Module 10 : OOP - Polymorphism

After completing this module, students are expected to:

- Understand the concept of polymorphism in OOP

- Implement compile-time polymorphism (function and operator overloading)

- Implement runtime polymorphism (virtual functions)

- Use abstract classes and pure virtual functions

- Apply polymorphism for flexible and maintainable code

1. Basic Concepts of Polymorphism

1.1 What is Polymorphism?

Polymorphism means "many forms" - the ability of objects to take on multiple forms or behave differently based on their type.

Real-World Analogy: Think of a smartphone's "share" button:

- Share a photo → Opens image sharing options
- Share a document → Opens document sharing options
- Share a location → Opens map sharing options
- Same button, different behavior based on what you're sharing

Types of Polymorphism in C++:

1. **Compile-Time Polymorphism** (Static Binding)
   - Function Overloading
   - Operator Overloading
2. **Runtime Polymorphism** (Dynamic Binding)
   - Virtual Functions
   - Abstract Classes

Benefits of Polymorphism:

- **Flexibility**: Write code that works with different types
- **Extensibility**: Add new types without changing existing code
- **Maintainability**: Reduce code duplication
- **Abstraction**: Hide implementation details

1.2 Compile-Time vs Runtime Polymorphism

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

// COMPILE-TIME POLYMORPHISM: Function Overloading
class Calculator {
public:
```

```
    // Same function name, different parameters
    int add(int a, int b) {
        return a + b;
    }

    double add(double a, double b) {
        return a + b;
    }

    int add(int a, int b, int c) {
        return a + b + c;
    }
};

// RUNTIME POLYMORPHISM: Virtual Functions
class Shape {
public:
    virtual void draw() {
        cout << "Drawing a shape" << endl;
    }

    virtual ~Shape() {}
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing a circle" << endl;
    }
};

class Rectangle : public Shape {
public:
    void draw() override {
        cout << "Drawing a rectangle" << endl;
    }
};

int main() {
    cout << "=== Compile-Time Polymorphism ===" << endl;
```

```
    Calculator calc;
    cout << "add(5, 3) = " << calc.add(5, 3) << endl;
    cout << "add(5.5, 3.2) = " << calc.add(5.5, 3.2) << endl;
    cout << "add(1, 2, 3) = " << calc.add(1, 2, 3) << endl;

    cout << "\n=== Runtime Polymorphism ===" << endl;
    Shape* shape1 = new Circle();
    Shape* shape2 = new Rectangle();

    shape1->draw();  // Calls Circle::draw()
    shape2->draw();  // Calls Rectangle::draw()

    delete shape1;
    delete shape2;

    return 0;
}
```

Output:

```
=== Compile-Time Polymorphism ===
add(5, 3) = 8
add(5.5, 3.2) = 8.7
add(1, 2, 3) = 6

=== Runtime Polymorphism ===
Drawing a circle
Drawing a rectangle
```

2. Compile-Time Polymorphism

2.1 Function Overloading

Definition: Multiple functions with the same name but different parameters.

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

class Printer {
public:
    // Overloaded print functions
    void print(int value) {
        cout << "Printing integer: " << value << endl;
    }

    void print(double value) {
        cout << "Printing double: " << value << endl;
    }

    void print(string value) {
        cout << "Printing string: " << value << endl;
    }

    void print(int value, int times) {
        cout << "Printing " << value << " for " << times << " times: ";
        for (int i = 0; i < times; i++) {
            cout << value << " ";
        }
        cout << endl;
    }
};

int main() {
    Printer p;
```

```
    p.print(42);
    p.print(3.14);
    p.print("Hello, World!");
    p.print(7, 3);

    return 0;
}
```

Rules for Function Overloading:

1. Functions must have different parameter lists
2. Return type alone is NOT enough to differentiate
3. Parameter types, number, or order must differ

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

class Example {
public:
    // VALID overloads
    void func(int x) { cout << "int: " << x << endl; }
    void func(double x) { cout << "double: " << x << endl; }
    void func(int x, int y) { cout << "two ints: " << x << ", " << y << endl; }

    // INVALID: Only return type differs
    // int func(int x) { return x; }  // ERROR!

