**CS Fundamental -> OOPS**

**You must know how to IMPLEMENT all these concepts**

[**CS\_FOR\_ALL**](https://www.youtube.com/watch?v=cCyMs96dOkQ&ab_channel=CSFORALL)**( class object constructor destructor )**

**-** [**CodingwithShivam**](https://www.youtube.com/watch?v=3fLvC4cofcg&ab_channel=codingLifeWithShivam) **(Interview Questions )**

**want to learn practical implementation, watch the above video or search each topic in YT.**

[**https://www.codingninjas.com/codestudio/library/oops**](https://www.codingninjas.com/codestudio/library/oops)

[**https://www.codingninjas.com/codestudio/guided-paths/oops-in-c**](https://www.codingninjas.com/codestudio/guided-paths/oops-in-c)

## ● Object-Oriented Programming

is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts defined below :

## ● Class

is a user-defined data type which defines its properties and its functions. Class is the only logical representation of the data. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space till the time an object is instantiated.

* Class can be used and accessed by creating an instance of that case.

Class by default private hota hai you need to change its access modifier.

It represent a set of properties or method common to all objects of same time.

Getter and setter ka concept isme hai. Read and manipulate properties, data member

C++ Syntax (for class):

#include<bits/stdc++.h>

using namespace std;

class Student{

Public:

string nickname;

int roll; // data member

int mobile;

string result;

void PrintInformation(){

cout<<"Name:"<<nickname<<endl;

cout<<"Roll no:"<<roll<<endl;

cout<<"Mobile no:"<<mobile<<endl;

}

int add(int x, int y){ // member functions

return x + y;

}

};

int main(){

Student devesh; // Object

devesh.nickname="Rishu"; // features

devesh.roll=1;

devesh.mobile=7903114;

devesh.result="Pass";

**devesh.PrintInformation(); // calling print function**

Student Abhinav;// Object

Abhinav.nickname="Suraj";

Abhinav.roll =2;// features

Abhinav.mobile= 789456133;// features

Abhinav.result="Pass";// features

**Abhinav.PrintInformation();**

return 0;

// getter and setter ka bhi ues karte hai to inset data

}

## ● Object

is a run-time entity. It is an instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions, It has state and behavior

C++ Syntax (for object):

student s = new student() // dynamic allocation

Note : When an object is created using a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created without a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.

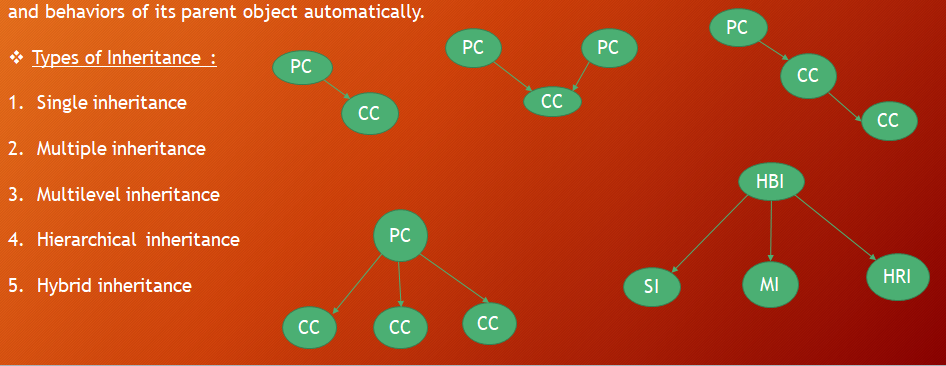
## \* Features of OOPS

### 1. **Inheritance**

Inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such a way, you can reuse, extend or modify the attributes and behaviors which are defined in other classes.

In C++ , the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

C++ Syntax : class derived\_class :: visibility-mode base\_class; visibility-modes = {private, protected, public}



**Types of Inheritance :**

1. **Single inheritance :** When one class inherits another class, it is known as single level inheritance

2. **Multiple inheritance :** Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.

3. **Hierarchical inheritance :** Hierarchical inheritance is defined as the process of deriving more than one class from a base class.

