# 4. Practical Applications

## 4.1 Complete Example: Drawing Application

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

// Abstract base class
class Shape {
protected:
    string color;
    double x, y;  // Position

public:
    Shape(string c, double px, double py) : color(c), x(px), y(py) {}

    // Pure virtual functions
    virtual double getArea() = 0;
    virtual double getPerimeter() = 0;
    virtual void draw() = 0;
    virtual string getType() = 0;

    // Concrete functions
    void move(double dx, double dy) {
        x += dx;
        y += dy;
        cout << getType() << " moved to (" << x << ", " << y << ")" << endl;
    }

    void setColor(string c) {
        color = c;
        cout << getType() << " color changed to " << color << endl;
    }
```

```
    string getColor() { return color; }

    virtual void displayInfo() {
        cout << getType() << " at (" << x << ", " << y << ")" << endl;
        cout << "Color: " << color << endl;
        cout << "Area: " << getArea() << endl;
        cout << "Perimeter: " << getPerimeter() << endl;
    }

    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

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

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

    double getPerimeter() override {
        return 2 * 3.14159 * radius;
    }

    void draw() override {
        cout << "Drawing " << color << " circle at (" << x << ", " << y
             << ") with radius " << radius << endl;
    }

    string getType() override {
        return "Circle";
    }

    void displayInfo() override {
        Shape::displayInfo();
```

```
        cout << "Radius: " << radius << endl;
    }
};

class Rectangle : public Shape {
private:
    double width, height;

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

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

    double getPerimeter() override {
        return 2 * (width + height);
    }

    void draw() override {
        cout << "Drawing " << color << " rectangle at (" << x << ", " << y
             << ") with width " << width << " and height " << height << endl;
    }

    string getType() override {
        return "Rectangle";
    }

    void displayInfo() override {
        Shape::displayInfo();
        cout << "Width: " << width << ", Height: " << height << endl;
    }
};

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

public:
```

```
    Triangle(string c, double px, double py, double b, double h)
        : Shape(c, px, py), base(b), height(h) {}

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

    double getPerimeter() override {
        // Simplified: assumes isosceles triangle
        double side = sqrt((base/2)*(base/2) + height*height);
        return base + 2*side;
    }

    void draw() override {
        cout << "Drawing " << color << " triangle at (" << x << ", " << y
             << ") with base " << base << " and height " << height << endl;
    }

    string getType() override {
        return "Triangle";
    }

    void displayInfo() override {
        Shape::displayInfo();
        cout << "Base: " << base << ", Height: " << height << endl;
    }
};

// Canvas class to manage shapes
class Canvas {
private:
    vector<Shape*> shapes;

public:
    void addShape(Shape* shape) {
        shapes.push_back(shape);
        cout << shape->getType() << " added to canvas" << endl;
    }

    void drawAll() {
```

```
        cout << "\n=== Drawing All Shapes ===" << endl;
        for (Shape* shape : shapes) {
            shape->draw();
        }
    }

    void displayAllInfo() {
        cout << "\n=== All Shapes Information ===" << endl;
        for (size_t i = 0; i < shapes.size(); i++) {
            cout << "\nShape " << (i+1) << ":" << endl;
            shapes[i]->displayInfo();
            cout << "----------------------------------" << endl;
        }
    }

    double getTotalArea() {
        double total = 0;
        for (Shape* shape : shapes) {
            total += shape->getArea();
        }
        return total;
    }

    void removeShape(int index) {
        if (index >= 0 && index < shapes.size()) {
            delete shapes[index];
            shapes.erase(shapes.begin() + index);
            cout << "Shape removed" << endl;
        }
    }

    ~Canvas() {
        for (Shape* shape : shapes) {
            delete shape;
        }
    }
};

int main() {
    Canvas canvas;
```

```
    cout << "=== Creating Shapes ===" << endl;
    canvas.addShape(new Circle("Red", 10, 10, 5));
    canvas.addShape(new Rectangle("Blue", 20, 20, 10, 8));
    canvas.addShape(new Triangle("Green", 30, 30, 6, 4));
    canvas.addShape(new Circle("Yellow", 40, 40, 7));

    canvas.drawAll();
    canvas.displayAllInfo();

    cout << "\nTotal area of all shapes: " << canvas.getTotalArea() << endl;