    // VALID: Different parameter order
    void process(int x, double y) { cout << "int, double" << endl; }
    void process(double x, int y) { cout << "double, int" << endl; }
};

int main() {
    Example ex;

    ex.func(10);
    ex.func(3.14);
    ex.func(5, 7);

    ex.process(5, 3.14);
```

```
    ex.process(3.14, 5);

    return 0;
}
```

2.2 Operator Overloading

Definition: Redefining operators to work with user-defined types.

Basic Syntax:

```
return_type operator symbol (parameters) {
    // implementation
}
```

Example: Complex Number Class

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

class Complex {
private:
    double real;
    double imag;

public:
    Complex(double r = 0, double i = 0) : real(r), imag(i) {}

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

    // Overload - operator
    Complex operator-(const Complex& other) {
        return Complex(real - other.real, imag - other.imag);
    }

    // Overload * operator
    Complex operator*(const Complex& other) {
```

```
        return Complex(
            real * other.real - imag * other.imag,
            real * other.imag + imag * other.real
        );
    }

    // Overload == operator
    bool operator==(const Complex& other) {
        return (real == other.real && imag == other.imag);
    }

    // Overload << operator (friend function)
    friend ostream& operator<<(ostream& out, const Complex& c) {
        out << c.real;
        if (c.imag >= 0)
            out << " + " << c.imag << "i";
        else
            out << " - " << -c.imag << "i";
        return out;
    }

    void display() {
        cout << *this << endl;
    }
};

int main() {
    Complex c1(3, 4);
    Complex c2(1, 2);

    cout << "c1 = " << c1 << endl;
    cout << "c2 = " << c2 << endl;

    Complex c3 = c1 + c2;
    cout << "c1 + c2 = " << c3 << endl;

    Complex c4 = c1 - c2;
    cout << "c1 - c2 = " << c4 << endl;

    Complex c5 = c1 * c2;
```

```
    cout << "c1 * c2 = " << c5 << endl;

    if (c1 == c2)
        cout << "c1 equals c2" << endl;
    else
        cout << "c1 not equals c2" << endl;

    return 0;
}
```

Common Operators to Overload:

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

class Vector2D {
private:
    double x, y;

public:
    Vector2D(double x = 0, double y = 0) : x(x), y(y) {}

    // Arithmetic operators
    Vector2D operator+(const Vector2D& v) {
        return Vector2D(x + v.x, y + v.y);
    }

    Vector2D operator-(const Vector2D& v) {
        return Vector2D(x - v.x, y - v.y);
    }

    Vector2D operator*(double scalar) {
        return Vector2D(x * scalar, y * scalar);
    }

    // Compound assignment operators
    Vector2D& operator+=(const Vector2D& v) {
        x += v.x;
        y += v.y;
```

```
    return *this;
}

// Unary operators
Vector2D operator-() {
    return Vector2D(-x, -y);
}

// Increment/Decrement
Vector2D& operator++() {  // Prefix
    ++x;
    ++y;
    return *this;
}

Vector2D operator++(int) {  // Postfix
    Vector2D temp = *this;
    ++(*this);
    return temp;
}

// Comparison operators
bool operator==(const Vector2D& v) {
    return (x == v.x && y == v.y);
}

bool operator!=(const Vector2D& v) {
    return !(*this == v);
}

// Subscript operator
double& operator[](int index) {
    if (index == 0) return x;
    return y;
}

// Stream operators
friend ostream& operator<<(ostream& out, const Vector2D& v) {
    out << "(" << v.x << ", " << v.y << ")";
    return out;
```

```
    }

    friend istream& operator>>(istream& in, Vector2D& v) {
        in >> v.x >> v.y;
        return in;
    }
};

int main() {
    Vector2D v1(3, 4);
    Vector2D v2(1, 2);

    cout << "v1 = " << v1 << endl;
    cout << "v2 = " << v2 << endl;

    Vector2D v3 = v1 + v2;
    cout << "v1 + v2 = " << v3 << endl;

    Vector2D v4 = v1 * 2;
    cout << "v1 * 2 = " << v4 << endl;

    v1 += v2;
    cout << "v1 after += v2: " << v1 << endl;

    Vector2D v5 = -v1;
    cout << "-v1 = " << v5 << endl;

