1. Introduction to OOPS

Definition:
Object-Oriented Programming (OOP) is a programming paradigm based on the concept of **objects**, which contain data (fields/attributes) and methods (functions).

Key Idea:
Computation is carried out using **objects** that interact with each other.

2. Class & Object

Class:

- Blueprint for creating objects
- Contains data members and member functions

Object:

- Instance of a class
- Memory allocated only when object is created

Example:

```cpp
#include <bits/stdc++.h>
using namespace std;

class Person {
public:
    string name;
    void printname() {
        cout << "Person name is: " << name;
    }
};

int main() {
    Person obj1;
    obj1.name = "Thanos";
    obj1.printname();
}
```

Output:

```
Person name is: Thanos
```

3. Constructor

Special member function that initializes objects.

Types:

1. **Default Constructor** – No parameters
2. **Parameterized Constructor** – Takes arguments
3. **Copy Constructor** – Initializes object using another object

Characteristics:

- Same name as class
- No return type
- Automatically called
- Can be overloaded
- Cannot be virtual

Example:

```cpp
#include <bits/stdc++.h>
using namespace std;

class Student {
    string name;
public:
    int age;
    bool gender;

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

    Student(string s, int a, int g) { // Parameterized
        name = s;
        age = a;
        gender = g;
        cout << "Parameterized Constructor" << endl;
    }
```

```
    Student(Student &p) { // Copy
        name = p.name;
        age = p.age;
        gender = p.gender;
        cout << "Copy Constructor" << endl;
    }

    void printinfo() {
        cout << "Name = " << name << "\nAge = " << age << "\nGender = " << gender << "\n\n";
    }
};

int main() {
    Student s1;
    s1.printinfo();

    Student s2("Sumeet", 20, 1);
    s2.printinfo();

    Student s3(s2);
    s3.printinfo();
}
```

Output:

```
Default Constructor
Name =
Age = 0
Gender = 0

Parameterized Constructor
Name = Sumeet
Age = 20
Gender = 1

Copy Constructor
Name = Sumeet
Age = 20
Gender = 1
```

4. Destructor

Special member function that destroys objects when they go out of scope.

Characteristics:

- Name preceded by ~
- No parameters, no return type
- Automatically called
- Cannot be overloaded

Example:

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

int count = 0;

class Num {
public:
    Num() {
        count++;
        cout << "Constructor called for object " << count << endl;
    }
    ~Num() {
        cout << "Destructor called for object " << count << endl;
        count--;
    }
};

int main() {
    Num n1, n2, n3;
}
```

5. Four Pillars of OOPS

1. Inheritance
2. Encapsulation
3. Abstraction
4. Polymorphism

6. Inheritance

Ability of a class to acquire properties of another class.

Types:

Type	Description	Example
Single	One base → One derived	class B: public A
Multiple	Derived from multiple bases	class C: public A, public B
Multilevel	Derived from a derived class	A → B → C

Hierarchical	Multiple derived from same base	B, C: public A
Hybrid	Combination of types	Multiple + Multilevel

Example – Single Inheritance:

```
class A { public: void funcA() { cout << "Base Class\n"; } };
class B : public A { public: void funcB() { cout << "Derived Class\n"; } };
int main() { B obj; obj.funcA(); obj.funcB(); }
```

Output:

```
Base Class
Derived Class
```

7. Encapsulation

Wrapping data & methods together.

Advantages:

- Data hiding
- Modular code
- Increased security

Example:

```
class Encapsulation {
private:
    int x;
public:
    void set(int a) { x = a; }
    int get() { return x; }
};
```

8. Abstraction

Hiding implementation details and showing only necessary info.

Advantages:

- Security
- Reusability
- Simpler interface

Example:

```
class Abstraction {
private:
    int a, b;
public:
    void set(int x, int y) { a = x; b = y; }
    void display() { cout << "a = " << a << ", b = " << b; }
};
```

9. Polymorphism

Many forms – same interface, different implementation.

Types:

1. **Compile-time** → Function overloading, Operator overloading
2. **Run-time** → Function overriding with virtual functions

Example – Function Overloading:

```
class Demo {
public:
    void show() { cout << "No args\n"; }
    void show(int x) { cout << "Int arg\n"; }
};
```

10. Abstract Class & Pure Virtual Function

- Abstract Class: Has at least one pure virtual function
- Pure Virtual Function: `virtual void func() = 0;`

11. Friend Class & Friend Function

- **Friend Class**: Can access private members of another class
- **Friend Function**: Non-member function with access to private data

12. Access Modifiers

Modifier	Access Scope	Modifier
Private	Within class only	Private
Protected	Class + derived classes	Protected
Public	Everywhere	Public

Some Practice Questions :

1. Abstract Class with Pure Virtual Function

Demonstrates abstraction using a pure virtual function and runtime polymorphism.

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

// Abstract class
class Shape {
public:
    virtual void area() = 0; // Pure virtual function
};

class Circle : public Shape {
private:
    float radius;
public:
    Circle(float r) : radius(r) {}
    void area() override {
        cout << "Area of Circle: " << 3.14 * radius * radius << endl;
    }
};
```

```
class Rectangle : public Shape {
private:
    float length, breadth;
public:
    Rectangle(float l, float b) : length(l), breadth(b) {}
    void area() override {
        cout << "Area of Rectangle: " << length * breadth << endl;
    }
};

int main() {
    Shape* s1 = new Circle(5);
    Shape* s2 = new Rectangle(4, 6);
    s1->area();
    s2->area();
    return 0;
}
```

Output:

```
Area of Circle: 78.5
Area of Rectangle: 24
```

2. Protected Inheritance + Constructor Types

Shows constructor overloading, copy constructor, getters/setters, and protected inheritance.

Link

3. Function Overloading

Same function name with different parameter lists (compile-time polymorphism).

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

class Calculator {
public:
    int add(int a, int b) { return a + b; }
    double add(double a, double b) { return a + b; }
    int add(int a, int b, int c) { return a + b + c; }
};

int main() {
    Calculator calc;
    cout << "Sum of 2 integers: " << calc.add(5, 3) << endl;
    cout << "Sum of 2 doubles: " << calc.add(5.5, 2.3) << endl;
    cout << "Sum of 3 integers: " << calc.add(1, 2, 3) << endl;
    return 0;
}
```

Output:

```
Sum of 2 integers: 8
Sum of 2 doubles: 7.8
Sum of 3 integers: 6
```

4. Multiple, Multilevel, Hierarchical & Hybrid Inheritance

Demonstrates various inheritance types and resolving diamond problem using scope resolution.

Link

5. Runtime Polymorphism using Virtual Functions

Base class pointer calls derived class overridden methods at runtime.

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

class Animal {
public:
    virtual void speak() { cout << "Animal speaks" << endl; }
};

class Dog : public Animal {
public:
    void speak() override { cout << "Dog barks" << endl; }
};

class Cat : public Animal {
public:
    void speak() override { cout << "Cat meows" << endl; }
};

void makeItSpeak(Animal* a) {
    a->speak(); // Resolved at runtime
}
```

```cpp
int main() {
    Dog d; Cat c;
    Animal* a1 = &d;
    Animal* a2 = &c;
    makeItSpeak(a1);
    makeItSpeak(a2);
    return 0;
}
```

Output :

```
Dog barks
Cat meows
```