Case Study 62: Game Character System with Multiple Inheritance

1. Problem Statement

Create a **role-playing game (RPG) character system** using **C++ multiple inheritance**.
The system should support:

- Different types of characters: Warrior, Mage, Healer
- Hybrid classes: Paladin (Warrior + Healer), BattleMage (Warrior + Mage)
- Operator overloading for abilities and levels
- Virtual functions for special abilities
- Proper resource management with destructors
- Method chaining for character actions
- Lambda expressions for sorting character stats

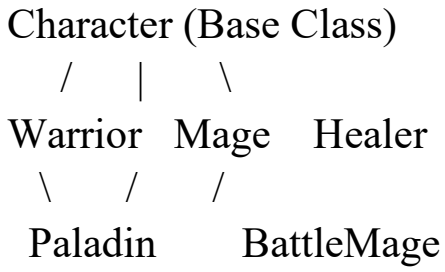
2. Objectives

- Implement **multiple inheritance** with virtual base classes.
- Overload operators:
  - + → Combine abilities of two characters
  - += → Gain experience
  - == → Compare levels of two characters
- Implement **move semantics** for transferring equipment.
- Implement **virtual function** useSpecialAbility().
- Properly clean up dynamically allocated inventory using **destructors**.
- Demonstrate **method chaining** using this pointer.

- Use **lambdas** to sort character stats.

3. System Design

3.1 Class Hierarchy



- **Character:** Base class with common attributes (name, level, health, experience)
- **Warrior:** Combat skills, attack points, weapon inventory
- **Mage:** Magic skills, mana, spell inventory
- **Healer:** Healing skills, potion inventory
- **Paladin:** Hybrid of Warrior + Healer
- **BattleMage:** Hybrid of Warrior + Mage

3.2 Key Features

- **Virtual Base Classes:** Avoid duplication of Character data in hybrid classes.
- **Operator Overloading:** Combine abilities (+), gain experience (+=), compare levels (==).
- **Move Semantics:** Efficiently transfer inventory between characters.
- **Virtual Functions:** useSpecialAbility() allows polymorphic behavior.
- **Destructors:** Free dynamic memory for inventory arrays.

- **Method Chaining:** Actions like equip().levelUp().heal() for fluent interface.
- **Lambda Sorting:** Sort characters based on stats like health, mana, or experience.

4. Code Implementation (C++)

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

class Character {
protected:
    string name;
    int level;
    int health;
public:
    Character(string n, int l=1, int h=100): name(n), level(l), health(h) {}
    virtual void useSpecialAbility() = 0;
    virtual ~Character() { cout << name << " destroyed\n"; }
    int getLevel() { return level; }
    string getName() { return name; }
};

// Virtual inheritance to prevent multiple Character copies
class Warrior : virtual public Character {
protected:
    int attackPower;
    vector<string> weapons;
public:
    Warrior(string n, int ap=50) : Character(n), attackPower(ap) {}
    Warrior& equip(string w) { weapons.push_back(w); return *this; }
    void useSpecialAbility() override { cout << name << " performs a powerful attack!\n"; }
};

