2. Types of Inheritance and Method Overriding

2.1 Single Inheritance

Definition: One derived class inherits from one base class.

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

class Person {
protected:
    string name;
    int age;

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

    void introduce() {
        cout << "Hi, I'm " << name << ", " << age << " years old." << endl;
    }
};

class Student : public Person {
private:
    string studentId;
    double gpa;

public:
    Student(string n, int a, string id, double g)
        : Person(n, a), studentId(id), gpa(g) {}

    void study() {
        cout << name << " is studying..." << endl;
```

```
    }

    void showGPA() {
        cout << "GPA: " << gpa << endl;
    }
};

int main() {
    Student s("Alice", 20, "S001", 3.8);
    s.introduce();  // Inherited
    s.study();      // Own method
    s.showGPA();    // Own method

    return 0;
}
```

2.2 Multilevel Inheritance

Definition: A class is derived from another derived class.

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

// Level 1: Base class
class LivingBeing {
protected:
    bool isAlive;

public:
    LivingBeing() : isAlive(true) {
        cout << "LivingBeing created" << endl;
    }

    void breathe() {
        cout << "Breathing..." << endl;
    }
};

// Level 2: Derived from LivingBeing
```

```
class Animal : public LivingBeing {
protected:
    string species;

public:
    Animal(string s) : species(s) {
        cout << "Animal created: " << species << endl;
    }

    void move() {
        cout << species << " is moving..." << endl;
    }
};

// Level 3: Derived from Animal
class Dog : public Animal {
private:
    string name;

public:
    Dog(string n) : Animal("Canine"), name(n) {
        cout << "Dog created: " << name << endl;
    }

    void bark() {
        cout << name << " is barking!" << endl;
    }

    void showCapabilities() {
        breathe();  // From LivingBeing
        move();     // From Animal
        bark();     // From Dog
    }
};

int main() {
    Dog myDog("Buddy");
    cout << "\nDog capabilities:" << endl;
    myDog.showCapabilities();
```

```
    return 0;
}
```

Output:

```
LivingBeing created
Animal created: Canine
Dog created: Buddy

Dog capabilities:
Breathing...
Canine is moving...
Buddy is barking!
```

2.3 Multiple Inheritance

Definition: A class inherits from multiple base classes.

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

class Engine {
protected:
    int horsepower;

public:
    Engine(int hp) : horsepower(hp) {
        cout << "Engine: " << horsepower << " HP" << endl;
    }

    void start() {
        cout << "Engine started: " << horsepower << " HP" << endl;
    }
};

class GPS {
protected:
    string currentLocation;
```

```
public:
    GPS(string loc) : currentLocation(loc) {
        cout << "GPS initialized at: " << loc << endl;
    }

    void navigate(string destination) {
        cout << "Navigating from " << currentLocation
             << " to " << destination << endl;
    }
};

class SmartCar : public Engine, public GPS {
private:
    string model;

public:
    SmartCar(string m, int hp, string loc)
        : Engine(hp), GPS(loc), model(m) {
        cout << "SmartCar created: " << model << endl;
    }

    void drive(string destination) {
        cout << "\n=== Driving " << model << " ===" << endl;
        start();            // From Engine
        navigate(destination); // From GPS
        cout << "Arrived at destination!" << endl;
    }
};

int main() {
    SmartCar tesla("Tesla Model S", 670, "New York");
    tesla.drive("Boston");

    return 0;
}
```

Output:

```
Engine: 670 HP
GPS initialized at: New York
```

```
SmartCar created: Tesla Model S

=== Driving Tesla Model S ===
Engine started: 670 HP
Navigating from New York to Boston
Arrived at destination!
```

Diamond Problem in Multiple Inheritance:

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

class Device {
protected:
    int powerConsumption;

public:
    Device(int power) : powerConsumption(power) {
        cout << "Device: " << power << "W" << endl;
    }
};

// Problem: Both inherit from Device
class Printer : public Device {
public:
    Printer(int power) : Device(power) {}
};

class Scanner : public Device {
public:
    Scanner(int power) : Device(power) {}
};

// This creates two copies of Device!
class AllInOne : public Printer, public Scanner {
public:
    AllInOne(int pPower, int sPower)
        : Printer(pPower), Scanner(sPower) {}
    // Now we have ambiguity!
};
```

```
// Solution: Virtual Inheritance
class DeviceVirtual {
protected:
    int powerConsumption;

public:
    DeviceVirtual(int power) : powerConsumption(power) {
        cout << "Device: " << power << "W" << endl;
    }
};

class PrinterVirtual : virtual public DeviceVirtual {
public:
    PrinterVirtual(int power) : DeviceVirtual(power) {}
};

class ScannerVirtual : virtual public DeviceVirtual {
public:
    ScannerVirtual(int power) : DeviceVirtual(power) {}
};

class AllInOneVirtual : public PrinterVirtual, public ScannerVirtual {
public:
    AllInOneVirtual(int power)
        : DeviceVirtual(power), PrinterVirtual(power), ScannerVirtual(power) {}
    // Now only ONE copy of DeviceVirtual
};

int main() {
    AllInOneVirtual device(50);

    return 0;
}
```

2.4 Hierarchical Inheritance

Definition: Multiple derived classes inherit from a single base class.

