Advanced Object-Oriented Design in C++

Composition, Encapsulation, and Best Practices

LEARNING OBJECTIVES

What We'll Cover Today

Composition

Building complex systems with HAS-A relationships

Encapsulation

Separating interface from implementation

API Design

Creating clear, safe interfaces with const-correctness

Advanced Keywords

Understanding this, friend, and static

Design Principles

Applying OOP best practices

Code Review

Effective peer feedback techniques

SECTION 1

Composition and HAS-A Design

What is Composition?

Composition is the practice of building objects from other objects as data members. It represents a HAS-A relationship, distinct from IS-A relationships found in inheritance.

Real-World Analogy

A student organization HAS-A roster of members, a calendar of events, and officer positions. Each component is a separate object that the organization contains and manages.

- Members are independent objects
- Events exist as their own entities
- Organization coordinates these parts

Key Concept

HAS-A vs. IS-A

Composition creates flexible, maintainable systems by combining simple building blocks into complex structures.

Composition Example

Student Organization System

```
class Member {
    std::string name;
    std::string major;
    int graduationYear;
    bool isDuesActive;
public:
    // Member interface...
};

class StudentOrganization {
    std::vector<Member> members;      // HAS-A
    std::vector<Event> upcomingEvents; // HAS-A
public:
    void addMember(const Member& member);
    Member* findMember(const std::string& name);
    void scheduleEvent(const Event& event);
};
```

The StudentOrganization class demonstrates composition by containing vectors of Member and Event objects, establishing clear HAS-A relationships.

Composition Design Principles

Ownership

Determine who creates and destroys the contained objects. Clear ownership prevents memory leaks and undefined behavior.

Encapsulation

Keep internal objects private. External code shouldn't directly access or modify contained objects without going through the container's interface.

Responsibility

Each class should have a clear, single purpose. Member manages member data, StudentOrganization coordinates members.

Delegation

StudentOrganization delegates member-specific operations to the Member class rather than implementing them itself.

Benefits of Composition

Code Reuse

Use existing classes as building blocks without rewriting functionality.

Flexibility

Easier to modify than deep inheritance hierarchies. Change components independently.

Maintainability

Changes to Member don't require changing StudentOrganization interface.

Testing

Test each component independently before integrating into larger systems.

SECTION 1.5

Destructors & Resource Management

What is a Destructor?

Definition & Purpose

- A special member function that executes automatically when an object is destroyed.
- Its primary purpose is to clean up resources (e.g., memory, file handles, network connections) acquired during the object's lifetime.

Automatic Invocation

Destructors are automatically called when an object goes out of scope (for stack-allocated objects) or is explicitly deleted (for heap-allocated objects).

Syntax & Example

```
class MyClass {
public:
    MyClass() { /* Constructor */ }
    ~MyClass() { /* Destructor */ } // Note the tilde (~)
};

void func() {
    MyClass obj1; // Constructor called
    // ...
} // Destructor for obj1 called here

MyClass* obj2 = new MyClass(); // Constructor called
// ...
delete obj2; // Destructor for obj2 called here
```

The destructor uses a tilde (~) prefix before the class name. It ensures proper resource management and prevents leaks.

Destructors in Action

The Problem: Memory Leaks

When an object allocates dynamic memory (e.g., using `new`), but doesn't explicitly free it before being destroyed, a memory leak occurs. The allocated memory remains occupied and unusable.

```
class Member {
public:
    std::string* name; // Pointer to dynamically allocated string

    Member(const std::string& n) {
        name = new std::string(n); // Allocate memory for name
        std::cout << "Member " << *name << " created." << std::endl;
    }
    // No custom destructor: *name memory leaks when Member
object is destroyed
};

void simulateLeak() {
    Member m("Alice"); // Constructor called, memory allocated
    // When 'm' goes out of scope, no destructor is called to free
'name'
    // This leads to a memory leak.
}
```

In `simulateLeak()`, the memory for "Alice" is allocated but never released, consuming system resources unnecessarily.

The Solution: Proper Deallocation

A destructor is explicitly defined to free any dynamically allocated resources. It ensures cleanup happens automatically when the object's lifetime ends.

```
class Member {
public:
    std::string* name;

    Member(const std::string& n) {
        name = new std::string(n);
        std::cout << "Member " << *name << " created." << std::endl;
    }

    // Destructor: called automatically when object is destroyed
    ~Member() {
        delete name; // Deallocate memory pointed to by 'name'
        name = nullptr; // Good practice to prevent dangling pointers
        std::cout << "Member destroyed, memory freed." << std::endl;
    }
};

void solveLeak() {
    Member m("Bob"); // Constructor called, memory allocated
    // When 'm' goes out of scope, ~Member() is called automatically
    // The memory for 'name' is properly freed.
}
```

With the destructor, when `m` in `solveLeak()` is destroyed, its allocated memory for `name` is correctly released.

