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;
}
```

Revision #3
Created 2025-11-25 03:34:16 UTC by DS
Updated 2025-11-25 04:00:19 UTC by DS