Inheritance and Polymorphism in Object-Oriented Programming

A Introduction

Inheritance is a fundamental concept in Object-Oriented Programming (OOP) that allows a new class (called the derived class or subclass) to inherit properties and behaviors from an existing class (called the base class or superclass). This promotes code reuse and establishes a hierarchical relationship between classes. The derived class can extend or modify the functionality of the base class by adding new attributes and methods or overriding existing ones. Inheritance can be classified into different types, such as single inheritance (one base class), multiple inheritance (multiple base classes), and hierarchical inheritance (one base class with multiple derived classes).

Example: Single Inheritance

```
class Animal {
protected:
    std::string name;
public:
    // Constructor
    Animal(std::string n) : name(n) {}

    // Method to make a sound
    virtual void makeSound() {
        std::cout << name << "␣makes␣a␣sound." << std::endl;
    }
};

class Dog : public Animal {
public:
    // Constructor
    Dog(std::string n) : Animal(n) {}

    // Overriding the makeSound method
    void makeSound() override {
        std::cout << name << "␣barks." << std::endl;
    }
};
class Cat : public Animal {
public:
    // Constructor
    Cat(std::string n) : Animal(n) {}

    // Overriding the makeSound method
    void makeSound() override {
        std::cout << name << "␣meows." << std::endl;
    }
};
int main() {
    // Creating objects of the derived classes
    Dog dog("Buddy");
    Cat cat("Whiskers");

    // Calling the makeSound method
    dog.makeSound(); // Output: Buddy barks.
    cat.makeSound(); // Output: Whiskers meows.

    return 0;
}
```

In this example, the `Animal` class is the base class with a protected attribute `name` and a virtual method `makeSound()`. The `Dog` and `Cat` classes are derived from the `Animal` class, inheriting its properties and methods. They override the `makeSound()` method to provide specific implementations for dogs and cats. When the `makeSound()` method is called on the derived class objects, the overridden methods are executed.

Note that the `makeSound()` method in the base class is declared as `virtual`, allowing for polymorphism. This means that when a derived class object is treated as an instance of the base class, the overridden method in the derived class will be called. This is called **method**

overriding and is a key feature of inheritance in OOP.

B Constructor and Destructor with Inheritance

When using inheritance, constructors and destructors play an important role in the initialization and cleanup of objects. In C++, when a derived class object is created, the base class constructor is called first, followed by the derived class constructor. Similarly, when the object is destroyed, the derived class destructor is called first, followed by the base class destructor. This ensures that the base part of the object is properly initialized and cleaned up.

If the base class constructor requires arguments, the derived class constructor must explicitly call the base class constructor using an initializer list.

Example: Constructor and Destructor Order

```
class Base {
public:
    Base() {
        std::cout << "Base␣constructor␣called." << std::endl;
    }
    ~Base() {
        std::cout << "Base␣destructor␣called." << std::endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        std::cout << "Derived␣constructor␣called." << std::endl;
    }
    ~Derived() {
        std::cout << "Derived␣destructor␣called." << std::endl;
    }
};

int main() {
    Derived obj;
    return 0;
}
```

Output:

```
Base constructor called.
Derived constructor called.
Derived destructor called.
Base destructor called.
```

This example demonstrates the order in which constructors and destructors are called in an inheritance hierarchy. The base class constructor is always called before the derived class constructor, and the derived class destructor is called before the base class destructor. This order ensures proper resource management and object lifecycle handling in inheritance.

C Multiple Inheritance

Multiple inheritance is a feature in Object-Oriented Programming (OOP) that allows a class (the derived class) to inherit properties and behaviors from more than one base class. This enables the derived class to combine functionalities from multiple sources, promoting code reuse and flexibility.

However, multiple inheritance can lead to complexities such as the ”diamond problem,” where a derived class inherits from two classes that share a common base class. To resolve this issue, many programming languages implement mechanisms like virtual inheritance or interfaces.

Example: Multiple Inheritance

```
class Base1 {
public:
    void displayBase1() {
        std::cout << "Base1 method called." << std::endl;
    }
};
class Base2 {
public:
    void displayBase2() {
        std::cout << "Base2 method called." << std::endl;
    }
};
class Derived : public Base1, public Base2 {
public:
    void displayDerived() {
        std::cout << "Derived method called." << std::endl;
    }
};
int main() {
    Derived obj;

    // Calling methods from base classes
    obj.displayBase1(); // Output: Base1 method called.
    obj.displayBase2(); // Output: Base2 method called.

    // Calling method from derived class
    obj.displayDerived(); // Output: Derived method called.

    return 0;
}
```

In this example, the `Derived` class inherits from both `Base1` and `Base2`. It can access methods from both base classes, demonstrating multiple inheritance. The derived class can also have its own methods, such as `displayDerived()`. However, care must be taken when using multiple inheritance to avoid ambiguity and maintain code clarity. In languages like C++, virtual inheritance can be used to resolve issues related to the diamond problem, ensuring that only one instance of the common base class is present in the derived class hierarchy.

Practice Exercise:

Create a class hierarchy with a base class called `Shape` that has a method `draw()`. Create two derived classes, `Circle` and `Square`, that override the `draw()` method. Then, create a third class called `ColoredShape` that inherits from both `Circle` and `Square`. Implement a method in `ColoredShape` that calls the `draw()` method of both base classes, so that when this method is called, it outputs the drawing behavior of both a circle and a square, clearly indicating which shape is being drawn. Ensure that the `draw()` method in `ColoredShape` correctly distinguishes between the two shapes.