Destructors and Composition

When a composed object is destroyed, its components' destructors are called in the reverse order of their construction. This ensures a complete and orderly cleanup of all associated resources.

1. Composed Object Destruction

The `StudentOrganization` object's destructor is invoked when it goes out of scope or is explicitly deleted.

2. Component Destructors Triggered

The `StudentOrganization`'s destructor automatically handles the destruction of its contained components, such as a `std::vector<Member>`.

3. Individual Component Cleanup

For each `Member` object within the vector, its own destructor is called, freeing any dynamically allocated memory or resources it managed.

This structured approach, facilitated by proper destructor design in composed classes, guarantees that all resources are cleaned up correctly, preventing memory leaks and ensuring application stability.

Destructor Best Practices

Following these guidelines ensures robust and leak-free resource management in your C++ applications.

Keep it Simple

Avoid complex logic; destructors should strictly manage resource deallocation, not perform business operations.

No Exceptions

Throwing exceptions from a destructor can lead to undefined behavior or program termination, especially during stack unwinding.

Virtual for Inheritance

Ensures the correct destructor is called when deleting a derived class object through a base class pointer, preventing resource leaks.

Embrace RAII

Manage resources through objects that acquire them in constructors and release them in destructors, guaranteeing proper cleanup.

Use Smart Pointers

Automate memory management and reduce common errors like leaks or double-deletions by leveraging `unique_ptr` and `shared_ptr`.

Nullify Pointers

After `delete`, set raw pointers to `nullptr` to prevent dangling pointer issues and clarify that the memory is no longer valid.

SECTION 2

Separating Interface from Implementation

Why Separate Interface and Implementation?

The Problem

Single-file classes expose implementation details to client code. Every change requires recompiling all dependent code, making maintenance difficult and time-consuming.

- Client sees private details
- Compilation dependencies multiply
- Harder to understand and maintain
- Slows development cycle

The Solution

Separate header files (.h) from source files (.cpp). Headers declare what the class can do, while source files implement how it does it.

- Clean public interface
- Hidden implementation details
- Faster compilation
- Better organization

Header Files (.h) - The Interface

Member.h - What the class can do

```
#ifndef MEMBER_H
#define MEMBER_H
#include <string>

class Member {
public:
    Member(std::string name, std::string major, int gradYear);
    void payDues();
    bool isDuesActive() const;
    std::string getName() const;
    int getGraduationYear() const;

private:
    std::string m_name;
    std::string m_major;
    int m_graduationYear;
    bool m_duesActive;
};

#endif
```

The header file provides the public interface while keeping implementation details private. Include guards prevent multiple definitions.

Source Files (.cpp) - The Implementation

Member.cpp - How it does it

```
#include "Member.h"

Member::Member(std::string name, std::string major, int gradYear)
    : m_name{name}, m_major{major}, m_graduationYear{gradYear},
      m_duesActive{false} {}

void Member::payDues() {
    m_duesActive = true;
}

bool Member::isDuesActive() const {
    return m_duesActive;
}

std::string Member::getName() const {
    return m_name;
}

int Member::getGraduationYear() const {
    return m_graduationYear;
}
```

The source file contains the actual implementation. Changes here don't affect client code that only includes the header.

Common Pitfalls and Solutions

1

Multiple Definition Errors

Use include guards (#ifndef/#define/#endif) in every header file to prevent the same code from being included multiple times.

2

Missing Includes

Each file should include what it uses. Don't rely on transitive includes from other headers.

3

Circular Dependencies

Use forward declarations when two classes reference each other. Declare the class name before defining it.

4

Linking Errors

Make sure all .cpp files are compiled and linked together. Missing source files cause undefined reference errors.

SECTION 3

Designing Clear Public APIs

What Makes a Good API?

A well-designed API communicates intent clearly and prevents misuse through thoughtful design choices.

Meaningful Names

Self-documenting names that clearly describe purpose

Appropriate Types

Parameters and return values match usage patterns

Const-Correctness

Use const to communicate immutability intent

Minimal but Complete

Provide necessary functionality without bloat

API Design: Before and After

❌ Unclear Intent

```
class Event {
public:
    void process(int x, std::string y);
    // What are x and y?

    int get();
    // Get what?

    void change(Event e);
    // Copies entire object!
};
```

Vague names and poor parameter choices make this API confusing and error-prone.

