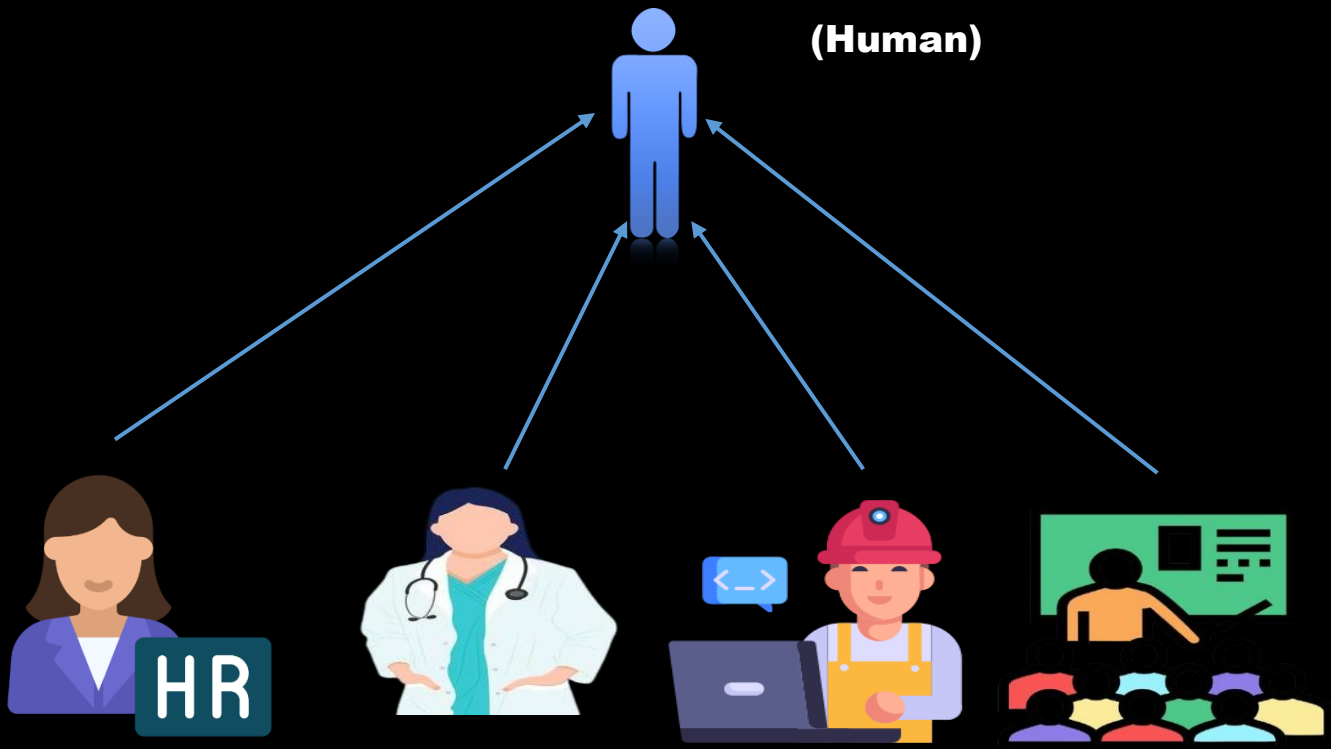
OOPs

My Hours of Learning — Compressed into a Few Powerful Pages



Satyendra Gautam

📚 Topics Covered

🧱 Core OOP Concepts (4 Pillars)

- ✅ **Class and Object**
- ✅ **Abstraction**
- ✅ **Encapsulation**
- ✅ **Inheritance**
- ✅ **Polymorphism**

🛠️ Constructors & Destructors

🔁 Inheritance (In-Depth)

- Single Inheritance
- Multilevel Inheritance
- Multiple Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance
- Diamond Problem (Mentioned)

🌀 Polymorphism (Compile-Time & Run-Time Polymorphism)

🎛️ Access Control & Data Handling

💉 Advanced OOP Concepts

- Abstract Class
- Interface (Pure Virtual Class)
- Virtual Functions and Destructors

👫 Friend Mechanisms

- Friend Function
- Friend Class

📌 Static Members

- Static Data Members
- Static Member Functions

🧠 OOPs Notes

Object-Oriented Programming (OOPs) is a **programming paradigm** (style of coding) where most of the code revolves around **objects** and **classes**.

