Introduction to OOP in C++

- **Object-Oriented Programming (OOP)** is a programming paradigm that organizes software design around **objects** rather than functions and logic.
- C++ is one of the most popular **object-oriented languages**, extending C with OOP features.
- The main idea is to model real-world entities as **classes (blueprints)** and **objects (instances)**.
- OOP improves **code reusability, modularity, security, and flexibility** compared to **procedural programming**.

Principles of OOP

1. Encapsulation

Wrapping data and functions into a single unit (class).
Protects data using **access specifiers** (`private`, `public`, `protected`).

2. Abstraction

Hiding complex implementation and showing only necessary details.
Achieved using **abstract classes** and **pure virtual functions**.

3. Inheritance

One class can acquire properties and behaviors of another class.
Promotes **code reusability**.

4. Polymorphism

The ability of a function or object to behave differently in different situations.
Types:
Compile-time (static): Function/Operator overloading.
Run-time (dynamic): Virtual functions & overriding.

Advantages of OOP

- **Modularity** → Code is organized into objects and classes.
- **Reusability** → Inheritance allows code to be reused in new programs.
- **Maintainability** → Easy to update and modify code.
- **Security** → Encapsulation and data hiding prevent unauthorized access.
- **Flexibility** → Polymorphism allows dynamic changes in behavior.
- **Real-world modeling** → Objects represent real-world entities directly.

Disadvantages of OOP

- **Complexity** → OOP concepts like inheritance, polymorphism, and abstraction can be difficult for beginners.
- **Memory overhead** → Objects and features like virtual functions consume more memory.

- **Slower execution** → Compared to procedural programming, due to dynamic binding and abstraction.
- **Not suitable for small programs** → For very small/simple tasks, procedural programming (C) may be faster and easier.

Class in C++ (Data Member and Member Function)

- A **class** is a **user-defined data type** in C++ that groups **data members (variables)** and **member functions (methods)** into a single unit.
- It is the **blueprint** for creating objects.
- Objects are instances of a class.

Example

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

// Defining a class
class Student {
private:
    int rollNo;       // data member
    string name;      // data member

public:
    // function to set values
    void setData(int r, string n) {
        rollNo = r;
        name = n;
    }

    // function to display values
    void displayData() {
        cout << "Roll No: " << rollNo << endl;
        cout << "Name: " << name << endl;
    }
};

int main() {
    Student s1;   // creating object of Student class
    s1.setData(101, "Neeraj");
    s1.displayData();

    return 0;
}
```

Output

```
Roll No: 101
Name: Neeraj
```

Important Points

- **Class** = blueprint, **Object** = instance.
- By default, members of a class are **private**.
- **Access specifiers** control visibility:

    - `public` → accessible everywhere.
    - `private` → accessible only inside the class.
    - `protected` → accessible in class + derived class.
- Member functions can be **defined inside or outside** the class.

Class vs Object

Term	Meaning
Class	Blueprint/template containing data & functions.
Object	Instance of a class, created in memory.

Access Modifiers in C++

Access modifiers (also called **Access Specifiers**) define the **visibility** or **accessibility** of class members (data & functions).

Types of Access Modifiers

1. Public

Members are accessible **from anywhere** in the program (inside & outside the class).

2. Private *(default in C++ classes)*

Members are accessible **only inside the class**.
They cannot be accessed directly by objects.

3. Protected

Members are accessible inside the class and in derived (child) classes, but not outside.

Example

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

class Demo {
private:
    int a;   // private member

protected:
    int b;   // protected member

public:
    int c;   // public member

    void setValues(int x, int y, int z) {
        a = x;
        b = y;
        c = z;
    }
```

```
    void display() {
        cout << "a = " << a << endl;
        cout << "b = " << b << endl;
        cout << "c = " << c << endl;
    }
};

int main() {
    Demo d;
    d.setValues(10, 20, 30);
    d.display();

    // d.a = 50;  Error: 'a' is private
    // d.b = 60;  Error: 'b' is protected
    d.c = 70;   //  Allowed (public)
    d.display();
    return 0;
}
```

Scope Resolution Operator (::) in C++

Definition

The **scope resolution operator (::)** is used to define or access something **outside the scope of its class or namespace**.

Uses of Scope Resolution Operator (::)

1. To define member functions outside a class

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

class Test {
public:
    void show();   // function declaration
};

// function defined outside class using ::
void Test::show() {
    cout << "Function defined outside class" << endl;
}

int main() {
    Test t;
    t.show();
    return 0;
}
```

2. To access global variables when a local variable has the same name

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

int num = 100; // global variable

int main() {
```

```
    int num = 50; // local variable
    cout << "Local num = " << num << endl;
    cout << "Global num = " << ::num << endl; // using ::
    return 0;
}
```

Output:

```
Local num = 50
Global num = 100
```

3. With namespaces (e.g., `std::cout`)

To access identifiers inside a specific namespace.

Quick Summary

Concept	Meaning	Example
Public	Accessible everywhere	`object.member`
Private	Accessible only in class	object can't access
Protected	Accessible in class & derived class	used in inheritance
Scope Resolution (::)	Used to define/access global or class members outside their scope	`ClassName::function()`, `::globalVar`

Array of Objects in C++

- An array of objects is a collection of objects of the same class stored in contiguous memory locations.
- Just like arrays of `int`, `char`, etc., we can create arrays of class-type.