✅ Clear, Safe Design

```
class Event {
public:
    void scheduleEvent(
        int attendeeCount,
        const std::string& location);

    int getEventId() const;

    void updateLocation(
        const std::string& newLocation);
};
```

Descriptive names and const references create a self-documenting, efficient interface.

API Design Checklist

Descriptive Method Names

Names should describe actions or queries clearly. Avoid abbreviations and single-letter names.

Meaningful Parameters

Use descriptive parameter names and appropriate types. Pass by const reference for efficiency.

Appropriate Return Types

Return types should match what callers need. Consider const references for large objects.

Const Wherever Possible

Mark methods const when they don't modify state. Use const parameters to prevent changes.

Hide Implementation

Avoid exposing internal implementation details through the public interface.

SECTION 4

Const-Correctness

What is Const-Correctness?

Const-correctness uses the const keyword to prevent unintended changes to data. The compiler enforces immutability, catching bugs at compile time rather than runtime.

Const Objects

Cannot be modified after creation

Const Member Functions

Promise not to modify object state

Const Parameters

Prevent function from changing arguments

Compiler Enforcement

Catches violations at compile time

Const Objects

```
const Member president{"Kamala Harris", "Political Science", 2026};

president.payDues(); //
```

Const Member Functions

```
class Member {
public:
    std::string getName() const {  // Promises not to modify
        return m_name;
    }

    void payDues() {  // NOT const - modifies state
        m_duesActive = true;
    }

private:
    std::string m_name;
    bool m_duesActive;
};
```

Const member functions promise not to modify the object's state. They can be called on const objects, while non-const functions cannot. This distinction helps the compiler enforce correctness and enables optimizations.

Const Parameters

Pass by Const Reference

```
// Efficient and safe
void printMemberInfo(
 const Member& member) {

 std::cout << member.getName();
 //
```

Const-Correctness Best Practices

01

Mark Non-Modifying Functions Const

All member functions that don't change object state should be marked const.

02

Use Const References for Parameters

Pass objects by const reference to avoid expensive copies while preventing modifications.

03

Make Objects Const When Appropriate

Declare objects const when they shouldn't change after initialization.

04

Enable Compiler Optimizations

Const-correctness improves code safety and enables compiler optimizations.

SECTION 5

The this Pointer

Understanding Pointers

In C++, a **pointer** is a special type of variable that stores the memory address of another variable. Think of it like a street address:

It doesn't hold the house itself, but rather the exact location where you can find the house.

Key Operations:

- `&` (Address-of Operator): Used to get the memory address of a variable.
- `*` (Dereference Operator): Used to access the value stored at the memory address a pointer holds.

Pointers allow for indirect access to objects, enabling powerful features like dynamic memory allocation and efficient data manipulation. They are also fundamental to understanding concepts like the **this pointer**.

Basic Syntax & Example:

```
int myValue = 42;  // Declare an integer variable

// Declare a pointer to an integer.
// 'ptr' will store the memory address of an 'int'.
int* ptr;

// Assign the address of 'myValue' to 'ptr'
ptr = &myValue;

// Now, let's use the pointer:
std::cout << "Value: " << myValue << std::endl;
// Output: 42

std::cout << "Address of myValue: " << &myValue << std::endl;
// Output: (memory address, e.g., 0x7ffee5c01b1c)

std::cout << "Pointer holds: " << ptr << std::endl;
// Output: (same memory address as &myValue)

std::cout << "Value at pointer: " << *ptr << std::endl;
// Output: 42 (dereferences 'ptr' to get the value)
```

What is this?

The this pointer is implicitly available in all non-static member functions. It points to the object on which the member function is called.

Key Properties

- Type: ClassName* const
- Available in member functions
- Points to current object
- Usually implicit

When It's Useful

While this is always available, explicit use is only necessary in specific scenarios: resolving name conflicts, returning the current object, or passing it to other functions.

Implicit vs. Explicit Use

```
class Member {
    std::string m_name;

public:
    void setName(const std::string& name) {
        // Implicit use (normal, preferred)
        m_name = name;

        // Explicit use (usually unnecessary)
        this->m_name = name;
    }
};
```

In most cases, implicit use of this is clearer and more idiomatic. The compiler automatically uses this to access member variables. Explicit use is only needed in specific situations.

Resolving Name Ambiguity

Using this

```
class Member {
    std::string name;

public:
    void setName(std::string name) {
        this->name = name;
        // Distinguish from parameter
    }
};
```

When parameter names match member names, this resolves the ambiguity.

