Object-Oriented Programming in C++

1. Classes and Objects

- **Class**: A blueprint for creating objects (instances). A class encapsulates data for the object and methods to manipulate that data.
- **Object**: An instance of a class. When a class is defined, no memory is allocated until an object of that class is created.

cpp

```
class Car {
public:
    // Attributes
    std::string brand;
    std::string model;
    int year;

    // Methods
    void displayInfo() {
        std::cout << "Brand: " << brand << ", Model: " <<
model << ", Year: " << year << std::endl;
    }
};

int main() {
    // Create an object of Car
    Car myCar;
    myCar.brand = "Toyota";
    myCar.model = "Corolla";
    myCar.year = 2020;

    // Call a method
```

```
    myCar.displayInfo();  // Output: Brand: Toyota,
Model: Corolla, Year: 2020

    return 0;
}
```

- **Attributes**: Variables inside a class are known as attributes (or data members).
- **Methods**: Functions inside a class are known as methods (or member functions).
- **Access Specifiers**: Control the access level of class members (`public`, `private`, `protected`).

2. Encapsulation

- **Encapsulation** is the bundling of data and methods that operate on the data within a single unit, or class, and restricting access to some of the object's components.
- Encapsulation is implemented in C++ by using access specifiers:
    - **`public`**: Members are accessible from outside the class.
    - **`private`**: Members cannot be accessed (or viewed) from outside the class.
    - **`protected`**: Members are accessible within the class and by derived classes.

cpp

```
class Employee {
private:
    std::string name;
    int id;

public:
    void setName(std::string empName) {
        name = empName;
    }
```

```
    std::string getName() {
        return name;
    }

    void setID(int empID) {
        id = empID;
    }

    int getID() {
        return id;
    }
};

int main() {
    Employee emp;
    emp.setName("John Doe");
    emp.setID(12345);

    std::cout << "Employee Name: " << emp.getName() <<
std::endl;  // Output: Employee Name: John Doe
    std::cout << "Employee ID: " << emp.getID() <<
std::endl;      // Output: Employee ID: 12345

    return 0;
}
```

- **Getter and Setter Methods**: These methods allow controlled access to private data members, ensuring encapsulation.

3. Inheritance

- **Inheritance** allows a new class (derived class) to inherit attributes and methods from an existing class (base class). This promotes code reuse and establishes a natural hierarchy.
- **Types of Inheritance**:
    - **Single Inheritance**: A class inherits from one base class.

    - **Multiple Inheritance**: A class inherits from more than one base class.
    - **Multilevel Inheritance**: A class is derived from a class that is also derived from another class.
    - **Hierarchical Inheritance**: Multiple classes are derived from a single base class.
    - **Hybrid Inheritance**: A combination of two or more types of inheritance.

cpp

```
class Vehicle {  // Base class
public:
    std::string brand = "Ford";

    void honk() {
        std::cout << "Beep beep!" << std::endl;
    }
};

class Car : public Vehicle {  // Derived class
public:
    std::string model = "Mustang";
};

int main() {
    Car myCar;
    myCar.honk();  // Inherited from Vehicle
    std::cout << myCar.brand << " " << myCar.model <<
std::endl;  // Output: Ford Mustang

    return 0;
}
```

- **Base Class**: The class being inherited from.
- **Derived Class**: The class that inherits from the base class.
- **Access Control in Inheritance**:

    - **`public` inheritance**: Public and protected members of the base class remain public and protected in the derived class.
    - **`protected` inheritance**: Public and protected members of the base class become protected in the derived class.
    - **`private` inheritance**: Public and protected members of the base class become private in the derived class.

4. Polymorphism

- **Polymorphism** means "many forms" and allows methods to do different things based on the object they are acting upon.
- **Compile-time Polymorphism** (Static Binding):
    - Achieved through **function overloading** and **operator overloading**.
- **Runtime Polymorphism** (Dynamic Binding):
    - Achieved through **virtual functions** and **inheritance**.
- **Function Overloading**: Multiple functions can have the same name with different parameters.

cpp

```
class Print {
public:
    void show(int i) {
        std::cout << "Integer: " << i << std::endl;
    }
    void show(double d) {
        std::cout << "Double: " << d << std::endl;
    }
    void show(std::string s) {
        std::cout << "String: " << s << std::endl;
    }
};

int main() {
    Print obj;
    obj.show(5);             // Output: Integer: 5
    obj.show(9.99);          // Output: Double: 9.99
```

```
    obj.show("Hello");       // Output: String: Hello

    return 0;
}
```

- **Virtual Functions**: Allow for dynamic binding, where the function call is determined at runtime.

cpp

```
class Animal {
public:
    virtual void sound() {
        std::cout << "Some generic animal sound" <<
std::endl;
    }
};

class Dog : public Animal {
public:
    void sound() override {  // Overriding base class
method
        std::cout << "Woof woof!" << std::endl;
    }
};

int main() {
    Animal* animal = new Dog();
    animal->sound();  // Output: Woof woof!

    delete animal;
    return 0;
}
```

- **Pure Virtual Functions and Abstract Classes**:

    - A **pure virtual function** is a function that has no implementation in the base class and must be overridden in derived classes.
    - A class containing at least one pure virtual function is considered an **abstract class** and cannot be instantiated.

cpp

```cpp
class Shape {
public:
    virtual void draw() = 0;  // Pure virtual function
};

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

int main() {
    Circle c;
    c.draw();  // Output: Drawing a circle

    return 0;
}
```

5. Abstraction

- **Abstraction** is the concept of hiding the complex implementation details and showing only the essential features of an object.
- Achieved using **abstract classes** and **interfaces** in C++.
- **Abstract Class**: A class that cannot be instantiated and often contains pure virtual functions.

cpp

```cpp
class AbstractEmployee {
```

```
public:
    virtual void askForPromotion() = 0;  // Pure virtual
function
};

class Employee : public AbstractEmployee {
private:
    std::string name;
    int age;

public:
    Employee(std::string empName, int empAge) :
name(empName), age(empAge) {}

    void askForPromotion() override {
        if (age > 30)
            std::cout << name << " got promoted!" <<
std::endl;
        else
            std::cout << name << ", sorry, no promotion."
<< std::endl;
    }
};

int main() {
    Employee emp1("John", 35);
    emp1.askForPromotion();  // Output: John got
promoted!

    return 0;
}
```

- **Interfaces in C++**: While C++ does not have built-in interfaces like some other languages (e.g., Java), you can achieve a similar effect using abstract classes with only pure virtual functions.

Summary

- **Classes and Objects**: Core components of OOP, where classes are blueprints for creating objects.
- **Encapsulation**: Bundles data and methods, restricting access to internal data using private members and exposing only what is necessary through public methods.
- **Inheritance**: Allows new classes to inherit attributes and methods from existing classes, promoting code reuse.
- **Polymorphism**: Allows functions to process objects differently based on their data type or class.
- **Abstraction**: Hides complex implementation details, showing only essential features through abstract classes and interfaces.