4. **Multilevel inheritance :** Multilevel inheritance is a process of deriving a class from another derived class.

5. **Hybrid inheritance :** Hybrid inheritance is a combination of simple, multiple inheritance and hierarchical inheritance.

### 2. **Encapsulation**

Encapsulation is the process of combining data and functions into a single unit called class. In Encapsulation, the data is not accessed directly; it is accessed through the functions present inside the class. In simpler words,

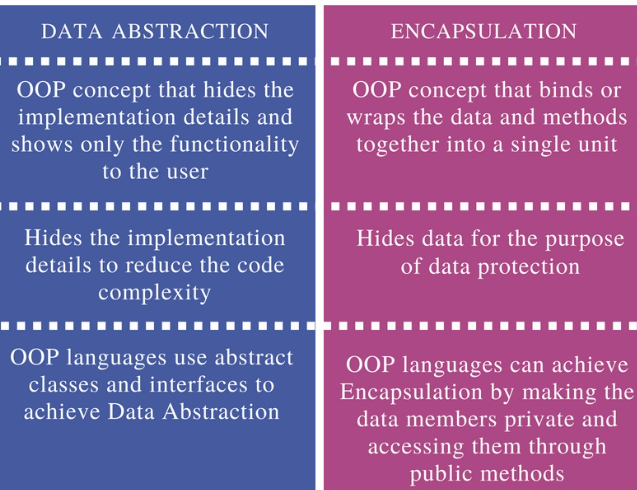
attributes of the class are kept private and public getter and setter methods are provided to manipulate these attributes. Thus, encapsulation makes the concept of data hiding possible.(Data hiding: a language feature to restrict access to members of an object, reducing the negative effect due to dependencies. e.g. "protected", "private" feature in C++).

### 3. **Abstraction**

Abstraction is the process of hiding the internal details of an application which are unnecessary details to user only essential details are displayed to the user. Abstraction is used to describe things in simple terms. 1.Reduce the complexity of view, 2.Increase the security.

Ex- Internal working of car - Lift - coffee machine

-**Data binding :** Data binding is a process of binding the application UI and business logic. Any change made in the business logic will reflect directly to the application UI.



### 4. **Polymorphism**

The word itself defines the meaning Poly means ‘many’ and morphism means ‘forms’.

So, Polymorphism is the ability to present the same interface for different underlying forms (data types). With polymorphism, each of these classes will have different underlying data.

*So the same Person posse's different behavior.*

Types of Polymorphism IMP

● **Compile Time Polymorphism (Static):** The polymorphism which is implemented at the compile time is known as compile-time polymorphism. Example - Function Overloading

(Fast hota hai kyu ki compile time me hi ho jaa raha hai)

Function Overloading : Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality. Method overloading can be possible on the following basis:

1. The return type of the overloaded function.

2. The type of the parameters passed to the function.

3. The number of parameters passed to the function.

● **Runtime Polymorphism** (Dynamic)**:** Runtime polymorphism is also known as dynamic polymorphism. Function overriding is an example of runtime polymorphism.

Function overriding means when the child class contains the method which is already present in the parent class. Hence, the child class overrides the method of the parent class. In case of function overriding, parent and child classes both contain the same function with a different definition. The call to the function is determined at runtime is known as runtime polymorphism.

**When it redefines a function of the base class in a derived class with the same signature i.e., name, return type, and parameter but with a different definition**, it is called function overriding

| **Compile-Time Polymorphism** | **Runtime Polymorphism** |
| --- | --- |
| 1. It is also termed static binding and early binding. | It is also termed Dynamic binding and Late binding. |
| 2. It is fast because execution is known early at compile time. | It is slow as compared to compile-time because execution is known at runtime. |
| 3. It is achieved by function overloading and operator overloading. | It is achieved by virtual functions and function overriding. |
| **int** **add**(**int** a, **int** b){  **return** a+b;  }  **int** **add**(**int** a, **int** b, **int** c){  **return** a+b+c;  }  **int** **main**(){  cout<<add(2,3)<<endl;  cout<<add(2,3,4)<<endl;  **return** 0;  } | **class** **A**{  **public**:  **virtual** **void** **fun**(){  cout<<"base ";  }  };  **class** **B**: **public** A{  **public**: **void** **fun**(){  cout<<"derived "; } };  **int** **main**(){  A \*a=**new** B;  a->fun();  **return** 0;  } |