Better Approach

```
class Member {
    std::string m_name;

public:
    void setName(std::string name) {
        m_name = name;
        // No ambiguity
    }
};
```

Using member prefixes (m_) avoids ambiguity without needing this.

Cascading Member Functions

Fluent Interface Pattern

```
class Event {
 std::string m_location;
 int m_capacity;
 std::string m_description;

public:
 Event& setLocation(const std::string& loc) {
 m_location = loc;
 return *this; // Return reference to current object
 }

 Event& setCapacity(int capacity) {
 m_capacity = capacity;
 return *this;
 }

 Event& setDescription(const std::string& desc) {
 m_description = desc;
 return *this;
 }
};

// Enable chaining:
event.setLocation("Student Union")
 .setCapacity(100)
 .setDescription("Welcome Back Mixer");
```

When to Use this

✅ Resolving Name Conflicts

When parameter names match member names (though prefixes are better)

✅ Returning Current Object

For cascading/fluent interfaces that chain method calls

✅ Passing to Other Functions

When you need to pass the current object as an argument

❌ Overusing Explicit this

Avoid using this-> everywhere when implicit access is clearer

SECTION 6

Friend Functions and Classes

What is a friend?

The friend keyword grants special access to private and protected members of a class. Friends bypass normal encapsulation rules.

```
class Member {
private:
    bool m_duesActive;

    friend void generateMembershipReport(const Member& member);
    friend class StudentOrganization;
};

void generateMembershipReport(const Member& member) {
    std::cout << "Dues Active: " << member.m_duesActive;
    // Can access private members
}
```

Friend declarations appear inside the class but grant external functions or classes access to private data.

When Friends Make Sense

Operator Overloading

Friend functions enable natural syntax for operators like operator<< for output streams.

Testing Frameworks

Test frameworks may need internal access to verify private state without exposing it publicly.

Tightly Coupled Classes

Classes working together closely, like StudentOrganization managing Member status, may justify friend access.

```
class Event {
 friend std::ostream& operator<<(std::ostream&, const Event&);
};
```

The Risks of Friendship

Friend declarations break encapsulation and create tight coupling between classes. Use them sparingly.

❌ Breaks Encapsulation

Friend functions can access all private members, violating information hiding principles.

❌ Implementation Coupling

Changes to private implementation affect friends, increasing maintenance burden.

❌ Harder to Maintain Invariants

Friends can bypass validation logic, potentially creating invalid object states.

❌ Can't Restrict Access

Once granted, friend access is all-or-nothing. You can't limit which private members are accessible.

Guideline: Use friends sparingly and only when necessary. Prefer public interfaces when possible.

Alternatives to friend

❌ Using friend

```
class Member {
private:
    bool m_duesActive;
    std::string m_name;

    friend void generateReport(
        const Member& member);
};
```

Friend access exposes all private data, creating tight coupling.

✅ Using Public Interface

```
class Member {
public:
    bool isDuesActive() const {
        return m_duesActive;
    }
    std::string getName() const {
        return m_name;
    }
};

void generateReport(const Member& m) {
    std::cout << m.getName() << ": "
        << (m.isDuesActive() ?
            "Active" : "Inactive");
}
```

Controlled access through public interface maintains encapsulation.

SECTION 7

Static Members

What are Static Members?

Static members belong to the class itself rather than to individual objects. One copy is shared by all instances.

Regular Members

Each object has its own copy of regular data members. Creating 100 Member objects means 100 separate copies of m_name, m_memberId, etc.

Static Members

Only one copy exists regardless of how many objects are created. All instances share the same static data.

Static Data Members

```
class Member {
private:
    static int s_nextMemberId;  // Shared by all
    int m_memberId;             // Unique per instance
    std::string m_name;

public:
    Member(std::string name) : m_name{name} {
        m_memberId = s_nextMemberId++;
    }

    int getMemberId() const { return m_memberId; }
};

// Must define static members outside class
int Member::s_nextMemberId = 1000;
```

Static data members must be defined in a .cpp file. The static keyword appears only in the class declaration, not in the definition.

Static Member Functions

```
class StudentOrganization {
private:
    static int s_totalOrganizations;
    static int s_totalMembers;

public:
    static int getTotalOrganizations() {  // No 'this' pointer
        return s_totalOrganizations;
    }

    static int getTotalMembers() {
        return s_totalMembers;
    }
};

// Call without an object:
int orgCount = StudentOrganization::getTotalOrganizations();
```

Static member functions can be called without an object instance. They have no this pointer and can only access static members.

