Chapter 13: Deterministic Resource Management

This chapter covers the features that define modern C++ resource management. Building on RAII, move semantics and smart pointers provide the tools to create safe, exception-friendly, and highly efficient code that expresses resource ownership directly in the type system, largely eliminating the need for manual `new` and `delete`.

13.1 Move Semantics

Why was this needed? In C++03, returning a resource-heavy object like a `std::vector` from a function was incredibly expensive. It required a deep copy, even if the object being returned was a temporary that was about to be destroyed. The compiler had no standard way to take advantage of the fact that the source object was temporary. Move semantics was introduced in C++11 to solve this by allowing a cheap "pilfering" of resources from temporary objects (rvalues) instead of copying them.

Rvalue References (`&&`)

The feature that enables move semantics is the **rvalue reference (`&&`)**. It's a new kind of reference that can *only* bind to temporary objects (rvalues). This allows us to create function overloads that are chosen by the compiler only when the source object is a temporary.

Move Constructor and Move Assignment

These are two new special member functions that take an rvalue reference. Their job is not to copy, but to **move**—to steal the internal resources from the source object and then leave the source object in a valid but "hollow" state.

```
class MyString {
public:
    // ... (Destructor, Copy Constructor, etc.)

    // Move Constructor: Steals resources from `other`
    MyString(MyString&& other) noexcept { // `noexcept` is a critical optimization
        // 1. Copy the pointer and size from the source object.
        data_ = other.data_;
        size_ = other.size_;

        // 2. CRUCIAL: Reset the source object to a valid, destructible state.
        // This prevents the source's destructor from double-freeing the memory.
        other.data_ = nullptr;
        other.size_ = 0;
    }

    // Move Assignment Operator is similar...
};
```

What `std::move` Really Does

`std::move` does not move anything. It is simply a **cast**. It unconditionally casts its argument into an rvalue reference. It is a promise to the compiler: "I know I have an lvalue here, but I am done with it. You have my permission to treat it like a temporary and cannibalize it." This is how you explicitly invoke a move operation on a non-temporary object.

The "Moved-From" State

After an object has been moved from, it must be in a **valid state**. This means it must be safe to be destroyed. It also means you can assign a new value to it. However, you should make no other assumptions about its contents. It is not guaranteed to be null or zero. Using a moved-from object in any way that depends on its value (other than assignment or destruction) is a bug.

13.2 Smart Pointers: RAII in Action

Why use them? Smart pointers are the embodiment of RAII for dynamic memory. They are wrapper classes that own a raw pointer and manage its lifetime automatically. They completely eliminate the need for manual `new` and `delete`, which is the single largest source of bugs (memory leaks, double-frees, dangling pointers) in C++.

`std::unique_ptr`

- **What it is:** Represents **unique, exclusive ownership** of a resource. It is a lightweight, zero-overhead abstraction. In a release build, it compiles down to the same machine code as a raw pointer.
- **The Rules:** It cannot be copied. It can only be **moved**. This enforces unique ownership at compile time.
- **How to use it:** Always prefer `std::make_unique<T>()` (from C++14) to create one. To transfer ownership, return it by value from a function or pass it to another function using `std::move`.

```cpp
#include <memory>

// A factory function that creates and returns a Widget.
std::unique_ptr<Widget> create_widget() {
    return std::make_unique<Widget>(); // Efficiently returns ownership to the caller
}

// A function that takes ownership of a Widget.
void process_widget(std::unique_ptr<Widget> w) {
    // ... now this function owns the widget ...
} // `w` goes out of scope here, and its destructor automatically deletes the Widget.

int main() {
    std::unique_ptr<Widget> my_widget = create_widget();
    process_widget(std::move(my_widget)); // Explicitly move to transfer ownership.
    // `my_widget` is now nullptr.
}
```

`std::shared_ptr`

- **What it is:** Represents **shared ownership**. It uses reference counting to keep track of how many `shared_ptr`s are pointing to the same object. The last `shared_ptr` to be destroyed is the one that calls `delete`.
- **The Overhead:** This convenience comes at a cost. A `shared_ptr` requires a separate allocation for a **control block**, which stores the reference count. This means more memory overhead and slightly slower creation and destruction than a `unique_ptr`.
- **How to use it:** Always prefer `std::make_shared<T>()`. This is a crucial optimization that allocates the object and its control block in a single, contiguous memory block, which is much more efficient.

`std::weak_ptr`

- **Why it exists:** To break **circular references**. If two objects hold `shared_ptr`s to each other, their reference counts will never reach zero, and they will leak, even though they are unreachable by the rest of the program.
- **What it is:** A `weak_ptr` is a non-owning, "observing" pointer. It does not affect the reference count.
- **How to use it:** You cannot access the object directly through a `weak_ptr`. You must first try to "promote" it to a `shared_ptr` by calling its `lock()` method. If the object still exists, `lock()` returns a valid `shared_ptr`. If the object has already been deleted, `lock()` returns an empty `shared_ptr`.

```cpp
// Classic example: A tree where nodes point to children, and children point back to the parent.
class Node {
public:
    std::vector<std::shared_ptr<Node>> children;
    std::weak_ptr<Node> parent; // MUST be weak_ptr to prevent a cycle!
};
```

Projects for Chapter 13

Project 1: The Movable `Vector`

- **Problem Statement:** Implement a simple `Vector<T>` class template that manages a dynamic array. Implement the "Rule of Five": a destructor, a copy constructor (deep copy), a copy assignment operator, a **move constructor**, and a **move assignment operator**. The move operations should steal the internal pointer and size from the source object and null out the source. Add print statements to each of the five functions to see when they are called. In `main`, create

a function that creates and returns a large `Vector` by value. Observe that the move constructor is called, avoiding an expensive deep copy.
- **Core Concepts to Apply:** The Rule of Five, move constructor, move assignment operator, `std::move`, rvalue references.

Project 2: The `unique_ptr` Factory

- **Problem Statement:** Create a `struct GameObject`. Write a factory function `std::unique_ptr<GameObject> create_player();` that creates a `GameObject` on the heap and returns it via a `unique_ptr`. In `main`, call this function to get a player object. Then, create a `std::vector<std::unique_ptr<GameObject>>` and `std::move` the player object into the vector. Finally, loop through the vector and use the objects.
- **Core Concepts to Apply:** `std::unique_ptr`, `std::make_unique`, `std::move` for ownership transfer, factory functions, storing smart pointers in containers.

Project 3: The `shared_ptr` Circular Reference

- **Problem Statement:** Create two classes, `Person` and `Apartment`. A `Person` should have a `std::shared_ptr<Apartment> apartment_`. An `Apartment` should have a `std::shared_ptr<Person> tenant_`. In the destructors of both classes, print a message (e.g., `"Person destroyed"`). In `main`, create a `Person` and an `Apartment` using `std::make_shared`. Then, link them together by setting their respective `shared_ptr` members, creating a cycle. When `main` exits, observe that neither destructor is called, proving the memory leak. Fix the problem by changing one of the members to be a `std::weak_ptr` and observe that the destructors are now called correctly.
- **Core Concepts to Apply:** `std::shared_ptr`, `std::weak_ptr`, identifying and breaking circular references.