OOP

1. Class and Object

- **Class:** A class is a blueprint for creating objects (a particular data structure). It encapsulates data for the object and methods to manipulate that data.
- **Object:** An object is an instance of a class. Think of a class as a blueprint of a car, and an object as the actual car built from the blueprint.

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

// Define a class named 'Car'
class Car {
    public:
        string brand;
        string model;
        int year;
};

// Create objects from the class
int main() {
    Car car1;
    car1.brand = "Toyota";
    car1.model = "Corolla";
    car1.year = 2020;

    Car car2;
    car2.brand = "Honda";
    car2.model = "Civic";
    car2.year = 2022;
```

```
    cout << car1.brand << " " << car1.model << " " << car1.ye
ar << endl;
    cout << car2.brand << " " << car2.model << " " << car2.ye
ar << endl;

    return 0;
}
```

Explanation:

- We created a class `Car` with three attributes: `brand`, `model`, and `year`.
- Then, we created two objects, `car1` and `car2`, and assigned values to their attributes.

2. Access Modifiers

Access modifiers define how the members of the class (attributes and methods) can be accessed. The three main types are:

- **Public:** Members are accessible from outside the class.
- **Private:** Members are only accessible from within the class.
- **Protected:** Members are accessible within the class and by derived classes.

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

class Student {
    private:
        int studentID; // Private member
    public:
        string name; // Public member

        // Public method to set the private studentID
        void setID(int id) {
            studentID = id;
        }
```

```
        // Public method to get the private studentID
        int getID() {
            return studentID;
        }
};

int main() {
    Student student1;
    student1.name = "John Doe";
    student1.setID(12345);

    cout << "Name: " << student1.name << endl;
    cout << "Student ID: " << student1.getID() << endl;

    return 0;
}
```

Explanation:

- The `studentID` attribute is private, so it cannot be accessed directly from outside the class.
- We use public methods `setID()` and `getID()` to set and get the value of `studentID`.

3. Constructor

A constructor is a special method that is automatically called when an object is created. It is used to initialize the object.

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

class Book {
    public:
        string title;
        string author;
```

```
        int pages;

        // Constructor
        Book(string t, string a, int p) {  //Parameterized Co
nstructor
            title = t;
            author = a;
            pages = p;
        }
};

int main() {
    // Creating an object and passing values to the construct
or
    Book book1("The Great Gatsby", "F. Scott Fitzgerald", 18
0);

    cout << "Title: " << book1.title << endl;
    cout << "Author: " << book1.author << endl;
    cout << "Pages: " << book1.pages << endl;

    return 0;
}
```

Explanation:

- We created a constructor for the `Book` class that initializes the `title`, `author`, and `pages` attributes when an object is created.

4. Destructor

A destructor is a special method that is automatically called when an object is destroyed. It is used to perform any cleanup necessary before the object is removed from memory.

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

```
class Sample {
    public:
        // Constructor
        Sample() {
            cout << "Constructor called!" << endl;
        }

        // Destructor
        ~Sample() {
            cout << "Destructor called!" << endl;
        }
};

int main() {
    Sample obj; // Creating an object calls the constructor
    cout << "In the main function." << endl;

    // The destructor is automatically called at the end of t
he scope (when the object is destroyed).
    return 0;
}
```

Explanation:

- When the object `obj` is created, the constructor is called.
- When `obj` goes out of scope (at the end of `main()`), the destructor is automatically called.

5. Relatable Example

Imagine you're designing a class for a "Smartphone." The class would have attributes like `brand`, `model`, `batteryLife`, etc., and methods like `makeCall()`, `sendText()`, etc. Each smartphone you create (like `iPhone` or `Samsung Galaxy`) would be an object of the `Smartphone` class, each with its own unique set of attributes.

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

class Smartphone {
    private:
        int batteryLife;
    public:
        string brand;
        string model;

        // Constructor
        Smartphone(string b, string m, int battery) {
            brand = b;
            model = m;
            batteryLife = battery;
        }

        void makeCall(string number) {
            cout << "Calling " << number << " from " << brand
<< " " << model << endl;
        }

        void showBatteryLife() {
            cout << "Battery life is " << batteryLife << " ho
urs." << endl;
        }
};

int main() {
    Smartphone phone1("Apple", "iPhone 13", 20);
    phone1.makeCall("123-456-7890");
    phone1.showBatteryLife();
```

```
    return 0;
}
```