    ++v2;
    cout << "++v2 = " << v2 << endl;

    cout << "v1[0] = " << v1[0] << ", v1[1] = " << v1[1] << endl;

    return 0;
}
```

2.3 Practical Example: Fraction Class

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

```
class Fraction {
private:
    int numerator;
    int denominator;

    // Helper function to find GCD
    int gcd(int a, int b) {
        if (b == 0) return a;
        return gcd(b, a % b);
    }

    // Simplify the fraction
    void simplify() {
        int g = gcd(abs(numerator), abs(denominator));
        numerator /= g;
        denominator /= g;

        // Keep denominator positive
        if (denominator < 0) {
            numerator = -numerator;
            denominator = -denominator;
        }
    }

public:
    Fraction(int num = 0, int den = 1) : numerator(num), denominator(den) {
        if (denominator == 0) {
            cout << "Error: Denominator cannot be zero!" << endl;
            denominator = 1;
        }
        simplify();
    }

    // Arithmetic operators
    Fraction operator+(const Fraction& f) {
        int num = numerator * f.denominator + f.numerator * denominator;
        int den = denominator * f.denominator;
        return Fraction(num, den);
    }
```

```
Fraction operator-(const Fraction& f) {
    int num = numerator * f.denominator - f.numerator * denominator;
    int den = denominator * f.denominator;
    return Fraction(num, den);
}

Fraction operator*(const Fraction& f) {
    return Fraction(numerator * f.numerator, denominator * f.denominator);
}

Fraction operator/(const Fraction& f) {
    return Fraction(numerator * f.denominator, denominator * f.numerator);
}

// Comparison operators
bool operator==(const Fraction& f) {
    return (numerator == f.numerator && denominator == f.denominator);
}

bool operator<(const Fraction& f) {
    return (numerator * f.denominator < f.numerator * denominator);
}

bool operator>(const Fraction& f) {
    return f < *this;
}

// Stream operators
friend ostream& operator<<(ostream& out, const Fraction& f) {
    if (f.denominator == 1)
        out << f.numerator;
    else
        out << f.numerator << "/" << f.denominator;
    return out;
}

friend istream& operator>>(istream& in, Fraction& f) {
    char slash;
    in >> f.numerator >> slash >> f.denominator;
```

```
        f.simplify();
        return in;
    }
};

int main() {
    Fraction f1(1, 2);  // 1/2
    Fraction f2(3, 4);  // 3/4

    cout << "f1 = " << f1 << endl;
    cout << "f2 = " << f2 << endl;

    Fraction sum = f1 + f2;
    cout << "f1 + f2 = " << sum << endl;

    Fraction diff = f1 - f2;
    cout << "f1 - f2 = " << diff << endl;

    Fraction prod = f1 * f2;
    cout << "f1 * f2 = " << prod << endl;

    Fraction quot = f1 / f2;
    cout << "f1 / f2 = " << quot << endl;

    if (f1 < f2)
        cout << f1 << " is less than " << f2 << endl;
    else
        cout << f1 << " is not less than " << f2 << endl;

    return 0;
}
```

2.4 Stream Operator (Advanced Theory)

The **stream insertion operator (`<<`) is typically overloaded as a friend function** because it **does not logically belong to the class** and **needs access to the class's private data**. Let's break down why.

2.4.1 Reason 1 — The left operand is not the class

When you write:

```
cout << obj;
```

The operator's **left-hand side** is `cout` (an `ostream` object), **not** your class.

So the operator signature must look like:

```
ostream& operator<<(ostream& os, const MyClass& obj);
```

This means:

- It **cannot** be a member of `MyClass` (because then `MyClass` would be the left operand).
- It **must** be a free-standing function.

But this free function still needs to access `obj`'s private data → so you typically declare it as a **friend**.

Example Without Friend

It becomes impossible to access private members:

```
class Point {
private:
    int x, y;   // private
};

ostream& operator<<(ostream& os, const Point& p) {
    os << p.x;   // ERROR — x is private
    return os;
}
```

Because `operator<<` is not a member function, it has **no access** to private members.

2.4.2 Reason 2 — Friend gives access to private data

To solve that, you declare it as a friend:

```
class Point {
private:
    int x, y;

public:
    Point(int x, int y) : x(x), y(y) {}
```

```
    friend ostream& operator<<(ostream& os, const Point& p);
};
```

Now the non-member function has access to private members.

2.4.3 Reason 3 — Makes syntax natural

Overloading as a friend function allows this natural C++ syntax:

```
cout << obj1 << obj2;
```

If it were a member function, you'd need to write something like:

```
obj << cout;   // awkward and reversed operands
```

This is not how streams are meant to be used.

2.4.4 Reason 4 — Works with chaining

Friend/global versions support chaining:

```
cout << a << b << c;
```

Which relies on:

```
return os;
```

So the next `<<` works.

2.4.5 Stream Operator Summary

Reason	Explanation
Left operand is `ostream`	So cannot be a member of your class
Needs access to private data	So it's declared as friend
Natural syntax	Allows `cout << obj` instead of reversed order
Supports chaining	Returning `ostream&` works smoothly

3. Runtime Polymorphism

3.1 Virtual Functions

Definition: Functions that can be overridden in derived classes and are resolved at runtime.

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

class Animal {
protected:
    string name;

public:
    Animal(string n) : name(n) {}

    // Virtual function
    virtual void makeSound() {
        cout << name << " makes a sound" << endl;
    }

    // Non-virtual function
    void eat() {
        cout << name << " is eating" << endl;
    }

    virtual ~Animal() {}
};

class Dog : public Animal {
public:
    Dog(string n) : Animal(n) {}

    void makeSound() override {
        cout << name << " barks: Woof! Woof!" << endl;
    }
```

```
};

class Cat : public Animal {
public:
    Cat(string n) : Animal(n) {}

    void makeSound() override {
        cout << name << " meows: Meow! Meow!" << endl;
    }
};

int main() {
    // Runtime polymorphism with pointers
    Animal* animal1 = new Dog("Buddy");
    Animal* animal2 = new Cat("Whiskers");

    cout << "=== Using Pointers ===" << endl;
    animal1->makeSound();  // Calls Dog::makeSound()
    animal2->makeSound();  // Calls Cat::makeSound()

    animal1->eat();  // Calls Animal::eat() (non-virtual)
    animal2->eat();

    delete animal1;
    delete animal2;

    // Runtime polymorphism with references
    cout << "\n=== Using References ===" << endl;
    Dog dog("Max");
    Cat cat("Luna");

    Animal& ref1 = dog;
    Animal& ref2 = cat;

    ref1.makeSound();  // Calls Dog::makeSound()
    ref2.makeSound();  // Calls Cat::makeSound()

    return 0;
}
```

How Virtual Functions Work:

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

class Base {
public:
    virtual void func1() { cout << "Base::func1()" << endl; }
    virtual void func2() { cout << "Base::func2()" << endl; }
    void func3() { cout << "Base::func3()" << endl; }
};

class Derived : public Base {
public:
    void func1() override { cout << "Derived::func1()" << endl; }
    // func2() not overridden - uses Base version
    void func3() { cout << "Derived::func3()" << endl; }
};

int main() {
    Base* ptr = new Derived();

    ptr->func1();  // Derived::func1() - virtual, overridden
    ptr->func2();  // Base::func2() - virtual, not overridden
    ptr->func3();  // Base::func3() - not virtual

    delete ptr;
    return 0;
}
```

3.2 Abstract Classes and Pure Virtual Functions

Definition: A class with at least one pure virtual function. Cannot be instantiated directly. Or else, it will return a Compile Time Error.

Syntax:

```
virtual return_type function_name() = 0;
```

Example: Payment System

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

// Abstract base class
class Payment {
protected:
    double amount;
    string transactionId;

public:
    Payment(double amt, string id) : amount(amt), transactionId(id) {}

    // Pure virtual functions
    virtual void processPayment() = 0;
    virtual void displayReceipt() = 0;
    virtual string getPaymentMethod() = 0;

    // Concrete function
    void showAmount() {
        cout << "Amount: $" << amount << endl;
    }

    virtual ~Payment() {}
};

class CreditCardPayment : public Payment {
private:
    string cardNumber;
    string cvv;

public:
    CreditCardPayment(double amt, string id, string card, string c)
        : Payment(amt, id), cardNumber(card), cvv(c) {}

    void processPayment() override {
        cout << "Processing credit card payment..." << endl;
        cout << "Card: ****" << cardNumber.substr(cardNumber.length() - 4) << endl;
        cout << "Payment of $" << amount << " approved!" << endl;
```

```
    }

    void displayReceipt() override {
        cout << "\n=== Credit Card Receipt ===" << endl;
        cout << "Transaction ID: " << transactionId << endl;
        showAmount();
        cout << "Payment Method: " << getPaymentMethod() << endl;
        cout << "Status: Completed" << endl;
    }

    string getPaymentMethod() override {
        return "Credit Card";
    }
};

class PayPalPayment : public Payment {
private:
    string email;

public:
    PayPalPayment(double amt, string id, string e)
        : Payment(amt, id), email(e) {}

    void processPayment() override {
        cout << "Processing PayPal payment..." << endl;
        cout << "Account: " << email << endl;
        cout << "Payment of $" << amount << " approved!" << endl;
    }

    void displayReceipt() override {
        cout << "\n=== PayPal Receipt ===" << endl;
        cout << "Transaction ID: " << transactionId << endl;
        showAmount();
        cout << "PayPal Email: " << email << endl;
        cout << "Payment Method: " << getPaymentMethod() << endl;
        cout << "Status: Completed" << endl;
    }

    string getPaymentMethod() override {
        return "PayPal";
```

```
    }
};

class BankTransferPayment : public Payment {
private:
    string accountNumber;
    string bankName;

public:
    BankTransferPayment(double amt, string id, string acc, string bank)
        : Payment(amt, id), accountNumber(acc), bankName(bank) {}

    void processPayment() override {
        cout << "Processing bank transfer..." << endl;
        cout << "Bank: " << bankName << endl;
        cout << "Account: ****" << accountNumber.substr(accountNumber.length() - 4) << endl;
        cout << "Payment of $" << amount << " initiated!" << endl;
        cout << "Note: Transfer may take 1-3 business days" << endl;
    }

    void displayReceipt() override {
        cout << "\n=== Bank Transfer Receipt ===" << endl;
        cout << "Transaction ID: " << transactionId << endl;
        showAmount();
        cout << "Bank: " << bankName << endl;
        cout << "Payment Method: " << getPaymentMethod() << endl;
        cout << "Status: Pending" << endl;
    }

    string getPaymentMethod() override {
        return "Bank Transfer";
    }
};

// Payment processor using polymorphism
void executePayment(Payment* payment) {
    payment->processPayment();
    payment->displayReceipt();
}
```

```
int main() {
    // Cannot instantiate abstract class
    // Payment* p = new Payment(100, "TXN001");  // ERROR!

    Payment* payment1 = new CreditCardPayment(250.50, "TXN001", "1234567890123456", "123");
    Payment* payment2 = new PayPalPayment(89.99, "TXN002", "user@example.com");
    Payment* payment3 = new BankTransferPayment(1500.00, "TXN003", "9876543210", "Bank of
America");

    cout << "=== Payment 1 ===" << endl;
    executePayment(payment1);

    cout << "\n=== Payment 2 ===" << endl;
    executePayment(payment2);

    cout << "\n=== Payment 3 ===" << endl;
    executePayment(payment3);

    delete payment1;
    delete payment2;
    delete payment3;

    return 0;
}
```

3.3 Polymorphism with Arrays

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

// Abstract base class
class Employee {
protected:
    string name;
    int id;

public:
```

```
    Employee(string n, int i) : name(n), id(i) {}

    virtual double calculateSalary() = 0;
    virtual void displayInfo() = 0;
    virtual string getType() = 0;

    string getName() { return name; }

    virtual ~Employee() {}
};

class FullTimeEmployee : public Employee {
private:
    double monthlySalary;

public:
    FullTimeEmployee(string n, int i, double salary)
        : Employee(n, i), monthlySalary(salary) {}

    double calculateSalary() override {
        return monthlySalary;
    }

    void displayInfo() override {
        cout << "Full-Time: " << name << " (ID: " << id << ")" << endl;
        cout << "Monthly Salary: $" << monthlySalary << endl;
    }

