OOPS

1. **What is meant by the term OOPs?**

OOPs stands for Object-Oriented Programming. It’s a programming paradigm where we design and structure our code around objects. Each object represents a real-world entity and contains both data; which we call attributes; and behavior; which we call methods. This approach makes the code modular, reusable, and easier to maintain compared to procedural programming.

2. **Why OOPs?**

The main reason OOPs is preferred is that it makes code more modular, readable, and maintainable, especially for large-scale software. By using concepts like encapsulation, we can change the internal implementation without affecting how other parts of the system interact with it. This makes maintenance easier. OOP also aligns closely with how we think about real-world entities, so both developers and non-technical users can understand the system design better. In short, it improves clarity, reusability, and scalability

3. **What are some major Object-Oriented Programming languages?**

Some of the most widely used Object-Oriented Programming languages include Java, C++, Python, C#, Ruby, and Swift.

4. **What are some other programming paradigms other than OOPs?**

Apart from Object-Oriented Programming, there are two broad categories of programming paradigms - Imperative and Declarative.

- **Imperative Paradigm** focuses on ***how* to achieve a goal** and includes styles like procedural programming and structured programming.

- **Declarative Paradigm** focuses on ***what* the outcome should be**, without specifying control flow. This includes functional programming and database query languages.

5. **What is meant by Structured Programming?**

Structured Programming is a programming approach where the control flow of the program is organized into clear, logical blocks. These blocks usually include constructs like `if-else` statements, loops such as `while` and `for`, and functions. The idea is to make code easier to read, debug, and maintain.

6. **What is the difference between Structured Programming and Object-Oriented Programming?**

Structured Programming is a procedural approach where the program is divided into functions or modules, and execution follows a top-down flow. Object-Oriented Programming, on the other hand, is based on objects that combine both data and behavior. The key differences are:

- **Approach:** Structured programming is top-down; OOP follows a bottom-up approach.
- **Data Security:** OOP uses encapsulation to restrict data access; structured programming has no such restriction.
- **Reusability:** OOP achieves it through inheritance and polymorphism; structured programming relies on functions and loops.
  Overall, OOP makes it easier to maintain and extend large, complex systems compared to structured programming.

7. **What are the advantages and disadvantages of OOPs?**

The advantages of OOPs are:

- **Code reusability** through inheritance and polymorphism.
- **Easier maintenance and updates** because objects encapsulate related data and behavior.
- **Better data security** by controlling access using access modifiers.

- **Lower complexity** for large systems by modeling them as real-world entities. However, OOPs also has some disadvantages:
- It requires **skilled design and object-oriented thinking**, which can have a learning curve.
- **Proper planning** is essential; without it, the benefits can be lost.
- It's **not always suitable** for every type of problem, especially small or straightforward tasks.
- The **program size** can be larger compared to procedural approaches.

8. **Why is OOPs so popular?**

OOPs is popular because it makes it easier to design, write, and maintain complex software. By organizing code around objects and using concepts like abstraction, encapsulation, inheritance, and polymorphism, we can model real-world scenarios more naturally, reuse code efficiently, and make updates with minimal impact on other parts of the system. This combination of clarity, scalability, and reusability is why OOPs is widely adopted across many programming languages and large-scale projects

9. **What are the main features of OOPs?**

The main features of OOPs are:

- **Inheritance:**
  - Inheritance is the process by which one class (child or subclass) acquires the properties and behaviors (fields and methods) of another class (parent or superclass).
  - It promotes **code reusability** and establishes a parent-child relationship.
- **Encapsulation:**
  - Encapsulation is the concept of **wrapping data (variables) and methods** that operate on the data into a single unit (class), while **restricting direct access** to that data.
  - It's achieved using **access modifiers** (`private` , `protected` , `public`).
- **Abstraction:**
  - Abstraction is the process of **hiding implementation details** and showing only the essential features of an object to the user.

- It is implemented using **abstract classes** or **interfaces** in Java.

- **Polymorphism:**
  - Polymorphism is made of 2 words "Poly" means "many" and "morph" means "forms". So, polymorphism is a feature which allows us to perform the same operation in different ways depending on the object or context.
  - We have 2 types of polymorphism: **Compile-time Polymorphism**(method overloading) and **Runtime Polymorphism** (method overriding).
  - **Compile-time Polymorphism** *(Static polymorphism / Early binding)*
    - The method to be executed is decided at compile time.
    - Achieved using **method overloading** and **operator overloading**.
    - Example:

      ```
      void print(int x) { cout << x; }
      void print(string s) { cout << s; }
      ```

      Here, the correct `print` function is chosen at compile time.
  - **Runtime Polymorphism** *(Dynamic polymorphism / Late binding)*
    - The method to be executed is decided at runtime based on the actual object type.
    - Achieved using **method overriding** with **virtual functions** in C++.
    - Example:

      ```
      class Base {
          virtual void show() { cout << "Base"; }
      };
      class Derived : public Base {
          void show() override { cout << "Derived"; }
      };
      ```

      - If a `Base*` points to a `Derived` object, the `Derived` version runs at runtime.
  - Compile-time means decision taken before the program runs and Runtime means decision taken while it runs.

