy - Stores copies of the object's value.

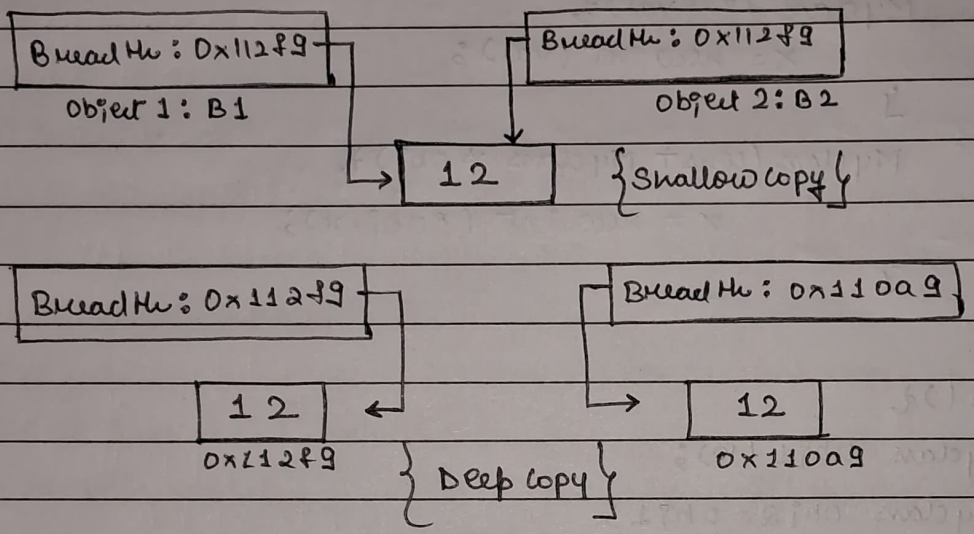
- Shallow copy reflects changes made to the new/copied object in the original object.
- Deep copy doesn't reflect changes made to the new/copied object in the original object.

// Copy constructor

```
Geeks obj1 (obj);
or
Geeks obj1 = obj
```

// Default assignment operator

```
Geeks obj2;
obj2 = obj1;
```




```
#include <iostream>
using namespace std;
class Myclass {
    Public:
        int * x;
        Myclass (int val){
            x = new int (val);
        }
        Myclass (const Myclass &obj){
            x = obj.x;
        }
};
int main () {
    Myclass obj1 (10);
    Myclass obj2 = obj1;
    *obj2.x = 20;
    cout << *obj1.x << " " << *obj2.x;
    return 0;
}
```

// 20 20

```
#include <iostream>
using namespace std;

class Myclass {
    public:
        int * x;
        Myclass (int val) {
            x = new int (val);
        }
        Myclass (const Myclass & obj) {
            x = new int (*obj.x);
        }
};
int main () {
    Myclass obj1 (10);
    Myclass obj2 = obj1;
    *obj2.x = 20;
    cout << *obj1.x << " " << *obj2.x;
    return 0;
}
```

// 10 20

Destructors

Destructors is an instance member function that is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

Example -

```
#include <iostream>
using namespace std;
class Test {
    public:
        Test () {
            cout << "constructor called" << endl;
        }
        ~Test () {
            cout << "Destructor called" << endl;
        }
};
int main () {
    Test t;
    return 0;
}
```

↳ className object()

constructor called
Destructor called

[// Static Allocation – Destructor automatically
// Dynamic allocation – Destructor manually]

Static keyword - when a variable is declared as ~~Stat~~ Static, space is get allocated for the lifetime of the program.

(*) when a static keyword is used, variable, data member, and functions can't be modified again. Static functions can directly used a class name. Static variables are variables defined by using static keywords, which consists a special behaviour -

i) Static variable are initialized by ones.

ii) Static variable can be defined by inside or outside

iii) Default value of the static member is zero.

→ Uses of Static keyword

The static keyword in C++ can be used in several ways -

1. Static keyword (variables) -

C++ static variables maintain their value until the end of program.

2. class object -

C++ static class object can be used repeatedly even when their scope end.

3. class variable -

Class variable declared as static can be used to define a constant class ~~pro~~ property as they are common to all class object.

