Runtime Polymorphism

Runtime polymorphism, also known as **dynamic polymorphism**, is a feature in object-oriented programming where the function call is resolved at runtime rather than compile time. It allows a program to decide which function to call based on the actual object type, not the reference or pointer type.

How It Works:

- Runtime polymorphism is achieved using **function overriding** and **virtual functions**.
- When a virtual function is called through a base class pointer or reference, the compiler generates code to determine the actual function to call at runtime (using a **vtable** or virtual table).

Key Points:

- It allows flexibility in program design by enabling the same interface to behave differently based on the derived class implementation.
- It is useful in scenarios where the exact type of the object is not known until runtime.

Example

```
class Animal {
public:
    virtual void makeSound() {
        cout << "Animal sound" << endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() override {
        cout << "Bark" << endl;
    }
};

class Cat : public Animal {
public:
    void makeSound() override {
        cout << "Meow" << endl;
    }
};
```

```cpp
class Mouse : public Animal {
public:
    void makeSound() override {
        cout << "Squeak" << endl;
    }
};

int main() {
    Animal* animals[3];
    animals[0] = new Dog();
    animals[1] = new Cat();
    animals[2] = new Mouse();

    for (int i = 0; i < 3; i++) {
        animals[i]->makeSound(); // Calls the appropriate derived class function
    }

    // Cleanup
    for (int i = 0; i < 3; i++) {
        delete animals[i];
    }
    return 0;
}
```

Function Overriding

Function overriding is a feature in object-oriented programming that allows a derived class to provide a specific implementation of a method that is already defined in its base class. This enables the derived class to customize or extend the behavior of the base class method.

Key Points

- The method in the base class must be declared as `virtual` to allow overriding.
- The method in the derived class must have the same **name**, **return type**, and **parameters** as the base class method.
- The `override` keyword (in C++11 and later) can be used to explicitly indicate that a function is intended to override a base class function.
- The `final` keyword can be used in the derived class to prevent further overriding of a method.

Example:

```
class Animal {
public:
    virtual void makeSound() {  // Virtual function allows overriding
        cout << "Animal sound" << endl;
    }
};

class Dog final : public Animal {  // Marking Dog as final prevents further
   inheritance
public:
    void makeSound() final override {  // Prevents overriding in further derived
       classes
        cout << "Bark" << endl;
    }
};

// Uncommenting this will cause a compilation error
// class Puppy : public Dog {
// public:
//     void makeSound() override {  // ERROR: Cannot override a final function
//         cout << "Small bark" << endl;
//     }
// };

int main() {
    Animal* animal = new Dog();
    animal->makeSound();  // Output: Bark (Derived class method is called)

    delete animal;
    return 0;
}
```

Virtual Functions, VTable, and VPtr

1. Virtual Functions

A virtual function is a member function declared in a base class using the `virtual` keyword. It can be redefined in derived classes. Virtual functions enable **dynamic binding** (or late binding), where the function call is resolved at runtime based on the type of the object being pointed to or referenced.

2. VTable (Virtual Table)

A VTable is a mechanism used by C++ to support dynamic binding (runtime polymorphism). It is a table of function pointers that is created by the compiler for each class that contains virtual functions. Each entry in the VTable points to the most derived function that should be called for that virtual function.

3. VPtr (Virtual Pointer)

Each object of a class with virtual functions contains a hidden pointer called VPtr. This VPtr points to the VTable of the class. When a virtual function is called, the VPtr is used to find the correct function in the VTable.

How It Works

- When a class has virtual functions, the compiler creates a VTable for that class.
- Each object of that class contains a VPtr that points to the VTable.
- When a virtual function is called through a base class pointer or reference, the VPtr is used to look up the correct function in the VTable.

Example

```
class Base {
public:
    virtual void func1() {
        cout << "Base::func1" << endl;
    }
    virtual void func2() {
        cout << "Base::func2" << endl;
    }};
class Derived : public Base {
public:
    void func1() override {
        cout << "Derived::func1" << endl;
    }
    void func2() override {
        cout << "Derived::func2" << endl;
    }};
```

```
int main() {
    Base* base = new Derived();
    base->func1(); // Output: Derived::func1
    base->func2(); // Output: Derived::func2
    delete base;
    return 0;
}
```

Diamond Problem in Multiple Inheritance

The diamond problem occurs in multiple inheritance when a class inherits from two or more classes that have a common base class. This leads to ambiguity in function calls and data members due to multiple copies of the base class being inherited.

