Prerequisites Object-Oriented Programming in C++

Object-Oriented Programming in C++: From Basic to Advanced

Table of Contents

Understanding Programming Paradigms

Programming paradigms are different approaches or styles of programming that provide guidelines on how to structure and organize code. Each paradigm represents a distinct way of thinking about and solving problems through code.

Major programming paradigms include:

1. **Procedural Programming**: Focuses on procedure calls (functions) where code is organized as a sequence of procedures that operate on data.
2. **Object-Oriented Programming**: Organizes code around "objects" which encapsulate data and behavior.
3. **Functional Programming**: Treats computation as the evaluation of mathematical functions, avoiding state changes and mutable data.
4. **Declarative Programming**: Expresses the logic of computation without describing its control flow.
5. **Event-Driven Programming**: Flow of the program is determined by events such as user actions, sensor outputs, or messages from other programs.

Procedural vs. Object-Oriented Programming

Procedural Programming

Procedural programming is based on the concept of procedure calls, where procedures (also known as routines, subroutines, or functions) contain a series of computational steps to be carried out.

Key Characteristics:

- Programs are structured around procedures or functions
- Uses top-down approach (breaking a program into smaller procedures)
- Data and procedures are separate entities
- Procedures operate on data passed to them
- Global data can be accessed by any procedure

Example in C:

```c
// Global data
float balance = 1000.0;

// Function to deposit money
void deposit(float amount) {
    if (amount > 0) {
        balance += amount;
        printf("Deposited: $%.2f\n", amount);
        printf("New Balance: $%.2f\n", balance);
    }
}

// Function to withdraw money
void withdraw(float amount) {
    if (amount > 0 && balance >= amount) {
        balance -= amount;
        printf("Withdrawn: $%.2f\n", amount);
        printf("New Balance: $%.2f\n", balance);
    } else {
        printf("Insufficient funds or invalid amount\n");
    }
```

```
}

int main() {
    deposit(500);
    withdraw(200);
    return 0;
}
```

Object-Oriented Programming

OOP is based on the concept of "objects" which contain data and code. Objects have state (data) and behavior (code) and can interact with each other.

Key Characteristics:

- Programs are organized around objects rather than actions
- Data and methods are encapsulated within objects
- Objects can maintain private state
- Objects interact by sending messages to each other
- Supports inheritance, polymorphism, encapsulation, and abstraction

Example in C++:

```
class BankAccount {
private:
    float balance;

public:
    BankAccount(float initialBalance) {
        balance = initialBalance;
    }

    void deposit(float amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited: $" << amount << std::endl;
            std::cout << "New Balance: $" << balance << std::endl;
        }
    }

    void withdraw(float amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            std::cout << "Withdrawn: $" << amount << std::endl;
            std::cout << "New Balance: $" << balance << std::endl;
        } else {
            std::cout << "Insufficient funds or invalid amount" << std::endl;
        }
    }
};
```

```
int main() {
    BankAccount account(1000);
    account.deposit(500);
    account.withdraw(200);
    return 0;
}
```

Key Concepts Before Learning OOP

Before diving into OOP, it's important to have a solid understanding of these foundational programming concepts:

Understanding Data Types

Data types specify what kind of data can be stored and manipulated within a program.

Primitive Data Types in C++:

- `int`: Integer numbers
- `float`, `double`: Floating-point numbers
- `char`: Single characters
- `bool`: Boolean values (true/false)

Compound Data Types:

- Arrays
- Structures
- Unions
- Enumerations

Example:

```
// Primitive types
int age = 30;
double salary = 50000.50;
char grade = 'A';
bool isEmployed = true;

// Compound types
int scores[5] = {95, 88, 75, 90, 82};

struct Person {
    char name[50];
    int age;
    double salary;
};

struct Person employee = {"John Doe", 30, 50000.50};
```

Control Structures

Control structures determine the flow of execution in a program.

Conditional Statements:

- If-else statements
- Switch statements

Loops:

- For loops
- While loops
- Do-while loops

Example:

```
// Conditional statements
if (score >= 90) {
    grade = 'A';
} else if (score >= 80) {
    grade = 'B';
} else {
    grade = 'C';
}

// Switch statement
switch (option) {
    case 1:
        std::cout << "Option 1 selected";
        break;
    case 2:
        std::cout << "Option 2 selected";
        break;
    default:
        std::cout << "Invalid option";
}

// Loops
for (int i = 0; i < 5; i++) {
    std::cout << scores[i] << std::endl;
}

int i = 0;
while (i < 5) {
    std::cout << scores[i] << std::endl;
    i++;
}
```

Functions and Procedures

Functions are blocks of code designed to perform a particular task. They help in code reusability and modularity.

Function Components:

- Return type
- Function name
- Parameters (optional)
- Function body

Example:

```
// Function definition
int add(int a, int b) {
    return a + b;
}

// Function with no return value (procedure)
void printMessage(std::string message) {
    std::cout << message << std::endl;
}

// Function call
int result = add(5, 3);
printMessage("Hello, World!");
```

Memory Management

Understanding how memory works is crucial, especially in C++ where you have direct control over memory allocation and deallocation.

Stack vs. Heap:

- **Stack**: Automatic memory management, stores local variables
- **Heap**: Dynamic memory management, requires manual allocation and deallocation

Example:

```
// Stack memory (automatically managed)
int stackArray[10];

// Heap memory (manually managed)
int* heapArray = new int[10];  // Allocate memory

// Perform operations with heapArray

delete[] heapArray;  // Deallocate memory
```

Pointers and References

Pointers and references are fundamental concepts in C++ that allow direct manipulation of memory.

Pointer: A variable that stores the memory address of another variable.

Reference: An alias for an existing variable.

Example:

```
int number = 10;

// Pointer
int* ptr = &number;  // ptr holds the address of number
*ptr = 20;  // Changes the value of number to 20

// Reference
int& ref = number;  // ref is an alias for number
ref = 30;  // Changes the value of number to 30
```

Why OOP?

Limitations of Procedural Programming

1. **Data Security**: Global data is accessible to all procedures, increasing the risk of unintended modifications.
2. **Code Reusability**: Limited mechanisms for code reuse apart from functions.
3. **Scalability**: As programs grow larger, maintaining procedural code becomes difficult.
4. **Real-World Modeling**: Procedural programming doesn't naturally model real-world entities.
5. **Code Organization**: As projects grow, organizing code becomes challenging.

Benefits of OOP

1. **Modularity**: Objects are self-contained, making code easier to understand and maintain.
2. **Data Hiding**: Encapsulation protects data from unintended interference.
3. **Code Reusability**: Inheritance allows code reuse and extension.
4. **Flexibility**: Polymorphism enables objects to take multiple forms depending on context.
5. **Real-World Mapping**: OOP naturally models real-world entities and relationships.
6. **Maintainable Code**: The structure of OOP makes large projects more manageable.

Transitioning from Procedural to OOP

When moving from procedural to object-oriented programming, follow these steps:

1. **Identify Objects**: Look for nouns in your problem domain that represent entities.
2. **Identify Attributes**: Determine what data each object should contain.
3. **Identify Methods**: Identify operations that objects can perform or that can be performed on objects.
4. **Establish Relationships**: Determine how objects interact with each other.
5. **Design Classes**: Create class structures based on the identified objects.

Example Transition:

Procedural approach:

```
// Global data
struct Student {
    int id;
    std::string name;
    float gpa;
};

// Functions operating on data
void displayStudent(const Student& s) {
    std::cout << "ID: " << s.id << ", Name: " << s.name << ", GPA: " << s.gpa <<
std::endl;
}

void updateGPA(Student& s, float newGPA) {
    s.gpa = newGPA;
}

int main() {
    Student s1 = {1, "John", 3.5};
    displayStudent(s1);
    updateGPA(s1, 3.8);
    displayStudent(s1);
    return 0;
}
```

Object-oriented approach:

```
class Student {
private:
    int id;
    std::string name;
    float gpa;

public:
    Student(int id, std::string name, float gpa)
        : id(id), name(name), gpa(gpa) {}

    void display() const {
        std::cout << "ID: " << id << ", Name: " << name << ", GPA: " << gpa <<
std::endl;
    }

    void updateGPA(float newGPA) {
        gpa = newGPA;
    }
};

int main() {
```

```
    Student s1(1, "John", 3.5);
    s1.display();
    s1.updateGPA(3.8);
    s1.display();
    return 0;
}
```

Basic Building Blocks for OOP

Before diving deep into OOP concepts like inheritance, polymorphism, etc., understand these fundamental building blocks:

1. Classes and Objects

A **class** is a blueprint or template that defines the characteristics and behaviors of an entity.

An **object** is an instance of a class, representing a specific entity.

Example:

```
// Class definition
class Car {
private:
    std::string brand;
    std::string model;
    int year;

public:
    // Constructor
    Car(std::string b, std::string m, int y)
        : brand(b), model(m), year(y) {}

    // Method
    void displayInfo() {
        std::cout << year << " " << brand << " " << model << std::endl;
    }
};

// Creating objects
Car car1("Toyota", "Corolla", 2020);
Car car2("Honda", "Civic", 2019);

// Using objects
car1.displayInfo();
car2.displayInfo();
```

2. Encapsulation

Encapsulation is the bundling of data and methods that operate on that data within a single unit (class), and restricting access to some of the object's components.

Example:

```
class BankAccount {
private:
    // Private data members
    std::string accountNumber;
    double balance;

public:
    // Public methods to interact with private data
    BankAccount(std::string accNum, double initialBalance)
        : accountNumber(accNum), balance(initialBalance) {}

    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
        }
    }

    bool withdraw(double amount) {
        if (amount > 0 && balance >= amount) {
            balance -= amount;
            return true;
        }
        return false;
    }

    double getBalance() const {
        return balance;
    }
};
```

3. Constructors and Destructors

Constructors initialize objects when they are created. **Destructors** clean up resources when objects are destroyed.

Example:

```
class Resource {
private:
    int* data;

public:
    // Constructor
    Resource() {
        std::cout << "Resource acquired" << std::endl;
        data = new int[100];  // Allocate memory
    }
```

```
    // Destructor
    ~Resource() {
        std::cout << "Resource released" << std::endl;
        delete[] data;  // Release memory
    }
};

// Using the class
void useResource() {
    Resource res;  // Constructor called
    // Use the resource
}  // Destructor called when res goes out of scope
```

4. Access Specifiers

Access specifiers control the visibility and accessibility of class members:

- `private`: Accessible only within the class
- `protected`: Accessible within the class and its derived classes
- `public`: Accessible from anywhere

Example:

```
class Base {
private:
    int privateVar;  // Accessible only within Base class

protected:
    int protectedVar;  // Accessible within Base and derived classes

public:
    int publicVar;  // Accessible from anywhere

    Base() : privateVar(1), protectedVar(2), publicVar(3) {}
};

class Derived : public Base {
public:
    void accessTest() {
        // privateVar = 10;  // Error: Cannot access private member
        protectedVar = 20;  // OK: Can access protected member
        publicVar = 30;     // OK: Can access public member
    }
};
```

Introduction to OOP

Object-Oriented Programming (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. It utilizes several techniques from previously established paradigms, including modularity, polymorphism, and encapsulation.

OOP was developed to increase the reusability and maintainability of code. By creating objects that can work independently, programmers can build programs from working components rather than having to start from scratch each time.

C++ was designed with OOP principles in mind, as an extension of the C language. Bjarne Stroustrup, the creator of C++, wanted to combine the efficiency and flexibility of C with the features of object-oriented languages.

Why OOP?

- **Modularity**: Encapsulation allows objects to be self-contained, making troubleshooting and collaborative development easier.
- **Reusability**: Through inheritance and composition, code can be reused, saving development time and effort.
- **Scalability**: OOP makes it easier to extend existing code with new features.
- **Maintenance**: The encapsulated nature of objects makes maintenance easier, as changes to one part of the code are less likely to affect other parts.
- **Security**: Through encapsulation and abstraction, sensitive data can be hidden from users, and only necessary functionalities are exposed.

Basic OOP Concepts

Classes and Objects

The foundation of OOP in C++ is the concept of classes and objects.

- A **class** is a blueprint or template that defines the structure and behavior that objects of its type will have.
- An **object** is an instance of a class. It is a concrete entity based on a class and has state and behavior.

```
// Class definition
class Car {
private:
    // Data members (attributes)
    std::string brand;
    std::string model;
    int year;

public:
    // Constructor
    Car(std::string b, std::string m, int y) {
        brand = b;
        model = m;
        year = y;
    }
```

```
    // Methods
    void accelerate() {
        std::cout << brand << " " << model << " is accelerating!" << std::endl;
    }

    void brake() {
        std::cout << brand << " " << model << " is braking!" << std::endl;
    }

    // Getter methods
    std::string getBrand() { return brand; }
    std::string getModel() { return model; }
    int getYear() { return year; }
};

// Creating objects (instances of Car class)
Car myCar("Toyota", "Corolla", 2020);
Car yourCar("Honda", "Civic", 2019);

// Using object methods
myCar.accelerate();  // Output: Toyota Corolla is accelerating!
yourCar.brake();     // Output: Honda Civic is braking!

// Accessing object attributes via getters
std::cout << "My car is a " << myCar.getYear() << " " << myCar.getBrand() << " "
<< myCar.getModel() << std::endl;
```

Encapsulation

Encapsulation is the bundling of data and methods that operate on that data within a single unit (the class). It also involves restricting direct access to some of the object's components, which is a means of preventing accidental modification of data.