4. Members functions -

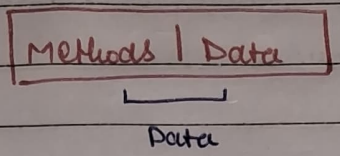
C++ static member functions can be called without using an object in a class.

(#) Encapsulation - (Information hiding / data hiding)

It refers to the bundling of data (variable) and methods (function) in a single unit, called a class. Encapsulation restricts direct access to some of an object's components, which is a means of preventing unintended interference of and misuses of data.

fully encapsulated class - all the data member is marked as private to prevent the data from any leaked and get accessed off it by getter and setter.

Encapsulation



Example -

```
#include<iostream>
using namespace std;
class student{
  private:
    string name;
    int age;
  public:
    void setName(string n){
        name = n
    }
    void setage(int a){
        age = a
    }
    string getName(){
        return name;
    }
    int getage(){
        return age;
    }
};
int main(){
    Student std1;
    std1.setName("Alice");
    std1.setage(20);
    cout << std1.getName() << endl;
    cout << std1.getage() << endl;
```

Can't accessed outside the class

setter

getter

helps to access the private class

// Alice

// 20

Abstraction – Implementation hiding

Abstraction only display the essential information and hide the other details and information.

Example –

```
#include <iostream>
using namespace std;
class Animal {
  public:
      virtual void makesound =0
};
Class Dog: public Animal {
   Public :                                  → override
      void makesound() override {
            cout << " Woof! " << endl;
      }
};

int main() {
   Animal *a1 = new Dog();
    a1 → make Sound();
    delete a1;
    return 0;
}
```

Polymorphism – means many form

We can define Polymorphism as the ability of an entity to behave different in different scenarios. Person at the same time can have different characteristics.

Two types of Polymorphism –

1. Compile time polymorphism – achieved using function overloading and operator overloading.

2. Run-time Polymorphism – Achieved using inheritance and virtual functions.

A. Function Overloading – Features of object-oriented programming where two or more functions can have same name but behave differently for different parameters. Such function are said to be overloaded – Function Overloading.

① Changing the no. of arguments or changing the type of arguments.

Example –

```
class Geeks {
    public:
        void add (int a, int b) {
            cout << " Integer Sum = " << a+b << endl;
        }
        void add (double a, double b) {
            cout << "float Sum = " << a+b << endl;
        }
};
```

B. Operator overloading –

The ability to provide the operators with special meaning for particular data type, this ability are known as "Operator Overloading".

Addition operator (+) for String to concatenate two strings and for integer to add two integers.

Example –

```
class complex {
  public:
    int real, img;
    complex (int r, int i);
    real(r), image (i) {}
```

// overloading the (+) operator -

```
complex operator + (const Complex &obj) {
    return Complex (real + obj.real, imag + obj.imag);
}
};
```

2. Runtime Polymorphism -

Also known as late binding and dynamic Polymorphism, the function call in runtime polymorphism is resolved at runtime in contrast with compile time polymorphism, where the compiler determines which function call to bind at compilation.

Runtime polymorphism is implemented using function overridding with virtual function.

Example -

```
class Animal {
    public:
    void speak() {
        cout << "speaking" << endl;
    }
};

Class Dog : Public Animal {
    public:
    void speak() {          → runtime polymorphism
        cout << "barking" << endl;
    }
};
```

Inheritance

It allows a class (derived/child class) to inherit properties and behaviours (data members) from another class (base/parent class).

Access modifier in Inheritance —

Inheritance Type	Base class (Public)	Protected	Private
	Base class		
Public	Public	Protected	not accessible
Protected	Protected	Protected	not accessible
Private	Private	Private	not accessible

Example —

```
Class Base {
    Public :
    int n ;
    void print N () {
        cout << n << endl;
    }
};
// Inheriting Base class Publicly
Class Derived : Public Base {
    public
    void fun() {
        n = 22
    }
};
```

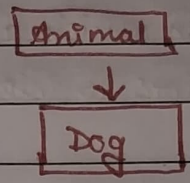
Types of Inheritance

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical inheritance
- Hybrid Inheritance

⊛ Single Inheritance —
a class is allowed to inherit from only one class.