Explanation:

- The `Smartphone` class has attributes like `brand`, `model`, and `batteryLife`.
- We created a constructor to initialize these attributes.
- The `makeCall()` method simulates making a call, and `showBatteryLife()` displays the battery life.

These examples introduce the fundamental concepts of OOP in an easy and relatable way, helping fresh students understand the importance and utility of classes, objects, constructors, destructors, and access modifiers.

Some Additional Topics related to Constructor

(can ignore for now , will be teaching you in class hopefully)

▼ Constructor Overloading and Copy Constructor

1. Constructor Overloading

Constructor overloading in C++ allows you to define multiple constructors in a class, each with a different parameter list. This enables creating objects in various ways depending on the arguments passed.

Example:

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

class Rectangle {
    private:
        int length;
        int width;
```

```
    public:
        // Default constructor
        Rectangle() {
            length = 0;
            width = 0;
        }

        // Parameterized constructor 1
        Rectangle(int l, int w) {
            length = l;
            width = w;
        }

        // Parameterized constructor 2 (Square case)
        Rectangle(int side) {
            length = side;
            width = side;
        }

        // Method to calculate area
        int area() {
            return length * width;
        }
};

int main() {
    Rectangle rect1;           // Calls the default constr
uctor
    Rectangle rect2(10, 5);    // Calls the parameterized
constructor 1
    Rectangle rect3(7);        // Calls the parameterized
constructor 2

    cout << "Area of rect1: " << rect1.area() << endl; //
0
```

```
    cout << "Area of rect2: " << rect2.area() << endl; //
50
    cout << "Area of rect3: " << rect3.area() << endl; //
49

    return 0;
}
```

Explanation:

- **Default Constructor:** `Rectangle()` sets `length` and `width` to 0.
- **Parameterized Constructor 1:** `Rectangle(int l, int w)` allows creating a rectangle with specific `length` and `width`.
- **Parameterized Constructor 2:** `Rectangle(int side)` is for creating a square (same length and width).

2. Copy Constructor

A **copy constructor** is a special constructor in C++ used to create a new object as a copy of an existing object. It is automatically invoked when an object is passed by value, returned by value, or explicitly copied.

Example:

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

class Point {
    private:
        int x, y;

    public:
        // Parameterized constructor
        Point(int a, int b) {
            x = a;
            y = b;
```

```
        }

        // Copy constructor
        Point(const Point &p) {
            x = p.x;
            y = p.y;
            cout << "Copy constructor called!" << endl;
        }

        void display() {
            cout << "(" << x << ", " << y << ")" << endl;
        }
};

int main() {
    Point p1(10, 20); // Parameterized constructor is call
ed
    Point p2 = p1;    // Copy constructor is called

    cout << "Point p1: ";
    p1.display();

    cout << "Point p2: ";
    p2.display();

    return 0;
}
```

Explanation:

- **Parameterized Constructor:** `Point(int a, int b)` initializes the coordinates `x` and `y`.
- **Copy Constructor:** `Point(const Point &p)` creates a new object `p2` as a copy of `p1`.

- When `Point p2 = p1;` is executed, the copy constructor is invoked, and the object `p2` is created with the same values as `p1`.

Summary

- **Constructor Overloading** allows creating objects in different ways depending on the arguments.
- **Copy Constructor** is used to create a copy of an existing object, typically when an object is passed or returned by value.