Example showing all 4 features of OOPS:

```
class Animal{
    protected:
    string name; // Encapsulated - not directly accessible
```

```
    public:
    Animal() {
        name = "";
    }
    Animal(string s){
        name = s; // Assigning inside constructor
    }
    void setName(string s){
        name = s;
    }
    string getName(){ // Encapsulation: Accessing via method
        return name;
    }

    virtual void makeSound() = 0; // Abstraction

    // Compile-time Polymorphism: function overloading
    void eat(){
        cout << name << "is eating" << endl;
    }
    void eat(string food){
        cout << name << "is eating" << food << endl;
    }
};

// Inheritance
class Dog: public Animal{
    public:
    Dog(string n){
        setName(n);
    }

    void makeSound() override{
        cout << name << "I am a dog" << endl;
    }
};

class Cat: public Animal{
    public:
    Cat(string n){
        setName(n);
    }

    void makeSound() override{
```

```
        cout << name << "I am a cat" << endl;
    }
};

int main(){
    Animal* a1 = new Dog("Tommy");
    Animal* a2 = new Cat("Mikey");

    a1->makeSound(); // Method Overriding
    a2->makeSound();

    a1->eat(); // Method Overloading
    a2->eat("fish");

    delete a1;
    delete a2;
    return 0;
}
```

- *Encapsulation: name is protected, accessed through methods.*
- *Abstraction: makeSound() is a pure virtual method; only the interface is known.*
- *Inheritance: Dog and Cat inherit from Animal.*
- *Polymorphism: makeSound() is overridden (runtime), eat() is overloaded (compile-time).*
- The reason we are writing "delete" at the end is to prevent memory leaks. Memory leaks means the memory stays allocated even after my program can't access it anymore. C++ doesn't have garbage collection. For dynamic allocation with `new`, you must use `delete`

10. **What is a class?**

A class is a template in Object-Oriented Programming that defines the structure and behavior of objects. It contains **member variables** to store state and **member functions** to define behavior. When we create an object from a class, the object gets its own copy of the data and can use the methods defined in the class.

11. **What is an object?**

An object is an instance of a class. It represents a specific entity with its own state and behavior, defined by the class it's created from. We can't directly use a class's members. We first create an object, and then access its data members and methods through that object.
For example:

```
class Student {
public:
    string name;
};

int main(){
    // Creating object
    Student student1;

    // Assigning member some value
    student1.name = "Rahul";
    cout << "student1.name: " << student1.name;

    return 0;
}
```

12. **What is encapsulation?**

Encapsulation is the process of bundling variables (data) and the methods that operate on that data into a single unit i.e. a class, while restricting direct access to some of the object's components.
It serves two main purposes:

1. **Data Hiding** – It prevents unauthorized access to sensitive data, usually using access modifiers like `private` or `protected`.
2. **Data Binding** – Keeping related data and behavior together so they can be managed as a whole.
   For example:

```
class BankAccount{
    private:
    double balance; // Hidden data
```

```
    public:
    BankAccount(double initialBalance) {
        balance = initialBalance;
    }

    // Data Binding: balance is accessed/modified only via methods
    void deposit(double amount) {
        balance += amount;
    }
    void withdraw(double amount) {
        balance -= amount;
    }
    double getBalance() {
        return balance;
    }
};
int main() {
    BankAccount account(1000); // Data is hidden inside object
    account.deposit(500);      // Accessing via methods
    account.withdraw(200);
    cout << "Final Balance: " << account.getBalance() << endl;

    // account.balance = 5000; // ❌ Not allowed (data hidden)
}
```

- **Data Hiding -** `balance` is `private`, so it can't be accessed directly from outside.
- **Data Binding** - The only way to interact with `balance` is through the class methods (`deposit`, `withdraw`, `getBalance`), binding the data and behavior together.

13. **What is abstraction? How is data abstraction accomplished?**

Abstraction is about showing **only the necessary details** to the user and hiding the internal complexity. It focuses on **what** an object does rather than **how** it does it. For example:

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

```
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Circle" << endl; // Internal details
hidden from user
    }
};

int main() {
    Shape* shape = new Circle();
    shape->draw(); // User just calls draw(), doesn't know HOW it's
drawn
    delete shape;
    return 0;
}
```

Explanation:

- `Shape` is an **abstract class** because it has a **pure virtual function** (`draw()`).
- The **user only calls** `draw()` without knowing the steps to draw a circle.
- The **internal logic** (like coordinates, algorithms, etc.) is hidden in the `Circle` class.
  We can also achieve **data abstraction** using **access modifiers**:

```
class BankAccount {
private:
    double balance; // Hidden data
public:
    BankAccount(double initialBalance) : balance(initialBalance) {}
    void deposit(double amount) {
        balance += amount; // Internal calculation hidden
    }
    double getBalance() {
        return balance; // Only necessary info exposed
    }
};
```

This way, the user **cannot directly change** `balance` but can only interact through provided methods.

