Part III: The C++ Memory Model and Resource Management

This part delves into the intricacies of C++ memory management and resource handling. It begins by demystifying raw pointers, dynamic allocation, and the risks associated with manual memory management. The guide then explores value categories, lvalue and rvalue references, and the mechanics of move semantics, equipping you to write efficient and safe code. Finally, it introduces smart pointers and the RAII (Resource Acquisition Is Initialization) paradigm, demonstrating modern techniques for automatic and robust resource management in C++.

Table of Contents

6. Raw Pointers and Dynamic Allocation

This chapter marks the transition into the core of C++: **manual memory management**. For an experienced developer coming from a managed language, this is the most significant conceptual shift. In C++, you must understand *where* your data lives and *who* is responsible for its destruction. While modern C++ aims to minimize the use of **raw pointers**, they remain the foundational mechanism for dynamic memory control.

6.1 The Stack vs. The Heap vs. The Data Segment

In contrast to the abstract "managed heap" of the CLR, a C++ program explicitly divides its working memory into three main regions, each dictating a different **Storage Duration** (Chapter 3.3).

Memory Region	Storage Duration	Allocation/Deallocation	C++ Keywords	Use Case
The Stack	**Automatic**	Automatically managed by the compiler (LIFO).	Local variables, function arguments, objects without `new`.	Small, local variables; **RAII** objects.
The Heap	**Dynamic**	Manually allocated by the programmer.	`new`, `delete`, `malloc`, `free`.	Large objects, data whose size is unknown until runtime, shared resources.
Data Segment	**Static**	Exists for the entire program lifetime.	`static`, global variables, string literals.	Constants, global application state.

The Crucial Distinction: Objects created on the **Stack** have **Automatic Storage Duration** and are *deterministically destroyed* when they go out of scope (Chapter 5.3). Objects created on the **Heap** have **Dynamic Storage Duration** and exist until they are **explicitly destroyed** by the programmer using `delete`.

6.2 Raw Pointers: Declaration, Dereferencing, and the Null State

A **raw pointer** is a variable that holds the **memory address** of another variable. It is the C++ equivalent of an `IntPtr` used in `unsafe` C# code, representing direct control over memory.

Declaration and Address-of

The asterisk (`*`) is used when **declaring** a pointer. The ampersand (`&`) is the **address-of operator**, which retrieves the memory address of a variable.

```
int number = 42; // An integer with automatic storage (on the Stack)

// Declaration: ptr is a pointer to an integer
int* ptr = &number;
```

Dereferencing

The asterisk (*) is also used to **dereference** a pointer, meaning to access or modify the data at the address the pointer holds.

```
// Access the value at the address held by ptr (which is 42)
std::cout << "Value via pointer: " << *ptr << "\n";

// Change the value of 'number' via the pointer alias
*ptr = 100;
std::cout << "New value of number: " << number << "\n"; // Output: 100
```

Syntax Alert: `T* ptr` declares a pointer. `*ptr` accesses the data pointed to. The star has two separate meanings based on context.

The Null State

In Modern C++, the correct way to represent a pointer that does not point to any valid memory address is using `nullptr` (introduced in C++11), which is a distinct type for null pointer constants.

```
int* data_ptr = nullptr; // Always initialize pointers to nullptr
```

Never use the C-style `NULL` macro, as it is an integer literal (`0`) and can cause ambiguous overload resolution errors.

6.3 Manual Allocation and Deallocation (`new` and `delete`)

To create an object on the **Heap** (Dynamic Storage Duration), you use the `new` operator. `new` allocates memory and calls the object's constructor. It returns a **raw pointer** to the newly created object.

```
// Allocation on the Heap
// ptr points to a Heap object. Its lifetime is independent of its scope.
Point* ptr = new Point(10, 20);

// Access members via the arrow operator (syntactic sugar for (*ptr).x)
ptr->x = 50;
```