In C++, encapsulation is achieved using access specifiers:

- `private`: Members can only be accessed within the class
- `protected`: Members can be accessed within the class and by derived classes
- `public`: Members can be accessed from anywhere

```
class BankAccount {
private:
    // Private data members
    double balance;
    std::string accountNumber;

public:
    // Public constructor
    BankAccount(std::string accNum, double initialBalance) {
        accountNumber = accNum;
        // Validate initial balance
```

```
        if (initialBalance >= 0) {
            balance = initialBalance;
        } else {
            balance = 0;
            std::cout << "Initial balance cannot be negative. Setting to 0." <<
std::endl;
        }
    }

    // Public methods to interact with private data
    void deposit(double amount) {
        if (amount > 0) {
            balance += amount;
            std::cout << "Deposited: $" << amount << std::endl;
        } else {
            std::cout << "Cannot deposit negative amount." << std::endl;
        }
    }

    bool withdraw(double amount) {
        if (amount > 0 && amount <= balance) {
            balance -= amount;
            std::cout << "Withdrawn: $" << amount << std::endl;
            return true;
        } else {
            std::cout << "Invalid withdrawal amount or insufficient funds." <<
std::endl;
            return false;
        }
    }

    // Getter method
    double getBalance() const {
        return balance;
    }

    std::string getAccountNumber() const {
        // Only return a masked version for security
        return "XXXX" + accountNumber.substr(accountNumber.length() - 4);
    }
};

BankAccount myAccount("1234567890", 1000);
std::cout << "Account: " << myAccount.getAccountNumber() << ", Balance: $" <<
myAccount.getBalance() << std::endl;
myAccount.deposit(500);
std::cout << "New balance: $" << myAccount.getBalance() << std::endl;
```

Inheritance

Inheritance is a mechanism where a new class (derived or child class) is created from an existing class (base or parent class), inheriting its properties and behaviors. This promotes code reusability and establishes a

relationship between the base and derived classes.

```
// Base class
class Vehicle {
protected:
    std::string brand;

public:
    Vehicle(std::string b) : brand(b) {}

    void honk() {
        std::cout << "Beep! Beep!" << std::endl;
    }

    std::string getBrand() const {
        return brand;
    }
};

// Derived class
class Car : public Vehicle {
private:
    int numDoors;

public:
    Car(std::string brand, int doors) : Vehicle(brand), numDoors(doors) {}

    void displayInfo() {
        std::cout << "Brand: " << brand << ", Doors: " << numDoors << std::endl;
    }
};

// Another derived class
class Motorcycle : public Vehicle {
private:
    bool hasSidecar;

public:
    Motorcycle(std::string brand, bool sidecar) : Vehicle(brand),
hasSidecar(sidecar) {}

    void wheelie() {
        std::cout << brand << " motorcycle is doing a wheelie!" << std::endl;
    }
};

Car myCar("Toyota", 4);
myCar.honk();       // Inherited method
myCar.displayInfo(); // Own method

Motorcycle myBike("Harley-Davidson", false);
myBike.honk();      // Inherited method
myBike.wheelie();   // Own method
```

Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common base class. It enables one interface to be used for a general class of actions. The specific action is determined by the exact nature of the situation.

There are two types of polymorphism in C++:

1. **Compile-time Polymorphism (Static Binding)**:
    - Function overloading
    - Operator overloading
2. **Runtime Polymorphism (Dynamic Binding)**:
    - Function overriding using virtual functions

```
// Base class
class Shape {
public:
    virtual double area() const {
        return 0;
    }

    virtual void display() const {
        std::cout << "This is a shape." << std::endl;
    }

    // Virtual destructor (important for proper cleanup)
    virtual ~Shape() {}
};

// Derived class
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    // Override area method
    double area() const override {
        return 3.14159 * radius * radius;
    }

    void display() const override {
        std::cout << "This is a circle with radius " << radius << std::endl;
    }
};
```

```
// Another derived class
class Rectangle : public Shape {
private:
    double width;
    double height;

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

    // Override area method
    double area() const override {
        return width * height;
    }

    void display() const override {
        std::cout << "This is a rectangle with width " << width << " and height "
<< height << std::endl;
    }
};

// Polymorphic behavior
void printArea(const Shape& shape) {
    std::cout << "Area: " << shape.area() << std::endl;
    shape.display();
}

Circle myCircle(5);
Rectangle myRectangle(4, 6);

printArea(myCircle);     // Will call Circle::area() and Circle::display()
printArea(myRectangle);  // Will call Rectangle::area() and Rectangle::display()
```

Abstraction

Abstraction is the concept of hiding the complex implementation details and showing only the necessary features of an object. It helps in reducing programming complexity and effort.

In C++, abstraction is achieved using:

- Abstract classes (with pure virtual functions)
- Interfaces (abstract classes with only pure virtual functions)

```

// Abstract class
class AbstractEmployee {
protected:
    std::string name;
    int id;

public:
    AbstractEmployee(std::string n, int i) : name(n), id(i) {}
```

```
    // Pure virtual functions (must be implemented by derived classes)
    virtual double calculateSalary() const = 0;
    virtual void displayDetails() const = 0;

    // Regular method
    std::string getName() const {
        return name;
    }

    virtual ~AbstractEmployee() {}
};

// Concrete class implementing the abstract class
class FullTimeEmployee : public AbstractEmployee {
private:
    double monthlySalary;

public:
    FullTimeEmployee(std::string name, int id, double salary)
        : AbstractEmployee(name, id), monthlySalary(salary) {}

    double calculateSalary() const override {
        return monthlySalary;
    }

    void displayDetails() const override {
        std::cout << "Full-time Employee: " << name << ", ID: " << id
                  << ", Monthly Salary: $" << monthlySalary << std::endl;
    }
};

// Another concrete class
class PartTimeEmployee : public AbstractEmployee {
private:
    double hourlyRate;
    int hoursWorked;

public:
    PartTimeEmployee(std::string name, int id, double rate, int hours)
        : AbstractEmployee(name, id), hourlyRate(rate), hoursWorked(hours) {}

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

    void displayDetails() const override {
        std::cout << "Part-time Employee: " << name << ", ID: " << id
                  << ", Hourly Rate: $" << hourlyRate
                  << ", Hours Worked: " << hoursWorked << std::endl;
    }
};

// Cannot instantiate abstract class
// AbstractEmployee emp("John", 1);  // Error!
```

```
FullTimeEmployee fte("John", 1001, 5000);
PartTimeEmployee pte("Jane", 2001, 20, 80);

fte.displayDetails();
std::cout << "Salary: $" << fte.calculateSalary() << std::endl;

pte.displayDetails();
std::cout << "Salary: $" << pte.calculateSalary() << std::endl;

// Polymorphism with abstract class pointers
AbstractEmployee* employees[2] = {&fte, &pte};
for (AbstractEmployee* emp : employees) {
    emp->displayDetails();
    std::cout << "Salary: $" << emp->calculateSalary() << std::endl;
}
```

Advanced Class Features

Constructors and Destructors

Types of Constructors

1. **Default Constructor**: Takes no parameters, or all parameters have default values.
2. **Parameterized Constructor**: Takes parameters for initializing objects.
3. **Copy Constructor**: Creates a new object as a copy of an existing object.
4. **Move Constructor**: Transfers resources from a temporary object to a new object.
5. **Delegating Constructor**: One constructor calls another constructor of the same class.

```
class Person {
private:
    std::string name;
    int age;
    double* salaryHistory;
    int historySize;

public:
    // Default constructor
    Person() : name("Unknown"), age(0), salaryHistory(nullptr), historySize(0) {
        std::cout << "Default constructor called" << std::endl;
    }

    // Parameterized constructor
    Person(std::string n, int a) : name(n), age(a), historySize(0) {
        std::cout << "Parameterized constructor called" << std::endl;
        salaryHistory = new double[10]; // Allocate memory
        historySize = 10;
    }
```

```
    // Copy constructor
    Person(const Person& other) : name(other.name), age(other.age),
historySize(other.historySize) {
        std::cout << "Copy constructor called" << std::endl;
        // Deep copy
        if (other.salaryHistory != nullptr) {
            salaryHistory = new double[historySize];
            for (int i = 0; i < historySize; i++) {
                salaryHistory[i] = other.salaryHistory[i];
            }
        } else {
            salaryHistory = nullptr;
        }
    }

    // Move constructor
    Person(Person&& other) noexcept : name(std::move(other.name)), age(other.age),
                                      salaryHistory(other.salaryHistory),
historySize(other.historySize) {
        std::cout << "Move constructor called" << std::endl;
        // Transfer ownership and leave other in valid state
        other.salaryHistory = nullptr;
        other.historySize = 0;
    }

    // Destructor
    ~Person() {
        std::cout << "Destructor called for " << name << std::endl;
        delete[] salaryHistory; // Free allocated memory
    }

    // Methods
    void setSalary(int index, double salary) {
        if (index >= 0 && index < historySize) {
            salaryHistory[index] = salary;
        }
    }

    void display() const {
        std::cout << "Name: " << name << ", Age: " << age << std::endl;
    }
};

// Usage
Person p1;                  // Default constructor
Person p2("Alice", 30);     // Parameterized constructor
Person p3 = p2;             // Copy constructor
Person p4 = std::move(p3);  // Move constructor
```

Initialization Lists

Initialization lists are used to initialize class members directly, which is more efficient than assigning values in the constructor body.

```
class Rectangle {
private:
    double width;
    double height;
    const double MAX_WIDTH;

public:
    // Using initialization list
    Rectangle(double w, double h)
        : width(w), height(h), MAX_WIDTH(100) {
        // Constructor body
        if (width > MAX_WIDTH) {
            width = MAX_WIDTH;
        }
    }
};
```

Destructors

A destructor is a special member function that is called when an object's lifetime ends. It is used to clean up resources, especially those that were dynamically allocated.

```
class ResourceManager {
private:
    int* data;

public:
    ResourceManager() {
        data = new int[100];
        std::cout << "Resources allocated" << std::endl;
    }

    ~ResourceManager() {
        delete[] data;
        std::cout << "Resources freed" << std::endl;
    }
};

void testResourceManagement() {
    ResourceManager rm;
    // When rm goes out of scope, destructor will be called
}
```

Copy and Move Semantics

Copy and move semantics define how objects are copied or moved when they are passed by value, returned by value, or when an object is created based on another object.

Copy Constructor and Copy Assignment Operator

```
class MyString {
private:
    char* data;
    size_t length;

public:
    // Constructor
    MyString(const char* str = nullptr) {
        if (str == nullptr) {
            data = new char[1];
            data[0] = '\0';
            length = 0;
        } else {
            length = strlen(str);
            data = new char[length + 1];
            strcpy(data, str);
        }
    }

    // Copy constructor
    MyString(const MyString& other) {
        length = other.length;
        data = new char[length + 1];
        strcpy(data, other.data);
        std::cout << "Copy constructor called" << std::endl;
    }

    // Copy assignment operator
    MyString& operator=(const MyString& other) {
        if (this != &other) { // Self-assignment check
            // Free existing resources
            delete[] data;

            // Copy resources
            length = other.length;
            data = new char[length + 1];
            strcpy(data, other.data);
        }
        std::cout << "Copy assignment operator called" << std::endl;
        return *this;
    }

    // Destructor
    ~MyString() {
        delete[] data;
    }
```

```
    // Utility method
    void display() const {
        std::cout << data << std::endl;
    }
};

MyString s1("Hello");
MyString s2 = s1;            // Copy constructor called
MyString s3;
s3 = s1;                     // Copy assignment operator called
```

Move Constructor and Move Assignment Operator

```
class MyString {
    // ... (previous code)

    // Move constructor
    MyString(MyString&& other) noexcept {
        data = other.data;           // Transfer ownership
        length = other.length;

        // Leave other in a valid state
        other.data = new char[1];
        other.data[0] = '\0';
        other.length = 0;

        std::cout << "Move constructor called" << std::endl;
    }

    // Move assignment operator
    MyString& operator=(MyString&& other) noexcept {
        if (this != &other) {
            // Free existing resources
            delete[] data;

            // Transfer ownership
            data = other.data;
            length = other.length;

            // Leave other in a valid state
            other.data = new char[1];
            other.data[0] = '\0';
            other.length = 0;
        }
        std::cout << "Move assignment operator called" << std::endl;
        return *this;
    }
};

MyString s1("Hello");
MyString s2 = std::move(s1);  // Move constructor called
```

```
MyString s3;
s3 = std::move(s2);           // Move assignment operator called
```

