OBJECT ORIENTED PROGRAMMING(OOP)

Object-Oriented Programming (OOP) is presented as a **better way to write code**. It's a very important concept commonly questioned in technical interviews for positions like Software Engineering. A strong understanding of OOP improves programming skills and is often implemented in companies. OOP concepts allow for the **representation of many real-life scenarios in C++ code** more easily. Libraries in C++'s Standard Template Library (STL), such as `vector`, `string`, and `stack`, use OOP concepts internally for their implementation.

Two foundational terms in OOP are **Class** and **Object**.
•An **Object** is any entity from the real world. Examples include a pen, a laptop, or a phone. In C++ code, these entities are also created as objects. Objects occupy space in memory.
•A **Class** is essentially a **blueprint for objects**. It specifies how an object should look. A class can also be thought of as a **group of objects**. For instance, a class can define the structure for a 'Teacher' object, just as a car company like Toyota uses a blueprint for its cars. Unlike objects, classes do not occupy specific space in memory until an object is created from them.

Objects have associated **Properties** (also called **Attributes**) and **Methods** (also called **Member Functions**).
•**Properties** are values associated with an object, like a teacher's name, department, subject, or salary.
•**Methods** are functions written inside a class that define the behaviour of an object, such as a function to change a teacher's department or calculate tax. Using classes and objects helps avoid repetitive code and improves **code reusability**, as the same class blueprint can be used to create many objects.

OOP concepts like Classes and Objects are not exclusive to C++; they exist in languages like Java, Python, and JavaScript, and are heavily used in day-to-day programming in companies.

Access Modifiers are special keywords used in C++ to control the accessibility of data and methods within a class. There are three types in C++:
•**Private:** Members declared as private are only accessible *inside* the class. By default, members in a C++ class are private.
•**Public:** Members declared as public are accessible *inside* the class and *outside* the class (e.g., from the `main` function or other classes).
•**Protected:** Members declared as protected are accessible *inside* the class and within *derived* classes during inheritance. This is particularly useful for members that should be inherited but not publicly accessible outside the class.

Data Hiding is the concept of hiding sensitive information. Encapsulation facilitates data hiding, primarily by using the `private` or `protected` access modifiers to restrict external access to sensitive data members like passwords or bank balances. While data hiding specifically focuses on concealing sensitive data, **Abstraction** hides unnecessary details *and* shows only the important parts. Access modifiers are a way to implement abstraction.

Constructors are special methods that are automatically invoked by C++ when a new object is created. Their primary role is to **initialise the object**.
•Properties of Constructors: They have the **same name as the class** and **no return type** (not even `void`). If a constructor is not explicitly defined by the programmer, the compiler creates a **default constructor**. Constructors are typically declared as **public** so they can be called automatically when objects are created outside the class (e.g., in `main`). A constructor is called **only once** at the time of object creation. Memory allocation for the object occurs when the constructor is called.

•Types of Constructors:
◦**Non-parameterised Constructor:** Takes no arguments.
◦**Parameterised Constructor:** Takes one or more arguments to initialise the object's properties.
◦**Copy Constructor:** A special constructor used to **copy the properties of one object into another**. C++ provides a **default copy constructor** if one is not explicitly defined.

•**Constructor Overloading:** A class can have **multiple constructors** with the same name, provided they have a different number or type of parameters. This is an example of **Polymorphism**.

The **`this` pointer** in C++ is a special pointer that **points to the current object**. It is used, for example, within a member function or constructor to refer to the object for which that function/constructor was called. This is particularly useful when a parameter name is the same as a class member name; `this->memberName` clarifies that you are referring to the object's member. The arrow operator (`->`) is syntactic sugar for dereferencing the `this` pointer and accessing a member. Understanding pointers in C++ is helpful for grasping the `this` pointer concept.

When objects are copied, there are two types of copies: **Shallow Copy** and **Deep Copy**.
•A **Shallow Copy** copies the values of all data members from one object to another. The default copy constructor provided by C++ performs a shallow copy. This is usually fine for simple data types, but it causes issues when a class contains **pointers** to **dynamically allocated memory** (memory allocated on the **heap** using `new`). In a shallow copy, the pointers in both the original and copied objects will point to the *same* block of dynamically allocated memory. Modifying the data through the pointer in one object will affect the data accessed by the pointer in the other object. Memory allocated with `new` resides on the heap, distinct from the stack where local variables are typically stored.

•A **Deep Copy** involves creating a **new block of dynamic memory** for the copied object's pointer and then copying the *contents* of the original object's dynamically allocated memory into this new block. This requires writing a **custom copy constructor**. Deep copying ensures that the original and copied objects have their own independent copies of the dynamic data, preventing unintended side effects when one object's data is modified. Deep copy is preferred when working with classes that manage dynamic memory.

A **Destructor** is a special function in C++ that is automatically called when an object goes out of scope or is explicitly deleted. Its purpose is to **deallocate the memory** that the object occupied. It is the **opposite of a constructor**.

•Properties of Destructors: They have the **same name as the class, prefixed with a tilde (~)**. They have **no return type**. The compiler provides a **default destructor** if one is not defined. The default destructor deallocates memory allocated on the stack.
•For **dynamically allocated memory** (on the heap using `new`), the programmer must **explicitly use the `delete` keyword** within the destructor to free that memory. Failure to deallocate dynamically allocated memory can lead to **memory leaks**.
•Destructors are called automatically when an object is no longer needed (e.g., at the end of its scope). In inheritance, the **child class destructor is called before the parent class destructor**.