14. **What is an abstract class?**

An abstract class is a class that can’t be instantiated on its own and may contain one or more abstract methods; methods that are declared but not implemented. Its main purpose is to provide a common base for derived classes, ensuring they implement specific functionality.
For example, (refer above)

15. **What is an interface?**

An interface is essentially a contract that defines a set of method signatures without providing their implementation. It can’t hold state and you can’t create objects from it directly. Any class implementing an interface must provide concrete implementations for all its declared methods. This is useful for achieving abstraction and ensuring multiple classes follow the same structure, which is key for polymorphism.

16. **How is an abstract class different from an interface?**

- An **abstract class** can have both abstract methods (without implementation) and concrete methods (with implementation). It can also have fields, constructors, and access modifiers. When a class inherits from an abstract class, it is not mandatory to implement all abstract methods unless they are actually used. For example:

```
class Animal {   // Abstract class
public:
    virtual void makeSound() = 0; // abstract method
    void breathe() {              // concrete method
        cout << "Breathing...\n";
    }
};
class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Woof!\n";
```

```
    }
};
int main() {
    Dog d;
    d.makeSound();  // Woof!
    d.breathe();    // Breathing...
}
```

- An **interface** is purely a contract — it contains only method declarations (and default/static methods in some languages) without implementation. When a class implements an interface, it must provide implementations for **all** its methods. Also, a class can implement multiple interfaces but can inherit from only one abstract class. For example:
  (In Java)

```
interface Vehicle {
    void start();   // only declaration
    void stop();
}
class Car implements Vehicle {
    public void start() { System.out.println("Car starting..."); }
    public void stop()  { System.out.println("Car stopping..."); }
}
class Bike implements Vehicle {
    public void start() { System.out.println("Bike starting..."); }
    public void stop()  { System.out.println("Bike stopping..."); }
}
public class Main {
    public static void main(String[] args) {
        Vehicle v = new Car();
        v.start(); // Car starting...
        v.stop();  // Car stopping...
    }
}
```

(In C++)

```
// Interface
class Shape {
public:
    // pure virtual function = interface method
    virtual void draw() = 0;
```

```
};

// Class implementing the interface
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a Circle\n";
    }
};
class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing a Square\n";
    }
};

int main() {
    Shape* s1 = new Circle();
    Shape* s2 = new Square();

    s1->draw();  // Output: Drawing a Circle
    s2->draw();  // Output: Drawing a Square

    delete s1;
    delete s2;
}
```

- Note: In C++ a subclass can implement multiple abstract classes (i.e., multiple interfaces). Since C++ supports multiple inheritance, the derived class must provide concrete definitions for all the pure virtual functions it inherits. However, we need to be careful about ambiguity issues like the diamond problem, which can be resolved using virtual inheritance.

17. **What is the difference between a structure and a class in C++?**

Both structures and classes in C++ are user-defined data types, but there are key differences:

- **Default Access Modifier:** In a structure, members are `public` by default, while in a class, they're `private` by default.

- **Keyword Used:** We use `struct` to define a structure and `class` to define a class.

18. **What are friend functions and friend classes?**

Friend functions are special non-member functions that are granted access to the private and protected members of a class. Similarly, a friend class is a class that has access to another class's private and protected members. They're useful when two classes or functions need close cooperation without making everything public. For example:

```
class B;
class A{
    int a;
    public:
    friend void fun(A, B);
};
class B{
    int b;
    public:
    friend void fun(A, B);
};
void fun(A o1, B o2){
    cout << o1.a + o2.b;
}
int main(){
    A obj1; B obj2;
    obj1.setData(2); obj2.setData(3);
    fun(obj1, obj2);
}
```

19. **What are the various types of inheritance?**

Inheritance in OOP can be classified into five main types:

- **Single Inheritance** – A child class inherits directly from one base class.
- **Multilevel Inheritance** – A class inherits from another derived class, forming a chain.

- **Multiple Inheritance** – A child class inherits from more than one base class. *(Note: Some languages like Java don't support this directly, but simulate it using interfaces.)*
- **Hierarchical Inheritance** – Multiple child classes inherit from the same base class.
- **Hybrid Inheritance** – A combination of two or more of the above types.
  The type supported depends on the programming language in use.

20. **Are there any limitations on Inheritance?**

As for limitations of inheritance: While it's powerful for code reusability, it comes with trade-offs:

- **Performance overhead** – The program may take longer to process because it needs to navigate through multiple class layers.
- **Tight coupling** – The child and base classes become closely linked, so changes in the base class can force changes in all derived classes.
- **Complexity** – Inheritance hierarchies can become hard to manage, and if not implemented carefully, they can lead to unexpected bugs or unintended behaviors.

21. **What is a subclass?**

A subclass is a class that inherits properties and behaviors (fields and methods) from another class, called the superclass. It can also have its own additional features or override the inherited ones. Subclasses are also known as child classes or derived classes.

22. **Define a superclass.**

A superclass is a class whose properties and behaviors are inherited by one or more subclasses. It defines the common features that can be reused by its subclasses. Super classes are also called parent classes or base classes.