Operator Overloading

Operator overloading allows operators to be redefined for user-defined types, making the code more intuitive and readable.

```
class Complex {
private:
    double real;
    double imag;

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

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

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

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

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

    // Overload != operator for inequality comparison
    bool operator!=(const Complex& other) const {
        return !(*this == other);
    }

    // Overload << operator for output stream
    friend std::ostream& operator<<(std::ostream& os, const Complex& c) {
        os << c.real;
        if (c.imag >= 0) {
            os << " + " << c.imag << "i";
        } else {
            os << " - " << -c.imag << "i";
```

```
        }
        return os;
    }

    // Overload >> operator for input stream
    friend std::istream& operator>>(std::istream& is, Complex& c) {
        is >> c.real >> c.imag;
        return is;
    }

    // Overload prefix increment (++c)
    Complex& operator++() {
        ++real;
        return *this;
    }

    // Overload postfix increment (c++)
    Complex operator++(int) {
        Complex temp = *this;
        ++(*this);  // Call prefix increment
        return temp;
    }

    // Overload assignment operators
    Complex& operator+=(const Complex& other) {
        real += other.real;
        imag += other.imag;
        return *this;
    }
};

Complex c1(3, 2);
Complex c2(1, 4);
Complex c3 = c1 + c2;        // Uses operator+
std::cout << c3 << std::endl; // Uses operator<<
c1 += c2;                    // Uses operator+=
++c1;                        // Uses prefix increment
c2++;                        // Uses postfix increment
```

Friend Functions and Classes

A friend function or class can access private and protected members of a class in which it is declared as a friend.

```
class Box {
private:
    double width;
    double height;
    double depth;

public:
```

```
    Box(double w, double h, double d) : width(w), height(h), depth(d) {}

    // Friend function declaration
    friend double calculateVolume(const Box& box);

    // Friend class declaration
    friend class BoxManager;
};

// Friend function definition
double calculateVolume(const Box& box) {
    // Can access private members of Box
    return box.width * box.height * box.depth;
}

// Friend class definition
class BoxManager {
public:
    void printBoxDimensions(const Box& box) {
        // Can access private members of Box
        std::cout << "Width: " << box.width
                  << ", Height: " << box.height
                  << ", Depth: " << box.depth << std::endl;
    }

    void modifyBox(Box& box) {
        // Can modify private members of Box
        box.width *= 2;
        box.height *= 2;
        box.depth *= 2;
    }
};

Box myBox(2, 3, 4);
std::cout << "Volume: " << calculateVolume(myBox) << std::endl;

BoxManager manager;
manager.printBoxDimensions(myBox);
manager.modifyBox(myBox);
manager.printBoxDimensions(myBox);
```

Static Members

Static members belong to the class itself rather than to instances of the class. They are shared among all objects of the class.

```
class Employee {
private:
    std::string name;
    int id;
    double salary;
```

```
    // Static data member
    static int employeeCount;
    static double totalSalary;

public:
    Employee(std::string n, int i, double s) : name(n), id(i), salary(s) {
        employeeCount++;
        totalSalary += salary;
    }

    ~Employee() {
        employeeCount--;
        totalSalary -= salary;
    }

    // Static method
    static int getEmployeeCount() {
        return employeeCount;
    }

    static double getAverageSalary() {
        if (employeeCount > 0) {
            return totalSalary / employeeCount;
        }
        return 0;
    }

    void display() const {
        std::cout << "ID: " << id << ", Name: " << name << ", Salary: " << salary
<< std::endl;
    }
};

// Initialize static data members
int Employee::employeeCount = 0;
double Employee::totalSalary = 0;

std::cout << "Initial employee count: " << Employee::getEmployeeCount() <<
std::endl;

{
    Employee e1("Alice", 1001, 5000);
    Employee e2("Bob", 1002, 6000);

    std::cout << "Current employee count: " << Employee::getEmployeeCount() <<
std::endl;
    std::cout << "Average salary: $" << Employee::getAverageSalary() << std::endl;

    // Scope ends, e1 and e2 will be destroyed
}

std::cout << "Final employee count: " << Employee::getEmployeeCount() <<
std::endl;
```

Inheritance Mechanisms

Types of Inheritance

C++ supports different types of inheritance:

1. **Single Inheritance**: A derived class inherits from a single base class.
2. **Multiple Inheritance**: A derived class inherits from multiple base classes.
3. **Multilevel Inheritance**: A derived class inherits from a base class, which in turn inherits from another base class.
4. **Hierarchical Inheritance**: Multiple derived classes inherit from a single base class.
5. **Hybrid Inheritance**: A combination of two or more types of inheritance.

Single Inheritance

```
class Animal {
protected:
    std::string name;

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

    void eat() {
        std::cout << name << " is eating." << std::endl;
    }

    void sleep() {
        std::cout << name << " is sleeping." << std::endl;
    }
};

class Dog : public Animal {
private:
    std::string breed;

public:
    Dog(std::string name, std::string b) : Animal(name), breed(b) {}

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

    void display() {
        std::cout << "Name: " << name << ", Breed: " << breed << std::endl;
    }
};

Dog myDog("Rex", "German Shepherd");
```

```
myDog.eat();    // From Animal class
myDog.bark();   // From Dog class
myDog.display();
```

Multiple Inheritance

```
class Printer {
protected:
    std::string model;

public:
    Printer(std::string m) : model(m) {}

    void print() {
        std::cout << "Printing document with " << model << std::endl;
    }
};

class Scanner {
protected:
    int dpi;

public:
    Scanner(int d) : dpi(d) {}

    void scan() {
        std::cout << "Scanning document at " << dpi << " DPI" << std::endl;
    }
};

class AllInOne : public Printer, public Scanner {
private:
    bool hasFax;

public:
    AllInOne(std::string model, int dpi, bool fax)
        : Printer(model), Scanner(dpi), hasFax(fax) {}

    void copy() {
        std::cout << "Copying document" << std::endl;
        scan();
        print();
    }

    void fax() {
        if (hasFax) {
            std::cout << "Sending fax..." << std::endl;
        } else {
            std::cout << "This device doesn't have fax capability." << std::endl;
        }
    }
```

```
};

AllInOne device("HP OfficeJet", 600, true);
device.print();  // From Printer
device.scan();   // From Scanner
device.copy();   // From AllInOne
device.fax();    // From AllInOne
```

Multilevel Inheritance

```
class Vehicle {
protected:
    std::string brand;

public:
    Vehicle(std::string b) : brand(b) {}

    void start() {
        std::cout << "Vehicle started" << std::endl;
    }
};

class Car : public Vehicle {
protected:
    int numWheels;

public:
    Car(std::string brand, int wheels) : Vehicle(brand), numWheels(wheels) {}

    void drive() {
        std::cout << brand << " car is driving" << std::endl;
    }
};

class SportsCar : public Car {
private:
    int topSpeed;

public:
    SportsCar(std::string brand, int wheels, int speed)
        : Car(brand, wheels), topSpeed(speed) {}

    void accelerate() {
        std::cout << brand << " sports car is accelerating to " << topSpeed << "
mph" << std::endl;
    }
};

SportsCar myCar("Ferrari", 4, 200);
myCar.start();      // From Vehicle
```

```
myCar.drive();      // From Car
myCar.accelerate(); // From SportsCar
```

Hierarchical Inheritance

```
class Shape {
protected:
    double x, y;  // Position coordinates

public:
    Shape(double xPos, double yPos) : x(xPos), y(yPos) {}

    virtual void draw() const {
        std::cout << "Drawing a shape at position (" << x << ", " << y << ")" <<
std::endl;
    }

    virtual double area() const = 0;
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double xPos, double yPos, double r) : Shape(xPos, yPos), radius(r) {}

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

    double area() const override {
        return 3.14159 * radius * radius;
    }
};

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

public:
    Rectangle(double xPos, double yPos, double w, double h)
        : Shape(xPos, yPos), width(w), height(h) {}

    void draw() const override {
        std::cout << "Drawing a rectangle at (" << x << ", " << y
                  << ") with width " << width << " and height " << height <<
std::endl;
    }
```

```
    double area() const override {
        return width * height;
    }
};

Circle circle(10, 20, 5);
Rectangle rectangle(30, 40, 15, 10);

circle.draw();
std::cout << "Circle area: " << circle.area() << std::endl;

rectangle.draw();
std::cout << "Rectangle area: " << rectangle.area() << std::endl;
```

Virtual Functions

Virtual functions allow a derived class to override methods from a base class, enabling runtime polymorphism. When a virtual function is called using a pointer or reference to a base class, the appropriate derived class's function is executed.

```
class Animal {
protected:
    std::string name;

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

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

    // Non-virtual function
    void sleep() const {
        std::cout << name << " is sleeping" << std::endl;
    }

    virtual ~Animal() {
        std::cout << "Animal destructor called" << std::endl;
    }
};

class Dog : public Animal {
private:
    std::string breed;

public:
    Dog(std::string name, std::string b) : Animal(name), breed(b) {}

    // Override the virtual function
```

```
    void makeSound() const override {
        std::cout << name << " (" << breed << ") says: Woof! Woof!" << std::endl;
    }

    ~Dog() {
        std::cout << "Dog destructor called" << std::endl;
    }
};

class Cat : public Animal {
public:
    Cat(std::string name) : Animal(name) {}

    // Override the virtual function
    void makeSound() const override {
        std::cout << name << " says: Meow! Meow!" << std::endl;
    }

    ~Cat() {
        std::cout << "Cat destructor called" << std::endl;
    }
};

// Polymorphic behavior with pointers
Animal* animals[3];
animals[0] = new Animal("Generic Animal");
animals[1] = new Dog("Rex", "German Shepherd");
animals[2] = new Cat("Whiskers");

for (int i = 0; i < 3; i++) {
    animals[i]->makeSound();  // Will call the appropriate derived class's
makeSound()
    animals[i]->sleep();      // Will always call Animal::sleep()
}

// Proper cleanup
for (int i = 0; i < 3; i++) {
    delete animals[i];  // Will call the appropriate destructor
}
```

Abstract Classes and Pure Virtual Functions

An abstract class is a class that contains at least one pure virtual function, marked with `= 0`. Abstract classes cannot be instantiated directly and must be derived from. Any class derived from an abstract class must implement all pure virtual functions, or it will also be abstract.

```
// Abstract class
class Shape {
protected:
    double x, y;  // Position
```

```
public:
    Shape(double xPos, double yPos) : x(xPos), y(yPos) {}

    // Pure virtual function
    virtual double area() const = 0;

    // Pure virtual function
    virtual double perimeter() const = 0;

    // Regular virtual function
    virtual void display() const {
        std::cout << "Position: (" << x << ", " << y << ")" << std::endl;
    }

    // Non-virtual function
    void move(double newX, double newY) {
        x = newX;
        y = newY;
    }

    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double xPos, double yPos, double r) : Shape(xPos, yPos), radius(r) {}

    // Implement pure virtual function
    double area() const override {
        return 3.14159 * radius * radius;
    }

    // Implement pure virtual function
    double perimeter() const override {
        return 2 * 3.14159 * radius;
    }

    // Override regular virtual function
    void display() const override {
        Shape::display();  // Call base class method
        std::cout << "Circle with radius " << radius << std::endl;
    }
};

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

public:
    Rectangle(double xPos, double yPos, double w, double h)
```

```
        : Shape(xPos, yPos), width(w), height(h) {}

    // Implement pure virtual function
    double area() const override {
        return width * height;
    }

    // Implement pure virtual function
    double perimeter() const override {
        return 2 * (width + height);
    }

    // Override regular virtual function
    void display() const override {
        Shape::display();  // Call base class method
        std::cout << "Rectangle with width " << width << " and height " << height
<< std::endl;
    }
};

// Cannot instantiate abstract class
// Shape shape(0, 0);  // Error!

Circle circle(10, 20, 5);
Rectangle rectangle(30, 40, 15, 10);

// Polymorphic array of shapes
Shape* shapes[2] = {&circle, &rectangle};

for (Shape* shape : shapes) {
    shape->display();
    std::cout << "Area: " << shape->area() << std::endl;
    std::cout << "Perimeter: " << shape->perimeter() << std::endl;
}
```

Multiple Inheritance

Multiple inheritance allows a class to inherit from more than one base class, combining their behaviors and attributes.

```
class Person {
protected:
    std::string name;
    int age;

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

    void introduce() const {
        std::cout << "My name is " << name << " and I am " << age << " years old."
<< std::endl;
```

```
    }
};

class Employee {
protected:
    int employeeId;
    double salary;

public:
    Employee(int id, double s) : employeeId(id), salary(s) {}

    void work() const {
        std::cout << "Employee #" << employeeId << " is working." << std::endl;
    }

    double getSalary() const {
        return salary;
    }
};

class Manager : public Person, public Employee {
private:
    std::string department;

public:
    Manager(std::string name, int age, int id, double salary, std::string dept)
        : Person(name, age), Employee(id, salary), department(dept) {}

    void manage() const {
        std::cout << name << " is managing the " << department << " department."
<< std::endl;
    }

    void displayDetails() const {
        introduce();
        std::cout << "Employee ID: " << employeeId << std::endl;
        std::cout << "Department: " << department << std::endl;
        std::cout << "Salary: $" << getSalary() << std::endl;
    }
};

Manager manager("John Smith", 45, 1001, 85000, "Engineering");
manager.introduce();  // From Person
manager.work();       // From Employee
manager.manage();     // From Manager
manager.displayDetails();
```