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

class Shape {
protected:
    string color;

public:
    Shape(string c) : color(c) {}

    void displayColor() {
        cout << "Color: " << color << endl;
    }

    virtual double getArea() = 0;
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(string c, double r) : Shape(c), radius(r) {}

    double getArea() override {
        return 3.14159 * radius * radius;
    }

    void display() {
        cout << "Circle - ";
        displayColor();
        cout << "Radius: " << radius << endl;
        cout << "Area: " << getArea() << endl;
    }
};

class Rectangle : public Shape {
private:
```

```
    double width, height;

public:
    Rectangle(string c, double w, double h)
        : Shape(c), width(w), height(h) {}

    double getArea() override {
        return width * height;
    }

    void display() {
        cout << "Rectangle - ";
        displayColor();
        cout << "Width: " << width << ", Height: " << height << endl;
        cout << "Area: " << getArea() << endl;
    }
};

class Triangle : public Shape {
private:
    double base, height;

public:
    Triangle(string c, double b, double h)
        : Shape(c), base(b), height(h) {}

    double getArea() override {
        return 0.5 * base * height;
    }

    void display() {
        cout << "Triangle - ";
        displayColor();
        cout << "Base: " << base << ", Height: " << height << endl;
        cout << "Area: " << getArea() << endl;
    }
};

int main() {
    Circle circle("Red", 5.0);
```

```
    Rectangle rect("Blue", 4.0, 6.0);
    Triangle tri("Green", 8.0, 5.0);

    circle.display();
    cout << endl;
    rect.display();
    cout << endl;
    tri.display();

    return 0;
}
```

2.5 Method Overriding

Definition: Redefining a base class method in a derived class.

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

class Account {
protected:
    string accountNumber;
    double balance;

public:
    Account(string acc, double bal)
        : accountNumber(acc), balance(bal) {}

    // Method to be overridden
    virtual void withdraw(double amount) {
        if (amount <= balance) {
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
        } else {
            cout << "Insufficient funds!" << endl;
        }
    }

    virtual void displayInfo() {
```

```
        cout << "Account: " << accountNumber << endl;
        cout << "Balance: $" << balance << endl;
    }

    double getBalance() { return balance; }
};

class SavingsAccount : public Account {
private:
    double minimumBalance;

public:
    SavingsAccount(string acc, double bal, double minBal)
        : Account(acc, bal), minimumBalance(minBal) {}

    // Override withdraw with additional constraint
    void withdraw(double amount) override {
        if (balance - amount >= minimumBalance) {
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
        } else {
            cout << "Cannot withdraw: Minimum balance requirement!" << endl;
            cout << "Minimum balance: $" << minimumBalance << endl;
        }
    }

    void displayInfo() override {
        Account::displayInfo();  // Call base class method
        cout << "Minimum Balance: $" << minimumBalance << endl;
        cout << "Account Type: Savings" << endl;
    }
};

class CheckingAccount : public Account {
private:
    double overdraftLimit;

public:
    CheckingAccount(string acc, double bal, double overdraft)
        : Account(acc, bal), overdraftLimit(overdraft) {}
```

```
    // Override withdraw with overdraft feature
    void withdraw(double amount) override {
        if (balance + overdraftLimit >= amount) {
            balance -= amount;
            cout << "Withdrawn: $" << amount << endl;
            if (balance < 0) {
                cout << "Warning: Overdraft used! Balance: $" << balance << endl;
            }
        } else {
            cout << "Cannot withdraw: Exceeds overdraft limit!" << endl;
        }
    }

    void displayInfo() override {
        Account::displayInfo();
        cout << "Overdraft Limit: $" << overdraftLimit << endl;
        cout << "Account Type: Checking" << endl;
    }
};

int main() {
    SavingsAccount savings("SA001", 5000, 1000);
    CheckingAccount checking("CA001", 2000, 500);

    cout << "=== Savings Account ===" << endl;
    savings.displayInfo();
    cout << "\nTrying to withdraw $4500..." << endl;
    savings.withdraw(4500);  // Should fail (below minimum)
    cout << "\nTrying to withdraw $3000..." << endl;
    savings.withdraw(3000);  // Should succeed

    cout << "\n=== Checking Account ===" << endl;
    checking.displayInfo();
    cout << "\nTrying to withdraw $2300..." << endl;
    checking.withdraw(2300);  // Should succeed with overdraft

    return 0;
}
```

Revision #1
Created 2025-11-17 14:29:28 UTC by DS
Updated 2025-11-17 14:29:55 UTC by DS