The Complete C++ Mastery Handbook - Volume 2

Object-Oriented Programming, STL, and Memory Management

Chapter 6: Classes and Objects Deep Dive

6.1 Dual Explanation Approach

Layman's Explanation

A class is like a blueprint for a house. The blueprint defines rooms (data members) and what you can do in them (member functions). Each actual house built from that blueprint is an object—they all follow the same design but can have different furniture (different data values). Encapsulation is like having walls with doors—outsiders can only access what you allow them to through specific entryways (public interface).

Technical Breakdown

Class Structure:

```
class ClassName {
private:      // Implementation details (hidden)
    // Data members
    // Private helper functions

protected:    // Accessible to derived classes
    // Protected members

public:       // Interface (exposed)
    // Constructors
    // Public member functions
    // Destructor
};
```

Memory Layout:

```
Object Memory Layout:
┌─────────────────────┐
│ Data Member 1       │  ← Offset 0
├─────────────────────┤
│ Data Member 2       │  ← Offset 4 (if int)
├─────────────────────┤
│ Data Member 3       │  ← Offset 8
├─────────────────────┤
│ Padding (alignment) │  ← As needed
└─────────────────────┘
```

```
Note: Member functions NOT stored in object
      (stored once in code segment)
```

6.2 Complete Class Example: Bank Account

```
#include &lt;iostream&gt;
#include &lt;string&gt;
#include &lt;ctime&gt;
#include &lt;iomanip&gt;

/**
 * BankAccount class demonstrating OOP principles
 *
 * Features:
 * - Encapsulation (private data)
 * - Constructor overloading
 * - Const correctness
 * - Member functions
 * - Static members (shared across all instances)
 */
class BankAccount {
private:
    // Data members (instance variables)
    std::string account_number;
    std::string owner_name;
    double balance;
    std::time_t created_at;

    // Static member (shared across all instances)
    static int total_accounts;

    // Private helper function
    void log_transaction(const std::string&amp; type, double amount) const {
        std::cout &lt;&lt; "[" &lt;&lt; owner_name &lt;&lt; "] " &lt;&lt; type &lt;&lt; '
                  &lt;&lt; std::fixed &lt;&lt; std::setprecision(2) &lt;&lt; amount
                  &lt;&lt; " | New balance: $" &lt;&lt; balance &lt;&lt; std::endl;
    }

public:
    // Default constructor
    BankAccount()
        : account_number("UNKNOWN"),
          owner_name("Anonymous"),
          balance(0.0),
          created_at(std::time(nullptr)) {
        ++total_accounts;
        std::cout &lt;&lt; "Default constructor called\n";
    }

    // Parameterized constructor
    BankAccount(const std::string&amp; acc_num, const std::string&amp; name, double initi
        : account_number(acc_num),
          owner_name(name),
          balance(initial),
          created_at(std::time(nullptr)) {
        ++total_accounts;
        std::cout &lt;&lt; "Parameterized constructor called for " &lt;&lt; name &lt;&lt;
    }

    // Destructor
    ~BankAccount() {
        --total_accounts;
        std::cout &lt;&lt; "Destructor called for " &lt;&lt; owner_name &lt;&lt; "\n";
    }

    // Deposit method
    void deposit(double amount) {
        if (amount &gt; 0) {
            balance += amount;
            log_transaction("Deposit", amount);
        } else {
            std::cout &lt;&lt; "Invalid deposit amount\n";
        }
    }

    // Withdraw method
    bool withdraw(double amount) {
        if (amount &lt;= 0) {
            std::cout &lt;&lt; "Invalid withdrawal amount\n";
            return false;
        }

        if (amount &gt; balance) {
            std::cout &lt;&lt; "Insufficient funds\n";
            return false;
        }

        balance -= amount;
        log_transaction("Withdrawal", amount);
        return true;
    }

    // Getter methods (const - don't modify object)
    double get_balance() const {
        return balance;
    }

    std::string get_owner() const {
        return owner_name;
    }

    std::string get_account_number() const {
        return account_number;
    }

    // Print account info (const method)
    void print_info() const {
        std::cout &lt;&lt; "\n=== Account Information ===\n";
        std::cout &lt;&lt; "Account Number: " &lt;&lt; account_number &lt;&lt; "\n";
        std::cout &lt;&lt; "Owner: " &lt;&lt; owner_name &lt;&lt; "\n";
        std::cout &lt;&lt; "Balance: $" &lt;&lt; std::fixed &lt;&lt; std::setprecision(2)
                  &lt;&lt; balance &lt;&lt; "\n";
        std::cout &lt;&lt; "===========================\n\n";
    }

    // Static method (can be called without instance)
    static int get_total_accounts() {
        return total_accounts;
    }
};

// Initialize static member (must be done outside class)
int BankAccount::total_accounts = 0;

int main() {
    std::cout &lt;&lt; "Total accounts: " &lt;&lt; BankAccount::get_total_accounts() &lt;

    // Create accounts
    BankAccount acc1("ACC001", "Alice", 1000.0);
    BankAccount acc2("ACC002", "Bob", 500.0);

    std::cout &lt;&lt; "\nTotal accounts: " &lt;&lt; BankAccount::get_total_accounts() &l

    // Perform operations
    acc1.deposit(250.0);
    acc1.withdraw(100.0);
    acc1.print_info();

    acc2.deposit(1000.0);
    acc2.withdraw(200.0);
    acc2.print_info();

    std::cout &lt;&lt; "Total accounts before scope exit: "
              &lt;&lt; BankAccount::get_total_accounts() &lt;&lt; "\n\n";

    return 0;
    // Destructors automatically called here
}
```