🌼 Pillars of OOP

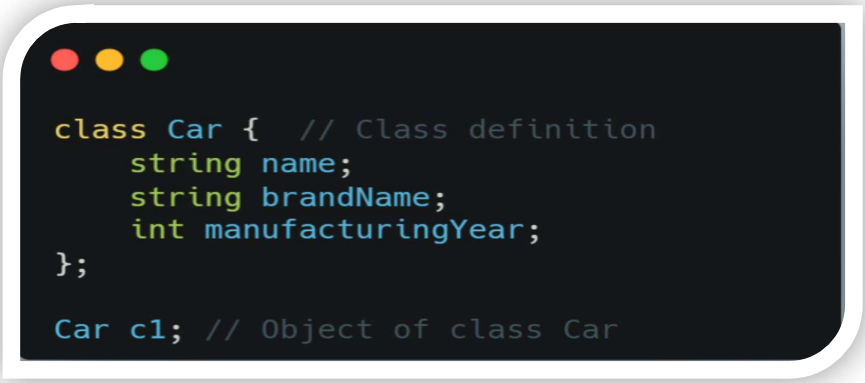
- Abstraction
- Encapsulation
- Inheritance
- Polymorphism

✅ 1. Class and Object

- **Class**: A class is a **user-defined data type** that serves as a **blueprint** for creating objects. It defines both the **structure** and **behaviour** of objects through data members and member functions.
- **Object**: An **object** is an **instance** of a class that represents a real-world entity and contains actual data.

📝 Notes:

- A class **does not occupy memory** until an object is created from it.
- An **empty class** (a class with no data members or functions) still takes **1 byte** of memory.
- The **size of a class object** depends on its data members and may include **padding** for memory alignment.



```
class Car {  // Class definition
    string name;
    string brandName;
    int manufacturingYear;
};

Car c1; // Object of class Car
```

🚫 2. Abstraction

Abstraction means **hiding the internal implementation details** and exposing **only the essential features** to the user.

✅ Key Idea:

Users should not need to understand how things work internally — they should just know how to use it. If anyone know then it is fine but not necessary.

📌 Achieved by:

- Keeping data **private**
- Exposing only a **public interface** (functions)

🔌 Real-Life Analogy:

If a user wants to turn on the TV, we don't ask them to connect red and green wires—we just provide a simple "Power" button.

```
class Television {
private:
    bool isOn;
    int currentChannel;
    int currentVolume;
    string brand;

public:
    void TurnOn();        // Turns the TV on
    void TurnOff();       // Turns the TV off
    void VolumeUp();      // Increases the volume
    void VolumeDown();    // Decreases the volum
};
```

🔐 3. Encapsulation

Encapsulation refers to **wrapping data and methods** that operate on that data into a **single unit** (i.e., a class). It also helps in **data protection** by restricting direct access to internal variables.

✅ Key Idea:

- Make **data members private**
- Use public **getter** and **setter** functions to access and modify them

🎯 Benefits:

- Protects data from **unauthorized access or misuse**
- Improves code **maintainability** and **security**

```
class Engineer {
private:
    // These variables can't be accessed directly from outside the class
    string name;
    string role;
    int yearOfExperience;

public:
    // Setter methods
    void setName(string a) { name = a; }
    void setRole(string r) { role = r; }
    void setExperience(int year) { yearOfExperience = year; }

    // Getter methods
    string getName() { return name; }
    string getRole() { return role; }
    int getExperience() { return yearOfExperience; }
};

// Creating an object of class Engineer
Engineer E1;
```

🔑 Access Modifiers

Access modifiers determine the **visibility** of class members (variables and functions).

Modifier	Description
public	Accessible from **anywhere** — inside or outside the class.
private	Accessible only within the class. Not accessible from outside.
protected	Accessible **within the class** and its **derived** (child) classes.

⚙ Getters & Setters

Getter: A function used to access **private data members** from outside the class.

Setter: A function used to set or **initialize private data members** from outside the class.

📌 They provide **controlled access** to class variables and help in maintaining **data integrity.**

🛠 Constructor

A constructor is a **special function** that is **automatically invoked** when an object of a class is created.

✅ **Key Points:**

- Its name is the **same as** the class name.
- It has **no return type** (not even void).
- Used to initialize objects.

