Q. What is Object-Oriented Programming Paradigm?

Object-Oriented Programming (OOP) is a programming style (or paradigm) that is based on the concept of **"objects"**, which can contain **data** (in the form of variables, called **attributes**) and **code** (in the form of functions, called **methods**). Instead of writing procedures or functions that operate on data, in OOP, we group both the data and the functions together into one unit — called an **object**.

This approach is inspired by how we understand and interact with the real world. For example, a **Car** can be considered an object — it has properties like color, model, speed, and it performs actions like drive(), brake(), etc.

OOP helps make programs more **modular**, **reusable**, and **easier to manage**, especially in large software systems.

Key Features of Object-Oriented Programming

1. **Class and Object**
   - A **class** is like a blueprint or template that defines the structure and behavior of objects.
   - An **object** is an actual instance of a class.
   - For example, class Car is a blueprint, and Car car1; is an object made from that blueprint.
2. **Encapsulation**
   - Encapsulation means **binding data and methods** that work on the data into a single unit (class).
   - It helps in **data hiding** — the internal details of an object are hidden from the outside world.
   - We use **access specifiers** like private, public, and protected to control access.
3. **Abstraction**
   - Abstraction means **hiding complex details** and showing only the essential features.
   - For example, when you drive a car, you just use the steering and pedals — you don’t need to know how the engine works internally.
4. **Inheritance**
   - Inheritance allows a class (**child**) to **inherit** properties and behaviors from another class (**parent**).
   - It promotes **code reusability**.
   - Example: A SportsCar class can inherit from the Car class.
5. **Polymorphism**

- Polymorphism means **one function or method behaves differently** depending on the context.
- There are two types:
  - **Compile-time polymorphism** (function overloading, operator overloading)
  - **Run-time polymorphism** (using virtual functions and inheritance)

Conclusion

OOP makes programming more **organized**, **realistic**, and **manageable**. It is the backbone of modern programming languages like **C++, Java, Python**, etc., and is widely used in software development today.

Q. Basic Class Structure in C++

In C++, a **class** is a user-defined data type that acts as a **blueprint for creating objects**. It groups **data members** (variables) and **member functions** (methods) together. These members can have different access levels: public, private, or protected.

Syntax of a Basic Class:

```
class ClassName {
  // Access specifier
  public:
    // Data members
    int a;
    // Member functions
    void display() {
      cout << "Value of a: " << a;
    }
};
```

You can create an **object** from a class to access its members:

```
ClassName obj;
obj.a = 5;
obj.display();
```

Access Specifiers:

- **public**: Members are accessible from anywhere.
- **private**: Members are accessible only inside the class.
- **protected**: Accessible in the class and its derived classes (used in inheritance).

Base Class vs Derived Class

C++ supports **inheritance**, where one class (called a **derived class**) inherits the properties and behaviors (data and functions) of another class (called a **base class**).

Feature	**Base Class**	**Derived Class**
Definition	The original class whose features are inherited.	The class that inherits from the base class.

Feature	Base Class	Derived Class
Access	Members are accessed directly inside the base class.	Inherits accessible members (public/protected) from base class.
Purpose	Provides common features to be reused.	Extends or adds new features to the base class.
Declaration	class Base { /* members */ };	class Derived : access Base { /* members */ };

Example:

```
// Base class
class Animal {
  public:
    void sound() {
      cout << "Animals make sound\n";
    }
};
// Derived class
class Dog : public Animal {
  public:
    void bark() {
      cout << "Dog barks\n";
    }
};

int main() {
  Dog d;
  d.sound();  // Inherited from Animal
  d.bark();   // Dog's own function
}
```

Here, Animal is the base class and Dog is the derived class. Dog can use both its own bark() function and the inherited sound() function.

Q. What is a Constructor in C++?

A **constructor** is a **special function** in C++ that is **automatically called** when an object of a class is created. Its main purpose is to **initialize the data members** of the class.