The Diamond Problem

The diamond problem occurs in multiple inheritance when a class inherits from two classes that have a common base class, creating ambiguity.

```cpp
// Base class
class Person {
protected:
    std::string name;

public:
    Person(std::string n) : name(n) {
        std::cout << "Person constructor called" << std::endl;
    }

    void identify() const {
        std::cout << "I am " << name << std::endl;
    }

    virtual ~Person() {
        std::cout << "Person destructor called" << std::endl;
    }
};

// First derived class
class Student : public virtual Person {
protected:
    std::string studentId;

public:
    Student(std::string name, std::string id)
        : Person(name), studentId(id) {
        std::cout << "Student constructor called" << std::endl;
    }

    void study() const {
        std::cout << name << " with student ID " << studentId << " is studying" <<
std::endl;
    }

    ~Student() {
        std::cout << "Student destructor called" << std::endl;
    }
};

// Second derived class
class Employee : public virtual Person {
protected:
    int employeeId;

public:
    Employee(std::string name, int id)
        : Person(name), employeeId(id) {
        std::cout << "Employee constructor called" << std::endl;
    }

    void work() const {
        std::cout << name << " with employee ID " << employeeId << " is working"
```

```
<< std::endl;
    }

    ~Employee() {
        std::cout << "Employee destructor called" << std::endl;
    }
};

// Derived from both Student and Employee
class TeachingAssistant : public Student, public Employee {
private:
    std::string course;

public:
    TeachingAssistant(std::string name, std::string studentId, int employeeId,
std::string c)
        : Person(name),  // Only one Person object is created
          Student(name, studentId),
          Employee(name, employeeId),
          course(c) {
        std::cout << "TeachingAssistant constructor called" << std::endl;
    }

    void teach() const {
        std::cout << name << " is teaching " << course << std::endl;
    }

    ~TeachingAssistant() {
        std::cout << "TeachingAssistant destructor called" << std::endl;
    }
};

TeachingAssistant ta("Alice", "S12345", 1001, "C++ Programming");
ta.identify();  // No ambiguity thanks to virtual inheritance
ta.study();
ta.work();
ta.teach();
```

Polymorphism in Depth

Runtime Polymorphism

Runtime polymorphism is achieved through virtual functions. The decision about which function to call is taken at runtime based on the type of the object.

```
class Base {
public:
    virtual void show() {
        std::cout << "Base class show method" << std::endl;
    }
```

```
    void display() {
        std::cout << "Base class display method" << std::endl;
    }

    virtual ~Base() {
        std::cout << "Base destructor" << std::endl;
    }
};

class Derived : public Base {
public:
    void show() override {
        std::cout << "Derived class show method" << std::endl;
    }

    void display() {
        std::cout << "Derived class display method" << std::endl;
    }

    ~Derived() {
        std::cout << "Derived destructor" << std::endl;
    }
};

Base* basePtr = new Derived();
basePtr->show();     // Will call Derived::show()
basePtr->display();  // Will call Base::display() (not virtual)

delete basePtr;      // Will call Derived::~Derived() followed by Base::~Base()
```

Compile-time Polymorphism

Compile-time polymorphism is achieved through function overloading and templates. The decision about which function to call is taken at compile time.

```
class Calculator {
public:
    // Function overloading
    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;
    }
```

```
    // Template function
    template<typename T>
    T multiply(T a, T b) {
        return a * b;
    }
};

Calculator calc;
std::cout << calc.add(5, 3) << std::endl;             // Calls int add(int, int)
std::cout << calc.add(5.5, 3.5) << std::endl;         // Calls double add(double,
double)
std::cout << calc.add(1, 2, 3) << std::endl;          // Calls int add(int, int,
int)
std::cout << calc.multiply<int>(5, 3) << std::endl;   // Calls multiply<int>
std::cout << calc.multiply<double>(2.5, 3.0) << std::endl; // Calls
multiply<double>
```

Virtual Destructors

Virtual destructors are necessary when deleting a derived class object through a base class pointer. Without a virtual destructor, only the base class destructor would be called, potentially leading to resource leaks.

```
class Base {
public:
    Base() {
        std::cout << "Base constructor" << std::endl;
    }

    // Non-virtual destructor - problematic
    ~Base() {
        std::cout << "Base destructor" << std::endl;
    }
};

class Derived : public Base {
private:
    int* data;

public:
    Derived() {
        std::cout << "Derived constructor" << std::endl;
        data = new int[100];
    }

    ~Derived() {
        std::cout << "Derived destructor" << std::endl;
        delete[] data;  // This won't be called if Base::~Base() is not virtual
    }
};

// Problematic case
```

```
Base* ptr1 = new Derived();
delete ptr1;  // Only calls Base::~Base(), leading to memory leak

// Correct approach with virtual destructor
class BaseWithVirtual {
public:
    BaseWithVirtual() {
        std::cout << "BaseWithVirtual constructor" << std::endl;
    }

    virtual ~BaseWithVirtual() {
        std::cout << "BaseWithVirtual destructor" << std::endl;
    }
};

class DerivedFromVirtual : public BaseWithVirtual {
private:
    int* data;

public:
    DerivedFromVirtual() {
        std::cout << "DerivedFromVirtual constructor" << std::endl;
        data = new int[100];
    }

    ~DerivedFromVirtual() {
        std::cout << "DerivedFromVirtual destructor" << std::endl;
        delete[] data;  // This will be called properly
    }
};

BaseWithVirtual* ptr2 = new DerivedFromVirtual();
delete ptr2;  // Calls both DerivedFromVirtual::~DerivedFromVirtual() and
BaseWithVirtual::~BaseWithVirtual()
```

Virtual Function Tables

C++ implements virtual functions using a virtual function table (vtable). Each class with virtual functions has a vtable, and each object of that class contains a pointer to its vtable.

```

// This is a conceptual representation of how vtables work
class Base {
public:
    virtual void func1() { std::cout << "Base::func1()" << std::endl; }
    virtual void func2() { std::cout << "Base::func2()" << std::endl; }
    void func3() { std::cout << "Base::func3()" << std::endl; }
};

class Derived : public Base {
public:
    void func1() override { std::cout << "Derived::func1()" << std::endl; }
```

```
    // func2() is inherited from Base
    void func3() { std::cout << "Derived::func3()" << std::endl; }
    virtual void func4() { std::cout << "Derived::func4()" << std::endl; }
};

/*
Conceptual representation of vtables:

Base vtable:
    func1() -> &Base::func1
    func2() -> &Base::func2

Derived vtable:
    func1() -> &Derived::func1
    func2() -> &Base::func2
    func4() -> &Derived::func4

Note: func3() is not in the vtable because it's not virtual
*/

Base* basePtr = new Derived();
basePtr->func1();  // Calls Derived::func1() through vtable
basePtr->func2();  // Calls Base::func2() through vtable
basePtr->func3();  // Calls Base::func3() directly (not through vtable)

delete basePtr;
```

Memory Management in OOP

Dynamic Memory Allocation

Dynamic memory allocation in C++ is done using the `new` and `delete` operators.

```
class DynamicArray {
private:
    int* data;
    int size;

public:
    // Constructor
    DynamicArray(int s) : size(s) {
        data = new int[size];  // Allocate memory
        for (int i = 0; i < size; i++) {
            data[i] = 0;
        }
        std::cout << "Memory allocated for array of size " << size << std::endl;
    }

    // Destructor
    ~DynamicArray() {
```

```
        delete[] data;  // Free memory
        std::cout << "Memory freed for array of size " << size << std::endl;
    }

    // Access elements
    int& operator[](int index) {
        if (index < 0 || index >= size) {
            throw std::out_of_range("Index out of bounds");
        }
        return data[index];
    }

    int getSize() const {
        return size;
    }
};

// Usage
void testDynamicArray() {
    DynamicArray arr(10);

    // Modify array elements
    for (int i = 0; i < arr.getSize(); i++) {
        arr[i] = i * 2;
    }

    // Access array elements
    for (int i = 0; i < arr.getSize(); i++) {
        std::cout << "arr[" << i << "] = " << arr[i] << std::endl;
    }

    // Memory is automatically freed when arr goes out of scope
}

testDynamicArray();
```

RAII Principle

Resource Acquisition Is Initialization (RAII) is a programming technique where resource management is tied to the lifetime of an object. Resources are acquired during object initialization and released during object destruction.

```
class FileHandle {
private:
    FILE* file;
    std::string filename;

public:
    FileHandle(const std::string& name, const std::string& mode) : filename(name)
{
        file = fopen(name.c_str(), mode.c_str());
```

```
        if (!file) {
            throw std::runtime_error("Failed to open file: " + name);
        }
        std::cout << "Opened file: " << filename << std::endl;
    }

    ~FileHandle() {
        if (file) {
            fclose(file);
            std::cout << "Closed file: " << filename << std::endl;
        }
    }

    // Prevent copying
    FileHandle(const FileHandle&) = delete;
    FileHandle& operator=(const FileHandle&) = delete;

    // Allow moving
    FileHandle(FileHandle&& other) noexcept : file(other.file),
filename(std::move(other.filename)) {
        other.file = nullptr;
    }

    FileHandle& operator=(FileHandle&& other) noexcept {
        if (this != &other) {
            if (file) {
                fclose(file);
            }
            file = other.file;
            filename = std::move(other.filename);
            other.file = nullptr;
        }
        return *this;
    }

    // Write to file
    void write(const std::string& data) {
        if (!file) {
            throw std::runtime_error("File not open");
        }
        fputs(data.c_str(), file);
        fflush(file);
    }

    // Read from file
    std::string read(size_t maxSize) {
        if (!file) {
            throw std::runtime_error("File not open");
        }
        char* buffer = new char[maxSize + 1];
        size_t bytesRead = fread(buffer, 1, maxSize, file);
        buffer[bytesRead] = '\0';
        std::string result(buffer);
        delete[] buffer;
```

```
        return result;
    }
};

void testFileHandle() {
    try {
        // Resource is acquired when FileHandle is constructed
        FileHandle file("example.txt", "w");
        file.write("Hello, RAII!");

        // Resource is released when file goes out of scope
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << std::endl;
    }
}

testFileHandle();
```

Smart Pointers

Smart pointers are objects that act like pointers but provide automatic memory management. C++11 introduced three main types of smart pointers:

1. `std::unique_ptr`: Represents exclusive ownership of a dynamically allocated resource.
2. `std::shared_ptr`: Represents shared ownership of a resource using reference counting.
3. `std::weak_ptr`: Holds a non-owning reference to a resource managed by a `std::shared_ptr`.

```
#include <memory>

class Resource {
private:
    std::string name;

public:
    Resource(const std::string& n) : name(n) {
        std::cout << "Resource " << name << " acquired" << std::endl;
    }

    ~Resource() {
        std::cout << "Resource " << name << " released" << std::endl;
    }

    void use() {
        std::cout << "Using resource " << name << std::endl;
    }
};

void testUniquePtr() {
    std::cout << "=== Testing unique_ptr ===" << std::endl;

    // std::unique_ptr with direct initialization
```

```
    std::unique_ptr<Resource> res1(new Resource("One"));

    // Preferred way: std::make_unique (C++14)
    auto res2 = std::make_unique<Resource>("Two");

    res1->use();
    res2->use();

    // Transfer ownership
    std::unique_ptr<Resource> res3 = std::move(res1);

    // res1 is now null
    if (!res1) {
        std::cout << "res1 is null after move" << std::endl;
    }

    res3->use();

    // Unique pointers are automatically deleted when they go out of scope
}

void testSharedPtr() {
    std::cout << "=== Testing shared_ptr ===" << std::endl;

    // Create a shared_ptr
    auto res1 = std::make_shared<Resource>("Shared");
    std::cout << "Reference count: " << res1.use_count() << std::endl;

    {
        // Create another shared_ptr pointing to the same resource
        auto res2 = res1;
        std::cout << "Reference count: " << res1.use_count() << std::endl;

        // Create a third shared_ptr
        auto res3 = res2;
        std::cout << "Reference count: " << res1.use_count() << std::endl;

        res3->use();
    } // res2 and res3 go out of scope, but the resource is not deleted

    std::cout << "After inner scope, reference count: " << res1.use_count() <<
std::endl;
    res1->use();

} // res1 goes out of scope, and the resource is deleted

void testWeakPtr() {
    std::cout << "=== Testing weak_ptr ===" << std::endl;

    // Create a shared_ptr
    auto sharedRes = std::make_shared<Resource>("Weak");

    // Create a weak_ptr from the shared_ptr
    std::weak_ptr<Resource> weakRes = sharedRes;
```

```
    std::cout << "Weak pointer expired? " << weakRes.expired() << std::endl;

    // Use the resource through the weak_ptr
    if (auto tempShared = weakRes.lock()) {
        tempShared->use();
        std::cout << "Reference count: " << tempShared.use_count() << std::endl;
    } else {
        std::cout << "Resource is no longer available" << std::endl;
    }

    // Reset the shared_ptr, which deletes the resource
    sharedRes.reset();

    std::cout << "After reset, weak pointer expired? " << weakRes.expired() <<
std::endl;

    // Try to use the resource again
    if (auto tempShared = weakRes.lock()) {
        tempShared->use();
    } else {
        std::cout << "Resource is no longer available" << std::endl;
    }
}

testUniquePtr();
testSharedPtr();
testWeakPtr();
```

Memory Leaks and Prevention

Memory leaks occur when dynamically allocated memory is not properly freed. This can lead to inefficient resource usage and potential program crashes.

```
class MemoryLeakDemo {
private:
    int* data;

public:
    // Problematic constructor
    MemoryLeakDemo() {
        data = new int[100];
        std::cout << "Memory allocated" << std::endl;

        // If an exception is thrown here, data won't be freed
        // throw std::runtime_error("Exception in constructor");
    }

    // Missing destructor - will cause a memory leak
    // ~MemoryLeakDemo() {
    //     delete[] data;
```

```
    //     std::cout << "Memory freed" << std::endl;
    // }
};

class MemoryLeakFix {
private:
    std::unique_ptr<int[]> data;  // Using smart pointer instead

public:
    MemoryLeakFix() {
        data = std::make_unique<int[]>(100);
        std::cout << "Memory allocated safely" << std::endl;

        // Even if an exception is thrown, memory will be freed
        // throw std::runtime_error("Exception in constructor");
    }

    // No need for explicit destructor - smart pointer handles cleanup
};

void memoryLeakTest() {
    // Leaks memory
    MemoryLeakDemo* leaky = new MemoryLeakDemo();
    // Forgot to delete leaky

    // No leak - smart pointer handles cleanup
    auto safe = std::make_unique<MemoryLeakFix>();

    // Alternatively, use RAII with automatic variables
    MemoryLeakFix safeObject;
}

memoryLeakTest();
```

Templates and Generic Programming

Templates are a powerful feature of C++ that enables generic programming, allowing you to write code that works with any data type. They form the foundation of the Standard Template Library (STL) and are essential for creating reusable, type-safe components.