    string getType() override {
        return "Full-Time";
    }
};

class PartTimeEmployee : public Employee {
private:
    double hourlyRate;
    int hoursWorked;

public:
    PartTimeEmployee(string n, int i, double rate, int hours)
```

```
        : Employee(n, i), hourlyRate(rate), hoursWorked(hours) {}

    double calculateSalary() override {
        return hourlyRate * hoursWorked;
    }

    void displayInfo() override {
        cout << "Part-Time: " << name << " (ID: " << id << ")" << endl;
        cout << "Hourly Rate: $" << hourlyRate << ", Hours: " << hoursWorked << endl;
        cout << "Total Pay: $" << calculateSalary() << endl;
    }

    string getType() override {
        return "Part-Time";
    }
};

class Contractor : public Employee {
private:
    double projectFee;
    int projectsCompleted;

public:
    Contractor(string n, int i, double fee, int projects)
        : Employee(n, i), projectFee(fee), projectsCompleted(projects) {}

    double calculateSalary() override {
        return projectFee * projectsCompleted;
    }

    void displayInfo() override {
        cout << "Contractor: " << name << " (ID: " << id << ")" << endl;
        cout << "Project Fee: $" << projectFee << ", Projects: " << projectsCompleted << endl;
        cout << "Total Earnings: $" << calculateSalary() << endl;
    }

    string getType() override {
        return "Contractor";
    }
};
```

```
int main() {
    // Polymorphic array
    vector<Employee*> employees;

    employees.push_back(new FullTimeEmployee("Alice Johnson", 101, 5000));
    employees.push_back(new PartTimeEmployee("Bob Smith", 102, 25, 80));
    employees.push_back(new Contractor("Carol White", 103, 3000, 4));
    employees.push_back(new FullTimeEmployee("David Brown", 104, 6000));
    employees.push_back(new PartTimeEmployee("Eve Davis", 105, 30, 60));

    cout << "=== All Employees ===" << endl;
    double totalPayroll = 0;

    for (Employee* emp : employees) {
        emp->displayInfo();
        totalPayroll += emp->calculateSalary();
        cout << "-----------------------------------" << endl;
    }

    cout << "\nTotal Payroll: $" << totalPayroll << endl;

    // Count by type
    int fullTime = 0, partTime = 0, contractors = 0;
    for (Employee* emp : employees) {
        string type = emp->getType();
        if (type == "Full-Time") fullTime++;
        else if (type == "Part-Time") partTime++;
        else if (type == "Contractor") contractors++;
    }

    cout << "\n=== Employee Distribution ===" << endl;
    cout << "Full-Time: " << fullTime << endl;
    cout << "Part-Time: " << partTime << endl;
    cout << "Contractors: " << contractors << endl;

    // Cleanup
    for (Employee* emp : employees) {
        delete emp;
    }
```

```
    return 0;
}
```

3.4 The `override` Keyword

`override` is a contextual keyword in C++ (since C++11) that you place on a virtual function in a derived class to indicate your intention to override a virtual function declared in a base class. It does not change runtime semantics, but it instructs the compiler to verify that a matching virtual function exists in some base class. If no matching base virtual function is found, compilation fails.

Why `override` matters

- It turns silent mistakes (typos, wrong parameter types, wrong cv/ref qualifiers, incorrect exception specifications) into **compile-time errors**.
- It documents intent: future readers of the code see explicitly which functions are intended to participate in dynamic dispatch.
- It prevents subtle bugs where a derived method inadvertently creates a new function instead of overriding the base one.

Rules the compiler checks when you use `override`

- A base class has a function with the same name.
- The base function is `virtual` (or already overrides another virtual).
- The function signatures match exactly (parameter types, value category, cv-qualifiers).
- Return types are either identical or covariant (covariant return types allowed for pointers/references).
- Ref-qualifiers and `noexcept` are considered part of the function type for matching. If any of these checks fail, the compiler issues an error.

Basic example (correct override)

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

class Base {
public:
    virtual void speak() {
        cout << "Base speaking\n";
    }
    virtual ~Base() = default;
```

```
};

class Derived : public Base {
public:
    void speak() override {            // OK: matches Base::speak()
        cout << "Derived speaking\n";
    }
};
```

`Derived::speak()` is guaranteed by the compiler to override `Base::speak()`. If the signature differs, compilation fails.