When to Use Static Members

Tracking All Instances

Count total objects created, assign unique IDs, or maintain registries of all instances.

Shared Configuration

Constants or settings shared across all instances of the class.

Utility Functions

Class-related functions that don't need object state, like Event::formatDate(date).

Factory Methods

Static functions that create and return instances of the class with specific configurations.

Static Members Best Practices

Use for Class-Wide Data

Static members should represent information about the class as a whole, not individual instances.

Prefer Static Members Over Globals

Static member functions are better than global functions because they're scoped to the class.

Initialize in .cpp File

Define and initialize static data members in the source file, not the header.

Document Shared State

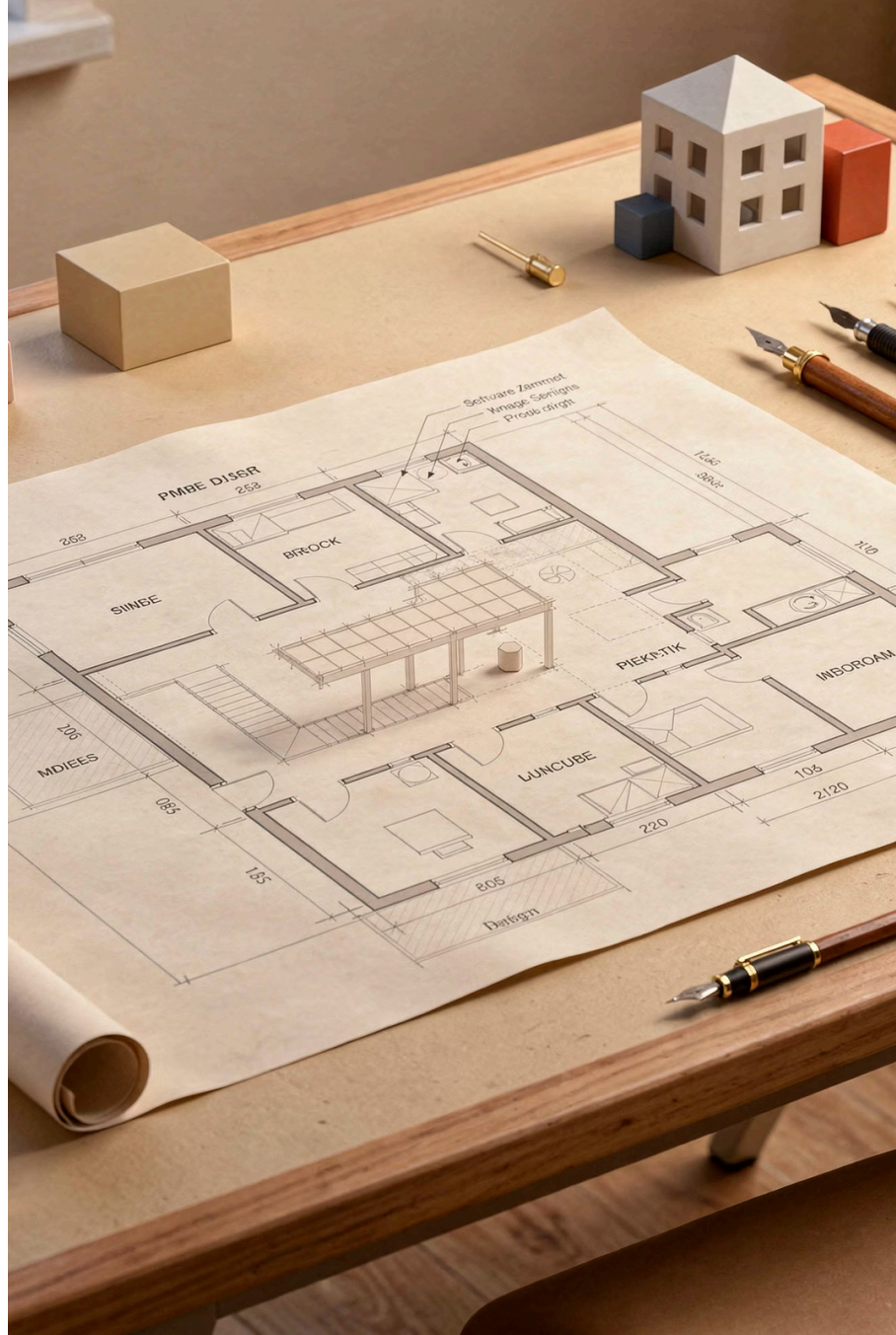
Clearly document when static state is shared across instances to avoid confusion.

⚠️ Thread Safety Concerns

Be cautious with mutable static data in multi-threaded programs. Consider synchronization.

