📘 C++ OOP Notes

🔷 Introduction

Definition:
C++ is a general-purpose, high-level programming language that supports **both procedural programming** and **object-oriented programming (OOP)**.
OOP makes programs more **modular, reusable, maintainable, and scalable**.
Why OOP?
Solves real-world problems using objects.
Provides features like **encapsulation, inheritance, and polymorphism**.
Reduces redundancy via **code reusability**.

🔷 OOPS Concepts

Object-Oriented Programming System (OOPS) includes 4 main pillars:
Encapsulation – Data hiding
Inheritance – Code reusability
Polymorphism – One name, many forms
Abstraction – Hiding implementation

🔷 Class & Object

Class: A blueprint that defines properties (data) and methods (functions).
Object: An instance of a class.
Example:

```
#include
using namespace std;

class Car {
public:
    string brand;
    int speed;

    void display() {
        cout << brand << " runs at " << speed << " km/h" << endl;
    }
};

int main() {
    Car c1;   // Object creation
    c1.brand = "BMW";
    c1.speed = 200;
    c1.display();
    return 0;
}
```

Output:

```
BMW runs at 200 km/h
```

🔷 Access Specifiers

They define **scope of members** in a class.
public → Accessible everywhere
private → Accessible only inside class
protected → Accessible in class & derived classes
Example:

```
#include
using namespace std;

class Student {
private:
    int rollNo;

public:
    void setRollNo(int r) { rollNo = r; }
    void display() { cout << "Roll No: " << rollNo << endl; }
};

int main() {
    Student s;
    // s.rollNo = 10; ❌ Error (private)
    s.setRollNo(10); // ✅
    s.display();
}
```

🔷 Encapsulation

Definition: Wrapping data and methods into a single unit (class).
Achieved using **classes & access specifiers**.
Ensures **data hiding & security**.
Example:

```
#include
using namespace std;

class BankAccount {
private:
    int balance;

public:
    BankAccount(int b) { balance = b; }
    void deposit(int amt) { balance += amt; }
    int getBalance() { return balance; }
};

int main() {
    BankAccount acc(1000);
    acc.deposit(500);
    cout << "Balance: " << acc.getBalance();
}
```

Output:

```
Balance: 1500
```

🔷 Constructor

Definition: Special function called automatically when object is created.

Types:

Default constructor

Parameterized constructor

Copy constructor

Example:

```
#include
using namespace std;

class Person {
public:
    string name;
    int age;

    Person() { cout << "Default Constructor\n"; }
    Person(string n, int a) { name = n; age = a; }
};

int main() {
    Person p1;               // Default
    Person p2("Alice", 20);  // Parameterized
}
```

🔷 this Pointer

Definition: A special pointer that refers to the **current object**.
Useful when **local variables shadow class members**.
Example:

```
#include
using namespace std;

class Student {
    int rollNo;
public:
    void setRollNo(int rollNo) {
        this->rollNo = rollNo;  // Resolves ambiguity
    }
    void display() { cout << "Roll No: " << rollNo; }
};

int main() {
    Student s;
    s.setRollNo(101);
    s.display();
}
```

🔷 Copy Constructor

Definition: A constructor that initializes an object by copying another object.
Example:

```
#include
using namespace std;

class Student {
    int marks;
public:
    Student(int m) { marks = m; }
    Student(const Student &s) { marks = s.marks; } // Copy Constructor
    void display() { cout << "Marks: " << marks; }
};

int main() {
    Student s1(90);
    Student s2 = s1;  // Copy constructor
    s2.display();
}
```

🔷 Shallow vs Deep Copy

✅ Already explained in detail earlier (with example, output, and comparison).

🔷 Destructor

Definition: Special function that is automatically invoked when object goes out of scope.
Name: `~ClassName()`
Used to free resources.
Example:

```
#include
using namespace std;

class Demo {
public:
    Demo() { cout << "Constructor called\n"; }
    ~Demo() { cout << "Destructor called\n"; }
};

int main() {
    Demo d;
}
```

Output:

```
Constructor called
Destructor called
```

🔷 Inheritance

Definition: Process of acquiring properties/methods from another class.
Promotes **code reusability**.
Example:

```
#include
using namespace std;

class Parent {
public:
    void greet() { cout << "Hello from Parent\n"; }
};

class Child : public Parent {
};

int main() {
    Child c;
    c.greet();
}
```

🔷 Modes of Inheritance

public → Public in base remains public, protected stays protected.
protected → Public & protected of base become protected.
private → Public & protected of base become private.

🔷 Types of Inheritance

Single
Multiple
Multilevel
Hierarchical
Hybrid

🔷 Polymorphism

Definition: "Many forms" → Ability of function/operator to behave differently.
Compile-time (static): Function overloading, Operator overloading.
Run-time (dynamic): Function overriding, Virtual functions.

🔷 Function Overloading

Definition: Same function name, different parameters.
Example:

```
#include
using namespace std;

class Math {
public:
    int add(int a, int b) { return a+b; }
    double add(double a, double b) { return a+b; }
};

int main() {
    Math m;
    cout << m.add(2,3) << endl;
    cout << m.add(2.5,3.5);
}
```

🔷 Function Overriding

Definition: Redefining base class function in derived class.
Example:

```
#include
using namespace std;

class Parent {
public:
    void show() { cout << "Parent class\n"; }
};
class Child: public Parent {
public:
    void show() { cout << "Child class\n"; }
};

int main() {
    Child c;
    c.show();
}
```

🔷 Virtual Function

Definition: Used to achieve **run-time polymorphism**.
Declared using `virtual` keyword in base class.
Example:

```
#include
using namespace std;

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

int main() {
    Base* b = new Derived();
    b->show();  // Calls Derived
}
```

🔷 Abstraction

Definition: Hiding implementation details & showing only essential features.
Achieved using **abstract classes** & **interfaces**.

🔷 Abstract Class

Definition: A class with **at least one pure virtual function**.
Cannot be instantiated.

Example:

```
#include
using namespace std;

class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};
class Circle : public Shape {
public:
    void draw() { cout << "Drawing Circle\n"; }
};

int main() {
    Shape* s = new Circle();
    s->draw();
}
```

🔷 Static Keyword

Uses in OOP:

Static variable in class → Shared by all objects.

Static member function → Can be called without object.

Example:

```
#include
using namespace std;

class Demo {
public:
    static int count;
    Demo() { count++; }
    static void show() { cout << "Count: " << count << endl; }
};
int Demo::count = 0;

int main() {
    Demo d1, d2;
    Demo::show();
}
```

Output:

```
Count: 2
```