Common mistakes `override` catches

1. Typo in function name:

```
class Derived : public Base {
public:
    void speek() override { } // error: no base function to override
};
```

2. Signature mismatch (parameter types, cv/ref qualifiers):

```
class Base {
public:
    virtual void f(int) {}
};
class Derived : public Base {
public:
    void f(double) override {} // error: signature does not match
};
```

3. Ref-qualifier mismatch:

```
class Base {
public:
    virtual void g() & {}   // lvalue-qualified
};
class Derived : public Base {
public:
    void g() override {}    // error: missing & qualifier, not matching
```

```
};
```

4. `noexcept` mismatch:

```
class Base {
public:
    virtual void h() noexcept {}
};
class Derived : public Base {
public:
    void h() override {} // error if noexcept mismatch is considered by the compiler
};
```

(Compilers may treat `noexcept` as part of the function type for override checks; using `override` helps catch inconsistencies.)

Covariant return types

Covariant returns are permitted when the return type in the derived override is a pointer or reference to a class derived from the base return type.

```
struct A { virtual ~A() = default; };
struct B : A {};

struct Base {
    virtual A* clone() { return new A; }
};

struct Derived : Base {
    B* clone() override { return new B; } // OK: covariant return type
};
```

`override` still applies; the compiler checks covariance rules.

`override` with `final`

You can combine `override` with `final` to both override a base virtual function and prevent further overrides in later derived classes.

```
struct Base {
    virtual void foo();
```

```
};

struct A : Base {
    void foo() override final; // overrides, and forbids further overrides
};

struct B : A {
    void foo() override; // error: foo() is final in A
};
```

When to use `override` — best practices

- Use `override` on every virtual function in a derived class that is intended to override a base virtual function. This is widely considered good style and recommended by C++ Core Guidelines.
- Prefer `override` even in small codebases; it catches bugs early and documents intent.
- Use `final` together with `override` when you want to block further overriding for design or performance reasons.
- Use `= default` or `= delete` for special member functions as appropriate; those are separate matters but keep interfaces explicit.

Interaction with pure virtual functions and abstract classes

`override` works with pure virtual functions as well:

```
struct Interface {
    virtual void op() = 0;
};

struct Impl : Interface {
    void op() override { /* implementation */ } // required to make Impl concrete
};
```

If a derived class fails to override a pure virtual function, the derived class remains abstract. Marking an implementation with `override` ensures you intended to implement that pure virtual function.

3.5 Virtual Destructors

Why Virtual Destructors are Important:

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

// WITHOUT virtual destructor (WRONG)
class BaseWrong {
public:
    BaseWrong() { cout << "Base constructed" << endl; }
    ~BaseWrong() { cout << "Base destructed" << endl; }
};

class DerivedWrong : public BaseWrong {
private:
    int* data;
public:
    DerivedWrong() {
        data = new int[100];
        cout << "Derived constructed" << endl;
    }
    ~DerivedWrong() {
        delete[] data;
        cout << "Derived destructed" << endl;
    }
};

// WITH virtual destructor (CORRECT)
class BaseCorrect {
public:
    BaseCorrect() { cout << "Base constructed" << endl; }
    virtual ~BaseCorrect() { cout << "Base destructed" << endl; }
};

class DerivedCorrect : public BaseCorrect {
private:
    int* data;
public:
    DerivedCorrect() {
        data = new int[100];
        cout << "Derived constructed" << endl;
    }
    ~DerivedCorrect() override {
```

```
        delete[] data;
        cout << "Derived destructed" << endl;
    }
};

int main() {
    cout << "=== WITHOUT Virtual Destructor (MEMORY LEAK!) ===" << endl;
    BaseWrong* ptr1 = new DerivedWrong();
    delete ptr1;  // Only Base destructor called!

    cout << "\n=== WITH Virtual Destructor (CORRECT) ===" << endl;
    BaseCorrect* ptr2 = new DerivedCorrect();
    delete ptr2;  // Both destructors called!

    return 0;
}
```

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();
}
```