Introduction to Object-Oriented Programming (OOP)

A Introduction

Object-Oriented Programming (OOP) is a programming paradigm that uses objects and classes to structure software. It allows for encapsulation, inheritance, and polymorphism, which help in organizing code and promoting reusability. OOP is based on the concept of objects, which can contain data and methods. Classes are blueprints for creating objects, defining their properties and behaviors. Key principles of OOP include:

- **Encapsulation:** Bundling data and methods that operate on that data within a single unit (class).
- **Inheritance:** Creating new classes based on existing ones, allowing for code reuse and extension.
- **Polymorphism:** Allowing objects to be treated as instances of their parent class, enabling flexibility in code.
- **Abstraction:** Hiding complex implementation details and exposing only the necessary parts of an object.

OOP is widely used in modern programming languages such as Python, Java, C++, and C#. It helps in building modular, maintainable, and scalable software systems.

B OOP vs Procedural Programming

Procedural Programming is a programming paradigm based on the concept of procedures or routines (also known as functions). In procedural programming, the focus is on writing sequences of instructions to perform tasks, and data is typically kept separate from the procedures that operate on it.
In contrast, Object-Oriented Programming (OOP) organizes code around objects, which bundle both data and the methods that operate on that data. This leads to several key differences:

- **Structure:** Procedural programming structures programs as a set of procedures and functions, while OOP structures programs as a collection of interacting objects.
- **Data Handling:** In procedural programming, data is often global or passed between functions, whereas in OOP, data is encapsulated within objects.
- **Reusability:** OOP promotes code reuse through inheritance and polymorphism, making it easier to extend and maintain code. Procedural programming relies on function reuse, which can be less flexible.
- **Modularity:** OOP encourages modularity by dividing code into classes and objects, while procedural programming divides code into functions and procedures.
- **Abstraction:** OOP provides higher levels of abstraction by modeling real-world entities as objects, whereas procedural programming focuses on the sequence of actions to be performed.

While both paradigms are widely used, OOP is generally preferred for large, complex, and scalable software systems, whereas procedural programming can be suitable for smaller, straightforward tasks.

C Basic Concepts of OOP

The basic concepts of Object-Oriented Programming (OOP) include:

- **Class:** A blueprint for creating objects, defining their properties (attributes) and behaviors (methods).
- **Object:** An instance of a class, representing a specific entity with its own state and behavior.
- **Attribute:** A variable that holds data associated with a class or object.
- **Method:** A function defined within a class that operates on the object's data.
- **Constructor:** A special method used to initialize an object when it is created.
- **Destructor:** A special method called when an object is destroyed, used for cleanup tasks.

Below is an example of a simple class declaration in C++:

Example: Class Declaration

```
class Person {
private:
    std::string name;
    int age;

public:
    // Constructor
    Person(std::string n, int a) : name(n), age(a) {}

    // Method to display information
    void display() {
        std::cout << "Name: " << name << ", Age: " << age <<
            std::endl;
    }
};
```

This example defines a `Person` class with private attributes `name` and `age`, a constructor to initialize them, and a method to display the person's information. In this example, encapsulation is demonstrated by declaring the attributes `name` and `age` as `private`. This means they cannot be accessed directly from outside the class. Instead, access to these attributes is controlled through public methods (such as the constructor and the `display()` method). Encapsulation helps protect the internal state of the object and ensures that data is only modified in controlled ways.

Practice Exercise:

Create a class called `Car` that has the following attributes: `make`, `model`, and `year`. Implement a constructor to initialize these attributes and a method called `displayInfo()` that prints the car's information in the format: "Make: [make], Model: [model], Year: [year]".

D Calling Methods

In Object-Oriented Programming, methods are functions defined within a class that operate on the data contained in the class's objects. To call a method, you need to create an instance (object) of the class and then use the dot operator to access the method.

Example: Calling Methods

```
class Dog {
private:
    std::string name;
    int age;
public:
    // Constructor
    Dog(std::string n, int a) : name(n), age(a) {}

    // Method to make the dog bark
    void bark() {
        std::cout << name << " says: Woof! Woof!" << std::endl;
    }

    // Method to display dog's information
    void displayInfo() {
        std::cout << "Dog Name: " << name << ", Age: " << age <<
           std::endl;
    }
};
int main() {
    // Creating an object of the Dog class
    Dog myDog("Buddy", 3);

    // Calling methods on the object
    myDog.bark();           // Output: Buddy says: Woof! Woof!
    myDog.displayInfo();    // Output: Dog Name: Buddy, Age: 3

    return 0;
}
```

In this example, the `Dog` class has two methods: `bark()` and `displayInfo()`. In the `main()` function, an object `myDog` of the `Dog` class is created. The methods are called using the dot operator, allowing access to the object's behavior. The output demonstrates how the methods operate on the object's data.

When calling methods, the object must be instantiated first. The dot operator is used to access the method, and the method can then perform actions or return values based on the object's state. Note that, if you are working with a pointer to an object, you would use the arrow operator (->) to call methods on the object pointed to by the pointer.