Unlike regular functions:

- A constructor has **the same name as the class**.
- It **does not have a return type** (not even void).
- It is **called automatically** when an object is created.

How is a Constructor Called?

You **don't call a constructor manually** like other functions.
It is **invoked automatically** when you create an object of the class.

For example:

```
class Student {
  public:
    Student() {  // Constructor
      cout << "Constructor called!" << endl;
    }
};

int main() {
  Student s1;  // Constructor is called automatically here
}
```

Output:

sql

CopyEdit

```
Constructor called!
```

Types of Constructors

1. **Default Constructor**
   - Takes **no parameters.**
   - Automatically created by the compiler if no constructor is defined.
2. **Parameterized Constructor**

    - Takes parameters to initialize objects with custom values.

3. **Copy Constructor**
    - Creates a new object as a **copy of an existing object**.

Example with Parameterized Constructor:

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

class Student {
  private:
    string name;
    int age;

  public:
    // Parameterized constructor
    Student(string n, int a) {
      name = n;
      age = a;
    }

    void display() {
      cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
  Student s1("Alice", 20);  // Constructor is called with arguments
  s1.display();
}
```

Output:

```
Name: Alice, Age: 20
```

Key Points:

- Constructors initialize objects automatically.
- They make code cleaner and more reliable.
- You can overload constructors by using different parameter sets.

Conclusion:

A **constructor** is an essential part of object-oriented programming in C++. It ensures that **every object starts with valid data**, saving us from writing repetitive initialization code. With types like default, parameterized, and copy constructors, you can control how your objects are created easily and effectively.

Q. What is Operator Overloading in C++?

Operator Overloading is a feature in C++ that allows us to **redefine the meaning of an operator** (like +, -, *, ==, etc.) when it is used with **user-defined data types** (like classes).

Normally, operators work on **built-in data types**. For example:

```
int a = 5, b = 3;
int c = a + b;  // '+' adds two integers
```

But what if we have two objects (say, of a Complex number class) and want to add them using the + operator?
That's where **operator overloading** comes in.

Why Use Operator Overloading?

- To make **custom classes behave like built-in types**.
- To write **clean, readable, and intuitive** code.
- To perform operations like +, -, == directly on objects.

Syntax of Operator Overloading

We use a **special function** with the keyword operator followed by the symbol.

```
return_type operator symbol (parameters)
```

It can be a **member function** or a **friend function**.

Example: Overloading + Operator for a Complex Number Class

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

class Complex {
  private:
    int real, imag;

  public:
    Complex(int r = 0, int i = 0) {
      real = r;
      imag = i;
    }
    // Overloading the '+' operator
    Complex operator + (Complex obj) {
      Complex temp;
      temp.real = real + obj.real;
      temp.imag = imag + obj.imag;
      return temp;
    }
    void display() {
      cout << real << " + " << imag << "i" << endl;
    }
};
int main() {
  Complex c1(2, 3), c2(1, 4);
  Complex c3 = c1 + c2;  // '+' operator works on objects
  c3.display();  // Output: 3 + 7i
}
```

Things to Remember:

- Not all operators can be overloaded (e.g., ::, .*, sizeof, ?:)
- Operator overloading should **not change the original meaning** of the operator drastically.
- It improves **code readability** but should be used wisely.

Conclusion:

Operator overloading in C++ gives us the power to use standard operators on **custom objects**, making our code **cleaner and more natural** to read. It's an important part of writing **object-oriented and intuitive programs**.

Q. Difference Between == (Equal To) and = (Assignment Operator) in C++

In C++, the symbols == and = look similar but serve **very different purposes**.

---

1. Assignment Operator =

- It is used to **assign a value** to a variable.
- It **stores** the value on the **right side** into the variable on the **left side**.

Example:

```
int a;
a = 5;  // Assigning value 5 to variable a
```

Here, a gets the value 5.

---

2. Equality Operator ==

- It is a **comparison operator**.
- It checks if the **two values are equal** or not.
- It returns a **boolean result**: true (1) or false (0).