Class Templates

Class templates allow you to define classes that can work with any data type, making your code more flexible and reusable.

Basic Syntax

```
template <typename T>
class Container {
```

```
private:
    T element;
public:
    Container(T arg) : element(arg) {}
    T getElement() const { return element; }
    void setElement(T arg) { element = arg; }
};
```

Using Class Templates

```
// Using with different types
Container<int> intContainer(5);
Container<std::string> stringContainer("Hello");
Container<double> doubleContainer(3.14);

std::cout << intContainer.getElement() << std::endl;      // Outputs: 5
std::cout << stringContainer.getElement() << std::endl;   // Outputs: Hello
std::cout << doubleContainer.getElement() << std::endl;   // Outputs: 3.14
```

Multiple Template Parameters

Class templates can have multiple template parameters of different types:

```
template <typename T, typename U, int Size = 10>
class Pair {
private:
    T first;
    U second;
    T data[Size];
public:
    Pair(T a, U b) : first(a), second(b) {}
    T getFirst() const { return first; }
    U getSecond() const { return second; }
};

// Usage
Pair<int, std::string, 5> pair(42, "Answer");
```

Template Type Constraints (C++20)

With concepts in C++20, you can constrain template parameters:

```
#include <concepts>

template <typename T>
concept Numeric = std::is_arithmetic_v<T>;
```

```
template <Numeric T>
class Calculator {
public:
    T add(T a, T b) { return a + b; }
    T subtract(T a, T b) { return a - b; }
};

// Only works with numeric types
Calculator<int> intCalc;     // Valid
Calculator<double> dblCalc;  // Valid
// Calculator<std::string> strCalc; // Error: std::string is not a numeric type
```

7.2 Function Templates

Function templates allow you to write a single function that can operate on different data types.

Basic Syntax

```
template <typename T>
T maximum(T a, T b) {
    return (a > b) ? a : b;
}
```

Template Argument Deduction

The compiler can often deduce the template arguments from the function arguments:

```
int a = 5, b = 7;
std::cout << maximum(a, b) << std::endl;     // T is deduced as int

double c = 3.14, d = 2.71;
std::cout << maximum(c, d) << std::endl;     // T is deduced as double
```

Explicit Template Arguments

Sometimes you need to specify template arguments explicitly:

```
template <typename T, typename U>
T custom_cast(U value) {
    return static_cast<T>(value);
}

double pi = 3.14159;
int rounded_pi = custom_cast<int>(pi);  // Explicitly specify T as int
```

Variadic Templates (C++11)

Variadic templates allow you to work with an arbitrary number of arguments:

```
template <typename T>
T sum(T value) {
    return value;
}

template <typename T, typename... Args>
T sum(T first, Args... args) {
    return first + sum(args...);
}

// Usage
int total = sum(1, 2, 3, 4, 5);  // Returns 15
```

7.3 Template Specialization

Template specialization allows you to provide a special implementation for a particular type.

Full Specialization

```
// Primary template
template <typename T>
class DataHandler {
public:
    void process(T data) {
        std::cout << "Processing generic data: " << data << std::endl;
    }
};

// Full specialization for std::string
template <>
class DataHandler<std::string> {
public:
    void process(std::string data) {
        std::cout << "Processing string data: " << data << std::endl;
        std::cout << "String length: " << data.length() << std::endl;
    }
};
```

Partial Specialization

Partial specialization is only available for class templates:

```
// Primary template
template <typename T, typename U>
```

```
class Converter {
public:
    void convert() {
        std::cout << "Generic conversion" << std::endl;
    }
};

// Partial specialization for when U is int
template <typename T>
class Converter<T, int> {
public:
    void convert() {
        std::cout << "Converting to int" << std::endl;
    }
};
```

Function Template Specialization

Function templates can only be fully specialized:

```
// Primary template
template <typename T>
void print(T value) {
    std::cout << "Generic: " << value << std::endl;
}

// Full specialization for char*
template <>
void print<char*>(char* value) {
    std::cout << "String: " << value << std::endl;
}
```

7.4 Template Metaprogramming

Template metaprogramming (TMP) is a technique that uses templates to perform computations at compile time.

Compile-Time Calculations

```
// Factorial at compile time
template <unsigned int N>
struct Factorial {
    static constexpr unsigned int value = N * Factorial<N-1>::value;
};

// Base case specialization
template <>
struct Factorial<0> {
    static constexpr unsigned int value = 1;
```

```
};

// Usage
constexpr unsigned int fact5 = Factorial<5>::value;  // 120 (computed at compile
time)
```

Type Traits

Type traits provide information about types at compile time:

```
#include <type_traits>

template <typename T>
void analyze_type() {
    if constexpr (std::is_integral_v<T>) {
        std::cout << "T is an integral type" << std::endl;
    } else if constexpr (std::is_floating_point_v<T>) {
        std::cout << "T is a floating-point type" << std::endl;
    } else {
        std::cout << "T is some other type" << std::endl;
    }
}

// Usage
analyze_type<int>();      // T is an integral type
analyze_type<double>();   // T is a floating-point type
analyze_type<std::string>();  // T is some other type
```

SFINAE (Substitution Failure Is Not An Error)

SFINAE allows you to selectively enable or disable function templates based on properties of the types:

```
// Using std::enable_if
template <typename T,
          typename = std::enable_if_t<std::is_arithmetic_v<T>>>
T safe_divide(T a, T b) {
    if (b == 0) throw std::runtime_error("Division by zero");
    return a / b;
}

// This will compile
double result = safe_divide(10.0, 2.0);

// This would cause a compile error
// std::string s1 = "Hello", s2 = "World";
// auto error = safe_divide(s1, s2); // Error: std::string is not arithmetic
```

Modern C++ Alternatives to SFINAE

In C++20, concepts provide a more readable alternative to SFINAE:

```
#include <concepts>

template <typename T>
concept Arithmetic = std::is_arithmetic_v<T>;

template <Arithmetic T>
T safe_divide(T a, T b) {
    if (b == 0) throw std::runtime_error("Division by zero");
    return a / b;
}
```

7.5 Advanced Template Techniques

Template Template Parameters

These allow you to parameterize a template with another template:

```
template <template <typename, typename> class Container,
          typename T,
          typename Allocator = std::allocator<T>>
class Stack {
private:
    Container<T, Allocator> elements;
public:
    void push(const T& value) {
        elements.push_back(value);
    }
    T pop() {
        T top = elements.back();
        elements.pop_back();
        return top;
    }
    bool empty() const {
        return elements.empty();
    }
};

// Usage with different containers
Stack<std::vector, int> vector_stack;
Stack<std::deque, double> deque_stack;
```

Curiously Recurring Template Pattern (CRTP)

CRTP is a design pattern where a class derives from a template specialized with the derived class itself:

```
template <typename Derived>
class Base {
public:
    void interface() {
        static_cast<Derived*>(this)->implementation();
    }

    // Default implementation
    void implementation() {
        std::cout << "Base implementation" << std::endl;
    }
};

class Derived : public Base<Derived> {
public:
    // Override implementation
    void implementation() {
        std::cout << "Derived implementation" << std::endl;
    }
};

// Usage
Derived d;
d.interface();  // Calls Derived::implementation()
```

Policy-Based Design

Policy-based design uses templates to compose classes from policy classes:

```
// Policy classes
class NoCheckPolicy {
public:
    static void check(int index, int size) { /* No check */ }
};

class BoundsCheckPolicy {
public:
    static void check(int index, int size) {
        if (index < 0 || index >= size) {
            throw std::out_of_range("Index out of bounds");
        }
    }
};

// Host class using policies
template <typename T, typename CheckPolicy>
class Array {
private:
    T* data;
    int size;
public:
```

```
    Array(int s) : size(s), data(new T[s]) {}
    ~Array() { delete[] data; }

    T& operator[](int index) {
        CheckPolicy::check(index, size);
        return data[index];
    }
};

// Usage with different policies
Array<int, NoCheckPolicy> fastArray(100);
Array<int, BoundsCheckPolicy> safeArray(100);
```

7.6 Real-World Applications

Container Classes

The STL containers are prime examples of template use:

```
#include <vector>
#include <list>
#include <map>

std::vector<int> numbers = {1, 2, 3, 4, 5};
std::list<std::string> names = {"Alice", "Bob", "Charlie"};
std::map<std::string, int> ages = {{"Alice", 30}, {"Bob", 25}};
```

Algorithms

The STL algorithms are function templates that work with any compatible container:

```
#include <algorithm>
#include <vector>

std::vector<int> numbers = {5, 2, 8, 1, 9};

// Sort the vector
std::sort(numbers.begin(), numbers.end());

// Find an element
auto it = std::find(numbers.begin(), numbers.end(), 8);

// Transform elements
std::transform(numbers.begin(), numbers.end(), numbers.begin(),
               [](int n) { return n * 2; });
```

Smart Pointers

Smart pointers use templates to provide type-safe memory management:

```
#include <memory>

// Unique pointer
std::unique_ptr<int> p1 = std::make_unique<int>(42);

// Shared pointer
std::shared_ptr<std::string> p2 = std::make_shared<std::string>("Hello");
std::shared_ptr<std::string> p3 = p2;  // Reference count is now 2
```

Function Objects and Lambdas

Function objects and lambdas are often used with templates:

```
#include <functional>
#include <vector>
#include <algorithm>

std::vector<int> numbers = {1, 2, 3, 4, 5};

// Using function object
std::transform(numbers.begin(), numbers.end(), numbers.begin(),
               std::negate<int>());

// Using lambda
std::for_each(numbers.begin(), numbers.end(),
              [](int& n) { n = n * n; });
```

7.7 Best Practices and Pitfalls

Best Practices

1. **Keep template code in header files**: Template definitions must be visible at the point of instantiation.
2. **Use type traits and concepts for constraints**: Make your templates work only with appropriate types.
3. **Provide clear error messages**: Use static_assert to provide meaningful error messages when template constraints are violated.

   ```
   template <typename T>
   class Vector {
       static_assert(std::is_default_constructible_v<T>,
                     "Vector requires a default-constructible type");
       // ...
   };
   ```

4. **Consider template parameter defaults**: Provide sensible defaults when appropriate.
5. **Document template requirements**: Clearly document what requirements your template parameters must satisfy.

Common Pitfalls

1. **Code bloat**: Each template instantiation creates a new copy of the code, which can lead to larger executables.
2. **Cryptic error messages**: Template errors can be notoriously difficult to understand.
3. **Slow compilation**: Templates can significantly increase compilation times.
4. **Implicit interface requirements**: Templates often have implicit requirements on the types they work with.
5. **Debugging difficulties**: Debugging templated code can be more challenging.

Mitigation Strategies

1. **Precompiled headers**: Use precompiled headers to reduce compilation times.
2. **Explicit instantiation**: Explicitly instantiate common template specializations to reduce code bloat.

```
// In a .cpp file
template class Vector<int>;
template class Vector<double>;
```

3. **Type erasure**: Use type erasure to hide template complexity at interface boundaries.
4. **Concept-based design**: In C++20, use concepts to make template requirements explicit.

Templates and generic programming are foundational to modern C++ and enable powerful abstractions while maintaining type safety and performance. Mastering these techniques is essential for writing flexible, reusable, and efficient C++ code.

Design Patterns

Design patterns are reusable solutions to common problems in software design. They represent best practices evolved over time by experienced software developers.

Creational Patterns

Creational patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation.