There are different types:

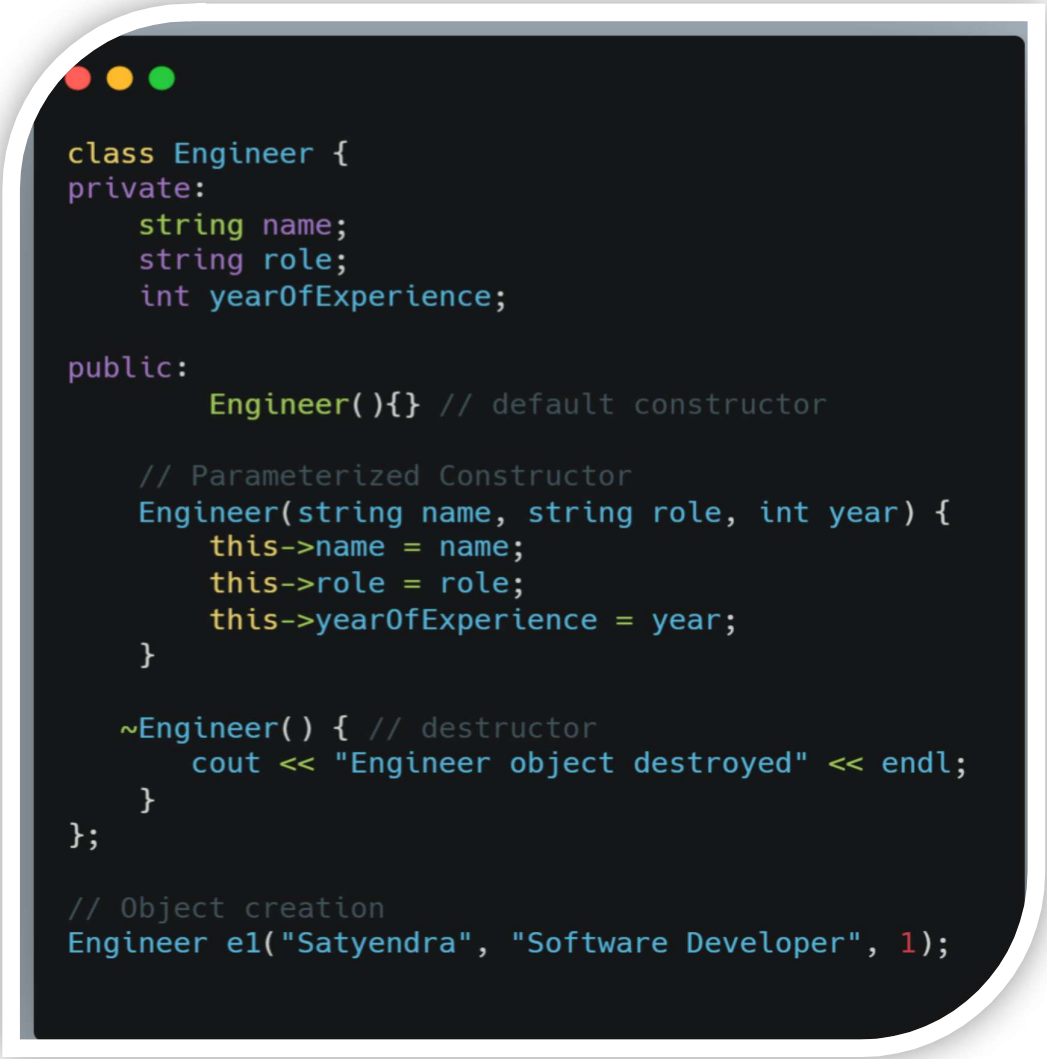
1. Default Constructor
2. Parameterized Constructor
3. Copy Constructor etc.

🧹 Destructor

A **destructor** is a **special function** that is **automatically called** when an object goes out of scope or is explicitly deleted.

✅ Key Points:

- Used to release resources acquired by the object.
- Its name is the same as the class name, prefixed with a tilde ~ .
- It does not take parameters and does not return anything.
- You usually don’t need to write one unless you're managing memory manually (e.g., using new / delete).



```
class Engineer {
private:
    string name;
    string role;
    int yearOfExperience;

public:
        Engineer(){} // default constructor

    // Parameterized Constructor
    Engineer(string name, string role, int year) {
        this->name = name;
        this->role = role;
        this->yearOfExperience = year;
    }

   ~Engineer() { // destructor
       cout << "Engineer object destroyed" << endl;
    }
};

// Object creation
Engineer e1("Satyendra", "Software Developer", 1);
```

🔁 4. Inheritance

Inheritance is a feature of OOP that allows a class to **acquire properties** and **behaviors** (i.e., data members and methods) from **another class**.

