OOP in C++ - Google Interview Preparation Guide

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) {
```

```c
        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        printf("Withdrawn: $%.2f\n", amount);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++:

```cpp
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;
```

```cpp
        } 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:

```cpp
// 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:

```cpp
// 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
```

```cpp
// 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:

```cpp
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;
    }
```

```cpp
    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:

```cpp
// 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:
```

```cpp
    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:

```cpp
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
```

```
    }
};
```

Preparing for Advanced OOP

Once you have a solid understanding of the basic building blocks, you'll be better prepared to tackle more advanced OOP concepts such as:

1. **Inheritance**: Creating new classes that inherit attributes and behaviors from existing classes.
2. **Polymorphism**: Allowing objects to take on many forms depending on the context.
3. **Abstraction**: Simplifying complex systems by modeling classes based on real-world entities.
4. **Interfaces**: Defining contracts that classes must adhere to.
5. **Design Patterns**: Standard solutions to common programming problems.

Each of these concepts builds upon the fundamental understanding of classes, objects, and encapsulation.

Remember that mastering OOP is a journey. Take time to practice implementing these concepts in small projects before tackling more complex applications.

Core OOP Concepts

Classes and Objects

Classes are the blueprint for objects, which are instances of classes. In C++, they're defined with the `class` keyword:

```cpp
class Employee {
private:
    // Data members (attributes)
    std::string name;
    int id;
    double salary;

public:
    // Constructor
    Employee(std::string name, int id, double salary)
        : name(name), id(id), salary(salary) {}

    // Methods
    void giveRaise(double percentage) {
        salary += salary * (percentage/100);
    }

    // Getters & Setters
    std::string getName() const { return name; }
    void setName(std::string newName) { name = newName; }
};
```

Encapsulation

Encapsulation hides implementation details and exposes only necessary interfaces, achieved through access specifiers:

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

```
class BankAccount {
private:
    double balance; // Hidden implementation detail

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

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

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

Inheritance

Inheritance allows a class to inherit attributes and methods from another class:

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

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

    void introduce() const {
        std::cout << "Hi, I'm " << name << ", " << age << " years old." <<
std::endl;
    }
};

class Student : public Person {
private:
    std::string studentId;
```

```
    double gpa;

public:
    Student(std::string name, int age, std::string id, double gpa)
        : Person(name, age), studentId(id), gpa(gpa) {}

    void study() {
        std::cout << name << " is studying hard!" << std::endl;
    }

    // Method overriding
    void introduce() const {
        std::cout << "Hi, I'm student " << name << ", my ID is " << studentId << 
std::endl;
    }
};
```

Polymorphism

Polymorphism allows objects to be treated as instances of their parent class while maintaining their derived behavior.

Runtime Polymorphism (using virtual functions)

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

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

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, height;

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

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

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

Compile-time Polymorphism (function overloading)

```
class Calculator {
public:
    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;
    }
};
```

Abstraction

Abstraction focuses on essential qualities rather than specifics, often implemented with abstract classes:

```
class AbstractDatabase {
public:
    virtual void connect() = 0; // Pure virtual function
    virtual void disconnect() = 0;
    virtual bool execute(const std::string& query) = 0;
    virtual ~AbstractDatabase() {}
};

class MySQLDatabase : public AbstractDatabase {
public:
    void connect() override {
        // MySQL-specific connection code
    }

    void disconnect() override {
        // MySQL-specific disconnection code
    }
```

```cpp
    bool execute(const std::string& query) override {
        // MySQL-specific execution code
        return true;
    }
};
```

Advanced OOP Concepts

SOLID Principles

Single Responsibility Principle

```cpp
// Bad: Class does too much
class ReportGenerator {
public:
    void generateReport(Data data) { /* ... */ }
    void saveToFile(std::string filename) { /* ... */ }
    void sendEmail(std::string recipient) { /* ... */ }
};

// Better: Separate responsibilities
class ReportGenerator {
public:
    Report generateReport(Data data) { /* ... */ }
};

class ReportSaver {
public:
    void saveToFile(const Report& report, std::string filename) { /* ... */ }
};

class EmailSender {
public:
    void sendEmail(const Report& report, std::string recipient) { /* ... */ }
};
```

Open/Closed Principle

```cpp
// Open for extension, closed for modification
class PaymentProcessor {
public:
    virtual bool processPayment(double amount) = 0;
};

class CreditCardProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
```

```cpp
        // Credit card processing logic
        return true;
    }
};

class PayPalProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        // PayPal processing logic
        return true;
    }
};