Singleton

Ensures a class has only one instance and provides a global point of access to it.

```
class Singleton {
private:
    static Singleton* instance;
    // Private constructor prevents direct instantiation
    Singleton() {}

public:
    // Delete copy constructor and assignment operator
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    static Singleton* getInstance() {
        if (instance == nullptr) {
            instance = new Singleton();
        }
        return instance;
    }

    void someBusinessLogic() {
        // ...
    }
};

// Initialize the static member
Singleton* Singleton::instance = nullptr;
```

Thread-safe version (C++11):

```
class Singleton {
private:
    Singleton() {}

public:
    Singleton(const Singleton&) = delete;
    Singleton& operator=(const Singleton&) = delete;

    static Singleton& getInstance() {
        // Guaranteed to be initialized only once
        static Singleton instance;
        return instance;
    }
};
```

Factory Method

Defines an interface for creating an object, but lets subclasses decide which class to instantiate.

```cpp
class Product {
public:
    virtual ~Product() {}
    virtual std::string operation() const = 0;
};

class ConcreteProductA : public Product {
public:
    std::string operation() const override {
        return "Result of ConcreteProductA";
    }
};

class ConcreteProductB : public Product {
public:
    std::string operation() const override {
        return "Result of ConcreteProductB";
    }
};

class Creator {
public:
    virtual ~Creator() {}
    virtual Product* factoryMethod() const = 0;

    std::string someOperation() const {
        Product* product = this->factoryMethod();
        std::string result = "Creator: " + product->operation();
        delete product;
        return result;
    }
};

class ConcreteCreatorA : public Creator {
public:
    Product* factoryMethod() const override {
        return new ConcreteProductA();
    }
};

class ConcreteCreatorB : public Creator {
public:
    Product* factoryMethod() const override {
        return new ConcreteProductB();
    }
};
```

Abstract Factory

Provides an interface for creating families of related or dependent objects without specifying their concrete classes.

```cpp
// Abstract products
class AbstractProductA {
public:
    virtual ~AbstractProductA() {}
    virtual std::string useProductA() const = 0;
};

class AbstractProductB {
public:
    virtual ~AbstractProductB() {}
    virtual std::string useProductB() const = 0;
};

// Concrete products
class ConcreteProductA1 : public AbstractProductA {
public:
    std::string useProductA() const override {
        return "Product A1";
    }
};

class ConcreteProductA2 : public AbstractProductA {
public:
    std::string useProductA() const override {
        return "Product A2";
    }
};

class ConcreteProductB1 : public AbstractProductB {
public:
    std::string useProductB() const override {
        return "Product B1";
    }
};

class ConcreteProductB2 : public AbstractProductB {
public:
    std::string useProductB() const override {
        return "Product B2";
    }
};

// Abstract factory
class AbstractFactory {
public:
    virtual AbstractProductA* createProductA() const = 0;
    virtual AbstractProductB* createProductB() const = 0;
    virtual ~AbstractFactory() {}
};

// Concrete factories
class ConcreteFactory1 : public AbstractFactory {
public:
```

```
    AbstractProductA* createProductA() const override {
        return new ConcreteProductA1();
    }
    AbstractProductB* createProductB() const override {
        return new ConcreteProductB1();
    }
};

class ConcreteFactory2 : public AbstractFactory {
public:
    AbstractProductA* createProductA() const override {
        return new ConcreteProductA2();
    }
    AbstractProductB* createProductB() const override {
        return new ConcreteProductB2();
    }
};
```

Builder

Separates the construction of a complex object from its representation, allowing the same construction process to create different representations.

```
class Product {
private:
    std::vector<std::string> parts;

public:
    void addPart(const std::string& part) {
        parts.push_back(part);
    }

    void showProduct() const {
        std::cout << "Product parts: ";
        for (const auto& part : parts) {
            std::cout << part << " ";
        }
        std::cout << std::endl;
    }
};

class Builder {
public:
    virtual ~Builder() {}
    virtual void buildPartA() = 0;
    virtual void buildPartB() = 0;
    virtual void buildPartC() = 0;
    virtual Product* getResult() = 0;
};

class ConcreteBuilder : public Builder {
```

```
private:
    Product* product;

public:
    ConcreteBuilder() {
        this->product = new Product();
    }

    ~ConcreteBuilder() {
        delete product;
    }

    void buildPartA() override {
        product->addPart("Part A");
    }

    void buildPartB() override {
        product->addPart("Part B");
    }

    void buildPartC() override {
        product->addPart("Part C");
    }

    Product* getResult() override {
        return product;
    }
};

class Director {
private:
    Builder* builder;

public:
    void setBuilder(Builder* builder) {
        this->builder = builder;
    }

    void buildMinimalProduct() {
        builder->buildPartA();
    }

    void buildFullProduct() {
        builder->buildPartA();
        builder->buildPartB();
        builder->buildPartC();
    }
};
```

Prototype

Creates new objects by copying an existing object, known as the prototype.

```
class Prototype {
public:
    virtual ~Prototype() {}
    virtual Prototype* clone() const = 0;
    virtual void printInfo() const = 0;
};

class ConcretePrototype1 : public Prototype {
private:
    std::string field;

public:
    ConcretePrototype1(const std::string& field) : field(field) {}

    Prototype* clone() const override {
        return new ConcretePrototype1(*this);
    }

    void printInfo() const override {
        std::cout << "ConcretePrototype1: " << field << std::endl;
    }
};

class ConcretePrototype2 : public Prototype {
private:
    std::string field;

public:
    ConcretePrototype2(const std::string& field) : field(field) {}

    Prototype* clone() const override {
        return new ConcretePrototype2(*this);
    }

    void printInfo() const override {
        std::cout << "ConcretePrototype2: " << field << std::endl;
    }
};
```

Structural Patterns

Structural patterns focus on how classes and objects are composed to form larger structures.

Adapter

Allows classes with incompatible interfaces to work together by wrapping an instance of one class with a new adapter class.

```
// Target interface
class Target {
```

```
public:
    virtual ~Target() {}
    virtual std::string request() const {
        return "Target: The default target's behavior.";
    }
};

// The class that needs adapting
class Adaptee {
public:
    std::string specificRequest() const {
        return "Adaptee: Special behavior.";
    }
};

// Adapter makes Adaptee compatible with Target
class Adapter : public Target {
private:
    Adaptee* adaptee;

public:
    Adapter(Adaptee* adaptee) : adaptee(adaptee) {}

    std::string request() const override {
        std::string result = adaptee->specificRequest();
        return "Adapter: (TRANSLATED) " + result;
    }
};
```

Bridge

Decouples an abstraction from its implementation so that the two can vary independently.

```
// Implementation
class Implementation {
public:
    virtual ~Implementation() {}
    virtual std::string operationImplementation() const = 0;
};

class ConcreteImplementationA : public Implementation {
public:
    std::string operationImplementation() const override {
        return "ConcreteImplementationA: Result on platform A.";
    }
};

class ConcreteImplementationB : public Implementation {
public:
    std::string operationImplementation() const override {
        return "ConcreteImplementationB: Result on platform B.";
```

```
    }
};

// Abstraction
class Abstraction {
protected:
    Implementation* implementation;

public:
    Abstraction(Implementation* implementation) : implementation(implementation)
{}

    virtual ~Abstraction() {}

    virtual std::string operation() const {
        return "Abstraction: Base operation with:\n" +
            implementation->operationImplementation();
    }
};

// Extended Abstraction
class ExtendedAbstraction : public Abstraction {
public:
    ExtendedAbstraction(Implementation* implementation) :
Abstraction(implementation) {}

    std::string operation() const override {
        return "ExtendedAbstraction: Extended operation with:\n" +
            implementation->operationImplementation();
    }
};
```

Composite

Composes objects into tree structures to represent part-whole hierarchies.

```
class Component {
protected:
    Component* parent;

public:
    Component() : parent(nullptr) {}
    virtual ~Component() {}

    void setParent(Component* parent) {
        this->parent = parent;
    }

    Component* getParent() const {
        return parent;
    }
```

```
    virtual bool add(Component* component) { return false; }
    virtual bool remove(Component* component) { return false; }
    virtual bool isComposite() const { return false; }

    virtual std::string operation() const = 0;
};

class Leaf : public Component {
public:
    std::string operation() const override {
        return "Leaf";
    }
};

class Composite : public Component {
private:
    std::vector<Component*> children;

public:
    bool add(Component* component) override {
        children.push_back(component);
        component->setParent(this);
        return true;
    }

    bool remove(Component* component) override {
        auto it = std::find(children.begin(), children.end(), component);
        if (it != children.end()) {
            children.erase(it);
            component->setParent(nullptr);
            return true;
        }
        return false;
    }

    bool isComposite() const override {
        return true;
    }

    std::string operation() const override {
        std::string result = "Branch(";
        for (const auto* c : children) {
            if (c == children.back()) {
                result += c->operation();
            } else {
                result += c->operation() + "+";
            }
        }
        result += ")";
        return result;
    }
};
```

Decorator

Attaches additional responsibilities to an object dynamically.

```
class Component {
public:
    virtual ~Component() {}
    virtual std::string operation() const = 0;
};

class ConcreteComponent : public Component {
public:
    std::string operation() const override {
        return "ConcreteComponent";
    }
};

class Decorator : public Component {
protected:
    Component* component;

public:
    Decorator(Component* component) : component(component) {}

    std::string operation() const override {
        return component->operation();
    }
};

class ConcreteDecoratorA : public Decorator {
public:
    ConcreteDecoratorA(Component* component) : Decorator(component) {}

    std::string operation() const override {
        return "ConcreteDecoratorA(" + Decorator::operation() + ")";
    }
};

class ConcreteDecoratorB : public Decorator {
public:
    ConcreteDecoratorB(Component* component) : Decorator(component) {}

    std::string operation() const override {
        return "ConcreteDecoratorB(" + Decorator::operation() + ")";
    }
};
```

Facade

Provides a simplified interface to a complex subsystem.

```
// Complex subsystem classes
class Subsystem1 {
public:
    std::string operation1() const {
        return "Subsystem1: Ready!\n";
    }

    std::string operationN() const {
        return "Subsystem1: Go!\n";
    }
};

class Subsystem2 {
public:
    std::string operation1() const {
        return "Subsystem2: Get ready!\n";
    }

    std::string operationZ() const {
        return "Subsystem2: Fire!\n";
    }
};

// Facade class
class Facade {
protected:
    Subsystem1* subsystem1;
    Subsystem2* subsystem2;

public:
    Facade(Subsystem1* subsystem1, Subsystem2* subsystem2)
        : subsystem1(subsystem1), subsystem2(subsystem2) {}

    ~Facade() {
        delete subsystem1;
        delete subsystem2;
    }

    std::string operation() {
        std::string result = "Facade initializes subsystems:\n";
        result += subsystem1->operation1();
        result += subsystem2->operation1();
        result += "Facade orders subsystems to perform the action:\n";
        result += subsystem1->operationN();
        result += subsystem2->operationZ();
        return result;
    }
};
```

Proxy

Provides a surrogate or placeholder for another object to control access to it.

```
class Subject {
public:
    virtual ~Subject() {}
    virtual void request() const = 0;
};

class RealSubject : public Subject {
public:
    void request() const override {
        std::cout << "RealSubject: Handling request." << std::endl;
    }
};

class Proxy : public Subject {
private:
    RealSubject* realSubject;

    bool checkAccess() const {
        // Some security checks
        std::cout << "Proxy: Checking access prior to firing a real request." <<
std::endl;
        return true;
    }

    void logAccess() const {
        std::cout << "Proxy: Logging the time of request." << std::endl;
    }

public:
    Proxy(RealSubject* realSubject) : realSubject(realSubject) {}

    ~Proxy() {
        delete realSubject;
    }

    void request() const override {
        if (this->checkAccess()) {
            this->realSubject->request();
            this->logAccess();
        }
    }
};
```

Behavioral Patterns

Behavioral patterns focus on algorithms and the assignment of responsibilities between objects.