| **Function Overloading** | **Function Overriding** |
| --- | --- |
| 1. In Function Overloading we declare more than one function with the same name and different type of parameter | In Function Overriding we declare a function in base class and derived class with the same name and same parameter. |
| 2. It can occurs without inheritance | It occurs when one class is inherited from another class |
| 3. In Function Overloading must have different signature function meaning that they have a different set of parameters. | In Function Overriding must have same signature |
| 4. It is a concept of compile time polymorphism | It is a concept of Runtime polymorphism |

## ● Constructor :

Definition - use - similarity

Constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as class or structure.

There can be three types of constructors in C++.

#include<bits/stdc++.h>

using namespace std;

class Student{

public:

string nickname;

int roll; // data member

int mobile;

string result;

Student(){

cout<<"Default Constructor"<<endl;

}

Student(string s, int r, int m){

nickname=s;

roll=r;

mobile=m;

cout<<"Parameterized Constructor"<<endl;

}

Student (Student &devesh){

nickname=devesh.nickname;

roll=devesh.roll;

mobile=devesh.mobile;

cout<<"Copy Constructor"<<endl;

}

~Student(){ // calling destructor

cout<<"Destructor "<<endl;

}

void PrintInformation(){

cout<<"Name:"<<nickname<<endl;

cout<<"Roll no:"<<roll<<endl;

cout<<"Mobile no:"<<mobile<<endl;

}

};

int main(){

Student devesh("Devesh", 1, 790311); // Parameterized constructor

devesh.PrintInformation();

Student Abhinav("Abhinav", 2, 321456);

Abhinav.PrintInformation();

Student D; // Default Constructor

Student Abhishek=devesh; // Copy Constructor;

Abhishek.PrintInformation();

return 0;

}

1.**Default constructor :** A constructor which has no argument is known as default constructor. It is invoked at the time of creating an object.

2.**Parameterized constructor :** Constructor which has parameters is called a parameterized constructor. It is used to provide different values to distinct objects.

3.**Copy Constructor :** A Copy constructor is an overloaded constructor used to declare and initialize an object from another object.

It is of two types - default copy constructor and user defined copy constructor.

## ● Destructor :

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically. A destructor is defined like a constructor. It must have the same name as class, prefixed with a tilde sign (~).

* Desctructor can t be static
* Destructor don't take any argument and don't have any return type

## ● **‘this’ Pointer :**

this is a keyword that refers to the current instance of the class.

There can be 3 main uses of ‘this’ keyword:

1. It can be used to pass the CO current object as a P parameter to AM another method

2. It can be used to refer to the current class instance variable. CCIV

3. It can be used to DI declare indexers.

## ● **Friend Function :**

Friend function acts as a friend of the class. The friend function is not a member of the class, A non-member function has no ability to access the private data of the class. Sometimes, it is necessary for the non-member function to access the data.

The friend function is a non-member function and has the ability to access the private and protected data of the class and members of the class.

Note :

1. A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.

2. Friend function uses objects as arguments.

## ● **Virtual Function IMP:**

A virtual function is used to replace the implementation provided by the base class. The replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer.

1. A virtual function is a member function which is present in the base class and redefined by the derived class.

2. When we use the same function name in both base and derived class, the function in base class is declared with a keyword virtual.

3. When the function is made virtual, then C++ determines at run-time which function is to be called based on the type of the object pointed by the base class pointer. Thus, by making the base class pointer to point to different objects, we can execute different versions of the virtual functions.

**Key Points :** 1. Virtual functions cannot be static.

2. A class may have a virtual destructor but it cannot have a virtual constructor.

### Can a virtual function be called from a constructor?

● **Abstract Classes :**

In C++ class is made abstract by declaring at least one of its functions as a pure virtual function. A pure virtual function is specified by placing "= 0" in its declaration. Its implementation must be provided by derived classes.