Example:

```
int a = 5;
int b = 5;
if (a == b) {
  cout << "Both are equal";
} else {
  cout << "Not equal";
}
```

Output:

```
Both are equal
```

Here, a == b checks **if** the value of a is equal to b.

Common Mistake:

Beginners often **mistakenly use = in place of ==** in conditions:

```
if (a = 10) {  // Wrong: assigns 10 to a, not compares!
  // This will always be true if a becomes non-zero
}
```

Correct version:

```
if (a == 10) {  // Correct: checks if a is equal to 10
  // Executes only if condition is true
}
```

Feature	**= (Assignment Operator)**	**== (Equal To Operator)**
Purpose	Assigns value	Compares values
Returns	The assigned value	Boolean: true or false
Used In	Assignment statements	Conditional statements
Example	a = 10;	if (a == 10)

Conclusion:

- Use = when you want to **store a value** in a variable.
- Use == when you want to **compare** two values.
- Even though they look similar, using the **wrong one can cause logical errors**, especially in conditional statements like if or while.

Q. Write a note on Comments in C++, Virtual function, Dynamic binding and friend function in C++.

✅ Comments in C++

Comments are notes written in the code to explain what the code does. They are **ignored by the compiler** and are used only for human understanding.

There are two types of comments in C++:

1. **Single-line comment**:
   Uses //

```
// This is a single-line comment
```

2. **Multi-line comment**:
   Uses /* */

```
/* This is a
  multi-line comment */
```

✅ **Purpose of Comments**:

- To make code easier to understand.
- To temporarily disable code during debugging.
- To provide documentation within the code.

✅ Virtual Function

A **virtual function** is a function that is **declared in the base class** using the keyword virtual, and is **redefined in the derived class**. It allows **run-time polymorphism**.

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

When using **base class pointers**, the version of the function that gets called is based on the **object type**, not pointer type.

✅ Dynamic Binding

Dynamic Binding (also called **late binding**) means that the function call is **resolved at runtime**, not at compile time. This happens when using **virtual functions**.

```
Base* ptr;
Derived d;
ptr = &d;
ptr->show();  // Derived class function is called (due to dynamic binding)
```

This is useful in implementing **polymorphism** in C++.

✅ Friend Function

A **friend function** is a function that is **not a member** of a class but is allowed to **access its private and protected members**.

To declare it, we use the keyword friend inside the class.

```
class Box {
private:
   int width;
public:
   Box() { width = 10; }
   friend void showWidth(Box b);  // Friend function
};
void showWidth(Box b) {
   cout << "Width: " << b.width;  // Can access private data
}
```

✅ **Use of Friend Function**:

- Useful when two classes need to work closely together.
- Helps access private data without making all members public.

Conclusion

Q. Constructor vs Destructor in C++.

Both **constructors** and **destructors** are **special member functions** in a class. They are automatically called when an object is **created** (constructor) and **destroyed** (destructor). They help manage the **lifecycle** of an object.

✅ Constructor

A **constructor** is a function that is **automatically called when an object is created**.
Its main purpose is to **initialize the data members** of the class.

Key Features:

- Same name as the class.
- **No return type**, not even void.
- Can be **overloaded** (i.e., multiple constructors with different parameters).
- Can have **parameters** (called **parameterized constructors**).

Example:

```
class Student {
public:
  Student() {
    cout << "Constructor called!" << endl;
  }
};
```

This constructor is called when you create an object like:

```
Student s1;
```

✅ Destructor

A **destructor** is a function that is **automatically called when an object goes out of scope** or is **explicitly deleted**.
Its main purpose is to **free up resources** (like memory or files) used by the object.

Key Features:

- Same name as the class, but with a **tilde (~)** symbol before it.
- **No return type** and **takes no parameters**.
- **Cannot be overloaded** — only one destructor per class.
- Called **automatically** at the end of the object's life.