Observer

Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

```
class Observer {
public:
    virtual ~Observer() {}
    virtual void update(const std::string& message) = 0;
};

class Subject {
private:
    std::vector<Observer*> observers;
    std::string message;

public:
    virtual ~Subject() {}

    void attach(Observer* observer) {
        observers.push_back(observer);
    }

    void detach(Observer* observer) {
        observers.erase(std::remove(observers.begin(), observers.end(), observer),
observers.end());
    }

    void notify() {
        for (Observer* observer : observers) {
            observer->update(message);
        }
    }

    void createMessage(const std::string& message) {
        this->message = message;
        notify();
    }
};

class ConcreteObserver : public Observer {
private:
    std::string observerState;
    Subject& subject;
    static int staticNumber;
    int number;

public:
    ConcreteObserver(Subject& subject) : subject(subject), number(++staticNumber)
{
        this->subject.attach(this);
        std::cout << "Observer " << number << " created" << std::endl;
    }
```

```
    ~ConcreteObserver() {
        std::cout << "Observer " << number << " removed" << std::endl;
    }

    void update(const std::string& message) override {
        observerState = message;
        std::cout << "Observer " << number << " updated: " << observerState <<
std::endl;
    }
};

int ConcreteObserver::staticNumber = 0;
```

Strategy

Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

```
class Strategy {
public:
    virtual ~Strategy() {}
    virtual std::string doAlgorithm(const std::vector<std::string>& data) const =
0;
};

class ConcreteStrategyA : public Strategy {
public:
    std::string doAlgorithm(const std::vector<std::string>& data) const override {
        std::string result;
        for (const auto& item : data) {
            result += item + " ";
        }
        return result;
    }
};

class ConcreteStrategyB : public Strategy {
public:
    std::string doAlgorithm(const std::vector<std::string>& data) const override {
        std::string result;
        for (const auto& item : data) {
            result = item + " " + result;
        }
        return result;
    }
};

class Context {
private:
    Strategy* strategy;

public:
```

```
    Context(Strategy* strategy = nullptr) : strategy(strategy) {}

    ~Context() {
        delete strategy;
    }

    void setStrategy(Strategy* strategy) {
        delete this->strategy;
        this->strategy = strategy;
    }

    void doSomeBusinessLogic() const {
        if (strategy) {
            std::cout << "Context: Sorting data using the strategy" << std::endl;
            std::string result = strategy->doAlgorithm({"a", "b", "c", "d", "e"});
            std::cout << result << std::endl;
        } else {
            std::cout << "Context: Strategy isn't set" << std::endl;
        }
    }
};
```

Command

Encapsulates a request as an object, thereby allowing for parameterization of clients with different requests, queue or log requests, and support undoable operations.

```
class Command {
public:
    virtual ~Command() {}
    virtual void execute() const = 0;
};

class SimpleCommand : public Command {
private:
    std::string payload;

public:
    SimpleCommand(std::string payload) : payload(payload) {}

    void execute() const override {
        std::cout << "SimpleCommand: Execute " << payload << std::endl;
    }
};

class Receiver {
public:
    void doSomething(const std::string& a) {
        std::cout << "Receiver: Working on (" << a << ")" << std::endl;
    }
```

```
    void doSomethingElse(const std::string& b) {
        std::cout << "Receiver: Also working on (" << b << ")" << std::endl;
    }
};

class ComplexCommand : public Command {
private:
    Receiver* receiver;
    std::string a;
    std::string b;

public:
    ComplexCommand(Receiver* receiver, std::string a, std::string b)
        : receiver(receiver), a(a), b(b) {}

    void execute() const override {
        std::cout << "ComplexCommand: Complex stuff should be done by a receiver
object" << std::endl;
        receiver->doSomething(a);
        receiver->doSomethingElse(b);
    }
};

class Invoker {
private:
    Command* onStart;
    Command* onFinish;

public:
    ~Invoker() {
        delete onStart;
        delete onFinish;
    }

    void setOnStart(Command* command) {
        onStart = command;
    }

    void setOnFinish(Command* command) {
        onFinish = command;
    }

    void doSomethingImportant() {
        if (onStart) {
            onStart->execute();
        }

        std::cout << "Invoker: doing something really important..." << std::endl;

        if (onFinish) {
            onFinish->execute();
        }
    }
};
```

Iterator

Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

```
template <typename T>
class Iterator {
public:
    virtual ~Iterator() {}
    virtual T* current() = 0;
    virtual void next() = 0;
    virtual bool hasNext() = 0;
};

template <typename T>
class Container {
public:
    virtual ~Container() {}
    virtual Iterator<T>* createIterator() = 0;
};

template <typename T>
class ConcreteIterator : public Iterator<T> {
private:
    std::vector<T*>& collection;
    size_t position;

public:
    ConcreteIterator(std::vector<T*>& collection)
        : collection(collection), position(0) {}

    T* current() override {
        return collection[position];
    }

    void next() override {
        position++;
    }

    bool hasNext() override {
        return position < collection.size();
    }
};

template <typename T>
class ConcreteContainer : public Container<T> {
private:
    std::vector<T*> collection;

public:
```

```
    void add(T* item) {
        collection.push_back(item);
    }

    Iterator<T>* createIterator() override {
        return new ConcreteIterator<T>(collection);
    }
};
```

State

Allows an object to alter its behavior when its internal state changes.

```
class Context;

class State {
protected:
    Context* context;

public:
    virtual ~State() {}

    void setContext(Context* context) {
        this->context = context;
    }

    virtual void handle1() = 0;
    virtual void handle2() = 0;
};

class Context {
private:
    State* state;

public:
    Context(State* state) : state(nullptr) {
        this->setState(state);
    }

    ~Context() {
        delete state;
    }

    void setState(State* state) {
        if (this->state != nullptr) {
            delete this->state;
        }
        this->state = state;
        this->state->setContext(this);
    }
```

```
    void request1() {
        state->handle1();
    }

    void request2() {
        state->handle2();
    }
};

class ConcreteStateA : public State {
public:
    void handle1() override;

    void handle2() override {
        std::cout << "ConcreteStateA handles request2." << std::endl;
    }
};

class ConcreteStateB : public State {
public:
    void handle1() override {
        std::cout << "ConcreteStateB handles request1." << std::endl;
    }

    void handle2() override {
        std::cout << "ConcreteStateB handles request2." << std::endl;
        std::cout << "ConcreteStateB wants to change the state of the context." <<
std::endl;
        context->setState(new ConcreteStateA());
    }
};

void ConcreteStateA::handle1() {
    std::cout << "ConcreteStateA handles request1." << std::endl;
    std::cout << "ConcreteStateA wants to change the state of the context." <<
std::endl;
    context->setState(new ConcreteStateB());
}
```

SOLID Principles

SOLID is an acronym for five design principles intended to make software designs more understandable, flexible, and maintainable.

Single Responsibility Principle

A class should have only one reason to change, meaning it should have only one job or responsibility.

Bad example:

```
class Employee {
public:
    std::string name;

    Employee(const std::string& name) : name(name) {}

    void saveToDatabase() {
        // Save employee to database
    }

    void generateReport() {
        // Generate a report about the employee
    }

    // Other employee-related methods
};
```

Good example:

```
class Employee {
public:
    std::string name;

    Employee(const std::string& name) : name(name) {}

    // Only employee-related methods
};

class EmployeeRepository {
public:
    void save(const Employee& employee) {
        // Save employee to database
    }
};

class EmployeeReportGenerator {
public:
    void generateReport(const Employee& employee) {
        // Generate a report about the employee
    }
};
```

Open/Closed Principle

Software entities should be open for extension, but closed for modification.

Bad example:

```
class Rectangle {
public:
    double width;
    double height;

    Rectangle(double width, double height) : width(width), height(height) {}
};

class AreaCalculator {
public:
    double calculateArea(const Rectangle& rectangle) {
        return rectangle.width * rectangle.height;
    }
};

// Now we need to add Circle support - we have to modify AreaCalculator
class Circle {
public:
    double radius;

    Circle(double radius) : radius(radius) {}
};

// Modified AreaCalculator
class AreaCalculator {
public:
    double calculateArea(const Rectangle& rectangle) {
        return rectangle.width * rectangle.height;
    }

    double calculateArea(const Circle& circle) {
        return 3.14159 * circle.radius * circle.radius;
    }
};
```

Good example:

```
class Shape {
public:
    virtual ~Shape() {}
    virtual double calculateArea() const = 0;
};

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

public:
    Rectangle(double width, double height) : width(width), height(height) {}
```

```
    double calculateArea() const override {
        return width * height;
    }
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double radius) : radius(radius) {}

    double calculateArea() const override {
        return 3.14159 * radius * radius;
    }
};

// AreaCalculator now works with any Shape without modification
class AreaCalculator {
public:
    double calculateArea(const Shape& shape) {
        return shape.calculateArea();
    }
};
```

Liskov Substitution Principle

Objects of a superclass should be replaceable with objects of its subclasses without affecting the correctness of the program.

Bad example:

```
class Bird {
public:
    virtual void fly() {
        std::cout << "Flying..." << std::endl;
    }
};

class Penguin : public Bird {
public:
    void fly() override {
        // Penguins can't fly!
        throw std::runtime_error("Penguins can't fly!");
    }
};

void makeBirdFly(Bird& bird) {
    bird.fly();  // This will fail with Penguin
}
```

Good example:

```
class Bird {
public:
    virtual ~Bird() {}
    // Common bird behaviors
};

class FlyingBird : public Bird {
public:
    virtual void fly() {
        std::cout << "Flying..." << std::endl;
    }
};

class NonFlyingBird : public Bird {
    // No fly method
};

class Sparrow : public FlyingBird {
    // Can use fly method
};

class Penguin : public NonFlyingBird {
    // Doesn't have fly method
};

void makeBirdFly(FlyingBird& bird) {
    bird.fly();  // Safe to call with any FlyingBird
}
```

Interface Segregation Principle

Clients should not be forced to depend on interfaces they do not use.

Bad example:

```
class Worker {
public:
    virtual void work() = 0;
    virtual void eat() = 0;
    virtual void sleep() = 0;
};

class Human : public Worker {
public:
    void work() override {
        std::cout << "Working..." << std::endl;
    }
```

```
    void eat() override {
        std::cout << "Eating..." << std::endl;
    }

    void sleep() override {
        std::cout << "Sleeping..." << std::endl;
    }
};

class Robot : public Worker {
public:
    void work() override {
        std::cout << "Working efficiently..." << std::endl;
    }

    void eat() override {
        // Robots don't eat
        throw std::runtime_error("Robots don't eat!");
    }

    void sleep() override {
        // Robots don't sleep
        throw std::runtime_error("Robots don't sleep!");
    }
};
```

Good example:

```
class Workable {
public:
    virtual void work() = 0;
};

class Eatable {
public:
    virtual void eat() = 0;
};

class Sleepable {
public:
    virtual void sleep() = 0;
};

class Human : public Workable, public Eatable, public Sleepable {
public:
    void work() override {
        std::cout << "Working..." << std::endl;
    }

    void eat() override {
        std::cout << "Eating..." << std::endl;
    }
```

```
    void sleep() override {
        std::cout << "Sleeping..." << std::endl;
    }
};

class Robot : public Workable {
public:
    void work() override {
        std::cout << "Working efficiently..." << std::endl;
    }
};
```

Dependency Inversion Principle

High-level modules should not depend on low-level modules. Both should depend on abstractions. Abstractions should not depend on details. Details should depend on abstractions.

Bad example:

```
class LightBulb {
public:
    void turnOn() {
        std::cout << "LightBulb: turned on" << std::endl;
    }

    void turnOff() {
        std::cout << "LightBulb: turned off" << std::endl;
    }
};

class Switch {
private:
    LightBulb bulb;
    bool state;

public:
    Switch() : state(false) {}

    void toggle() {
        state = !state;
        if (state) {
            bulb.turnOn();
        } else {
            bulb.turnOff();
        }
    }
};
```

Good example:

```cpp
class Switchable {
public:
    virtual ~Switchable() {}
    virtual void turnOn() = 0;
    virtual void turnOff() = 0;
};

class LightBulb : public Switchable {
public:
    void turnOn() override {
        std::cout << "LightBulb: turned on" << std::endl;
    }

    void turnOff() override {
        std::cout << "LightBulb: turned off" << std::endl;
    }
};

class Fan : public Switchable {
public:
    void turnOn() override {
        std::cout << "Fan: turned on" << std::endl;
    }

    void turnOff() override {
        std::cout << "Fan: turned off" << std::endl;
    }
};

class Switch {
private:
    Switchable& device;
    bool state;

public:
    Switch(Switchable& device) : device(device), state(false) {}

    void toggle() {
        state = !state;
        if (state) {
            device.turnOn();
        } else {
            device.turnOff();
        }
    }
};
```

Modern C++ OOP Features

C++11 and later standards introduced several new features that enhance object-oriented programming.

10.1 Lambda Expressions in Class Context

Lambda expressions provide a concise way to create anonymous function objects.

```
class Widget {
private:
    int value;

public:
    Widget(int v) : value(v) {}

    // Using lambda in a member function
    void process(const std::vector<int>& numbers) {
        // Capturing 'this' to access class members
        std::for_each(numbers.begin(), numbers.end(),
            [this](int num) {
                std::cout << "Processing " << num + value << std::endl;
            }
        );
    }

    // Storing lambda as a class member
    std::function<int(int)> transformer = [this](int x) {
        return x * value;
    };
};
```

10.2 Auto Return Type and Decltype

The `auto` and `decltype` keywords allow for more flexible return types.

```
class Calculator {
private:
    template <typename T, typename U>
    auto add_impl(T a, U b) -> decltype(a + b) {
        return a + b;
    }

public:
    // Using auto return type with trailing return type
    template <typename T, typename U>
    auto add(T a, U b) -> decltype(add_impl(a, b)) {
        return add_impl(a, b);
    }

    // C++14 allows auto return type without trailing return type
    template <typename T, typename U>
    auto multiply(T a, U b) {
        return a * b;
```

```
    }
};
```

10.3 Delegating Constructors

A constructor can call another constructor in the same class.

```
class Person {
private:
    std::string name;
    int age;
    std::string address;

public:
    // Primary constructor
    Person(const std::string& name, int age, const std::string& address)
        : name(name), age(age), address(address) {}

    // Delegating constructors
    Person(const std::string& name, int age)
        : Person(name, age, "Unknown") {}

    Person(const std::string& name)
        : Person(name, 0) {}

    Person()
        : Person("John Doe") {}
};
```

10.4 Default and Deleted Functions

Explicitly defaulting or deleting special member functions.

```
class NonCopyable {
public:
    NonCopyable() = default;  // Use compiler-generated default constructor

    // Delete copy constructor and assignment operator
    NonCopyable(const NonCopyable&) = delete;
    NonCopyable& operator=(const NonCopyable&) = delete;

    // Allow move semantics
    NonCopyable(NonCopyable&&) = default;
    NonCopyable& operator=(NonCopyable&&) = default;

    // Other members...
};
```

10.5 Constexpr and OOP

`constexpr` enables compile-time computation in classes.