✅ **Key Idea**:

One class (child/derived) inherits from another class (parent/base) to promote code reusability.

```cpp
class Car { // Base class (Parent)
protected:  // Accessible to derived classes
    string name;
    string brandName;
    int manufacturingYear;

public:
    void startEngine() {
        cout << "Engine started" << endl;
    }

    void stopEngine() {
        cout << "Engine stopped" << endl;
    }
};

class PetrolCar : public Car {
    // Inherits all public & protected members from Car
    // You can also add extra features here
};

class DieselCar : public Car {
    // Inherits all public & protected members from Car
};
```

🔢 Types of Inheritance in C++

Type	Description
Single	One class inherits from one base class. (class PetrolCar : public Car)
Multilevel	A class inherits from a derived class. ModernCar → PetrolCar → Car
Multiple	A class inherits from **more than one base class**. class A : public B, public C
Hierarchical	Multiple classes inherit from the same base class. PetrolCar, DieselCar ← Car
Hybrid	Combination of two or more types of inheritance.
Diamond Problem	Occurs in multiple inheritance when two base classes inherit from a common ancestor, causing ambiguity. Solved using **virtual inheritance**.

🌀 5. Polymorphism

- Polymorphism means "**one name, many forms**".
- It allows the same function or operator to behave differently based on the context.

🎯 Real-Life Analogy:

*In MS Paint, the **selector tool** can perform multiple actions like **draw, erase**, or move depending on the object — one tool, many actions.*

🔢 Types of Polymorphism in C++

Type	Description	Example
Compile-Time (Static)	Function resolution happens at compile time	Function/Operator Overloading
Run-Time (Dynamic)	Function resolution happens at runtime using virtual	Function Overriding

⚙️ Compile-Time Polymorphism

✅ Function Overloading

Multiple functions with the same name but different parameters.

```
class Car {
private:
    bool isEngineOn;
    int currentGear;

public:
    void changeGear() {
        cout << "Gear changed automatically" << endl;
    }

    void changeGear(int gear) {
        currentGear = gear;
        cout << "Gear changed to " << gear << endl;
    }

    ~Car() {
        cout << "Car object destroyed" << endl;
    }
};
```

✅ Operator Overloading

Redefining built-in operators for user-defined types (like + for complex numbers).

```
class Complex {
private:
    int real;
    int imaginary;

public:
    Complex(int real = 0, int img = 0) : real(real),
imaginary(img) {}

    void Display() {
        cout << "Complex number: " << real << " + i" <<
imaginary << endl;
    }

    // Overloading + operator
    Complex operator + (const Complex &c) {
        Complex temp;
        temp.real = real + c.real;
        temp.imaginary = imaginary + c.imaginary;
        return temp;
    }
};
```

⚙️ Run-Time Polymorphism

🚀 Function Overriding (Run-Time Polymorphism)

To achieve run-time polymorphism, we use function overriding, where a derived class provides its own version of a function that is already defined in the base class.

✅ Key Requirement

The base class function must be marked with the virtual keyword to allow dynamic dispatch at runtime.

🔁 Real-World Analogy:

Different animals make different sounds. We call Speak(), but each animal "speaks" in its own way — same interface, different behavior.

```
class Animal {
public:
    virtual void Speak() {
        cout << "Animal Speaking." << endl;
    }

    virtual ~Animal() {
        cout << "Animal Destructor" << endl;
    }
};

class Dog : public Animal {
public:
    void Speak() override {
        cout << "Bhou Bhou Bhoooooo" << endl;
    }

    ~Dog() {
        cout << "Dog Destructor" << endl;
    }
};
```

🧠 Why Use Virtual Destructors?

- When deleting a derived class object through a **base class pointer**, having a **virtual destructor** ensures that both destructors (derived → base) get called. Without it, only the base class destructor may run — leading to **resource leaks**.

🧱 Abstract Class

An abstract class is a class that contains **at least one pure virtual function**.

✅ Key Points:

- Acts as a **blueprint** for other classes.
- Cannot be instantiated (i.e., you **cannot create objects** of an abstract class).
- Derived classes **must override** all pure virtual functions to be concrete.

```
class Car {
protected:
    bool isEngineOn;
    string name;

public:
    Car(string name) : name(name), isEngineOn(false) {}

    void StartEngine() {
        isEngineOn = true;
        cout << "Car Engine started" << endl;
    }

    void StopEngine() {
        isEngineOn = false;
        cout << "Car Engine stopped" << endl;
    }

    // Pure virtual functions
    virtual void ChangeGear() = 0;
    virtual void Accelerate() = 0;

    virtual ~Car() {
        cout << "Car Destructor" << endl;
    }
};
```

🧩 Interface in C++

- An interface is a class where **all functions are pure virtual**.
- It's used to define a contract — what a class must do, without saying how.