Example

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

class Student {
    int roll;
    char name[20];
public:
    void getData() {
        cout << "Enter Roll No: ";
        cin >> roll;
        cout << "Enter Name: ";
        cin >> name;
    }
    void showData() {
        cout << "Roll: " << roll << ", Name: " << name << endl;
    }
};

int main() {
    Student s[3];  // array of 3 objects
```

```
    cout << "Enter details of students:\n";
    for(int i=0; i<3; i++) {
        s[i].getData();
    }

    cout << "\nDisplaying student details:\n";
    for(int i=0; i<3; i++) {
        s[i].showData();
    }
    return 0;
}
```

Here, `s[0], s[1], s[2]` are objects of `Student`.

Passing Objects to Functions in C++

- **Pass by Value** – A copy of the object is passed (changes inside function do not affect original object).
- **Pass by Reference** – The actual object is passed (changes inside function affect original object).
- **Pass by Pointer** – The address of object is passed.

Example: Pass by Value

```
void display(Student s) {  // copy of object passed
    s.showData();
}

int main() {
    Student s1;
    s1.getData();
    display(s1);   // passing object to function
}
```

Example: Pass by Reference

```
void update(Student &s) {  // reference passed
    cout << "Updating roll no to 100\n";
    s.setRoll(100);  // modifies original object
}
```

Example: Pass by Pointer

```
void display(Student *s) {  // pointer to object
    s->showData();
}
```

Key Points

- Arrays of objects allow handling multiple objects efficiently.
- Objects can be passed to functions **by value, reference, or pointer** depending on whether changes should reflect back.
- **Pass by reference** is most common in practice, as it avoids unnecessary copying.

Friend Function in C++

- A **friend function** is a function that is **not a member of a class** but is given special permission to access the **private and protected members** of that class.
- Declared inside the class with the keyword `friend`.

Characteristics of Friend Function

- ✓ Declared inside the class with `friend` keyword.
- ✓ It is **not** a member function of the class.
- ✓ Defined **outside the class** without the scope resolution operator (`::`).
- ✓ Can access **private** and **protected** members of the class.
- ✓ Can be a **normal function**, a **member of another class**, or even a **friend of multiple classes**.
- ✓ Invoked like a normal function (not using object).

Example

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

class Box {
    int length;
public:
    Box(int l) { length = l; }

    // friend function declaration
    friend void printLength(Box b);
};

// friend function definition
void printLength(Box b) {
    cout << "Length of box = " << b.length << endl;
}

int main() {
    Box b1(10);
    printLength(b1);  // calling friend function
    return 0;
}
```

Output:

```
Length of box = 10
```

Friend Function with Two Classes

```
class B;  // forward declaration

class A {
    int x;
public:
    A(int a) { x = a; }
    friend void add(A, B);  // friend function
};
```

```
class B {
    int y;
public:
    B(int b) { y = b; }
    friend void add(A, B);  // friend function
};

void add(A obj1, B obj2) {
    cout << "Sum = " << obj1.x + obj2.y << endl;
}
```

Key Points

- Provides controlled access to private members.
- Breaks **data hiding** concept to some extent (so use only when necessary).
- Useful in **operator overloading** (like `operator+`, `operator<<`, etc.).

Forward Declaration of Class

- Forward declaration means **declaring the name of a class before its full definition**.
- Used when one class needs to reference another class but **definition is not available yet**.

Syntax

```
class B;   // forward declaration

class A {
    int x;
public:
    void set(B obj);  // can use reference or pointer to class B
};

class B {
    int y;
public:
    B(int b) { y = b; }
    friend void A::set(B obj);  // defining later
};
```

Forward declaration avoids compilation errors when two classes reference each other.

Individually Making a Friend Function of Another Class

- A function of **one class** can be made a **friend of another class**.
- This allows **controlled access** to private data of one class from a function that belongs to another class.

Example

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

class B;  // forward declaration
```

```
class A {
    int x;
public:
    A(int a) { x = a; }
    // declaring a function of class B as friend
    friend void B::showA(A obj);
};

class B {
public:
    void showA(A obj) {  // function of B accessing private of A
        cout << "Value of A::x = " << obj.x << endl;
    }
};

int main() {
    A a1(100);
    B b1;
    b1.showA(a1);  // B's function accessing A's private
    return 0;
}
```

Here, `showA()` is a **member of B** but also a **friend of A**.

Friend Class

- A **friend class** is a class that is given access to the **private and protected members** of another class.
- Declared using `friend class ClassName;`.

Syntax

```
class A {
    int secret;
public:
    A(int s) { secret = s; }
    friend class B;  // B is friend of A
};

class B {
public:
    void showSecret(A obj) {
        cout << "Secret = " << obj.secret << endl;
    }
};
```

Key Points about Friend Class

- ✓ All functions of **friend class** get access to private and protected members of the other class.
- ✓ Friendship is **not mutual** — if `B` is a friend of `A`, it doesn't mean `A` is a friend of `B`.
- ✓ Friendship is **not inherited** — derived classes don't automatically become friends.
- ✓ Useful when two classes work closely together (e.g., **Linked List and Node class**).