    return 0;
}
```

# 4.2 Example: Game Character System

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

// Abstract base class
class GameCharacter {
protected:
    string name;
    int health;
    int maxHealth;
    int attackPower;

public:
    GameCharacter(string n, int hp, int atk)
        : name(n), health(hp), maxHealth(hp), attackPower(atk) {}

    // Pure virtual functions
    virtual void attack(GameCharacter* target) = 0;
    virtual void specialAbility() = 0;
    virtual string getClass() = 0;
```

```
    // Concrete functions
    void takeDamage(int damage) {
        health -= damage;
        if (health < 0) health = 0;
        cout << name << " takes " << damage << " damage! Health: " << health << endl;

        if (health == 0) {
            cout << name << " has been defeated!" << endl;
        }
    }

    void heal(int amount) {
        health += amount;
        if (health > maxHealth) health = maxHealth;
        cout << name << " heals " << amount << " HP! Health: " << health << endl;
    }

    bool isAlive() {
        return health > 0;
    }

    void displayStatus() {
        cout << name << " (" << getClass() << ")" << endl;
        cout << "Health: " << health << "/" << maxHealth << endl;
        cout << "Attack Power: " << attackPower << endl;
    }

    string getName() { return name; }
    int getAttackPower() { return attackPower; }

    virtual ~GameCharacter() {}
};

class Warrior : public GameCharacter {
private:
    int armor;

public:
    Warrior(string n) : GameCharacter(n, 150, 25), armor(10) {}
```

```
    void attack(GameCharacter* target) override {
        cout << name << " swings sword at " << target->getName() << "!" << endl;
        target->takeDamage(attackPower);
    }

    void specialAbility() override {
        cout << name << " uses Shield Bash!" << endl;
        cout << "Defense increased temporarily!" << endl;
        armor += 5;
    }

    string getClass() override {
        return "Warrior";
    }

    void takeDamage(int damage) {
        int reducedDamage = damage - armor;
        if (reducedDamage < 0) reducedDamage = 0;
        cout << name << "'s armor blocks " << armor << " damage!" << endl;
        GameCharacter::takeDamage(reducedDamage);
    }
};

class Mage : public GameCharacter {
private:
    int mana;

public:
    Mage(string n) : GameCharacter(n, 80, 35), mana(100) {}

    void attack(GameCharacter* target) override {
        if (mana >= 10) {
            cout << name << " casts Fireball at " << target->getName() << "!" << endl;
            target->takeDamage(attackPower);
            mana -= 10;
        } else {
            cout << name << " is out of mana!" << endl;
        }
    }
```

```
    void specialAbility() override {
        if (mana >= 30) {
            cout << name << " casts Meteor Storm!" << endl;
            cout << "Massive area damage!" << endl;
            mana -= 30;
        } else {
            cout << name << " doesn't have enough mana!" << endl;
        }
    }

    string getClass() override {
        return "Mage";
    }

    void displayStatus() {
        GameCharacter::displayStatus();
        cout << "Mana: " << mana << endl;
    }
};

class Archer : public GameCharacter {
private:
    int arrows;

public:
    Archer(string n) : GameCharacter(n, 100, 30), arrows(50) {}

    void attack(GameCharacter* target) override {
        if (arrows > 0) {
            cout << name << " shoots arrow at " << target->getName() << "!" << endl;
            target->takeDamage(attackPower);
            arrows--;
        } else {
            cout << name << " is out of arrows!" << endl;
        }
    }

    void specialAbility() override {
        if (arrows >= 5) {
            cout << name << " uses Multi-Shot!" << endl;
```

```
            cout << "Fires multiple arrows!" << endl;
            arrows -= 5;
        } else {
            cout << name << " doesn't have enough arrows!" << endl;
        }
    }

    string getClass() override {
        return "Archer";
    }

    void displayStatus() {
        GameCharacter::displayStatus();
        cout << "Arrows: " << arrows << endl;
    }
};

int main() {
    cout << "=== Character Creation ===" << endl;
    vector<GameCharacter*> party;

    party.push_back(new Warrior("Thorin"));
    party.push_back(new Mage("Gandalf"));
    party.push_back(new Archer("Legolas"));

    cout << "\n=== Party Status ===" << endl;
    for (GameCharacter* character : party) {
        character->displayStatus();
        cout << endl;
    }

    cout << "=== Battle Simulation ===" << endl;
    GameCharacter* enemy = new Warrior("Orc");
    cout << "\nEnemy appears: ";
    enemy->displayStatus();

    cout << "\n--- Round 1 ---" << endl;
    party[0]->attack(enemy);
    party[1]->attack(enemy);
    party[2]->attack(enemy);
```

```
    cout << "\n--- Round 2 ---" << endl;
    party[0]->specialAbility();
    party[1]->specialAbility();
    party[2]->specialAbility();

    // Cleanup
    for (GameCharacter* character : party) {
        delete character;
    }
    delete enemy;

    return 0;
}
```

# 4.3 Best Practices

### 1. Always Use Virtual Destructors in Base Classes

```
class Base {
public:
    virtual ~Base() {}  // CRITICAL!
};
```

### 2. Use `override` Keyword

```
class Derived : public Base {
public:
    void func() override {  // Compiler checks
        // implementation
    }
};
```

### 3. Use Abstract Classes for Interfaces

```
class IDrawable {
public:
    virtual void draw() = 0;
    virtual ~IDrawable() {}
};
```

### 4. Prefer References or Pointers for Polymorphism

```
// GOOD: Using pointer or reference
void process(Shape* shape) {
    shape->draw();
}

void process(Shape& shape) {
    shape.draw();
}

// BAD: Pass by value causes slicing
void process(Shape shape) {  // DON'T DO THIS
    shape.draw();
}
```

### 5. Check for Null Pointers

```
Shape* shape = findShape(id);
if (shape != nullptr) {
    shape->draw();
}
```

---

Revision #1
Created 2025-11-25 03:36:23 UTC by DS
Updated 2025-11-25 03:37:47 UTC by DS