Output:

```
Total accounts: 0

Parameterized constructor called for Alice
Parameterized constructor called for Bob

Total accounts: 2

[Alice] Deposit: $250.00 | New balance: $1250.00
[Alice] Withdrawal: $100.00 | New balance: $1150.00

=== Account Information ===
Account Number: ACC001
Owner: Alice
Balance: $1150.00
```

```
===========================

[Bob] Deposit: $1000.00 | New balance: $1500.00
[Bob] Withdrawal: $200.00 | New balance: $1300.00

=== Account Information ===
Account Number: ACC002
Owner: Bob
Balance: $1300.00
===========================

Total accounts before scope exit: 2

Destructor called for Bob
Destructor called for Alice
```

6.3 Constructor Types and Initialization

Member Initializer List (Preferred)

```
class Point {
private:
    const int x;  // Must be initialized (can't be assigned)
    int&amp; ref;     // References must be initialized

public:
    // CORRECT: Use initializer list
    Point(int x_val, int&amp; r) : x(x_val), ref(r) {
        // Constructor body (can do additional work)
    }

    // WRONG: Cannot compile (const and reference uninitialized)
    // Point(int x_val, int&amp; r) {
    //     x = x_val;    // Error: assignment to const
    //     ref = r;      // Error: reference must be initialized
    // }
};
```

Constructor Delegation (C++11)

```
class Rectangle {
private:
    int width;
    int height;

public:
    // Main constructor
    Rectangle(int w, int h) : width(w), height(h) {
        std::cout &lt;&lt; "Main constructor\n";
    }

    // Delegating constructor (calls main constructor)
    Rectangle() : Rectangle(0, 0) {
        std::cout &lt;&lt; "Default constructor delegates to main\n";
    }

    // Another delegating constructor
    Rectangle(int side) : Rectangle(side, side) {
        std::cout &lt;&lt; "Square constructor delegates to main\n";
    }
};
```

Copy Constructor

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

public:
    // Constructor
    MyString(const char* str) {
        length = std::strlen(str);
        data = new char[length + 1];
        std::strcpy(data, str);
        std::cout &lt;&lt; "Constructor: allocated " &lt;&lt; length + 1 &lt;&lt; " bytes
    }

    // Copy constructor (deep copy)
    MyString(const MyString&amp; other) {
        length = other.length;
        data = new char[length + 1];
        std::strcpy(data, other.data);
        std::cout &lt;&lt; "Copy constructor: deep copy of " &lt;&lt; length + 1 &lt;&lt;
    }

    // Destructor
    ~MyString() {
        delete[] data;
        std::cout &lt;&lt; "Destructor: freed memory\n";
    }

    // Getter
    const char* get_data() const { return data; }
};

int main() {
    MyString s1("Hello");
    MyString s2 = s1;  // Copy constructor called

    std::cout &lt;&lt; "s1: " &lt;&lt; s1.get_data() &lt;&lt; "\n";
    std::cout &lt;&lt; "s2: " &lt;&lt; s2.get_data() &lt;&lt; "\n";

    return 0;
}
```

Output:

```
Constructor: allocated 6 bytes
Copy constructor: deep copy of 6 bytes
s1: Hello
s2: Hello
Destructor: freed memory
Destructor: freed memory
```

Chapter 9: Standard Template Library (STL)

9.1 Dual Explanation Approach

Layman's Explanation

STL is like a toolbox of pre-built, tested components. Instead of building your own hammer (vector) or screwdriver (map) every time, you use proven tools. These tools are generic (work with any type) and optimized (professional-grade performance). Learning STL is like learning to use power tools—initially complex, but incredibly productive once mastered.

Technical Breakdown

STL Components:

1. **Containers:** Data structures (vector, map, set, etc.)
2. **Iterators:** Unified interface to traverse containers
3. **Algorithms:** Generic operations (sort, find, transform)
4. **Function Objects:** Callable objects for customization

9.2 Vector - Dynamic Array

Complete Vector Guide

```
#include &lt;vector&gt;
#include &lt;iostream&gt;
#include &lt;algorithm&gt;

void demonstrate_vector() {
    // Creation
    std::vector&lt;int&gt; v1;                    // Empty vector
    std::vector&lt;int&gt; v2(10);                // 10 elements, default-initialized to
    std::vector&lt;int&gt; v3(10, 42);            // 10 elements, all = 42
    std::vector&lt;int&gt; v4{1, 2, 3, 4, 5};    // Initializer list

    // Adding elements
    v1.push_back(10);      // Add to end - O(1) amortized
    v1.push_back(20);
    v1.emplace_back(30);   // Construct in-place (more efficient)

    // Accessing elements
    int first = v1[0];           // No bounds checking (fast)
    int second = v1.at(1);       // Bounds checking (throws exception)
    int last = v1.back();        // Last element

    // Size and capacity
    size_t size = v1.size();         // Number of elements
    size_t capacity = v1.capacity(); // Allocated space
    bool empty = v1.empty();

    // Reserve space (avoid reallocations)
    v1.reserve(100);  // Pre-allocate for 100 elements

    // Iteration
    for (size_t i = 0; i &lt; v1.size(); ++i) {
        std::cout &lt;&lt; v1[i] &lt;&lt; " ";
    }
    std::cout &lt;&lt; "\n";

    // Range-based for loop (C++11)
    for (int value : v1) {
        std::cout &lt;&lt; value &lt;&lt; " ";
    }
    std::cout &lt;&lt; "\n";

    // Iterator-based
    for (auto it = v1.begin(); it != v1.end(); ++it) {
        std::cout &lt;&lt; *it &lt;&lt; " ";
    }
    std::cout &lt;&lt; "\n";

    // Remove elements
    v1.pop_back();        // Remove last - O(1)
    v1.erase(v1.begin()); // Remove first - O(n)
    v1.clear();           // Remove all

    // Sorting
    std::vector&lt;int&gt; nums{5, 2, 8, 1, 9};
    std::sort(nums.begin(), nums.end());  // O(n log n)

    // Binary search (requires sorted vector)
    bool found = std::binary_search(nums.begin(), nums.end(), 5);

    // Finding elements
    auto it = std::find(nums.begin(), nums.end(), 8);
    if (it != nums.end()) {
        std::cout &lt;&lt; "Found at index: " &lt;&lt; (it - nums.begin()) &lt;&lt; "\n";
    }
}
```

Vector Memory Management

```
#include &lt;vector&gt;
#include &lt;iostream&gt;

void analyze_vector_growth() {
    std::vector&lt;int&gt; v;

    std::cout &lt;&lt; "Size\tCapacity\n";
    for (int i = 0; i &lt; 20; ++i) {
        v.push_back(i);
        std::cout &lt;&lt; v.size() &lt;&lt; "\t" &lt;&lt; v.capacity() &lt;&lt; "\n";
    }
}
```

Typical Output:

```
Size    Capacity
1       1
2       2
3       4
4       4
5       8
6       8
7       8
8       8
9       16
...
```

Growth Strategy: Capacity typically doubles (factor of 2) to achieve amortized O(1) push_back.

HFT-Optimized Vector Usage

```
#include &lt;vector&gt;

/**
 * Pre-allocated vector for low-latency scenarios
 *
 * Avoid reallocations in critical path by reserving upfront
 */
class MarketDataBuffer {
private:
    static constexpr size_t INITIAL_CAPACITY = 10000;
    std::vector&lt;double&gt; prices;

public:
    MarketDataBuffer() {
        prices.reserve(INITIAL_CAPACITY);  // One-time allocation
    }

    void add_price(double price) {
        prices.push_back(price);  // No reallocation if &lt; INITIAL_CAPACITY
    }

    // Access without bounds checking (faster)
    double operator[](size_t idx) const {
        return prices[idx];
    }

    size_t size() const {
        return prices.size();
    }
};
```

9.3 Map - Associative Container

Complete Map Guide

```
#include &lt;map&gt;
#include &lt;unordered_map&gt;
#include &lt;string&gt;
#include &lt;iostream&gt;

void demonstrate_map() {
    // std::map - Ordered (Red-Black Tree)
    // O(log n) insertion, lookup, deletion
    std::map&lt;std::string, int&gt; ages;

    // Insertion
    ages["Alice"] = 30;
    ages["Bob"] = 25;
    ages.insert({"Charlie", 35});
    ages.emplace("David", 28);  // Construct in-place

    // Lookup
    int alice_age = ages["Alice"];  // Creates entry if doesn't exist!

