MENTORBRO LEARNING LLP,
Email: mentorbroofficial@gmail.com
Ph no : + 91 8157867616

Mentor Bro Notes

📚 Object-Oriented Programming (OOP) in C++

Object-Oriented Programming (OOP) is a programming paradigm that focuses on **objects** instead of actions, and on **data** rather than logic. In OOP, everything revolves around the **concept of objects and classes**, making code **modular, reusable**, and easier to debug. 💻

✨ Key Principles of OOP

OOP in C++ is built upon the following four principles:

1️⃣ Encapsulation (🔒 Wrapping Data)

- Encapsulation refers to the **binding of data (variables)** and **methods (functions)** into a single unit called a **class**.
- It restricts **direct access** to the internal state of objects to ensure **data integrity**.
- Only authorized methods (like **getters** and **setters**) can access or modify private data.

Theory:

Encapsulation ensures that the internal implementation of a class is hidden from the outside. This is achieved by making class members **private** and providing **public** methods to interact with them.

Example:

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

class BankAccount {
private:
```

```
    double balance;  // Private variable

public:
    void setBalance(double amount) {
        if (amount >= 0)
            balance = amount;
        else
            cout << "Invalid balance!" << endl;
    }

    double getBalance() {
        return balance;
    }
};

int main() {
    BankAccount account;
    account.setBalance(5000);  // Encapsulated access
    cout << "Current Balance: $" << account.getBalance() << endl;
    return 0;
}
```

Benefits:

- **Improved security**: Direct access to data is restricted.
- **Better code organization**: Data and methods are packaged together.

2️⃣ Abstraction (👓 Hiding Complexity)

- Abstraction involves **hiding unnecessary details** from the user and exposing only **essential features**.
- It reduces complexity and improves the usability of the program.

Theory:

Abstraction is achieved in C++ using **classes**, **abstract classes**, and **interfaces**. The user interacts with an object without needing to understand how it works internally.

Example:

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

class Calculator {
public:
    void add(int a, int b) {
        cout << "Sum: " << a + b << endl;
    }
    void multiply(int a, int b) {
        cout << "Product: " << a * b << endl;
    }
};

int main() {
    Calculator calc;
    calc.add(5, 10);           // User doesn't see implementation
details
    calc.multiply(3, 4);
    return 0;
}
```

Benefits:

- **Simplifies the code** for the user.
- Focuses only on what the object **does**, not how it does it.

3️⃣ Inheritance (👫 Reusability)

- **Inheritance** is a mechanism where a new class (**derived class**) acquires properties and methods of an existing class (**base class**).
- It promotes **code reuse** and allows hierarchical relationships between classes.

Theory:

Inheritance establishes an "is-a" relationship between classes. For example, a **Car** "is a" type of **Vehicle**.

Types of Inheritance in C++:

1. **Single Inheritance**
2. **Multilevel Inheritance**
3. **Hierarchical Inheritance**
4. **Multiple Inheritance**

Example: Single Inheritance

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

class Vehicle {
public:
    void move() {
        cout << "Vehicle is moving!" << endl;
    }
};

class Car : public Vehicle {  // Derived class inherits from Vehicle
public:
    void fuel() {
        cout << "Car is refueling!" << endl;
    }
};

int main() {
    Car myCar;
    myCar.move();  // Access method from base class
    myCar.fuel();  // Access method from derived class
    return 0;
}
```

Benefits:

- **Code Reusability**: Reduces duplication of code.
- **Hierarchical Structure**: Organizes related classes.

4️⃣ Polymorphism (🔄 Many Forms)

- Polymorphism means **one interface, many implementations**.

- It allows methods to have **different behavior** depending on the object that calls them.

Theory:

Polymorphism is achieved through:

- **Compile-Time Polymorphism**: Method overloading and operator overloading.
- **Run-Time Polymorphism**: Method overriding using **virtual functions**.

Example: Function Overloading (Compile-Time Polymorphism)

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

class Printer {
public:
    void print(int value) {
        cout << "Integer: " << value << endl;
    }
    void print(double value) {
        cout << "Double: " << value << endl;
    }
};

int main() {
    Printer p;
    p.print(42);       // Calls print(int)
    p.print(3.14);     // Calls print(double)
    return 0;
}
```

Example: Virtual Function (Run-Time Polymorphism)

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

class Shape {
public:
    virtual void draw() {
        cout << "Drawing Shape" << endl;
    }
};
```

```
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};

int main() {
    Shape* s;
    Circle c;
    s = &c;
    s->draw();  // Calls derived class's draw()
    return 0;
}
```

Benefits:

- **Flexibility**: A single interface adapts to different objects.
- **Extensibility**: Easy to add new functionality.

📐 Classes and Objects

A **class** is a blueprint for creating objects, and an **object** is an instance of a class.

Theory:

A class defines the **attributes** (variables) and **methods** (functions) for an object. Objects have a **state** (data) and **behavior** (methods).

Example:

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

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

```
    void sound() {
        cout << name << " is making a sound!" << endl;
    }
};

int main() {
    Animal dog;  // Create an object
    dog.name = "Buddy";
    dog.age = 5;
    dog.sound();
    return 0;
}
```

🔑 Constructors and Destructors

Constructor:

- A special method used to **initialize an object**.
- It is called **automatically** when an object is created.

Destructor:

- A special method used to **clean up resources** when an object is destroyed.

Example:

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

class MyClass {
public:
    MyClass() { cout << "Constructor called!" << endl; }  //
Constructor
    ~MyClass() { cout << "Destructor called!" << endl; }  //
Destructor
};

int main() {
    MyClass obj;  // Constructor is automatically called
```

```
    return 0;  // Destructor is automatically called
}
```

Benefits:

- **Constructor** ensures proper initialization.
- **Destructor** cleans up resources and prevents memory leaks.

📌 Quick Summary

Principle	Description	Example Use Case
🔒 **Encapsulation**	Restricts access to data; uses methods	Securely managing account balance
🕶️ **Abstraction**	Hides implementation details	ATM machine interface
👯 **Inheritance**	Reuses code through class hierarchy	Car inheriting features from Vehicle
🔄 **Polymorphism**	One interface, multiple implementations	Shapes (Circle, Rectangle) using `draw()`
🔑 **Constructors**	Automatically initializes objects	Initializing a Car with default values