The `new`/`delete` Contract

Because the C++ heap is unmanaged, you are entirely responsible for cleaning up memory allocated with `new`. You **must** call `delete` on the pointer when you are finished with the object.

```
// Deallocation: calls the object's destructor, then frees the memory.
delete ptr;

// After deletion, ptr is a dangling pointer (see 6.4)
// ptr = nullptr; // Best practice: reset the pointer after deletion
```

Array Allocation and Deallocation

If you allocate an array of objects on the heap, you must use the array form of `new` and the array form of `delete`.

```
// Array Allocation: Allocates an array of 5 Point objects on the Heap
Point* points_array = new Point[5];

// Array Deallocation: MUST use delete[] to call the destructor for ALL 5 objects
delete[] points_array;
// delete points_array; // ERROR/Undefined Behavior: Destructor only called for
the first element, risking memory leaks.
```

6.4 Dangers: Memory Leaks, Double Deletion, and Dangling Pointers

The contract of manual memory management is fraught with peril. These errors are often silent at runtime but catastrophic for stability, and they are the primary reason Modern C++ avoids raw pointers for object ownership.

1. Memory Leaks

A **memory leak** occurs when memory is allocated on the heap but is never freed by `delete`, usually because the pointer owning the memory goes out of scope.

```
void leaky_function() {
    Point* p = new Point(1, 1);
    // ... function exits here ...
    // 'p' (the pointer) goes out of scope and is destroyed,
    // but the Point object on the Heap is NOT deleted.
    // The memory is now inaccessible—a leak.
}
```

2. Double Deletion

Double deletion occurs when `delete` is called more than once on the same pointer address.

```
Point* p = new Point(1, 1);
delete p;
// ... later ...
// The memory has been returned to the system, but the pointer still holds the
address.
delete p; // CRASH! (Undefined Behavior, usually a heap corruption error)
```

3. Dangling Pointers

A **dangling pointer** is a pointer that points to a memory location that has already been freed.

```
Point* p1 = new Point(1, 1);
Point* p2 = p1; // p2 now also points to the same object
delete p1; // The memory is freed

// Now p2 is a dangling pointer. Using *p2 here is Undefined Behavior.
p2->x = 5; // CRASH or corrupted data!
```

These inherent risks are why the guiding principle of C++ is: **"Never use raw pointers to manage ownership; use smart pointers instead."** (Chapter 9)

6.5 C-style Arrays, Pointer Arithmetic, and Decaying

Raw pointers are closely linked to the legacy **C-style array** (`int arr[N]`). While Modern C++ prefers `std::vector` (dynamic) or `std::array` (static), understanding C-style arrays is necessary.

C-style Arrays

C-style arrays are fixed-size and lack bounds checking, unlike C# arrays:

```
int data[10]; // An array of 10 integers (fixed size on the stack)
data[10] = 5; // No error from the compiler, but this is a serious error (Buffer
Overflow).
```

Array Decaying

A fundamental concept is **array decay**: when a C-style array is passed to a function or assigned to a pointer, it automatically **decays** into a raw pointer to its first element.

```
void process_array(int* ptr) {
    // This function has no idea that the array originally had 10 elements.
}

int main() {
    int arr[10];
    process_array(arr); // arr decays to an int* pointing to arr[0]
    return 0;
}
```

Pointer Arithmetic

Because arrays and pointers are intertwined, you can perform arithmetic directly on pointers to move through memory:

```
int arr[] = {10, 20, 30};
int* p = arr; // p points to arr[0] (value 10)

p++; // p now points to arr[1] (value 20)
std::cout << "Value: " << *p << "\n"; // Output: 20

// p += 2; // Move 2 integers forward
```

The increment `p++` actually increments the memory address by `sizeof(int)` bytes, making pointer arithmetic type-aware. This low-level capability is powerful but is another vector for buffer overflow errors if bounds are not manually checked.