23. **How does C++ support polymorphism?**

C++ supports polymorphism in two main ways — compile-time and runtime.

- **Compile-time polymorphism**: Achieved via *function overloading* and *operator overloading*. In this case, the decision of which function to call is made at **compile time**.
  Example:
  `void print(int x); void print(double y);`
  The compiler chooses which `print` to call based on the argument type.
- **Runtime polymorphism**: Achieved via *function overriding* using **virtual functions** and inheritance. In this case, the function call is resolved at **runtime** using the **virtual method table mechanism**.
  Example:

```
class Base {
    virtual void show();
};
class Derived : public Base {
    void show() override;
};
```

  Here, the function to be executed depends on the type of the object pointed to at runtime.

24. **What is the virtual function and pure virtual function?**

A virtual function is a member function in a base class that you expect to override in a derived class. It enables **runtime polymorphism** via the **vtable** mechanism in C++. We declare it with the `virtual` keyword in the base class. When called through a base class pointer or reference, the derived class's version is executed if overridden.
For example:

```
class Base {
public:
    virtual void display() { cout << "Base display\n"; }
};

class Derived : public Base {
public:
```

```
    void display() override { cout << "Derived display\n"; }
};
```

A pure virtual function is a virtual function with `= 0` at the end of its declaration, meaning it has **no implementation** in the base class and must be implemented in the derived class. A class with at least one pure virtual function becomes an **abstract class**, and cannot be instantiated.

```
class Shape {
public:
    virtual void draw() = 0; // Pure virtual
};
```

In short:

- **Virtual function** → Has a definition, can be overridden.
- **Pure virtual function** → No definition, must be implemented in derived classes."

25. **How much memory does a class occupy?**

In C++, a class by itself does not occupy memory — it's just a blueprint or template for creating objects. The memory is allocated only when we create an object, and the size depends on the non-static data members of that class. Static members are not part of the object's memory footprint; they are stored separately in the data segment.

26. **Is it always necessary to create objects from class?**

No, it's not always necessary. If the class contains only static methods or static data members, we can access them directly using the class name without creating an object. However, if we want to access non-static members, we must create an object because those members belong to the instance, not the class itself.

27. **What is a constructor?**

A constructor is a special member function in a class that is automatically invoked when an object is created. Its primary purpose is to initialize the object's data members. Unlike regular functions, constructors don't have a return type. Their name must exactly match the class name. For example, in C++:

```
class Base {
    public:
    Base() {
        cout << "Constructor called";
    }
};
```

Whenever we create an object of `Base`, this constructor will be executed.

28. **What are the various types of constructors in C++?**

In C++, we generally classify constructors into four main types:

- **Default Constructor** – A constructor with no parameters, automatically created by the compiler if we don't define any. It initializes members to default values. For example:

```
class A {
public:
    int x;
    // No constructor defined → compiler generates default one
};
int main() {
    A obj; // Calls default constructor
    cout << obj.x; // May print garbage value
}
```

- **Non-Parameterized Constructor** – A user-defined constructor with no arguments. Similar to a default constructor, but explicitly coded by the developer. For example:

```
class A {
public:
    int x;
    A() { // Non-parameterized constructor
```

```
        x = 10;
    }
};
int main() {
    A obj; // Calls our defined constructor
    cout << obj.x; // Output: 10
}
```

- **Parameterized Constructor** – Accepts arguments so we can initialize objects with specific values at creation.

```
class A {
public:
    int x;
    A(int val) { // Parameterized constructor
        x = val;
    }
};
int main() {
    A obj(42); // Pass value while creating
    cout << obj.x; // Output: 42
}
```

- **Copy Constructor** – Creates a new object as a copy of an existing one, usually taking a reference to an object of the same class.
  Example of Copy Constructor:

```
class A {
public:
    int x;
    A(int val) { // Parameterized
        x = val;
    }
    A(A &obj) { // Copy constructor
        x = obj.x;
    }
};
int main() {
    A obj1(50);  // Original
    A obj2 = obj1; // Calls copy constructor
    cout << obj2.x; // Output: 50
}
```

If we don’t define a copy constructor, C++ provides a default one, but in cases involving dynamic memory, we often implement it ourselves to avoid shallow copying issues.

29. **Can we overload the constructor in a class?**

Yes. Constructor overloading is possible in many OOP languages.
It means defining multiple constructors in the same class, but with **different parameter lists**—either by **number of parameters** or by **types of parameters**. This allows us to create objects in different ways depending on the available data.

```
class Example {
    Example() { /* default constructor */ }
    Example(int x) { /* parameterized constructor */ }
    Example(String s, int x) { /* another parameterized constructor
*/ }
}
```

30. **What is a destructor?**

A destructor is a special method that is called **automatically** when an object is explicitly destroyed. Its job is to release resources, close files, free memory, or perform cleanup tasks.

- **In C++:** Same name as the class but prefixed with `~`. Example:

```
class Base {
public:
    ~Base() { cout << "Destructor called"; }
};
```

31. **Can we overload the destructor in a class?**

No. A class can have **only one destructor**, and it **cannot be overloaded** because it doesn’t take parameters and cannot return a value. The language ensures there’s a single, predictable cleanup mechanism.