D Polymorphism

Polymorphism is a core concept in Object-Oriented Programming (OOP) that allows objects of different classes to be treated as objects of a common base class. It enables a single interface to represent different underlying forms (data types). Polymorphism can be achieved through method overriding and method overloading.

- **Method Overriding:** This occurs when a derived class provides a specific implementation of a method that is already defined in its base class. The base class method is declared as `virtual`, allowing the derived class to override it.
- **Method Overloading:** This allows multiple methods with the same name but different parameter lists to exist within the same class. The appropriate method is chosen based on the arguments passed during the method call.
- **Dynamic Polymorphism:** This is achieved through virtual functions and pointers or references to base class types. It allows the program to determine at runtime which method to invoke based on the actual object type.

Example: Polymorphism with Virtual Functions

```
class Shape {
public:
    // Virtual method for drawing the shape
    virtual void draw() {
        std::cout << "Drawing␣a␣generic␣shape." << std::endl;
    }
};
class Circle : public Shape {
public:
    // Overriding the draw method
    void draw() override {
        std::cout << "Drawing␣a␣circle." << std::endl;
    }
};
class Square : public Shape {
public:
    // Overriding the draw method
    void draw() override {
        std::cout << "Drawing␣a␣square." << std::endl;
    }
};
int main() {
    Shape* shape1 = new Circle(); // Pointer to base class
    Shape* shape2 = new Square(); // Pointer to base class

    // Calling the draw method on different shapes
    shape1->draw(); // Output: Drawing a circle.
    shape2->draw(); // Output: Drawing a square.

    // Clean up
    delete shape1;
    delete shape2;

    return 0;
}
```

In this example, the `Shape` class has a virtual method `draw()`. The `Circle` and `Square` classes override this method to provide their specific implementations. In the `main()` function, pointers of type `Shape` are used to refer to objects of the derived classes. When the `draw()` method is called on these pointers, the appropriate overridden method is executed based on the actual object type, demonstrating polymorphism.

E Abstract Classes and Interfaces

An abstract class is a class that cannot be instantiated directly and is meant to be subclassed. It can contain pure virtual functions (methods without implementation) that must be implemented by derived classes. Abstract classes are used to define a common interface for a group of related classes, allowing for polymorphism.

An interface is a contract that defines a set of methods that a class must implement. In C++, interfaces are typically created using abstract classes with only pure virtual functions. A class that implements an interface must provide concrete implementations for all the methods defined in the interface.

In the previous example, the `Shape` class can be considered an abstract class if it contains at least one pure virtual function. If we modify the `Shape` class to make the `draw()` method a pure virtual function, it becomes an abstract class.

Example: Abstract Class

```
class Shape {
public:
    // Pure virtual method for drawing the shape
    virtual void draw() = 0; // This makes Shape an abstract class
};
class Circle : public Shape {
public:
    // Implementing the pure virtual method
    void draw() override {
        std::cout << "Drawing a circle." << std::endl;
    }
};
class Square : public Shape {
public:
    // Implementing the pure virtual method
    void draw() override {
        std::cout << "Drawing a square." << std::endl;
    }
};
int main() {
    Shape* shape1 = new Circle(); // Pointer to base class
    Shape* shape2 = new Square(); // Pointer to base class

    // Calling the draw method on different shapes
    shape1->draw(); // Output: Drawing a circle.
    shape2->draw(); // Output: Drawing a square.

    // Clean up
    delete shape1;
    delete shape2;

    return 0;
}
```

In this modified example, the `Shape` class has a pure virtual method `draw()`, making it an abstract class. The derived classes `Circle` and `Square` provide concrete implementations of the `draw()` method. The main function demonstrates polymorphism by calling the `draw()` method on pointers of type `Shape`, which refer to objects of the derived classes.

Practice Exercise:

Create an abstract class called `Vehicle` that has a pure virtual method `startEngine()`. Implement two derived classes, `Car` and `Motorcycle`, that provide their own implementations of the `startEngine()` method. In the main function, create pointers of type `Vehicle` to refer to objects of both derived classes and call the `startEngine()` method on each.

F Operator Overloading

Operator overloading is a feature in Object-Oriented Programming (OOP) that allows developers to define custom behavior for operators (such as +, -, *, etc.) when applied to user-defined types (classes). This enables objects of these classes to be manipulated using familiar operators, enhancing code readability and expressiveness.

In C++, operator overloading is achieved by defining special member functions or friend functions that specify how the operator should behave for the class. The syntax for operator overloading involves using the `operator` keyword followed by the operator symbol.

Example: Operator Overloading

```
class Complex {
private:
    double real;
    double imag;
public:
    // Constructor
    Complex(double r, double i) : real(r), imag(i) {}

    // Overloading the + operator
    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    // Method to display the complex number
    void display() {
        std::cout << real << " + " << imag << "i" << std::endl;
    }
};
int main() {
    Complex c1(2.0, 3.0);
    Complex c2(1.5, 4.5);

    // Using the overloaded + operator
    Complex c3 = c1 + c2;

    // Displaying the result
    c3.display(); // Output: 3.5 + 7.5i

    return 0;
}
```

In this example, the `Complex` class represents a complex number with real and imaginary parts. The `operator+` function is defined to overload the + operator, allowing two `Complex`

objects to be added together using the + operator. The result is a new `Complex` object that represents the sum of the two complex numbers. The `display()` method is used to print the complex number in a readable format.