Example:

```
class Student {
public:
  ~Student() {
    cout << "Destructor called!" << endl;
  }
};
```

Feature	Constructor	Destructor
Purpose	Initializes object	Cleans up before object is destroyed
Name	Same as class name	Same as class name with ~
Called When	Object is created	Object goes out of scope
Parameters	Can have parameters	Cannot have parameters
Overloading	Can be overloaded	Cannot be overloaded
Return Type	No return type	No return type

✅ **Conclusion:**

- A **constructor** sets things **up** when the object is created.
- A **destructor cleans up** when the object is no longer needed.
  Together, they help in managing memory and resources automatically in **object-oriented programming**.

Q. What is a Pointer in C++?

A **pointer** is a **variable that stores the memory address** of another variable.
Instead of holding a direct value (like 10 or 'A'), a pointer holds the **location in memory** where a value is stored.

Basic Syntax:

```
int a = 10;

int* ptr = &a;  // ptr stores the address of variable a
```

- &a gives the address of variable a.
- ptr is a pointer to an integer.
- *ptr (called **dereferencing**) gives the value stored at that address.

Example:

```
#include <iostream>

using namespace std;


int main() {

  int num = 5;

  int* ptr = &num;


  cout << "Value: " << *ptr << endl;      // 5

  cout << "Address: " << ptr << endl;     // memory address of num

}
```

✅ **Advantages of Pointers**

1. **Efficient Memory Usage**
   - Allows dynamic memory allocation using new and delete.
   - Helps create flexible programs that don't waste memory.
2. **Faster Access**
   - Pointers allow direct access to memory, making some operations faster.
3. **Used in Data Structures**
   - Pointers are essential for building **linked lists**, **trees**, **graphs**, etc.
4. **Function Arguments (Call by Reference)**
   - You can pass large data (like arrays) to functions efficiently without copying.

5. **Dynamic Arrays**
    - Enables creating arrays with size decided at runtime.

❌ Disadvantages of Pointers

1. **Complex and Error-Prone**
    - Mistakes like **dangling pointers** or **wild pointers** can cause bugs or crashes.
2. **Difficult to Debug**
    - Pointer-related errors are hard to detect and fix (e.g., memory leaks).
3. **Security Risks**
    - If misused, pointers can corrupt memory or cause unexpected behavior.
4. **Manual Memory Management**
    - You must manually manage memory (new/delete), which increases chances of mistakes.

✅ Conclusion

Pointers are a powerful feature in C++ that allow **direct control over memory**, making programs more flexible and efficient. However, they require **careful use**, as pointer errors can lead to **serious problems** like crashes and memory leaks. Understanding pointers is essential for working with **advanced data structures** and **system-level programming**.

Q. Friend Function in C++.

A **friend function** is a function that is **not a member** of a class but is allowed to **access its private and protected members**.

To make a function a friend, we declare it inside the class with the keyword friend.

✅ **Syntax:**

```
class Box {
  private:
    int length;

  public:
    Box() { length = 10; }
    friend void showLength(Box b);  // Friend function
};

void showLength(Box b) {
  cout << "Length: " << b.length << endl;  // Can access private member
}
```

✅ **Uses:**

- When **two or more classes** need to share private data.
- For operator overloading (like << and >> for input/output).

Q.. Static Function in C++.

A **static function** (also called **static member function**) belongs to the **class, not to any specific object**. It can be called **without creating an object** of the class.

✅ **Syntax:**

```
class Test {
  public:
    static void show() {
      cout << "Static function called!" << endl;
    }
};
```

Usage:

```
Test::show();  // No need to create object
```

✅ **Key Points:**

- Can only access **static members** of the class.
- Useful for utility functions related to the class.

Q. Virtual Function in C++.

A **virtual function** is a function in the **base class** that is meant to be **overridden in derived classes**. It allows **run-time polymorphism**, meaning the correct function is chosen **at runtime**, depending on the object.

✅ **Syntax:**

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


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

Usage:

```
Base* ptr;
Derived d;
ptr = &d;
ptr->show();  // Output: Derived class (because show() is virtual)
```

✅ **Uses:**

- To achieve **dynamic binding**.
- For implementing **polymorphism** in object-oriented programming.

