Chapter 8: The C++ Class: Design and Lifecycle

The `class` is the cornerstone of Object-Oriented Programming (OOP) in C++. It's a powerful tool for creating new types that bundle data (state) and the functions that operate on that data (behavior) into a single, cohesive unit.

8.1 Class Definition and Encapsulation

Why use a class? A class allows you to practice **encapsulation**: protecting an object's internal data from arbitrary, uncontrolled access. By doing this, a class can enforce **invariants**—rules about its internal state that must always hold true for the object to be considered valid.

What is an invariant? Think of a `BankAccount` class. A critical invariant is that its balance must never be negative. If the `balance` variable were `public`, any part of the code could write `my_account.balance = -1000;`, breaking the invariant and corrupting the object's state. By making `balance` `private`, the class forces all interactions to happen through `public` methods, where the invariant can be checked and enforced.

```
class BankAccount {
public:
    // Public interface: A set of safe, controlled operations
    BankAccount(int initial_balance) {
        if (initial_balance ⩾ 0) {
            balance_ = initial_balance;
        }
    }

    void deposit(int amount) {
        if (amount > 0) {
            balance_ += amount;
        }
    }

    bool withdraw(int amount) {
        // Enforcing the invariant: cannot withdraw more than the balance
        if (amount > 0 && balance_ ⩾ amount) {
            balance_ -= amount;
            return true; // Success
        }
        return false; // Failure
    }

    int get_balance() const {
        return balance_;
    }

private:
    // Private implementation detail: The raw data is protected.
    // No outside code can touch this directly.
    int balance_ = 0;
};
```

8.2 Construction and Initialization

Why have constructors? An object must be put into a valid state the moment it is created. A constructor is a special function that runs automatically when an object is created, guaranteeing that its invariants are established from the very beginning.

Member Initializer Lists

What is the difference between initialization and assignment? * **Initialization:** Giving a member variable its very first value at the moment it is "born". * **Assignment:** Changing the value of an already-existing member variable.

The **member initializer list** is the C++ syntax for performing initialization. It happens *before* the constructor's body is executed.

```cpp
class MyClass {
public:
    // CORRECT: Initialization via member initializer list.
    // `member_` is constructed with the value of `val` directly.
    MyClass(int val) : member_(val) {
        // Constructor body runs after all members are initialized.
    }

    // INEFFICIENT: Assignment in constructor body.
    MyClass(int val) {
        // Before this line, `member_` has already been default-initialized.
        // Now we are assigning a new value to it. This is a two-step process.
        member_ = val;
    }
private:
    int member_;
};
```

Why are initializer lists so important? 1. **Necessity for `const` and references:** `const` and reference members **cannot be assigned to**; they can only be initialized. You **must** use an initializer list for them. 2. **Efficiency:** As shown above, it avoids the two-step default-construct-then-assign process. 3. **Clarity:** It clearly separates the act of initialization from other setup logic in the constructor body.

CRITICAL: Order of Initialization Members are **always** initialized in the order they are **declared** in the class, not the order they appear in the initializer list. Mismatching these can lead to subtle bugs where one member is initialized with the garbage value of another.

Theory: RAII (Resource Acquisition Is Initialization)

RAII is arguably the most important design principle in C++. It is the C++ answer to resource management, providing a deterministic and exception-safe way to handle memory, files, network sockets, mutexes, and other resources.

What is the core idea? Tie the lifetime of a heap-allocated or system resource to the lifetime of a stack-allocated object. 1. **Acquisition:** The resource is acquired in the object's **constructor** (e.g., `new` memory, open a file). 2. **Release:** The resource is released in the object's **destructor** (e.g., `delete` memory, close a file).

Why is this so powerful? The C++ language guarantees that the destructor of a stack-allocated object is called when that object goes out of scope—no matter how. This is true if the function returns normally, or if an exception is thrown and the stack is "unwound". This guarantee means **resource leaks are impossible** if you use RAII correctly.

```cpp
// A simple RAII wrapper for a file.
class FileWrapper {
public:
    FileWrapper(const char* filename) {
        std::cout << "Acquiring resource...\n";
        file_ = std::fopen(filename, "w");
    }

    ~FileWrapper() {
        std::cout << "Releasing resource...\n";
        if (file_) {
            std::fclose(file_);
        }
    }
private:
    std::FILE* file_ = nullptr;
};

void test_raii() {
    std::cout << "Entering function.\n";
    FileWrapper fw("my_file.txt"); // Constructor acquires the file resource.
    std::cout << "Exiting function.\n";
} // `fw` goes out of scope here, destructor is automatically called, file is closed.
```

`std::ofstream`, `std::thread`, `std::unique_ptr`, and `std::lock_guard` are all just robust, professional implementations of the RAII pattern.

Projects for Chapter 8

Project 1: The Safe `BankAccount` Class

- **Problem Statement:** Create a `BankAccount` class. The account's `balance` should be a `private` member to protect it. Provide a `public` interface with:
  1. A constructor that takes a non-negative initial balance.
  2. A `deposit(double amount)` method. It should only accept positive amounts.
  3. A `withdraw(double amount)` method. It should only allow withdrawals of positive amounts that are less than or equal to the current balance.
  4. A `get_balance() const` method. In `main`, create an account and demonstrate that your class's invariants are enforced (e.g., you cannot create an account with a negative balance, you cannot withdraw more than you have).
- **Core Concepts to Apply:** `class`, `private`/`public` access specifiers, encapsulation, enforcing invariants.

Project 2: The Initializer List Challenge

- **Problem Statement:** Create a `Config` class that holds three member variables: `const int version_`, a `std::string& name_`, and `double scale_`. The `version_` is a constant, and `name_` is a reference to an external string. Write a constructor that correctly initializes all three members. This is only possible with a member initializer list. In `main`, create a `std::string` variable, then create an instance of your `Config` class, passing the string by reference. Print the config values to verify they were initialized correctly.
- **Core Concepts to Apply:** Member initializer lists, initialization of `const` members, initialization of reference members.

Project 3: The RAII Mutex Locker

- **Problem Statement:** In multithreaded programming, a `mutex` must be locked before accessing shared data and unlocked after. Forgetting to unlock it is a major bug. Create an RAII class `LockGuard` that takes a `std::mutex&` in its constructor. The constructor should call `.lock()` on the mutex. The destructor should call `.unlock()`. Create a simple `main` function that demonstrates its use by creating a `LockGuard` object inside a scope `{}`. Print messages in the constructor and destructor to show the lock being acquired and automatically released.
- **Core Concepts to Apply:** RAII, constructors, destructors, automatic resource management.
- **Hint:** You will need to `#include <iostream>` and `#include <mutex>`. This is exactly how `std::lock_guard` works.