✅ **Notes**:

- You **can't create an object** of an interface.
- A class that implements ICar **must define all functions**.
- Though C++ doesn't have a formal interface keyword like Java or C#, we achieve the same with **pure virtual classes**.

```
class ICar {
public:
    virtual void StartEngine() = 0;
    virtual void StopEngine() = 0;
    virtual void ChangeGear() = 0;
    virtual void Accelerate() = 0;

    virtual ~ICar() {
        cout << "ICar Destructor" << endl;
    }
};
```

👥 Friend Function

A friend function is a **non-member function** that is given special access to the **private** and **protected members** of a class.

✅ **Key Points:**

- Declared using the friend keyword inside the class.
- It is **not a member**, but it **acts like one**.
- Often used when **external functions** need access to internal data.

```
class Customer {
private:
    string name;
    int balance;

public:
    Customer(const string &name, int balance) : name(name),
balance(balance) {}

    // Friend function declaration
    friend void SeeBalance(Customer &c);

    ~Customer() {
        cout << "Destructor called for " << name << endl;
    }
};

// Friend function definition (outside the class)
void SeeBalance(Customer &c) {
    cout << "The balance is: " << c.balance << endl;
}
```

👫 Friend Class

A friend class is a class that is **granted full access to another class's** private and protected members.

✅ Key Points:

- Declared using friend class ClassName; inside the class to be exposed.
- The friend class can access all private and protected members.
- Useful in tight coupling scenarios where two classes work closely.

```
class Human {
private:
    int age;

public:
    void secret() {
        cout << "This is a secret function in the Human class!"
<< endl;
    }

    // Granting access to Me class
    friend class Me;
};

class Me {
public:
    void DisplayAge(Human &ram) {
        ram.age = 15;
        ram.secret(); // Accessing private function
        cout << "My age is: " << ram.age << endl;
    }
};
```

🧠 When to Use Friend Function/Class?

- When **tight internal access** is needed between two classes.
- In **operator overloading** where access to private members is required.
- In **helper functions** that logically operate on the internal state of a class.

🔗 Static Data Members

A **static data member** is a member of a class that is **shared among all objects** of that class.

✅ Key Points:

- Declared using the static keyword.
- It is **not tied to any specific object**, but to the **class itself**.
- It is **shared** and retains its value across all object instances.
- Can be accessed using the class name (without creating an object).

🧠 Real-Life Analogy:

In a Customer class, you want to keep track of the total number of customers and total balance across all accounts — this is a class-level property, not per-object.

```
class Customer {
private:
    static int total_balance;
    static int total_account;

public:
    // Static member function to access static data
    static void AccessTotalAccount() {
        // Cannot access non-static members here
        cout << "Total accounts: " << total_account << endl;
        cout << "Total balance in Bank: " << total_balance <<
endl;
    }

    ~Customer() {
        // Destructor logic (if needed)
    }
};

// Static member definition outside the class
int Customer::total_balance = 0;
int Customer::total_account = 0;
```

🧭 Static Member Functions

A static member function is a class function that:

- Can be called using the class name (**no object needed**).
- Can **only access static data members** (not instance variables).

💡 Why Use It?

- To **manipulate or access static data members**
- To create **utility functions** that operate at the class level

```
class Customer
{
private:
    static int total_balance;
    static int total_account; // static data member
public:
    // static member function
    static void AccessTotalAccount()
    {
        // this function can't access non-static data types
        cout << "Total account " << total_account << endl;
        cout << "Total balance in Bank " << total_balance <<
endl;
    }
    // destructor
    ~Customer()
    {}
};

Customer::AccessTotalAccount(); // No object needed
```

⚠️ Notes:

- Static functions **cannot use this** pointer.
- Static members must be **defined outside** the class once (memory is allocated here).
- Static data is **common to all objects**, useful for counting, pooling, logging, etc

🙏 Final Note

If you've made it this far — **congratulations**!
You now own this PDF, and you're **completely free to use, study, share, or remix** it however you like.

- If this guide helped you understand Object-Oriented Programming better, I'm truly glad.
- I don't ask for likes, shares, or reposts —
  but if you found it helpful, I'd really appreciate a kind comment or suggestion.

If you spot any mistakes — **please forgive me**.
I'm a learner too, and like everyone else, I make mistakes sometimes 😊

📂 GitHub Repository

All the code examples from this PDF — and more — are available here:
🔗 https://github.com/satyendragautam901/OOPs-in-cpp

Thanks for reading — and keep building, keep learning 💡

— *Satyendra Gautam*