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

public:
    // Constexpr constructor
    constexpr Point(int x, int y) : x(x), y(y) {}

    // Constexpr member functions
    constexpr int getX() const { return x; }
    constexpr int getY() const { return y; }

    // Constexpr static member function
    static constexpr Point origin() { return Point(0, 0); }

    // In C++14 and later, constexpr functions can have more complex logic
    constexpr Point add(const Point& other) const {
        return Point(x + other.x, y + other.y);
    }
};

// Usage at compile time
constexpr Point p1(1, 2);
constexpr Point p2(3, 4);
constexpr Point p3 = p1.add(p2);  // Computed at compile time
constexpr int x = p3.getX();      // x = 4, computed at compile time
```

Exception Handling in OOP

Exception handling is crucial for writing robust C++ code, especially in object-oriented contexts.

Exception Safety Guarantees

There are four levels of exception safety:

1. **No-throw guarantee**: Operations never throw exceptions.
2. **Strong guarantee**: If an exception occurs, the program state remains unchanged.
3. **Basic guarantee**: If an exception occurs, the program is in a valid but unspecified state.
4. **No guarantee**: If an exception occurs, the program may be in an invalid state.

```
class SafeResource {
private:
    int* data;
```

```
    size_t size;

public:
    // Constructor with basic guarantee
    SafeResource(size_t size) : size(size) {
        data = new int[size];  // May throw std::bad_alloc
    }

    // Copy constructor with strong guarantee
    SafeResource(const SafeResource& other) : size(0), data(nullptr) {
        // Allocate before modifying the object
        int* new_data = new int[other.size];

        // Copy the data
        for (size_t i = 0; i < other.size; ++i) {
            new_data[i] = other.data[i];
        }

        // Only now do we modify the object
        data = new_data;
        size = other.size;
    }

    // Destructor with no-throw guarantee
    ~SafeResource() noexcept {
        delete[] data;
    }

    // Assignment operator with strong guarantee
    SafeResource& operator=(const SafeResource& other) {
        // Copy-and-swap idiom
        SafeResource temp(other);
        std::swap(data, temp.data);
        std::swap(size, temp.size);
        return *this;
    }
};
```

RAII and Exceptions

Resource Acquisition Is Initialization (RAII) is a programming idiom used to manage resources through object lifetime.

```
class File {
private:
    FILE* handle;

public:
    File(const char* filename, const char* mode) {
        handle = fopen(filename, mode);
        if (!handle) {
```

```
            throw std::runtime_error("Failed to open file");
        }
    }

    ~File() noexcept {
        if (handle) {
            fclose(handle);
        }
    }

    // Disable copying
    File(const File&) = delete;
    File& operator=(const File&) = delete;

    // Allow moving
    File(File&& other) noexcept : handle(other.handle) {
        other.handle = nullptr;
    }

    File& operator=(File&& other) noexcept {
        if (this != &other) {
            if (handle) {
                fclose(handle);
            }
            handle = other.handle;
            other.handle = nullptr;
        }
        return *this;
    }

    // File operations...
    void write(const char* data) {
        if (fputs(data, handle) == EOF) {
            throw std::runtime_error("Failed to write to file");
        }
    }
};

// Usage with RAII
void processFile() {
    // File is automatically closed when function exits, even if an exception is
thrown
    File file("data.txt", "w");
    file.write("Hello, World!");
    // No need to manually close the file
}
```

Exception Specifications

Modern C++ recommends using `noexcept` to specify functions that won't throw exceptions.

```cpp
class Database {
public:
    // Constructor may throw
    Database(const std::string& connectionString) {
        // Initialize database connection
        if (!connect(connectionString)) {
            throw std::runtime_error("Failed to connect to database");
        }
    }

    // Destructor should never throw
    ~Database() noexcept {
        try {
            disconnect();
        } catch (...) {
            // Log error but don't propagate exception
        }
    }

    // This function guarantees it won't throw
    bool isConnected() const noexcept {
        return connected;
    }

    // This function may throw (implied by absence of noexcept)
    void execute(const std::string& query) {
        if (!connected) {
            throw std::logic_error("Database not connected");
        }
        // Execute query
    }

    // C++17 conditional noexcept
    template <typename T>
    void serialize(const T& data) noexcept(noexcept(data.serialize())) {
        data.serialize();
    }

private:
    bool connected = false;

    bool connect(const std::string& connectionString) {
        // Implementation details
        connected = true;
        return true;
    }

    void disconnect() noexcept {
        connected = false;
    }
};
```

Best Practices and Common Pitfalls

Code Organization

Organizing your code well makes it more maintainable and easier to understand.

Header and Source File Organization

```
// Widget.h
#ifndef WIDGET_H
#define WIDGET_H

class Widget {
public:
    Widget();
    ~Widget();

    void doSomething();

private:
    int data;
};

#endif // WIDGET_H

// Widget.cpp
#include "Widget.h"

Widget::Widget() : data(0) {}

Widget::~Widget() {}

void Widget::doSomething() {
    // Implementation
}
```

Pimpl Idiom (Pointer to Implementation)

This idiom reduces build dependencies and hides implementation details.

```
// Widget.h
#ifndef WIDGET_H
#define WIDGET_H

#include <memory>

class Widget {
public:
    Widget();
```

```
    ~Widget();
    Widget(Widget&& other);
    Widget& operator=(Widget&& other);

    void doSomething();

private:
    class Impl;
    std::unique_ptr<Impl> pImpl;
};

#endif // WIDGET_H

// Widget.cpp
#include "Widget.h"
#include <string>
#include <vector>

class Widget::Impl {
public:
    std::string name;
    std::vector<int> data;

    void process() {
        // Implementation details
    }
};

Widget::Widget() : pImpl(std::make_unique<Impl>()) {}

// Need to define destructor where Impl is complete
Widget::~Widget() = default;

Widget::Widget(Widget&& other) = default;
Widget& Widget::operator=(Widget&& other) = default;

void Widget::doSomething() {
    pImpl->process();
}
```

Performance Considerations

Virtual Function Overhead

Virtual functions have a small runtime cost due to virtual table lookups.

```

// Base class with virtual functions
class Shape {
public:
    virtual ~Shape() {}
    virtual double area() const = 0;
```

```
};

// Derived classes
class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }
};

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

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

    double area() const override {
        return width * height;
    }
};

// Using virtual functions
void printArea(const Shape& shape) {
    std::cout << "Area: " << shape.area() << std::endl;
}
```

Move Semantics for Performance

Using move semantics can significantly improve performance for large objects.

```
class BigObject {
private:
    std::vector<int> data;

public:
    // Constructor
    BigObject(size_t size) : data(size) {}

    // Copy constructor (expensive)
    BigObject(const BigObject& other) : data(other.data) {
        std::cout << "Copy constructor called" << std::endl;
    }

    // Move constructor (cheap)
```

```
    BigObject(BigObject&& other) noexcept : data(std::move(other.data)) {
        std::cout << "Move constructor called" << std::endl;
    }

    // Copy assignment (expensive)
    BigObject& operator=(const BigObject& other) {
        std::cout << "Copy assignment called" << std::endl;
        data = other.data;
        return *this;
    }

    // Move assignment (cheap)
    BigObject& operator=(BigObject&& other) noexcept {
        std::cout << "Move assignment called" << std::endl;
        data = std::move(other.data);
        return *this;
    }
};

// Function that returns a BigObject
BigObject createBigObject() {
    BigObject obj(1000000);
    return obj;  // Return value optimization may apply
}

// Usage
void processObject() {
    // Move semantics used here
    BigObject obj = createBigObject();

    // Without move semantics
    BigObject obj1(100);
    BigObject obj2 = obj1;  // Copy

    // With move semantics
    BigObject obj3(100);
    BigObject obj4 = std::move(obj3);  // Move (obj3 is now in a valid but
unspecified state)
}
```

Common OOP Mistakes

Slicing Problem

When a derived class object is assigned to a base class object, the derived part is "sliced off".

```
class Base {
protected:
    int baseData;

public:
```

```
    Base(int data) : baseData(data) {}

    virtual void display() const {
        std::cout << "Base: " << baseData << std::endl;
    }
};

class Derived : public Base {
private:
    int derivedData;

public:
    Derived(int baseData, int derivedData)
        : Base(baseData), derivedData(derivedData) {}

    void display() const override {
        std::cout << "Derived: Base=" << baseData
                  << ", Derived=" << derivedData << std::endl;
    }
};

// Slicing problem
void demonstrateSlicing() {
    Derived d(1, 2);
    Base b = d;  // Slicing occurs here

    d.display();  // Outputs: Derived: Base=1, Derived=2
    b.display();  // Outputs: Base: 1 (derived part is sliced off)

    // Solution: use pointers or references
    Base* bp = &d;
    bp->display();  // Outputs: Derived: Base=1, Derived=2 (no slicing)

    Base& br = d;
    br.display();  // Outputs: Derived: Base=1, Derived=2 (no slicing)
}
```

Object Lifetimes and Dangling References

```
class ResourceManager {
private:
    int* resource;

public:
    ResourceManager() : resource(new int(0)) {}

    ~ResourceManager() {
        delete resource;
    }

    // Dangerous: returns a reference to internal data
```

```
    int& getResource() {
        return *resource;
    }
};

void demonstrateDanglingReference() {
    int* dangling;

    {
        ResourceManager rm;
        int& ref = rm.getResource();
        ref = 42;

        // Store the address for later (BAD PRACTICE)
        dangling = &ref;

        // rm goes out of scope here, resource is deleted
    }

    // UNDEFINED BEHAVIOR: dangling now points to deleted memory
    std::cout << *dangling << std::endl;

    // Solution: don't store references beyond the lifetime of the object
}
```

Memory Leaks in OOP

```
class Resource {
public:
    Resource() {
        std::cout << "Resource acquired" << std::endl;
    }

    ~Resource() {
        std::cout << "Resource released" << std::endl;
    }

    void use() {
        std::cout << "Resource used" << std::endl;
    }
};

class BadManager {
private:
    Resource* resource;

public:
    BadManager() : resource(new Resource()) {}

    // Missing destructor leads to memory leak
```

```
    void useResource() {
        resource->use();
    }
};

class GoodManager {
private:
    Resource* resource;

public:
    GoodManager() : resource(new Resource()) {}

    ~GoodManager() {
        delete resource;  // Properly clean up
    }

    // Even better: use smart pointers
};

class BestManager {
private:
    std::unique_ptr<Resource> resource;

public:
    BestManager() : resource(std::make_unique<Resource>()) {}

    // No need for custom destructor

    void useResource() {
        resource->use();
    }
};
```

Interview Tips for OOP Questions

Common Interview Questions

1. **What is object-oriented programming and its main principles?**
    - Focus on encapsulation, inheritance, polymorphism, and abstraction with concrete examples.
2. **What is the difference between composition and inheritance?**
    - Composition ("has-a") vs. inheritance ("is-a") relationship.
    - When to use each (favor composition over inheritance).
3. **What is a virtual function and when would you use it?**
    - Runtime polymorphism, dynamic binding.
    - Performance considerations vs. flexibility.
4. **Explain the concept of RAII.**

- Resource management through object lifetime.
- How it provides exception safety.

5. **What are some SOLID principles and why are they important?**
    - Pick one or two principles and explain in depth with examples.

Example Answers

Inheritance vs. Composition:

```
// Inheritance example
class Animal {
public:
    virtual void makeSound() const {
        std::cout << "Animal sound" << std::endl;
    }
};

class Dog : public Animal {
public:
    void makeSound() const override {
        std::cout << "Woof!" << std::endl;
    }
};

// Composition example
class Engine {
public:
    void start() {
        std::cout << "Engine started" << std::endl;
    }
};

class Car {
private:
    Engine engine;  // Composition: Car has-an Engine

public:
    void start() {
        engine.start();
        std::cout << "Car started" << std::endl;
    }
};
```

Virtual Functions and Polymorphism:

```
class Shape {
public:
    virtual double area() const = 0;
```

```
    virtual void draw() const {
        std::cout << "Drawing a shape" << std::endl;
    }
    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    double radius;

public:
    Circle(double r) : radius(r) {}

    double area() const override {
        return 3.14159 * radius * radius;
    }

    void draw() const override {
        std::cout << "Drawing a circle" << std::endl;
    }
};

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

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

    double area() const override {
        return width * height;
    }

    void draw() const override {
        std::cout << "Drawing a rectangle" << std::endl;
    }
};

// Demonstrating polymorphism
void drawShapes(const std::vector<std::unique_ptr<Shape>>& shapes) {
    for (const auto& shape : shapes) {
        shape->draw();
        std::cout << "Area: " << shape->area() << std::endl;
    }
}
```

Key Takeaways

1. **Understand the principles deeply**: Don't just memorize definitions.
2. **Provide concrete examples**: Show how principles apply in real code.
3. **Discuss trade-offs**: Every design decision has pros and cons.

4. **Showcase modern C++ knowledge**: Demonstrate familiarity with newer standards.
5. **Relate to real-world scenarios**: Connect theory to practical applications.

Object-oriented programming in C++ offers powerful tools for building complex, maintainable software. By understanding and applying these principles and best practices, you can write code that is not only functional but also robust, efficient, and easy to maintain.