Inheritance is one of the four major pillars of OOP. It is a mechanism where a new class (**derived class** or **child class**) inherits properties and methods from an existing class (**base class** or **parent class**). The primary benefit is **code reusability**.

•Base class **private members are never inherited**.
•When an object of a derived class is created, the **base class constructor is called first, followed by the derived class constructor**. The derived class constructor can explicitly call a specific base class constructor.

•**Modes of Inheritance** (Public, Protected, Private) control the access level of inherited members in the derived class:
◦**Public Inheritance:** Public members of the base class remain public in the derived class; protected members remain protected.
◦**Protected Inheritance:** Public and protected members of the base class become protected in the derived class.
◦**Private Inheritance:** Public and protected members of the base class become private in the derived class.

•Types of Inheritance:
◦**Single Inheritance:** A derived class inherits from a single base class (e.g., `Student` inheriting from `Person`).
◦**Multi-level Inheritance:** A class inherits from a derived class, creating a chain (e.g., `Person` -> `Student` -> `GradStudent`).
◦**Multiple Inheritance:** A derived class inherits from more than one base class (e.g., `TA` inheriting from both `Student` and `Teacher`). Different base classes and their inheritance modes are specified, separated by commas.
◦**Hierarchical Inheritance:** Multiple derived classes inherit from a single base class (e.g., `Student` and `Teacher` both inheriting from `Person`).
◦**Hybrid Inheritance:** Any combination of the above types.

Polymorphism is another pillar of OOP, meaning "multiple forms". It is the ability of objects to **behave differently based on the context**.

•Types of Polymorphism:
◦**Compile-time Polymorphism (Static Polymorphism):** The decision about which function or constructor to call is made during compilation. Examples include:
▪**Constructor Overloading:** Multiple constructors with the same name but different parameters.
▪**Function Overloading:** Multiple functions within the same class with the same name but different parameters (either in number or type). The return type does not differentiate overloaded functions.
▪**Operator Overloading:** Allowing standard operators (like +, −, =) to have different meanings or behaviours depending on the types of data they operate on. (Mentioned as a concept to explore).

◦**Run-time Polymorphism (Dynamic Polymorphism):** The decision about which function to call is made during program execution (at runtime). This usually involves function overriding and pointers/references. An example is:
▪**Function Overriding:** Occurs when a derived class provides its own implementation of a function that is already defined in its base class, having the same name and parameters. For an object of the derived class, the derived class's version of the function is called, effectively "overriding" the base class's version. This is dependent on inheritance.
▪**Virtual Functions:** Special member functions declared with the `virtual` keyword in a base class. They are specifically designed to be overridden by derived classes. When a virtual function is called through a pointer or reference to the base class, the decision about which version (base or derived) to call is made at runtime based on the actual type of the object being pointed to. A **pure virtual function** is a virtual function declared with `= 0` (e.g., `virtual void func() = 0;`). It has no implementation in the base class and makes the class an **Abstract Class**.

Abstraction is the fourth pillar of OOP. It means **hiding unnecessary details** and showing only the essential features.

•Methods of implementing Abstraction:
◦**Access Modifiers:** By making sensitive data private or protected and necessary methods/data public, we hide internal details while exposing only the required interface. Data hiding is a component of abstraction.
◦**Abstract Classes:** A class declared as `abstract` cannot be instantiated (no objects can be created directly from it). Abstract classes serve as **base classes** for inheritance, providing a **blueprint or template** for other classes (derived classes) to inherit from and implement specific functionalities.
◦**Pure Virtual Functions:** A class containing at least one pure virtual function (`= 0`) automatically becomes an abstract class. Pure virtual functions define an interface that derived classes **must implement**. For example, a `Shape` class might have a pure virtual `draw()` function, requiring any class that inherits from `Shape` (like `Circle` or `Square`) to provide its own specific implementation of `draw()`.

The **`static` keyword** in C++ has multiple uses in OOP:

•**Static Variables (within a function):** A variable declared `static` inside a function is created and initialised **only once** during the program's execution. Its value persists between multiple calls to that function.
•**Static Variables (within a class):** A member variable declared `static` within a class is **shared by all objects** of that class. There is only one copy of the static variable for the entire class, rather than a separate copy for each object.
•**Static Objects:** An object created with the `static` keyword exists for the **entire lifetime of the program**. Its constructor is called when the object is created within its scope (or when the program starts for global/namespace static objects), and its destructor is called when the program finishes.

1. Friend Function

A **friend function** is a function that is not a member of a class but has permission to access its private and protected members. It is declared inside the class using the `friend` keyword.

Syntax

Key Characteristics

- Declared inside a class using `friend` keyword.
- Defined outside the class like a normal function.
- Can access private and protected members of the class.
- Provides flexibility when non-member functions need class data access.

2. Friend Class

A **friend class** is a class that is allowed to access the private and protected members of another class. This is useful when two classes need to work closely together.

Syntax

Key Characteristics

- Declared inside a class using `friend class ClassName;`
- Allows one class to access private and protected members of another class.
- Enables close collaboration between classes without breaking encapsulation.

Differences Between Friend Function and Friend Class

Feature	Friend Function	Friend Class
Access Modifier	Uses `friend` keyword before a function declaration	Uses `friend class` keyword
Accessibility	Grants access to a specific function	Grants access to all members of another class
Scope	Defined as an independent function	Defined as an entire class
Use Cases	When an external function needs access to private members	When another class should have full access to private members

Use Cases

- **Friend functions** are useful in operator overloading and standalone functions needing access to class members.
- **Friend classes** help when two classes need deep integration and one must access the private details of another.