Key Takeaways

- **Heap Requires `new`/`delete`:** Memory allocated on the **Heap** using `new` has **Dynamic Storage Duration** and must be explicitly released using `delete` (or `delete[]` for arrays).
- **`nullptr` is Modern Null:** Always initialize raw pointers to `nullptr` (C++11) to safely indicate a non-pointing state. Avoid the C-style `NULL`.
- **The `new`/`delete` Contract:** You **must** pair `new T` with `delete T` and `new T[N]` with `delete[] T` to avoid **Undefined Behavior** and memory leaks.
- **Raw Pointer Ownership is Dangerous:** The use of raw pointers for ownership leads directly to the three great dangers: **memory leaks**, **double deletion**, and **dangling pointers**.
- **Array Decay:** C-style arrays automatically **decay** into raw pointers to their first element, enabling **pointer arithmetic** but sacrificing compile-time size information and bounds checking.

Exercises

1. **Memory Leak Simulation:** Write a function `create_and_leak()` that allocates a large array of 1,000 integers on the heap using `new int[1000]`, but forgets to call `delete[]`.
    - *Task:* Explain why this specific function call leads to a memory leak and where the memory is lost (which region?).
    - *Hint:* The pointer holding the address is destroyed on the stack, but the memory on the heap remains allocated and unreachable.
2. **Dangling Pointer Creation:** Write a simple program where a pointer `p1` is created, a second pointer `p2` is assigned to `p1`, and then `delete p1` is called.
    - *Task:* Reset the pointer `p1` to `nullptr` immediately after deletion, but leave `p2` pointing to the old address. Explain why `p2` is now a dangling pointer.
    - *Hint:* The memory is gone, but the address in `p2` is still the same.
3. **Pointer Arithmetic Safety:** Declare a C-style array `char text[4] = "abc";`. Declare a `char* p = text;`.
    - *Task:* Use pointer arithmetic (`p = p + 5;`) and then try to print `*p`. Explain the potential for **Undefined Behavior** (or a crash) from this operation.
    - *Hint:* You have moved the pointer outside the bounds of the original array, accessing unowned memory.

4. **Allocation Mismatch:** Write a function that calls `new int[10]` but mistakenly calls `delete ptr;` (the single-object form) instead of `delete[] ptr;`.
    - *Task:* Explain the specific C++ rule that is violated and the consequence (which is likely **Undefined Behavior**).
    - *Hint:* The array form of `delete` is required to properly call the destructors and manage the metadata for the array allocation.

---

7. Value Categories and References Deep Dive

Part III is dedicated to maximizing performance and safety in C++. The foundation of the C++ optimization strategy—**Move Semantics**—rests on a deep understanding of **Value Categories**. These categories determine an object's identity, lifetime, and, critically, whether its internal resources can be *stolen* instead of *copied*.

7.1 Lvalues, Rvalues, and Prvalues: Defining Object Identity

Every expression in C++ results in a value that belongs to one of three primary **value categories** (Lvalue, Rvalue, Prvalue). This is known as the **Lvalue/Rvalue dichotomy**.

Lvalues (Left-hand side values)

An **Lvalue** (ℓvalue) is an expression that designates a **named, identifiable region of storage** (a memory location) that persists beyond the current expression. Lvalues have **identity** and can be assigned to.

- **Rule:** If you can take the address of an expression using the `&` operator, it's generally an Lvalue.
- **Examples:** Named variables, references (`int x;`, `x` is an Lvalue), functions returning an Lvalue reference, class member access.

```
int x = 10;
int& ref = x; // x is an Lvalue; we can bind an Lvalue reference to it.
int* ptr = &x; // We can take its address.
```

Rvalues (Right-hand side values)

An **Rvalue** ($r$value) is a temporary value that is the result of an expression and is about to expire. Rvalues do **not** have a permanent, identifiable memory location that can be accessed later. They are often created on the stack only for the duration of a single expression.