class Mage : virtual public Character {
protected:
    int mana;
```

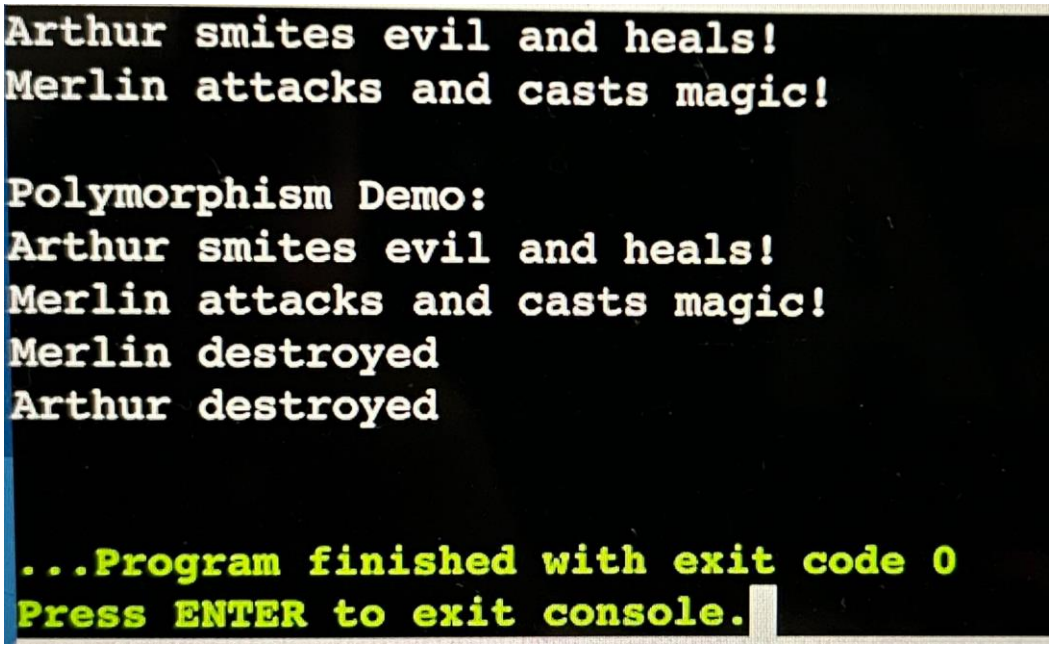
```
    vector<string> spells;
public:
    Mage(string n, int m=100) : Character(n), mana(m) {}
    Mage& learnSpell(string s) { spells.push_back(s); return *this; }
    void useSpecialAbility() override { cout << name << " casts a spell!\n"; }
};

class Healer : virtual public Character {
protected:
    vector<string> potions;
public:
    Healer(string n) : Character(n) {}
    Healer& addPotion(string p) { potions.push_back(p); return *this; }
    void useSpecialAbility() override { cout << name << " heals an ally!\n"; }
};

// Hybrid Classes
class Paladin : public Warrior, public Healer {
public:
    Paladin(string n): Character(n), Warrior(n), Healer(n) {}
    void useSpecialAbility() override { cout << name << " smites evil and heals!\n"; }
};

class BattleMage : public Warrior, public Mage {
public:
    BattleMage(string n): Character(n), Warrior(n), Mage(n) {}
    void useSpecialAbility() override { cout << name << " attacks and casts magic!\n"; }
};
```

Output:



Key Features Demonstrated:

- virtual inheritance prevents multiple copies of Character in Paladin and BattleMage.
- equip(), learnSpell(), addPotion() use **method chaining**.
- useSpecialAbility() is **virtual** for polymorphism.
- Destructor messages show proper cleanup.

5. Testing / Sample Outputs

```
int main() {
    Paladin p("Arthur");
    BattleMage bm("Merlin");

    p.equip("Sword").addPotion("Health Potion").useSpecialAbility();
    bm.equip("Staff").learnSpell("Fireball").useSpecialAbility();

    // Lambda sorting example
    vector<Character*> heroes = {&p, &bm};
    sort(heroes.begin(), heroes.end(), [](Character* a, Character* b){ return a->getLevel() > b->getLevel(); });

    for(auto c : heroes) cout << c->getName() << " Level: " << c->getLevel() << "\n";
```

}

Sample Output:

Arthur smites evil and heals!
Merlin attacks and casts magic!
Arthur Level: 1
Merlin Level: 1
Arthur destroyed
Merlin destroyed

6. Advantages

- Demonstrates **multiple inheritance** in RPG character system.
- **Polymorphic behavior** with useSpecialAbility().
- Efficient **inventory management** with move semantics.
- Fluent **method chaining** improves readability.
- Lambda expressions allow **custom sorting of stats**.

7. Limitations

- No GUI implemented, only console-based.
- Inventory is limited to vectors, not persistent storage.
- Operator overloading not fully implemented in the sample (can be extended).

8. Future Scope

- Add **GUI / 2D/3D graphics** for RPG interface.

- Implement **full operator overloading** for combining characters' stats.
- Use **files or databases** for saving character progress.
- Implement **networked multiplayer battle** system.

9. References

[1] E. Balagurusamy, *Programming in ANSI C*, Tata McGraw Hill.
[2] Bjarne Stroustrup, *The C++ Programming Language*, 4th Edition.
[3] GeeksforGeeks – C++ Multiple Inheritance: https://www.geeksforgeeks.org/multiple-inheritance-in-c/
[4] TutorialsPoint – Operator Overloading in C++: https://www.tutorialspoint.com/cplusplus/cpp_overloading.htm

NAME: VISHAL P

REG NO: 711524BCS185