32. **What are access specifiers and what is their significance?**

Access specifiers in C++ are keywords that define the accessibility of class members. The three main ones are `public`, `protected`, and `private`. They help implement encapsulation and data hiding by controlling how data and methods can be accessed from outside the class. For example, `private` members are accessible only within the class, `protected` members are accessible within the class and derived classes, and `public` members are accessible from anywhere. This is fundamental to maintaining control over how class data is used and modified.

33. **What is meant by garbage collection in OOPs world?**

In OOP, objects occupy memory, and if unused objects are not properly released, it can lead to memory leaks. Garbage collection is the process of automatically reclaiming memory occupied by objects that are no longer reachable or in use. While languages like Java have built-in garbage collectors, in C++ memory management is manual, meaning we explicitly deallocate memory using `delete` or `delete[]`. Proper memory handling ensures efficient resource usage and prevents system crashes due to memory exhaustion.

34. **What is an exception?**

An exception is an unexpected event that occurs during the execution of a program, disrupting its normal flow. It usually arises when the program encounters a situation it wasn't designed to handle, such as invalid input, division by zero, or accessing an out-of-bound array index. Essentially, it signals that something has gone wrong at runtime. For example:

```
int divide(int a, int b) {
    if (b == 0) {
        throw runtime_error("Division by zero is not allowed!");
    }
    return a / b;
}
```

```
int main() {
    try {
        int result = divide(10, 0);  // This will throw an
exception
        cout << "Result: " << result << endl;
    }
    catch (runtime_error& e) {
        cout << "Error caught: " << e.what() << endl;
    }

    cout << "Program continues..." << endl;
    return 0;
}
```

35. **What is meant by exception handling?**

Exception handling is the mechanism used to detect and manage unexpected situations without crashing the program. Instead of terminating abruptly, we can anticipate possible errors, catch them, and define alternative flows or recovery actions. The most common approach is using `try-catch` blocks, where the `try` section contains code that might throw an exception, and the `catch` section defines how to handle it. This ensures the program remains robust and user-friendly.

36. **Can we run a Java application without implementing the OOPs concept?**

Strictly speaking, a Java application must have at least one class, since Java is a class-based language. So while OOP is the core design paradigm of Java, it is *possible* to write and run small Java programs without actively applying all OOPs concepts — but you cannot avoid having at least one class definition.
Example:

```
public class Main {
    public static void main(String[] args) {
        int a = 5, b = 3;
        System.out.println("Sum: " + (a + b));
    }
}
```

This doesn't use OOP features, but still requires a class to run.

37. **Explain early binding and late binding in C++?**

In C++, **binding** means deciding which function to call for a given function call.

- **Early Binding** (also called **Static Binding**):
  - The function call is resolved **at compile time**.
  - This happens with **normal function calls** and **function overloading**.
  - Example:
  `class Base { public: void show() { // Non-virtual function cout << "Base class function (Early Binding)\n"; } }; int main() { Base obj; obj.show(); // Compiler decides at compile time which function to call }`
  Here, the compiler knows at compile time that `obj.show()` refers to `Base::show()` — that's **early binding**.
- **Late Binding** (also called **Dynamic Binding**):
  - The function call is resolved **at runtime**.
  - This happens when you use **virtual functions** and call them via **base class pointers/references**.
  - Example:

```
class Base {
public:
    virtual void show() { // Virtual function
        cout << "Base class function (Late Binding)\n";
    }
};

class Derived : public Base {
public:
    void show() override {
        cout << "Derived class function (Late Binding)\n";
    }
};

int main() {
    Base* ptr;
    Derived d;
    ptr = &d;
```

```
    ptr->show();  // Resolved at runtime (Late Binding)
}
```

Here, the compiler knows that `ptr` is a `Base*`, but it **doesn't decide at compile time** which version of `show()` to call.
At **runtime**, since `ptr` points to a `Derived` object, it calls `Derived::show()` — that's **late binding**.

38. **Explain deep copy and shallow copy ?**

In C++, when we talk about copying objects, we generally have two types: **shallow copy** and **deep copy**.

- **Shallow Copy**
  - It copies all member values **bit by bit**, including pointers.
  - This means if two objects share the same pointer, changes made through one object affect the other.
  - Default copy constructor provided by the compiler does a shallow copy.
  - Example:

```
class Shallow {
public:
    int* data;
    Shallow(int val) {
        data = new int(val); // dynamically allocate
    }
    // No custom copy constructor → compiler-generated shallow
copy
    ~Shallow() {
        delete data;
    }
};

int main() {
    Shallow obj1(10);
    Shallow obj2 = obj1;  // shallow copy → same pointer

    *(obj2.data) = 20;    // changes obj1 as well
    cout << *(obj1.data); // prints 20
}
```