SECTION 8

Design Best Practices



Key OOP Design Principles

SOLID principles guide the creation of maintainable, flexible object-oriented systems.

Single Responsibility

Each class does one thing well

Low Coupling

Classes minimize dependencies on each other

Strong Invariants

Class maintains valid state at all times

Consistent Style

Follow naming and formatting conventions

Single Responsibility Principle

❌ Bad Design - Multiple Responsibilities

```
class Member {
    void updateContactInfo();
    void payDues();
    void sendEmail();
    void generateTranscript();
    void saveToDatabase();
};
```

This class handles member data, payments, email, transcripts, and database operations - too many responsibilities.

✅ Good Design - Clear Responsibilities

```
class Member {
    // Manage member data
};

class MembershipPayments {
    // Handle dues
};

class EmailService {
    // Send notifications
};

class Database {
    // Persistence
};
```

Each class has a single, well-defined purpose.

Low Coupling Example

❌ Tightly Coupled (Fragile)

```
class Event {
    MySQLDatabase db;

    void save() {
        db.save(this);
    }
};
```

Event is hardcoded to use MySQL. Changing databases requires modifying Event class.

✅ Loosely Coupled (Flexible)

```
class Event {
    void save(Database& db) {
        db.save(this);
    }
};
```

Event accepts any database through dependency injection. Easy to swap implementations.

Low coupling makes systems more flexible and easier to test. Changes to one component don't ripple through the entire system.

Strong Class Invariants

Maintaining Valid State

```
class Member {
private:
 int m_graduationYear;
 std::string m_name;

public:
 Member(std::string name, int gradYear) {
 if (gradYear < 2020 || gradYear > 2030) {
 throw std::invalid_argument("Invalid graduation year");
 }
 if (name.empty()) {
 throw std::invalid_argument("Name cannot be empty");
 }
 m_name = name;
 m_graduationYear = gradYear;
 }

 void setGraduationYear(int year) {
 if (year < 2020 || year > 2030) {
 throw std::invalid_argument("Invalid graduation year");
 }
 m_graduationYear = year;
 }

 // Invariant: graduation year is always valid, name never empty
};
```

Code Style Consistency

```
class StudentOrganization {  // PascalCase for classes
private:
    std::string m_organizationName;  // m_ prefix for members
    std::vector<Member> m_members;
    int m_foundingYear;

public:
    void addMember(const Member& member);  // camelCase
    int getMemberCount() const;
    std::string getOrganizationName() const;
};
```

Why Consistency Matters

Code is read more often than written. Consistent naming and formatting make code easier to understand, maintain, and collaborate on. Choose a style and stick to it throughout your project.

Common Conventions

- PascalCase for classes
- camelCase for functions
- m_ prefix for members
- s_ prefix for static

Design Decision Example

Scenario: Student Organization Management System

1

Decision 1: Composition vs. Inheritance

✅ StudentOrganization HAS-A list of Members (composition)

❌ StudentOrganization IS-A List (doesn't make logical sense)

2

Decision 2: Handling Duplicates

✅ Strong invariant: Check for duplicates before adding

✅ Clear error: Throw exception if already exists

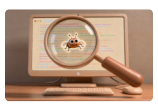
❌ Silently fail or return bool (unclear to caller)

Good design decisions are based on logical relationships, clear error handling, and maintainability considerations.

SECTION 9

Peer Code Review

Why Peer Code Review?

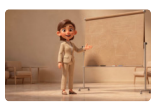


Catch Bugs Early

Find issues before they reach production, when they're cheaper and easier to fix.

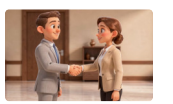
Learn from Others

Discover new approaches, techniques, and solutions by reviewing peers' code.



Practice Communication

Develop skills in articulating design decisions and technical concepts.



Professional Skills

Code review is a standard practice in industry. Build these skills early.

Effective Code Review - Giving Feedback

Framework: Actionable, Respectful, Criteria-Based

✅ Good Feedback

> "The processEvent function is 80 lines long, which violates the single responsibility principle. Consider extracting attendance tracking into a separate function."

Specific, actionable, explains the problem and suggests a solution.

❌ Poor Feedback

> "This code is messy and hard to read."

Vague, unhelpful, doesn't explain what's wrong or how to improve.

Code Review Criteria

01

Correctness

Does it work? Are edge cases handled? Are there potential bugs or logic errors?

02

Clarity

Can someone else understand it? Are names meaningful? Is the logic clear?

03

Structure

Is it organized logically? Are functions appropriately sized? Is there good separation of concerns?