    // Safe lookup
    auto it = ages.find("Eve");
    if (it != ages.end()) {
        std::cout &lt;&lt; "Eve's age: " &lt;&lt; it-&gt;second &lt;&lt; "\n";
    } else {
        std::cout &lt;&lt; "Eve not found\n";
    }

    // Iteration (sorted by key)
    for (const auto&amp; [name, age] : ages) {  // C++17 structured binding
        std::cout &lt;&lt; name &lt;&lt; ": " &lt;&lt; age &lt;&lt; "\n";
    }

    // Check if key exists
    if (ages.count("Alice") &gt; 0) {
        std::cout &lt;&lt; "Alice exists\n";
    }
    // Remove
    ages.erase("Bob");

    // std::unordered_map - Hash Table
    // O(1) average insertion, lookup, deletion
    std::unordered_map&lt;int, std::string&gt; symbols;
    symbols[1] = "AAPL";
    symbols[2] = "GOOGL";

    // Access
    std::string sym = symbols[1];  // "AAPL"
}
```

Map vs Unordered Map Comparison

Feature	std::map	std::unordered_map
Implementation	Red-Black Tree	Hash Table
Ordering	Sorted by key	Unordered
Lookup	O(log n)	O(1) average, O(n) worst
Insertion	O(log n)	O(1) average, O(n) worst
Iteration	Sorted order	Arbitrary order
Memory	Less overhead	More overhead (buckets)
Use Case	Need sorted keys	Fast lookup, don't care about order

HFT Example: Symbol Lookup

```
#include &lt;unordered_map&gt;
#include &lt;string&gt;

class SymbolLookup {
private:
    // Fast O(1) lookups
    std::unordered_map&lt;std::string, uint32_t&gt; symbol_to_id;
    std::unordered_map&lt;uint32_t, std::string&gt; id_to_symbol;

public:
    void add_symbol(const std::string&amp; symbol, uint32_t id) {
        symbol_to_id[symbol] = id;
        id_to_symbol[id] = symbol;
    }

    // O(1) symbol -&gt; ID conversion
    uint32_t get_id(const std::string&amp; symbol) const {
        auto it = symbol_to_id.find(symbol);
        return (it != symbol_to_id.end()) ? it-&gt;second : 0;
    }

    // O(1) ID -&gt; symbol conversion
    std::string get_symbol(uint32_t id) const {
        auto it = id_to_symbol.find(id);
        return (it != id_to_symbol.end()) ? it-&gt;second : "";
    }
};

int main() {
    SymbolLookup lookup;
    lookup.add_symbol("AAPL", 1);
    lookup.add_symbol("GOOGL", 2);

    uint32_t id = lookup.get_id("AAPL");  // Fast: O(1)
    std::string symbol = lookup.get_symbol(1);  // Fast: O(1)

    return 0;
}
```

9.4 Set - Unique Elements

```
#include &lt;set&gt;
#include &lt;unordered_set&gt;
#include &lt;iostream&gt;

void demonstrate_set() {
    // std::set - Ordered (Red-Black Tree)
    std::set&lt;int&gt; s;

    // Insertion
    s.insert(5);
    s.insert(2);
    s.insert(8);
    s.insert(2);  // Duplicate - ignored

    // Size = 3 (2 inserted only once)
    std::cout &lt;&lt; "Set size: " &lt;&lt; s.size() &lt;&lt; "\n";

    // Iteration (sorted order)
    for (int value : s) {
        std::cout &lt;&lt; value &lt;&lt; " ";  // Output: 2 5 8
    }
    std::cout &lt;&lt; "\n";

    // Find
    auto it = s.find(5);
    if (it != s.end()) {
        std::cout &lt;&lt; "Found: " &lt;&lt; *it &lt;&lt; "\n";
    }

    // Remove
    s.erase(2);

    // std::unordered_set - Hash Table
    std::unordered_set&lt;std::string&gt; words{"hello", "world", "hello"};
    std::cout &lt;&lt; "Unique words: " &lt;&lt; words.size() &lt;&lt; "\n";  // 2
}
```

9.5 Capstone Project: Order Book Implementation

Goal: Build a realistic limit order book for HFT systems.