    - **Problem:** When `obj1` and `obj2` go out of scope, `delete` is called twice on the same memory, leading to undefined behavior (often a crash).
- **Deep Copy**
    - Instead of copying the pointer as-it-is, we **allocate new memory** and copy the contents.
    - This ensures each object has its own independent copy of the data.
    - Example:

```
class Deep {
public:
    int* data;
    Deep(int val) {
        data = new int(val);
    }
    // Custom deep copy constructor
    Deep(const Deep &obj) {
        data = new int(*obj.data);  // new memory + copy value
    }
    ~Deep() {
        delete data;
    }
};

int main() {
    Deep obj1(10);
    Deep obj2 = obj1; // deep copy → independent memory

    *(obj2.data) = 20;
    cout << *(obj1.data) << endl; // prints 10 → unaffected
}
```

- Whenever the class contains **pointers or dynamically allocated memory** and we want each object to manage its own copy of data without interference, we must use deep copy. Otherwise, shallow copy can cause **double deletion**, **data corruption**, or **unexpected side effects**.

39. **Explain diamond problem with example?**

The diamond problem occurs in **hybrid inheritance** when a class inherits from two classes that have a common base class.

Let’s say we have a class structure like this:

```
    A
   / \
  B   C
   \ /
    D
```

- **A** is the base class.
- **B** and **C** both inherit from **A**.
- **D** inherits from both **B** and **C**.
  Now, without any special handling, **D** will end up with **two copies** of A’s members—one through B and one through C. This can cause **ambiguity**. For example:

```
class A {
public:
    void show() { cout << "Class A\n"; }
};

class B : public A {};
class C : public A {};
class D : public B, public C {};

int main() {
    D obj;
    // obj.show();  // ❌ Ambiguous — compiler doesn't know whether
to call B::A::show or C::A::show
}
```

We solve this problem by using the concept of **virtual inheritance**. By making B and C inherit **virtually** from A, C++ ensures there’s only **one shared instance** of A in the hierarchy. So the fix for above is:

```
class A {
public:
    void show() { cout << "Class A\n"; }
};

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

```
class D : public B, public C {};

int main() {
    D obj;
    obj.show();  // ✅ No ambiguity — single A instance &&
rightmost class value is considered
}
```

So in short,

- **Problem:** Multiple inheritance from a common base causes **duplicate copies** and ambiguity.
- **Solution:** Use **virtual inheritance** to ensure only **one copy** of the common base exists.

40. **Consider the following code snippet, and tell me the output?**

```
class A {
public:
    void func1();
};

class B : public A {
public:
    void func2();
};

class C : public A {
public:
    void func2();
};

class D : public B, public C {
public:
    void func3();
};

int main() {
    D obj;
    obj.func2();
}
```

We can resolve this by telling the compiler exactly which version you want:

- `obj.B::func2(); // calls B's func2`
- `obj.C::func2(); // calls C's func2`

41. **What is the Singleton design pattern?**

The Singleton class ensures that **only one instance** of a class exists in the program and provides a **global point of access** to it.
It's useful when exactly one object is needed, like a **logger, database connection, or configuration manager**. For example:

```
class Singleton {
private:
    static Singleton* instance;
    Singleton() {}
public:
    static Singleton* getInstance() {
        if (instance == nullptr) instance = new Singleton();
        return instance;
    }
};
Singleton* Singleton::instance = nullptr;
```

- Here, the constructor is private, so no one can directly create an object of `Singleton`.
- We hold a static pointer `instance` inside the class.
- When `getInstance()` is called the first time, we create the object. Any further calls just return the same instance.
  So:
- `Singleton* s1 = Singleton::getInstance();`
- `Singleton* s2 = Singleton::getInstance();`
  `s1` and `s2` will point to the **same object**, proving Singleton.

42. **Explain the Factory pattern?**

The Factory pattern provides an **interface for creating objects**, but lets subclasses decide which class to instantiate.

It hides object creation logic, so client code only depends on an abstract type, not concrete implementations. For example:

```
class Shape {
public:
    virtual void draw() = 0;
};
class Circle : public Shape {
public: void draw() { cout << "Circle\n"; } };
class Square : public Shape {
public: void draw() { cout << "Square\n"; } };

class ShapeFactory {
public:
    static Shape* getShape(string type) {
        if (type == "circle") return new Circle();
        else if (type == "square") return new Square();
        return nullptr;
    }
};
```

- The `ShapeFactory` hides object creation logic. Instead of writing: `Circle* c = new Circle();`
- The client just calls: `Shape* s = ShapeFactory::getShape("circle"); s->draw();`
- So the client doesn't worry about which concrete class to create. **Why?** If tomorrow I add a `Triangle`, client code doesn't change, only factory logic does.

43. **What is the Observer pattern?**

The Observer defines a **one-to-many dependency** so that when one object (Subject) changes state, all its dependents (Observers) are notified automatically. For example

- Think of YouTube: when a channel uploads, all subscribers are notified.