## ● **Namespaces in C++ :**

1. The namespace is a logical “ / ”division of the code which is designed to stop the naming conflict.

2. The namespace defines the scope where the identifiers such as variables, class, functions are declared.

3. The MP main purpose of using namespace in C++ is to remove the ambiguity. Ambiguity occurs when a different task occurs with the same name.

4. For example: if there are two functions with the same name such as add(). In order to prevent this ambiguity, the namespace is used. Functions are declared in different namespaces.

5. C++ consists of a standard namespace, i.e., std which contains inbuilt classes and functions. So, by using the statement "using namespace std;" includes the namespace "std" in our program.

## ● **Access Specifiers IMP :**

The AS access specifiers are used to define how functions and variables can be accessed outside the class.

There are three types of access specifiers:

1. **Private**: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared.

2. **Public**: Functions and variables declared under public can be accessed from anywhere.

3. **Protected**: Functions and variables declared as protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance

● **Operator overloading:** Operator overloading is defined as the standard operator can be redefined so that it has a different meaning when applied to the instances of a class.

● **Overloading** is static Binding, whereas Overriding is dynamic Binding. Overloading is nothing but the same method with different arguments, and it may or may not return the same value in the same class itself.

Pointer=> A Pointer is a variable that is used to store the address of another variable.

Reference can’t be null Pointer can be NULL

Reference can’t be void Pointer can be Void.

Once a reference is created it cannot be later made to reference to another object, it cant be reseted but in case of pointer we often change the pointer

Q Structure vs Class

Q Exception handling

Q

**Difference between Object Oriented Programming and Procedural Oriented Programming**

| OOP | POP |
| --- | --- |
| 1. Full form- Object Oriented Programming | Procedural Oriented Programming |
| 2. Approach- Bottom up | Top Down |
| 3. Emphasis on classes and object | Only on functions |
| 4. Features of OOPS E | Isme koi features nahi hota hai |

**Q=> Which language is pure Object Oriented .**

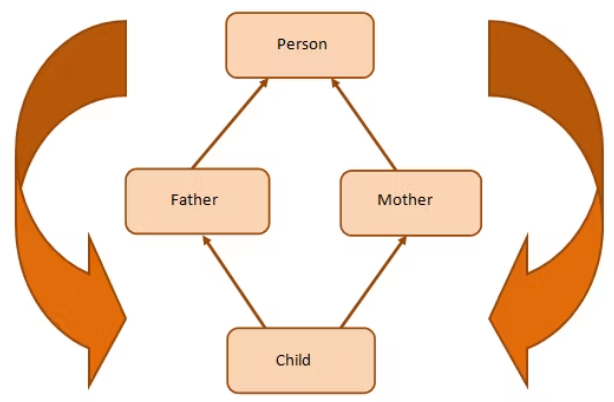
Ans=> No language is pure Object Oriented,

Java language is not a Pure Object Oriented Language as it contain these properties: Primitive Data Type

Python is not a Pure OOL as it does not support encapsulation.

C++ Main function is outside the class

**Q=> Diamond problem**

=> The Diamond Problem occurs when a child class inherits from two parent classes who both share a common grandparent class.

Here, we have a class **Child** inheriting from classes **Father**

and **Mother**. These two classes, in turn, inherit the class **Person**

because both Father and Mother are Person.

As shown in the figure, class Child inherits the traits of class

Person twice—once from Father and again from Mother. This

gives rise to ambiguity since the compiler fails to understand

which way to go.

This scenario gives rise to a diamond-shaped inheritance

graph and is famously called “The Diamond Problem.”