Example

```
class A {
public:
    void display() {
        cout << "Class A" << endl;
    }
};

class B : public A {};
class C : public A {};

class D : public B, public C {};
```

Problem:

If you create an object of class `D` and call `display()`, the compiler will not know which `display()` to call (from `B` or `C`).

Solution

Use **virtual inheritance** to ensure that only one instance of the common base class is inherited.

Fixed Example

```
class A {
public:
    void display() {
        cout << "Class A" << endl;
    }
};

class B : virtual public A {};
class C : virtual public A {};

class D : public B, public C {};

int main() {
    D obj;
    obj.display(); // No ambiguity now
    return 0;
}
```

Constructors and Destructors

1. Constructors

A constructor is a special member function that is automatically called when an object is created. In inheritance, the base class constructor is called first, followed by the derived class constructor.

2. Destructors

A destructor is a special member function that is automatically called when an object is destroyed. In inheritance, the derived class destructor is called first, followed by the base class destructor.

Example

```
class Base {
public:
    Base() {
        cout << "Base Constructor" << endl;
    }

    ~Base() {
        cout << "Base Destructor" << endl;
    }
};
```

```
class Derived : public Base {
public:
    Derived() {
        cout << "Derived Constructor" << endl;
    }
    ~Derived() {
        cout << "Derived Destructor" << endl;
    }
};

int main() {
    Derived obj; // Output: Base Constructor -> Derived Constructor
    return 0;    // Output: Derived Destructor -> Base Destructor
}
```

Virtual Destructors

Need for Virtual Destructors

- When a base class pointer points to a derived class object and the object is deleted, only the base class destructor is called if the destructor is not virtual. This can lead to resource leaks or undefined behavior.
- A virtual destructor ensures that the derived class destructor is called first, followed by the base class destructor.

Example

```
class Base {
public:
    Base() {
        cout << "Base Constructor" << endl;
    }
    virtual ~Base() {
        cout << "Base Destructor" << endl;
    }
};

class Derived : public Base {
public:
    Derived() {
        cout << "Derived Constructor" << endl;
    }
```

```
    ~Derived() {
        cout << "Derived Destructor" << endl;
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr; // Output: Derived Destructor -> Base Destructor
    return 0;
}
```

Abstract Classes

An abstract class is a class that cannot be instantiated and is used as a base class for other classes. It contains at least one **pure virtual function**, which is a virtual function declared with `= 0`.

Purpose

Abstract classes are used to define interfaces or common functionality that derived classes must implement.

Example

```
class Shape {
public:
    virtual void draw() = 0; // Pure virtual function
};
class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle" << endl;
    }
};
int main() {
    Shape* shape = new Circle();
    shape->draw(); // Output: Drawing Circle
    delete shape;
    return 0;
}
```

Interface in C++

In C++, an **interface** is implemented using an **abstract class** where all methods are pure virtual. This ensures that any class derived from the interface must provide its own implementation for all methods.

Key Features of an Interface:

- **Contains Only Pure Virtual Functions:** No function implementation in the base class.
- **Cannot be Instantiated:** You cannot create objects of an interface.
- **Forces Derived Classes to Implement Functions:** Ensures a uniform function signature across implementations.
- **Used for Polymorphism:** Enables dynamic function calls using base class pointers.

Example:

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

// Interface: Abstract class with pure virtual functions
class IShape {
public:
    virtual void draw() = 0;  // Pure virtual function
    virtual ~IShape() {}      // Virtual destructor
};

// Derived class: Implements the IShape interface
class Circle : public IShape {
public:
    void draw() override {
        cout << "Drawing a Circle" << endl;
    }
};

// Derived class: Implements the IShape interface
class Square : public IShape {
public:
    void draw() override {
        cout << "Drawing a Square" << endl;
    }
};
```

```
int main() {
    IShape* s1 = new Circle();
    IShape* s2 = new Square();

    s1->draw();
    s2->draw();