// Adding a new payment method doesn't modify existing code
class CryptocurrencyProcessor : public PaymentProcessor {
public:
    bool processPayment(double amount) override {
        // Cryptocurrency processing logic
        return true;
    }
};
```

Liskov Substitution Principle

```cpp
class Bird {
public:
    virtual void eat() { /* ... */ }
};

class FlyingBird : public Bird {
public:
    virtual void fly() { /* ... */ }
};

class Sparrow : public FlyingBird {
    // Sparrow can fly, so this inheritance is ok
};

class Penguin : public Bird {
    // Penguins can't fly, so we don't inherit from FlyingBird
};
```

Interface Segregation Principle

```cpp
// Bad: Forces classes to implement methods they don't need
class Worker {
public:
    virtual void work() = 0;
```

```cpp
    virtual void eat() = 0;
    virtual void sleep() = 0;
};

// Better: Segregated interfaces
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 {
    // Implements all interfaces
};

class Robot : public Workable {
    // Only implements what it needs
};
```

Dependency Inversion Principle

```cpp
// High-level modules depend on abstractions, not concrete implementations
class EmailSender {
public:
    virtual void sendEmail(const std::string& to, const std::string& message) = 0;
};

class GMail : public EmailSender {
public:
    void sendEmail(const std::string& to, const std::string& message) override {
        // Gmail-specific implementation
    }
};

class Outlook : public EmailSender {
public:
    void sendEmail(const std::string& to, const std::string& message) override {
        // Outlook-specific implementation
    }
};

class NotificationService {
```

```cpp
private:
    EmailSender& emailSender; // Depends on abstraction, not implementation

public:
    NotificationService(EmailSender& sender) : emailSender(sender) {}

    void notify(const std::string& user, const std::string& message) {
        emailSender.sendEmail(user, message);
    }
};
```

Design Patterns

Singleton Pattern

```cpp
class DatabaseConnection {
private:
    // Private constructor prevents direct instantiation
    DatabaseConnection() {
        // Initialize connection
    }

    // Static instance
    static DatabaseConnection* instance;

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

    // Access point for singleton instance
    static DatabaseConnection& getInstance() {
        if (instance == nullptr) {
            instance = new DatabaseConnection();
        }
        return *instance;
    }

    void query(const std::string& sql) {
        // Execute query
    }
};

// Initialize static member
DatabaseConnection* DatabaseConnection::instance = nullptr;
```

Factory Pattern

```cpp
class Vehicle {
public:
    virtual void drive() const = 0;
    virtual ~Vehicle() {}
};

class Car : public Vehicle {
public:
    void drive() const override {
        std::cout << "Driving a car" << std::endl;
    }
};

class Truck : public Vehicle {
public:
    void drive() const override {
        std::cout << "Driving a truck" << std::endl;
    }
};

class VehicleFactory {
public:
    static Vehicle* createVehicle(const std::string& type) {
        if (type == "car") {
            return new Car();
        } else if (type == "truck") {
            return new Truck();
        }
        return nullptr;
    }
};
```

Observer Pattern

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

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

public:
    void addObserver(Observer* observer) {
        observers.push_back(observer);
    }

    void removeObserver(Observer* observer) {
```

```cpp
        // Remove observer from vector
    }

    void notifyObservers(const std::string& message) {
        for (Observer* observer : observers) {
            observer->update(message);
        }
    }
};

class NewsAgency : public Subject {
private:
    std::string news;

public:
    void setNews(const std::string& newNews) {
        news = newNews;
        notifyObservers(news);
    }
};

class NewsChannel : public Observer {
private:
    std::string name;

public:
    NewsChannel(const std::string& channelName) : name(channelName) {}

    void update(const std::string& message) override {
        std::cout << name << " received news: " << message << std::endl;
    }
};
```

Memory Management

Resource Acquisition Is Initialization (RAII)

```cpp
class FileHandler {
private:
    FILE* file;

public:
    FileHandler(const std::string& filename) {
        file = fopen(filename.c_str(), "r");
        if (!file) {
            throw std::runtime_error("Failed to open file");
        }
    }

    ~FileHandler() {
        if (file) {
```

```
            fclose(file);  // Resource automatically released when object is
destroyed
        }
    }

    // Prevent copying to maintain RAII guarantees
    FileHandler(const FileHandler&) = delete;
    FileHandler& operator=(const FileHandler&) = delete;

    // Read data from file
    std::string readLine() {
        // Implementation
        return "";
    }
};
```