04

Style

Consistent naming, formatting, and conventions? Follows project standards?

05

Design

SOLID principles followed? Appropriate use of composition, encapsulation, const-correctness?

Code Review Example

Original Submission

```
class Org {
    int cnt;
    string nm;

public:
    void a(string n) {
        nm = n;
    }

    int g() {
        return cnt;
    }
};
```

Review Comments

1. **Clarity:** Class/method names are unclear. Suggest StudentOrganization, addMember, getMemberCount
2. **Encapsulation:** Data members should be initialized via constructor
3. **Const-correctness:** getMemberCount() should be const
4. **Type:** Use std::string with full namespace

Receiving and Using Feedback

✅ Read Carefully

Approach feedback non-defensively. It's about improving code, not criticizing you.

✅ Ask Questions

If feedback is unclear, ask for clarification. Understanding is key to improvement.

✅ Prioritize

Address critical issues first (correctness, security) before style concerns.

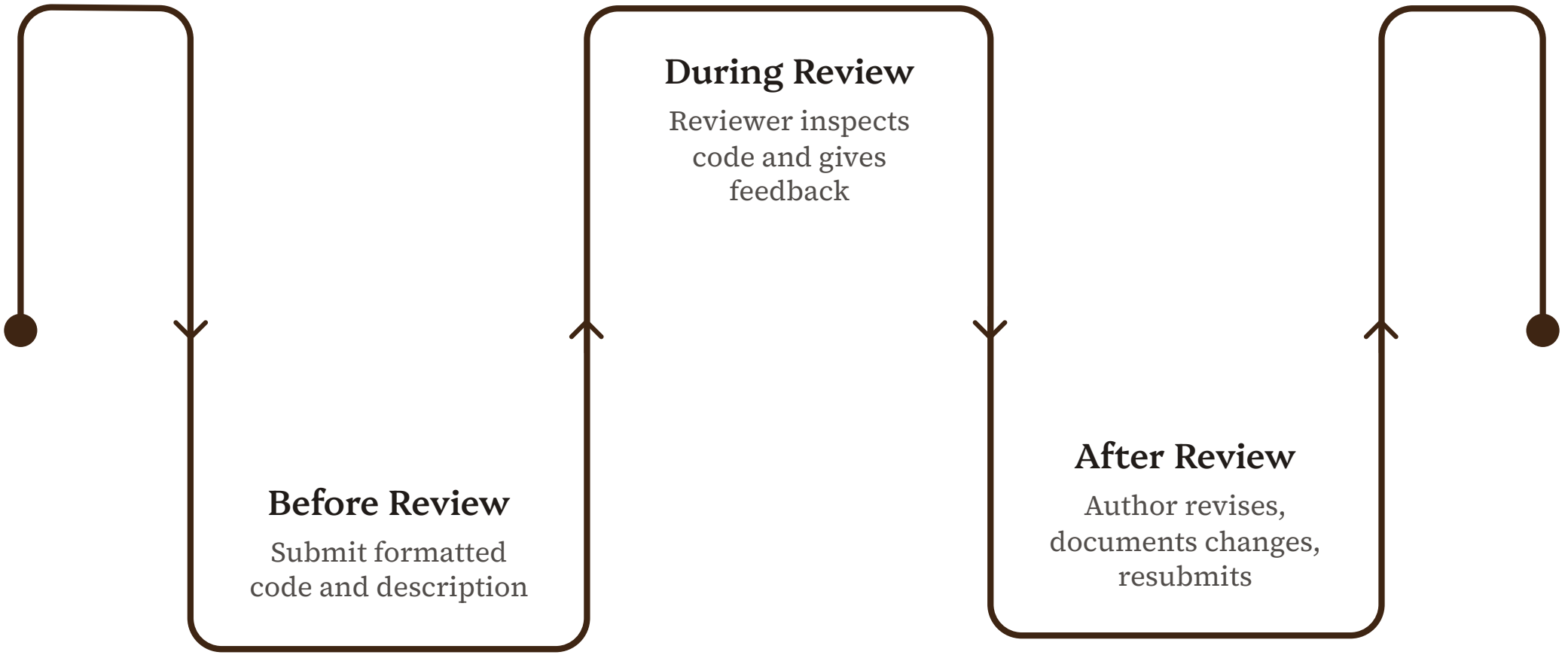
✅ Document Changes

Revise code, then explain what changed and why in your resubmission.

✅ Thank Reviewers

Acknowledge the time and effort reviewers invested in helping you improve.