```
class Observer {
public:
    virtual void update(string msg) = 0;
```

```
};
class Subscriber : public Observer {
public:
    void update(string msg) { cout << "Notification: " << msg <<
endl; }
};
class Channel {
    vector<Observer*> subs;
public:
    void subscribe(Observer* o) { subs.push_back(o); }
    void notify(string msg) {
        for(auto s: subs) s->update(msg);
    }
};
```

- We have a `Channel` (Subject) and many `Subscribers` (Observers). When the channel calls `notify("New Video")`, all subscribers automatically get updated.

```
Channel channel;
Subscriber s1, s2;
channel.subscribe(&s1);
channel.subscribe(&s2);
channel.notify("New Video Uploaded!");
```

```
1. `&s1` → address of `s1`, which is a `Subscriber` object (and
`Subscriber` inherits `Observer`).
2. - The pointer is passed as an `Observer* o`.
3. Inside `subscribe()`, that pointer is stored in the `subs`
vector.
```

- Both `s1` and `s2` get the notification.

44. **Explain procedural programming. How it is different from OOPS ?**

Procedural Programming is a programming paradigm that is **based on the concept of functions**. The main idea is to break down a program into a set of smaller, reusable functions that operate on data.

- The focus is **on functions**: you define procedures that take inputs, process them, and return outputs.
- Data and functions are usually **separate**.
- Execution is generally **top-down** and follows a sequence of steps.
  For example, in C language, I might write a function `calculateSum(int a, int b)` and call it wherever needed — the logic is organized around procedures.
  Difference from OOPS (Object-Oriented Programming):
- **Focus**
  - Procedural: Focus on **functions** (what to do).
  - OOPS: Focus on **objects** (data + behavior bundled together).
- **Data Handling**
  - Procedural: Data is usually **global** or passed explicitly to functions. Functions can operate on it freely.
  - OOPS: Data is **encapsulated** inside objects; access is restricted through methods.
- **Reusability**
  - Procedural: Code reuse happens mainly through function calls.
  - OOPS: Supports **inheritance, polymorphism**, which provide richer reusability and flexibility.
- **Example**
  - In C (procedural), I might have `int balance` as a variable and `deposit(balance, amount)` as a function.
  - In OOPS (say C++), I'd create a `BankAccount` class with `deposit()` and `withdraw()` methods, and the **balance stays protected inside the object**.

45. **What is the difference between C and C++?**

C is a **procedural programming language**, while **C++** is a **multi-paradigm language** that supports both procedural and object-oriented programming. So "C" has no classes, inheritance, polymorphism, or operator overloading, whereas, C++ has all of those, making it more suitable for large-scale applications.

46. **Explain SOLID principles ?**

The **SOLID principles** are five design principles in Object-Oriented Programming that help us write clean, maintainable, and extensible code. Let me go one by one:

- **S — Single Responsibility Principle (SRP)**
  - It says, a class should have only one reason to change, meaning it should only do one job. For example:

```
// ❌ Bad: This class handles both report generation and
saving to file
class Report {
public:
    void generateReport() {
        cout << "Generating Report..." << endl;
    }
    void saveToFile(string filename) {
        cout << "Saving Report to " << filename << endl;
    }
};

// ✅ Good: Split responsibilities
class Report {
public:
    void generateReport() {
        cout << "Generating Report..." << endl;
    }
};

class ReportSaver {
public:
    void saveToFile(string filename) {
        cout << "Saving Report to " << filename << endl;
    }
};
```

- **O — Open/Closed Principle (OCP)**
  - It says, classes should be open for extension but closed for modification i.e. we should be able to add new functionality without changing existing code. For example:

```
// ❌ Bad: Every time we add a new shape, we must modify
this class
class AreaCalculator {
public:
    double calculate(string shape, double a, double b=0) {
```

```
        if (shape == "rectangle") return a * b;
        else if (shape == "circle") return 3.14 * a * a;
        return 0;
    }
};

// ✅ Good: Open for extension via inheritance, closed for
modification
class Shape {
public:
    virtual double area() = 0;
};

class Rectangle : public Shape {
    double width, height;
public:
    Rectangle(double w, double h) : width(w), height(h) {}
    double area() override { return width * height; }
};

class Circle : public Shape {
    double radius;
public:
    Circle(double r) : radius(r) {}
    double area() override { return 3.14 * radius * radius;
}
};
```

- **L — Liskov Substitution Principle (LSP)**
  - It says, objects of a superclass should be replaceable with objects of a subclass without breaking the program. If a class `B` inherits from class `A`, then anywhere in the program where we use an object of `A`, we should be able to substitute it with an object of `B` without causing errors or unexpected behavior. For example:

```
class Bird {
public:
    virtual void fly() {
        cout << "Bird is flying" << endl;
    }
};

class Sparrow : public Bird {
public:
```

```
    void fly() override {
        cout << "Sparrow flies high" << endl;
    }
};

class Penguin : public Bird {
public:
    void fly() override {
        cout << "Penguin cannot fly!" << endl; // violates
LSP
    }
};
```

- **Where it breaks LSP:**
  - If a function expects a `Bird`, it assumes all birds can fly.
  - Substituting a `Penguin` breaks this expectation → violates LSP.
  - **Fix:** Don't force all birds to fly. Split the hierarchy:

```
class Bird { };
class FlyingBird : public Bird {
public:
    virtual void fly() = 0;
};
class NonFlyingBird : public Bird { };

class Sparrow : public FlyingBird {
public:
    void fly() override { cout << "Sparrow flies high"
<< endl; }
};

class Penguin : public NonFlyingBird { };
```

- **I — Interface Segregation Principle (ISP)**
  - It says, clients should not be forced to depend on interfaces they don't use. For example:

```
// ❌ Bad: One big interface
class Machine {
public:
    virtual void print() = 0;
    virtual void scan() = 0;
    virtual void fax() = 0;
};
```

```
// A simple printer is forced to implement scan & fax
unnecessarily
class SimplePrinter : public Machine {
public:
    void print() override { cout << "Printing...\n"; }
    void scan() override {}   // empty, irrelevant
    void fax() override {}    // empty, irrelevant
};

// ✅ Good: Split into smaller interfaces
class Printer {
public:
    virtual void print() = 0;
};

class Scanner {
public:
    virtual void scan() = 0;
};

class SimplePrinter2 : public Printer {
public:
    void print() override { cout << "Printing...\n"; }
};
```

- **D — Dependency Inversion Principle (DIP)**
  - It says, high-level modules should not depend on low-level modules, both should depend on abstractions. For example:

```
❌ Bad (violates DIP)
class EmailService {
public:
    void sendEmail(string msg) {
        cout << "Sending EMAIL: " << msg << endl;
    }
};

class Notification {
    EmailService email;  // tightly coupled to Email
public:
    void alert(string msg) {
        email.sendEmail(msg);
```

```
    }
};
```

- Here, `Notification` (a high-level class) is **dependent on EmailService** directly. If tomorrow we want to add **SMS** or **Push notifications**, we must modify `Notification` → breaks Open/Closed principle too.

```
✅ Good (follows DIP)
// Abstraction
class MessageService {
public:
    virtual void send(string msg) = 0;
};

// Low-level implementations
class EmailService : public MessageService {
public:
    void send(string msg) override {
        cout << "Sending EMAIL: " << msg << endl;
    }
};

class SMSService : public MessageService {
public:
    void send(string msg) override {
        cout << "Sending SMS: " << msg << endl;
    }
};

// High-level depends on abstraction
class Notification {
    MessageService* service;
public:
    Notification(MessageService* svc) : service(svc) {}
    void alert(string msg) {
        service->send(msg);
    }
};

int main() {
    EmailService email;
    SMSService sms;
    Notification n1(&email);
```

```
    n1.alert("Interview at 10 AM");
    Notification n2(&sms);
    n2.alert("Interview Reminder!");
}
```

- **Why is this better?**
  - `Notification` doesn't care whether the message goes via Email, SMS, or Push.
  - We can add new services without touching `Notification`.
  - Both high-level (`Notification`) and low-level (`EmailService`, `SMSService`) depend only on the abstraction (`MessageService`).

47. **Explain initializer list ?**

In C++, when we say **"initializer list"**, we usually mean a **constructor initializer list**. An initializer list initializes class members **before** the constructor body runs. It's the syntax with a colon (`:`) after the constructor signature, used to initialize data members **before** the constructor body executes. It's required for const members, references, and base classes, and it's also more efficient than assigning inside the constructor body. For example:

```
class Parent {
public:
    Parent(int x) { cout << "Parent constructor " << x << endl; }
};

class Child : public Parent {
    int y;
public:
    Child(int a, int b) : Parent(a), y(b) {   // calling Parent
constructor
        cout << "Child constructor" << endl;
    }
};
```

48. **Difference between *initialization* (using initializer list) and *assignment* (inside constructor body) in C++?**

In C++, **initialization** and **assignment** look similar but are different in how they work:

- **Initialization (Initializer List)**
  - Happens **before** the constructor body executes.
  - Members are **directly constructed** with the given values.
  - No extra overhead.

```
class Demo {
    int x;
public:
    Demo(int val) : x(val) {  // initialization
        // x is already constructed with 'val'
    }
};
```

- **Assignment (Inside Constructor Body)**
  - First, members are **default-constructed**.
  - Then, inside the constructor body, we **assign new values**.
  - Extra step → less efficient.

```
class Demo {
    int x;
public:
    Demo(int val) {
        x = val;   // assignment (after default
construction)
    }
};
```

- **Key Example with const and reference members**

```
class Example {
    const int a;
    int &b;
public:
    Example(int x, int &y) : a(x), b(y) { }  // ✅ works
    // Example(int x, int &y) { a = x; b = y; }  ❌ Error
};
```

  - **Const** and **reference** members **must** be initialized in the initializer list.
  - They cannot be assigned later inside the constructor body.
- **Performance difference**

```
class MyString {
    string str;
public:
    MyString(string s) : str(s) { }   // initialization
(construct directly with s)
    MyString(string s) { str = s; }   // assignment (first
default construct str, then assign s)
};
```

- First version: only one constructor call.
- Second version: default constructor + assignment → extra overhead.