- **Rule:** You generally cannot take the address of an Rvalue.
- **Examples:** Literal values (`10`, `"hello"`), the result of arithmetic operations (`x + y`), functions returning by value (`int get_val()`).

```
int result = x + 5;
// (x + 5) is an Rvalue: it's a temporary result with no name.
// result is an Lvalue (named variable).
```

The Modern Taxonomy (C++11/17)

For the purposes of Move Semantics, the categories are refined:

- **Prvalue (Pure Rvalue):** A temporary object produced by an expression (e.g., `x + 5`).
- **Xvalue (eXpiring Value):** An object that *has* an identity but whose resources are about to be destroyed and can thus be "stolen" (e.g., the result of `std::move(x)`).
- **Glvalue (Generalized Lvalue):** The union of Lvalues and Xvalues (objects that have identity).
- **Rvalue:** The union of Prvalues and Xvalues (objects whose contents can be moved from).

The critical takeaway: **Rvalues (Prvalues and Xvalues) are the targets of move semantics.**

7.2 The Need for Rvalue References

Before C++11, an Lvalue reference (`T&`) was the only way to pass an object without copying it (Chapter 4.4). However, Lvalue references **cannot bind to Rvalues** (temporaries), because binding to a temporary would allow its value to be modified—a violation of its transient nature.

```
void foo(int& ref) { /* ... */ } // Lvalue reference

int get_ten() { return 10; }

// foo(get_ten()); // COMPILE ERROR: Cannot bind Lvalue reference to Rvalue
(temporary)
```

Rvalue references (`T&&`) were introduced to solve this. An Rvalue reference is a new kind of reference that **can bind only to Rvalues**. This allows the compiler to specifically target temporary objects, enabling the efficient theft of resources (moving).

```
void bar(int&& ref) { /* ... */ } // Rvalue reference

int get_ten() { return 10; }

bar(get_ten()); // OK: The Rvalue 10 binds to the Rvalue reference.
// bar(x); // COMPILE ERROR: Cannot bind Rvalue reference to Lvalue (named
variable x).
```

The ability to bind an Rvalue reference to a temporary object is the **mechanism** that makes move constructors and move assignment operators (Chapter 8) possible.

7.3 Reference Collapsing and Forwarding References

When writing generic code (templates), it's often necessary to accept parameters that can be either an Lvalue or an Rvalue. This is where **Forwarding References** and **Reference Collapsing** come into play.

Forwarding References (formerly Universal References)

A **Forwarding Reference** is a template parameter declared as an Rvalue reference (`T&&`) where `T` is a deduced template type.

```
template <typename T>
void generic_func(T&& param) {
    // param is a Forwarding Reference
}
```

When you call `generic_func`, the compiler deduces `T` and applies the **reference collapsing rules** to determine the final type of `param`.

Reference Collapsing Rules

These rules dictate what happens when you combine two references (which only happens during template type deduction):

Original Call Type	Deduced T	Final Type of `param` (`T&&`)	Collapsed Result
Lvalue (`int&`)	`int&`	`(int&)&&`	$\rightarrow$ `int&` (Lvalue reference)
Rvalue (`int&&`)	`int`	`(int)&&`	$\rightarrow$ `int&&` (Rvalue reference)

- **The Key Rule:** An Lvalue reference combined with anything always results in an Lvalue reference. Only combining two Rvalue references results in an Rvalue reference.

This means a single signature (`T&& param`) can accept and preserve the exact value category (Lvalue or Rvalue) of the original argument.

7.4 The `std::move` Utility (A Cast, Not a Move)

The name `std::move` is highly misleading. It does **not** perform any data movement. It is a simple, unconditional **cast** to an Rvalue reference.

```
// Equivalent definition of std::move
template <typename T>
typename std::remove_reference<T>::type&& move(T&& t) noexcept {
    return static_cast<typename std::remove_reference<T>::type&&>(t);
}
```

The Purpose of `std::move`

`std::move` is used to convert an **Lvalue** into an **Xvalue** (an expiring value). This tells the compiler, "I know this is a named variable, but please treat it as a temporary so its resources can be stolen by a move constructor."

```
std::vector<int> source = {1, 2, 3};

// 1. source is an Lvalue (named).
```

```
// 2. std::move(source) casts 'source' into an Rvalue reference (Xvalue).
// 3. The move constructor of 'destination' is called, which steals source's
internal buffer.
std::vector<int> destination = std::move(source);
// WARNING: 'source' is now in a valid but unspecified state (its contents were
stolen).
```