E Class and Object

A class is a blueprint for creating objects, defining their properties (attributes) and behaviors (methods). An object is an instance of a class, representing a specific entity with its own state and behavior. Classes encapsulate data and functionality, allowing for modular and organized code. Objects are created from classes and can interact with each other through their methods.

Example: Class and Object

```
class Student {
private:
    std::string name;
    int id;
    float gpa;
public:
    // Constructor
    Student(std::string n, int i, float g) : name(n), id(i),
       gpa(g) {}

    // Method to display student information
    void displayInfo() {
        std::cout << "Name: " << name << ", ID: " << id << ", GPA:
           " << gpa << std::endl;
    }
};

int main() {
    // Creating an object of the Student class
    Student student1("Alice", 101, 3.8);

    // Displaying student information
    student1.displayInfo();

    return 0;
}
```

In this example, the `Student` class defines three private attributes: `name`, `id`, and `gpa`. The constructor initializes these attributes, and the `displayInfo()` method prints the student's information. In the `main()` function, an object (i.e. `student1`) of the `Student` class is created, and its information is displayed.

F Public, Private, and Protected Access Modifiers

Access modifiers in Object-Oriented Programming (OOP) control the visibility and accessibility of class members (attributes and methods). The three main access modifiers are:

- **Public:** Members declared as public can be accessed from outside the class. They are part of the class's interface and can be used by any code that has access to the class.
- **Private:** Members declared as private can only be accessed from within the class itself. They are not accessible from outside the class, which helps encapsulate the internal state and behavior of the object.
- **Protected:** Members declared as protected can be accessed within the class and by derived classes (subclasses). They are not accessible from outside the class hierarchy, allowing for controlled inheritance.
- **Default:** If no access modifier is specified, the default access level is private for class members in C++. In other languages like Java, the default access level is package-private (accessible within the same package).

Example: Access Modifiers

```
class BankAccount {
private:
    std::string accountNumber; // Private member
    double balance; // Private member
public:
    // Constructor
    BankAccount(std::string accNum, double initialBalance)
        : accountNumber(accNum), balance(initialBalance) {}

    // Public method to deposit money
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited: " << amount << ", New
               Balance: " << balance << std::endl;
        } else {
            std::cout << "Invalid deposit amount." << std::endl;
        }
    }

    // Public method to display account information
    void displayInfo() {
        std::cout << "Account Number: " << accountNumber
                  << ", Balance: " << balance << std::endl;
    }
};

int main() {
    // Creating an object of the BankAccount class
    BankAccount account("123456789", 1000.0);

    // Displaying account information
    account.displayInfo();

    // Depositing money
    account.deposit(500.0);

    // Attempting to access private members (will result in a
       compilation error)
    // std::cout << account.accountNumber; // Error:
       'accountNumber' is private within this context

    return 0;
}
```

In this example, the `BankAccount` class has private members `accountNumber` and `balance`, which cannot be accessed directly from outside the class. The public methods `deposit()` and `displayInfo()` provide controlled access to the account's functionality. Attempting to access the private members directly from the `main()` function would result in a compilation error, demonstrating the encapsulation provided by the private access modifier.

Practice Exercise:

Create a class called `Rectangle` that has private attributes `length` and `width`. Implement public methods to calculate the area and perimeter of the rectangle, and a method to display the rectangle's dimensions. Ensure that the attributes cannot be accessed directly from outside the class.

G Constructor and Destructor

A constructor is a special member function of a class that is automatically called when an object of that class is created. It is used to initialize the object's attributes and allocate resources if needed. Constructors can take parameters to allow for flexible initialization.
A destructor is another special member function that is called when an object is destroyed. It is used to release resources, perform cleanup tasks, and deallocate memory that was allocated during the object's lifetime.

Example: Constructor and Destructor

```
class Book {
private:
    std::string title;
    std::string author;
    int pages;
public:
    // Constructor
    Book(std::string t, std::string a, int p) : title(t),
       author(a), pages(p) {
        std::cout << "Book ’" << title << "’ created." <<
           std::endl;
    }

    // Destructor
    ~Book() {
        std::cout << "Book ’" << title << "’ destroyed." <<
           std::endl;
    }

    // Method to display book information
    void displayInfo() {
        std::cout << "Title: " << title << ", Author: " << author
                  << ", Pages: " << pages << std::endl;
    }
};
int main() {
    // Creating an object of the Book class
    Book book1("1984", "George Orwell", 328);

    // Displaying book information
    book1.displayInfo();

    // The destructor will be called automatically when the object
       goes out of scope
    return 0;
}
```

In this example, the `Book` class has a constructor that initializes the `title`, `author`, and

`pages` attributes. When an object of the `Book` class is created, the constructor is called, and a message is printed. The destructor is defined to print a message when the object is destroyed, which happens automatically when it goes out of scope (e.g., at the end of the `main()` function).

Practice Exercise:

Create a class called `Circle` that has a private attribute `radius`. Implement a constructor to initialize the radius, and public methods to calculate and return the area and circumference of the circle. Also, provide a method to display the circle's radius, area, and circumference.