Peer Review Process



A structured review process ensures consistent, high-quality feedback. Each phase has clear responsibilities and deliverables.

SECTION 10

Bringing It All Together

Case Study: Student Organization System

Member.h - Applying All Concepts

```
class Member {
private:
    static int s_nextMemberId;      // Static: shared counter
    int m_memberId;                 // Instance-specific
    std::string m_name;
    std::string m_major;
    int m_graduationYear;
    bool m_isDuesActive;

    friend class StudentOrganization;  // Friend: update dues

public:
    Member(std::string name, std::string major, int gradYear);

    int getMemberId() const;        // Const-correctness
    std::string getName() const;
    bool isDuesActive() const;

    Member& setMajor(const std::string& major);  // Returns *this
};
```

This header demonstrates composition, encapsulation, const-correctness, static members, friend access, and fluent interface design.

StudentOrganization Class - Composition

```
class StudentOrganization {
private:
 std::vector<Member> m_members; // HAS-A: composition
 std::vector<Event> m_events; // HAS-A: composition
 std::string m_name;
 int m_foundingYear;

public:
 StudentOrganization(std::string name, int foundingYear);

 void addMember(const Member& member); // const parameter
 Member* findMember(int memberId);
 int getMemberCount() const; // const member function
 void scheduleEvent(const Event& event);

 static int getTotalOrganizations(); // static function

private:
 static int s_organizationCount; // static data
};
```

Event Class - Fluent Interface

```
class Event {
private:
 std::string m_name;
 std::string m_location;
 int m_capacity;
 std::string m_date;

public:
 Event(std::string name);

 // Fluent interface - returns *this
 Event& setLocation(const std::string& location) {
 m_location = location;
 return *this;
 }

 Event& setCapacity(int capacity) {
 if (capacity <= 0) {
 throw std::invalid_argument("Capacity must be positive");
 }
 m_capacity = capacity;
 return *this;
 }

 Event& setDate(const std::string& date) {
 m_date = date;
 return *this;
 }

 // Const getters
 std::string getName() const { return m_name; }
 std::string getLocation() const { return m_location; }
 int getCapacity() const { return m_capacity; }
};
```

Using the System - Example

```
// Create organization
StudentOrganization naacp{"NAACP Chapter", 1950};

// Create and add members
Member member1{"John Lewis", "History", 2025};
Member member2{"Ruby Bridges", "Education", 2026};

naacp.addMember(member1);
naacp.addMember(member2);

// Create event with fluent interface
Event townHall{"Community Town Hall"};
townHall.setLocation("Auditorium")
        .setCapacity(200)
        .setDate("2026-03-15");

naacp.scheduleEvent(townHall);

// Query system
std::cout << "Total members: " << naacp.getMemberCount() << "\n";
std::cout << "Total organizations: "
          << StudentOrganization::getTotalOrganizations() << "\n";
```

This example demonstrates clean API usage, method chaining, and clear object relationships.

Design Justifications

1	**Composition** StudentOrganization HAS-A vector of Members and Events - appropriate logical relationships
2	**Friend Access** StudentOrganization is friend of Member to manage dues status directly - tight coupling justified for core operations
3	**Const-Correctness** All getters are const, parameters passed by const reference for efficiency and safety
4	**Static Members** Track total organizations/members with static counters shared across all instances
5	**Encapsulation** All data private, accessed through clear API that maintains invariants
6	**Fluent Interface** Event builder methods return *this for method chaining and improved readability
7	**Validation** Constructor and setters enforce invariants like positive capacity and valid years

Common Design Mistakes to Avoid

❌ God Classes

Classes that do everything - Member handling events, database, email

❌ Public Data

Exposing all data members as public breaks encapsulation

❌ Missing Const

Forgetting const on getters prevents use with const objects

❌ Overusing Friend

Too many friends breaks encapsulation and creates tight coupling

❌ Inconsistent Naming

Mixed naming conventions make code harder to read

❌ No Validation

Constructors/setters without validation allow invalid states

❌ Global Variables

Using globals instead of static members reduces encapsulation

❌ Wrong Storage

Storing members by value when pointers/references would be better

Key Takeaways

Recap the essential principles and practices for advanced object-oriented design in C++.

Master OOP Fundamentals

Leverage composition for robust object relationships and embrace const-correctness for safety and clarity in your C++ code.

Prioritize Design & API Excellence

Separate interfaces from implementations, design intuitive APIs, and apply SOLID principles like SRP and low coupling for maintainable systems.

Cultivate Collaborative Quality

Establish strong class invariants and utilize peer code reviews as a vital tool for continuous improvement and consistent code quality.