    delete s1;
    delete s2;
    return 0;
}
```

Lab exercises

Exercise 1

Package Delivery Management System: You are tasked with developing a program for a package delivery management system used by services like **FedEx and DHL**. The system should use inheritance and polymorphism to calculate shipping costs for different types of packages.

(a) Define Classes Using Inheritance

Create a **base class `Package`** and derive the following two classes from it:

- **`TwoDayPackage`**: Represents packages with a two-day delivery option.
- **`OvernightPackage`**: Represents packages with an overnight delivery option.

(b) Implement the Base Class (`Package`)

The `Package` class should have:

- **Attributes:** Sender and recipient details (name, address, city, state, ZIP code), `weight` (double), and `costPerOunce` (double).
- A constructor to initialize all attributes while ensuring positive values for weight and cost per ounce.
- A **`calculateCost()`** method that returns the shipping cost as `weight × costPerOunce`.

(c) Implement the Derived Classes

Each derived class must:

- Have **additional attributes** specific to their shipping method:
  - **`TwoDayPackage`**: `flatFee` (double) for two-day delivery.
  - **`OvernightPackage`**: `extraFeePerOunce` (double) for overnight shipping.
- A constructor that **initializes both inherited and new attributes**.
- A **`calculateCost()`** method that **overrides** the base class function:
  - **`TwoDayPackage`**: Returns `Package::calculateCost() + flatFee`.
  - **`OvernightPackage`**: Returns `weight × (costPerOunce + extraFeePerOunce)`.

(d) Implement the `main()` Function

- Create an array of `Package*` objects (standard, two-day, and overnight).
- Use **polymorphism** to call `calculateCost()` dynamically.
- Display sender/recipient details and the total shipping cost for each package.

Your program should prompt the user to enter package details, choose a shipping option, and display the calculated cost on runtime.

Exercise 2

Enterprise Payroll Management System: You are tasked with developing a program for an Enterprise Payroll Management System that categorizes employees based on their job roles using multilevel inheritance and run-time polymorphism. The system should dynamically compute salaries while ensuring proper class hierarchy and function overriding for structured employee information display.

(a) Define Classes Using Inheritance

Create a **base class `Person`** and derive the following two classes from it:

- **`Employee`**: Represents a general employee with a salary and experience.
- **`JobRole`**: Represents specific job roles and modifies salary calculations based on the role.

(b) Implement the Base Class (`Person`)

The `Person` class should have:

- **Attributes:** `name` (string), `personID` (int), and `age` (int).
- A constructor to initialize these attributes.
- A **`displayInfo()`** method declared as **virtual** for function overriding.

(c) Implement the Derived Classes

Each derived class must:

- **`Employee`** (inherits from `Person`):
  - **Attributes:** `baseSalary` (double), `yearsOfExperience` (int), and `department` (string).
  - Implements **`calculateSalary()`**: Adjusts salary based on experience.
  - Overrides **`displayInfo()`** to include employee details.
- **`JobRole`** (inherits from `Employee`):
  - **Attributes:** `jobTitle` (string).
  - Overrides **`calculateSalary()`** to adjust salary based on job type (`Salaried` or `Hourly`).
  - Overrides **`displayInfo()`** to include job role details.

(d) Implement the `main()` Function

- Create an **array of `JobRole*` objects**, each representing an employee with a different role.
- Use **run-time polymorphism** to dynamically call `calculateSalary()` and `displayInfo()`.
- Display each employee's structured details and final salary.

Your program should initialize employee data, categorize them into different job roles, and compute their salaries dynamically based on experience and job type.

Exercise 3

Smart Home Appliance Management System: You are tasked with developing a Smart Home Appliance Management System that monitors and controls different types of home appliances. The system should allow efficient memory management and ensure that every appliance implements essential functionalities while enforcing a common interface.

(a) Define Classes Using Inheritance

Create a **base class `Appliance`** and derive the following two classes from it:

- **`SmartLight`**: Represents a smart light that can be turned on/off and adjusted for brightness.
- **`SmartThermostat`**: Represents a smart thermostat that can regulate temperature.

(b) Implement the Base Class (`Appliance`)

The `Appliance` class should have:

- **Attributes:** `brand` (string), `model` (string), and `powerStatus` (bool) to track whether the appliance is on or off.
- A constructor to initialize these attributes.
- A **`turnOn()`** and **`turnOff()`** method to control power.
- A pure virtual function **`displayStatus()`** that must be implemented by all derived classes.
- A virtual destructor.

(c) Implement the Derived Classes

Each derived class must:

- **`SmartLight`**:
  - Additional attribute: `brightnessLevel` (int) to adjust brightness.
  - Implements **`displayStatus()`** to show power and brightness level.
- **`SmartThermostat`**:
  - Additional attribute: `temperature` (double) to regulate room temperature.
  - Implements **`displayStatus()`** to show power and temperature setting.

(d) Implement the `main()` Function

- Create an **array of `Appliance*` objects**, storing different appliance types dynamically.
- Use **polymorphism** to call `displayStatus()` dynamically for each appliance.
- Ensure proper memory management when deleting appliances.

Your program should allow the creation of different smart appliances, control their power status, and display their unique settings while ensuring safe memory management.