Smart Pointers

```
class Resource {
public:
    Resource() { std::cout << "Resource acquired" << std::endl; }
    ~Resource() { std::cout << "Resource released" << std::endl; }
    void use() { std::cout << "Resource being used" << std::endl; }
};

void useUniquePtr() {
    // Unique ownership - resource freed when ptr goes out of scope
    std::unique_ptr<Resource> ptr = std::make_unique<Resource>();
    ptr->use();

    // Transfer ownership - unique_ptr can't be copied
    std::unique_ptr<Resource> ptr2 = std::move(ptr);

    // ptr is now null, ptr2 owns the resource
    if (ptr2) ptr2->use();
}

void useSharedPtr() {
    // Shared ownership - resource freed when all ptrs go out of scope
    std::shared_ptr<Resource> ptr1 = std::make_shared<Resource>();

    {
        std::shared_ptr<Resource> ptr2 = ptr1; // Reference count = 2
        ptr2->use();
    } // ptr2 destroyed, reference count = 1

    ptr1->use(); // Still valid
} // ptr1 destroyed, reference count = 0, resource freed

void useWeakPtr() {
    std::shared_ptr<Resource> shared = std::make_shared<Resource>();
```

```cpp
    std::weak_ptr<Resource> weak = shared;

    // Check if resource still exists
    if (auto temp = weak.lock()) {
        temp->use();
    } else {
        std::cout << "Resource no longer available" << std::endl;
    }

    shared.reset(); // Resource destroyed

    // weak_ptr doesn't keep resource alive
    if (weak.expired()) {
        std::cout << "Resource has been destroyed" << std::endl;
    }
}
```

Multiple Inheritance and the Diamond Problem

```cpp
class Device {
protected:
    std::string serialNumber;

public:
    Device(const std::string& serial) : serialNumber(serial) {}
    virtual void turnOn() = 0;
};

class Printer : public virtual Device {
public:
    Printer(const std::string& serial) : Device(serial) {}
    void turnOn() override { std::cout << "Printer on" << std::endl; }
    void print() { std::cout << "Printing" << std::endl; }
};

class Scanner : public virtual Device {
public:
    Scanner(const std::string& serial) : Device(serial) {}
    void turnOn() override { std::cout << "Scanner on" << std::endl; }
    void scan() { std::cout << "Scanning" << std::endl; }
};

// Without virtual inheritance, would have two copies of Device
class PrinterScanner : public Printer, public Scanner {
public:
    PrinterScanner(const std::string& serial)
        : Device(serial), Printer(serial), Scanner(serial) {}

    void turnOn() override {
        std::cout << "PrinterScanner on" << std::endl;
        // Or call specific parent implementations
```

```
        // Printer::turnOn();
        // Scanner::turnOn();
    }
};
```

Templates and Generic Programming

```
template<typename T>
class Stack {
private:
    std::vector<T> elements;

public:
    void push(const T& item) {
        elements.push_back(item);
    }

    T pop() {
        if (elements.empty()) {
            throw std::out_of_range("Stack underflow");
        }

        T top = elements.back();
        elements.pop_back();
        return top;
    }

    bool isEmpty() const {
        return elements.empty();
    }

    size_t size() const {
        return elements.size();
    }
};

// Specialization for string type
template<>
class Stack<std::string> {
private:
    std::vector<std::string> elements;

public:
    void push(const std::string& item) {
        std::string copy = item;
        elements.push_back(std::move(copy));
    }

    std::string pop() {
        if (elements.empty()) {
            throw std::out_of_range("Stack underflow");
```

```
        }

        std::string top = std::move(elements.back());
        elements.pop_back();
        return top;
    }

    bool isEmpty() const {
        return elements.empty();
    }

    size_t size() const {
        return elements.size();
    }

    // String-specific method
    size_t totalLength() const {
        size_t total = 0;
        for (const auto& str : elements) {
            total += str.length();
        }
        return total;
    }
};
```

Interview Tips for OOP Questions

1. **Focus on design decisions**: Explain why you chose certain patterns or principles.
2. **Consider trade-offs**: Discuss advantages and disadvantages of your design choices.
3. **Show SOLID understanding**: Demonstrate how your solution adheres to SOLID principles.
4. **Be prepared for extensions**: Interviewers often ask how you'd extend your solution for new requirements.
5. **Watch for edge cases**: Consider memory management, thread safety, and error handling.