Q.What is String I/O and Object I/O?

String I/O refers to **input and output operations on strings**, usually using standard input/output functions like `cin`, `cout`, or string-specific classes like `stringstream`.

Example:

```
string name;

cout << "Enter your name: ";
cin >> name;
cout << "Hello, " << name;
```

Object I/O refers to **input and output of entire class objects**, usually done using **file streams** (ifstream, ofstream, fstream) or by **overloading operators** to read/write objects.

Example:

```
class Student {
  public:
    string name;
    int age;

    void getData() {
      cin >> name >> age;
    }

    void putData() {
      cout << name << " " << age;
    }
};
```

Feature	**String I/O**	**Object I/O**
Definition	Input/Output of string values	Input/Output of entire objects (class instances)
Data Type	Works with string or character arrays	Works with user-defined data types (objects)
Used Classes	iostream, string, sstream	fstream, ofstream, ifstream (for file I/O)

Feature	**String I/O**	**Object I/O**
Operators Used	cin, cout, getline()	Operator overloading (>>, <<) or member functions
File Handling	Generally used with standard input/output	Often involves reading/writing to **files**
Memory Size	Deals with characters or strings (fixed size)	Handles multiple data members (varied size)
Need for Overloading	No overloading needed	Often requires overloading of >> and <<
Example	cin >> name;	file >> studentObj; (after overloading)
Complexity	Simple and straightforward	More complex due to structure of the object
Formatting	Controlled using I/O manipulators (setw, endl)	Needs custom formatting logic (function definitions)

◆ **Summary:**

- **String I/O** is simple and deals with **text input/output**.
- **Object I/O** handles **complex data** and often involves **file operations** and **operator overloading**.
- Object I/O provides more flexibility for **storing and retrieving complete data structures**, making it useful for data storage, serialization, etc.

Q.What is Data Abstraction and Encapsulation ?

Data Abstraction is a key concept in object-oriented programming that refers to **hiding complex internal details** and showing only the **essential features** of an object.

Just like in real life — when you drive a car, you don’t need to know how the engine works; you just use the steering wheel, accelerator, and brakes. The **unnecessary internal complexity is hidden** — that’s abstraction.

✅ **In C++:**

- Abstraction is achieved using **classes**.
- You define **public methods** (which are exposed to the user) and **private data/methods** (which are hidden from the user).

🔹 **Example:**

```
class ATM {
  private:
    int pin;

  public:
    void withdrawMoney() {
      // logic to withdraw money
      cout << "Money withdrawn!" << endl;
    }
};
```

Here, the user only interacts with withdrawMoney() — they don’t need to know how the money is processed inside. That’s **abstraction**.

🔹 **What is Encapsulation?**

Encapsulation means **binding data and functions** that operate on that data into a **single unit** (i.e., a class). It also helps in **protecting data** by keeping it private and only allowing access through **public methods**.

Encapsulation is like a **capsule** — it wraps everything safely inside.

✅ **Key Features:**

- **Data Hiding**: Data members are kept private or protected.
- **Controlled Access**: Public methods are used to access or modify data.

🔹 **Example:**

```
class Student {
  private:
    int marks;

  public:
    void setMarks(int m) {
      marks = m;
    }

    int getMarks() {
      return marks;
    }
};
```

Here, marks is hidden from outside the class, and can only be accessed using setMarks() and getMarks() — this is **encapsulation**.

Feature	**Data Abstraction**	**Encapsulation**
Purpose	Hide internal implementation details	Bundle data and functions into one unit
Focus	What an object does	How an object does it
Access	Shows only essential information	Restricts direct access to data
Achieved By	Using abstract classes or public interfaces	Using classes and access specifiers

◆ **Conclusion:**

Both **abstraction** and **encapsulation** are essential parts of **object-oriented programming**.

- **Abstraction** helps in hiding **complex logic**.
- **Encapsulation** helps in **protecting data** and keeping the code **organized and secure**.