Example - → Base class

```
Class Animal {
  public:
    int age;
    int weight;
    void bark() {
        cout << "Barking" << endl;
    }
};
```



→ Single Inheritance.

```
Class Dog : public Animal {
};
```

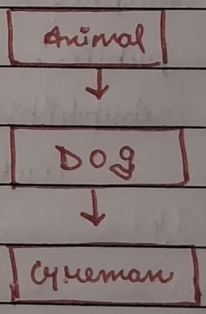
⊛ Multilevel Inheritance -

A derived class is created from another derived class and that derived class can be derived from a base class or any other derived class.

Ex -

→ Base class

```
Class Animal {
  public :
    int age ;
    int weight ;
    void break() {
        cout << "Braking" << endl;
    }
};
```



→ Single

```
Class Dog : Public Animal {
};
```

→ Multilevel Inheritance

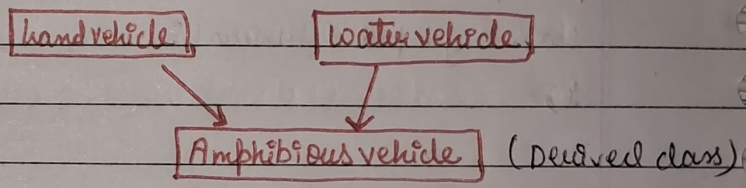
```
Class GermanShepherd : Public Dog {
};
```

⊛ Multiple Inheritance –

It is feature in which a class can inherit from more than one class, i.e one subclass is inherited from more than one base class.

Example –

```
class handvehicle {
    public :
     handvehicle (){
        cout << "handvehicle" << endl;
     }
};
class watervehicle {
    public:
     watervehicle (){
        cout << " This is watervehicle " <<endl;
     }
};
class Amphibiousvehicle : public watervehicle, public handvehicle {
    public :
     Amphibiousvehicle (){
        cout << " This is an Amphibiousvehicle " <<endl;
     }
};
```

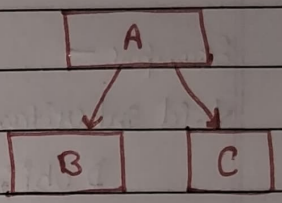


⊛ Hierarchical Inheritance

More than one class (subclass) is inherited from a single base class i.e., more than one derived class is created from a single base class.

Example -

```
class A{
public:
    void func1(){
        cout << "Inside the f1" <<endl;
    }
};
class B: public A{
    public:
        void func2(){
            cout<<" Inside the f2 " <<endl;
        }
};
class C: public A{
 Public:
     void func3(){
         cout << "Inside the f3" <<endl;
     }
};
```

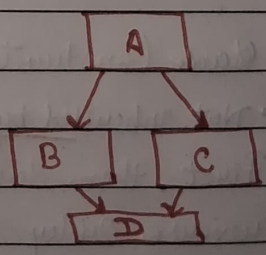


④ Hybrid Inheritance -

Implemented By combining more than one type of inheritance. For Example - Combining Hierarchical inheritance and Multiple Inheritance will create the hybrid inheritance C++.

Example -

```
class D : public B, Public C{
    public:
      void func4(){
          cout << "Inside the func4( Hybrid inheritance)"
      }
};
```



Inheritance Ambiguity –

When a derived class inherits from two classes that have a common base, ambiguity arises.

→ Class D inherits from both Class B and Class C.

Example –

```
Void inheritance Ambiguity Example (){
    D obj;
    obj.B::func1();  // call func1() from B's A
    obj.C::func1();  // call func1() from C's A
};
// obj.func1 → create a ambiguous, as func1() is inherited twice via B and C.
```

Effects of Inheritance

→ Static member –

• Belong to the class, not objects • Shared across all instances • Not inherited traditionally accessible via – 'className :: Static member'

→ Friend Function & class –

• Allow acces to private / protected members • Not inherited by derived classes • Friends of base class ≠ friends of derived class

→ Constructors & Destructors –

• Not inherited by derived class • Base class constructor called First, then derived class constructor – Destructors called in reverse Order • Can manually call base constructors / destructors in derived class

• Compiler generates default constructor / destructors if not defined

Example-

```
class Parent {
    public:
    parent() { cout << " Inside base class " <<endl; }
};
class child : public parent {
    public:
    child() { cout << " Inside sub class " <<endl; }
};
→ child obj;        ↔ Inside base class + Inside sub class
    return 0;
```