#include<iostream>

using **namespace** **std**;

**class** **Person** { //class Person

**public**:

    Person(**int** x) { cout << "Person::Person(int) called" << endl; }

};

**class** **Father** : public Person { //**class** **Father** **inherits** **Person**

**public**:

    Father(**int** x):Person(x) {

       cout << "Father::Father(int) called" << endl;

    }

};

**class** **Mother** : public Person { //**class** **Mother** **inherits** **Person**

**public**:

    Mother(**int** x):Person(x) {

        cout << "Mother::Mother(int) called" << endl;

    }

};

**class** **Child** : public Father, public Mother { //Child inherits Father **and** Mother

**public**:

    Child(**int** x):Mother(x), Father(x) {

        cout << "Child::Child(int) called" << endl;

    }

};

**int** **main**() {

    Child **child**(30);

}

Now you can see the ambiguity here. The Person class constructor is called twice: once when the Father class object is created and next when the Mother class object is created. The properties of the Person class are inherited twice, giving rise to ambiguity.

Since the Person class constructor is called twice, the destructor will also be called twice when the Child class object is destructed.

The solution to the diamond problem is to use the **virtual** keyword. We make the two parent classes (who inherit from the same grandparent class) into virtual classes in order to avoid two copies of the grandparent class in the child class.

#include<iostream>

using **namespace** **std**;

**class** **Person** { //class Person

**public**:

    Person() { cout << "Person::Person() called" << endl; } //Base constructor

    Person(**int** x) { cout << "Person::Person(int) called" << endl; }

};

**class** **Father** : virtual public Person { //**class** **Father** **inherits** **Person**

**public**:

    Father(**int** x):Person(x) {

       cout << "Father::Father(int) called" << endl;

    }

};

**class** **Mother** : virtual public Person { //**class** **Mother** **inherits** **Person**

**public**:

    Mother(**int** x):Person(x) {

        cout << "Mother::Mother(int) called" << endl;

    }

};

**class** **Child** : public Father, public Mother { //**class** **Child** **inherits** **Father** **and** **Mother**

**public**:

    Child(**int** x):Mother(x), Father(x) {

        cout << "Child::Child(int) called" << endl;

    }

};

**int** **main**() {

    Child **child**(30);

}

Here we have used the **virtual** keyword when classes Father and Mother inherit the Person class. This is usually called “virtual inheritance," which guarantees that only a single instance of the inherited class (in this case, the Person class) is passed on.

In other words, the Child class will have a single instance of the Person class, shared by both the Father and Mother classes. By having a single instance of the Person class, the ambiguity is resolved.

Virtual method se remove hota hai ambiguity

**// CLASS OBJECT**

#include<bits/stdc++.h>

using namespace std;

class Student{

public:

string Name;

int Roll;

bool gender;

void printInfo(){

cout<<"Name- "<<Name<<endl;

cout<<"Roll- "<<Roll<<endl;

cout<<"gender- "<<gender<<endl;

}

};

int main(){

Student A;

A.Name="Devesh";

A.Roll=21;

A.gender=false;

A.printInfo();

Student B;

B.Name="Abhinav";

B.Roll=11;

B.gender=false;

B.printInfo();

Student C;

C.Name="Shivani";

C.Roll=1;

C.gender=true;

C.printInfo();

return 0;

}

**// CONSTRUCTOR DESTRUCTOR**

#include<bits/stdc++.h>

using namespace std;

class Student{

public:

string Name;

int Roll;

bool gender;

Student(){

cout<<"Default Constructor "<<endl;

}

Student(string n, int r, bool g){

Name=n;

Roll=r;

gender=g;

cout<<"Paramaterized Constructor"<<endl;

}

Student(Student &C){

Name=C.Name;

Roll=C.Roll;

gender=C.gender;

cout<<"Copy Constructor"<<endl;

}

~Student(){

cout<<"Destructor Called"<<endl;

}

void printInfo(){

cout<<"Name:- "<<Name<<endl;

cout<<"Roll:- "<<Roll<<endl;

cout<<"Gender:- "<<gender<<endl;

}

};

int main(){

Student A("Raju", 10, false);

A.printInfo();

Student B("Devesh", 12, false);

B.printInfo();

Student C("Sita", 22, true);

C.printInfo();

Student D; // Default const

Student E=C;

E.printInfo();

return 0;

}