```
#include &lt;map&gt;
#include &lt;vector&gt;
#include &lt;iostream&gt;

enum class Side { BUY, SELL };

struct Order {
    uint64_t order_id;
    Side side;
    double price;
    uint32_t quantity;
    uint64_t timestamp;
};

/**
 * Simplified Order Book
 *
 * Features:
 * - Price-level aggregation
 * - Fast order insertion/cancellation
 * - Best bid/ask queries
 *
 * Data Structure:
 * - std::map for price levels (sorted)
 * - Vector for orders at each level
 */
class OrderBook {
private:
    // Buy side: descending prices (best bid = highest)
    std::map&lt;double, std::vector&lt;Order&gt;, std::greater&lt;double&gt;&gt; bids;

    // Sell side: ascending prices (best ask = lowest)
    std::map&lt;double, std::vector&lt;Order&gt;&gt; asks;

public:
    void add_order(const Order&amp; order) {
        if (order.side == Side::BUY) {
            bids[order.price].push_back(order);
        } else {
            asks[order.price].push_back(order);
        }
    }

    double get_best_bid() const {
        if (bids.empty()) return 0.0;
        return bids.begin()-&gt;first;
    }
    double get_best_ask() const {
        if (asks.empty()) return 0.0;
        return asks.begin()-&gt;first;
    }

    double get_spread() const {
        double bid = get_best_bid();
        double ask = get_best_ask();
        return (bid &gt; 0 &amp;&amp; ask &gt; 0) ? (ask - bid) : 0.0;
    }

    void print_book(int levels = 5) const {
        std::cout &lt;&lt; "\n=== Order Book ===\n";

        // Print asks (ascending)
        auto ask_it = asks.begin();
        for (int i = 0; i &lt; levels &amp;&amp; ask_it != asks.end(); ++i, ++ask_it) {
            uint32_t total_qty = 0;
            for (const auto&amp; order : ask_it-&gt;second) {
                total_qty += order.quantity;
            }
            std::cout &lt;&lt; "ASK: " &lt;&lt; ask_it-&gt;first &lt;&lt; " x " &lt;&lt;
        }

        std::cout &lt;&lt; "---SPREAD: " &lt;&lt; get_spread() &lt;&lt; "---\n";

        // Print bids (descending - already sorted)
        auto bid_it = bids.begin();
        for (int i = 0; i &lt; levels &amp;&amp; bid_it != bids.end(); ++i, ++bid_it) {
            uint32_t total_qty = 0;
            for (const auto&amp; order : bid_it-&gt;second) {
                total_qty += order.quantity;
            }
            std::cout &lt;&lt; "BID: " &lt;&lt; bid_it-&gt;first &lt;&lt; " x " &lt;&lt;
        }

        std::cout &lt;&lt; "==================\n\n";
    }
};

int main() {
    OrderBook book;

    // Add orders
    book.add_order({1, Side::BUY, 100.50, 100, 0});
    book.add_order({2, Side::BUY, 100.45, 200, 0});
    book.add_order({3, Side::BUY, 100.40, 150, 0});

    book.add_order({4, Side::SELL, 100.55, 100, 0});
    book.add_order({5, Side::SELL, 100.60, 200, 0});
    book.add_order({6, Side::SELL, 100.65, 150, 0});

    book.print_book();

    std::cout &lt;&lt; "Best Bid: " &lt;&lt; book.get_best_bid() &lt;&lt; "\n";
    std::cout &lt;&lt; "Best Ask: " &lt;&lt; book.get_best_ask() &lt;&lt; "\n";
    std::cout &lt;&lt; "Spread: " &lt;&lt; book.get_spread() &lt;&lt; "\n";

    return 0;
}
```

Output:

```
=== Order Book ===
ASK: 100.55 x 100
ASK: 100.6 x 200
ASK: 100.65 x 150
---SPREAD: 0.05---
BID: 100.5 x 100
BID: 100.45 x 200
BID: 100.4 x 150
==================

Best Bid: 100.5
Best Ask: 100.55
Spread: 0.05
```

This comprehensive Volume 2 continues with the same exhaustive depth covering all intermediate and advanced C++ topics including memory management, smart pointers, templates, move semantics, concurrency, and modern C++ features, always maintaining the dual explanation approach, annotated code, capstone projects, practice problems, interview questions, and cheat sheets.