7.5 The `std::forward` Utility

When using **Forwarding References** (Section 7.3) in generic code, we run into a problem: the generic parameter `param` inside the function body is always an **Lvalue** (it has a name inside the function). If we used `std::move(param)` to pass it to another function, we would *unconditionally* cast it to an Rvalue reference, which is wrong if the original argument was an Lvalue that should be copied.

`std::forward` solves this by performing a **conditional cast**.

- It checks the type `T` that was deduced by the reference collapsing rules.
- If T was deduced as an Lvalue reference (`int&`), `std::forward` casts the parameter back to an Lvalue reference (preserving the ability to copy).
- If T was deduced as a pure type (`int`), which happens when an Rvalue was passed, `std::forward` casts the parameter back to an Rvalue reference (enabling moving).

This is known as **Perfect Forwarding**: the inner function receives the argument with the exact same value category (Lvalue or Rvalue) as was passed to the outer function.

```
template <typename T>
void wrapper(T&& arg) { // arg is a Forwarding Reference
    // If the caller passed an Lvalue (copy), forward as Lvalue (copy)
    // If the caller passed an Rvalue (move), forward as Rvalue (move)
    process_resource(std::forward<T>(arg));
}
```

Key Takeaways

- **Lvalues vs. Rvalues:** Lvalues have identity (names and addresses); Rvalues are temporary, transient results.
- **Rvalue References (`T&&`):** This new type of reference binds only to Rvalues (temporaries), making it the essential mechanism to intercept and steal resources from objects about to be destroyed.
- **Forwarding References:** A specific form of `T&&` in template contexts that uses **reference collapsing** to perfectly preserve the Lvalue/Rvalue nature of the original argument.
- **`std::move` is a Cast:** `std::move` is an **unconditional cast** that converts a named Lvalue into an Rvalue, enabling its contents to be moved from. It does not perform the actual data transfer.
- **`std::forward` is Conditional:** `std::forward` is used within generic code (templates) to conditionally cast a forwarding reference back to its original value category, enabling **perfect forwarding** (copy if Lvalue, move if Rvalue).

Exercises

1. **Identify Value Categories:** For each expression below, state whether the result is an Lvalue or an Rvalue.
    - `std::string s1 = "hello";`
    - `s1 + " world"`
    - `s1`
    - `*(&s1)`
    - `std::move(s1)`
    - *Hint:* Only named variables and dereferenced pointers/references are Lvalues. `std::move` always yields an Rvalue.

2. **Rvalue Reference Binding Failure:** Write a small program with two variables, `int a = 5;` and `int b = 10;`. Attempt to declare an Rvalue reference: `int&& ref = a;`
    - *Task:* Observe the compiler error. Explain why an Rvalue reference cannot be initialized with the Lvalue `a`.
    - *Hint:* Binding an Rvalue reference to an Lvalue would let you modify a named object via a mechanism intended only for temporaries.

3. **The Misleading `std::move`:** Create a simple struct `Data` with a string member. Write a function `void process(Data&& d)` that takes an Rvalue reference. In `main()`, create a `Data d1`.
    - *Task:* Call the function using `process(std::move(d1))`. Explain why `d1` is an Lvalue *before* the call but is treated as an Rvalue *during* the call.
    - *Hint:* `std::move` is a cast that changes the *expression type*, not the *variable type*.

4. **Reference Collapsing Test:** Given the function template `template <typename T> void test_ref(T&& val);`.
    - *Task:* Call `test_ref` with `int i = 5;` and then with `5 + 5`. For each call, state the deduced type of `T` and the final, collapsed type of `val`.
    - *Hint:* Passing an Lvalue deduces T as $T&$; passing an Rvalue deduces T as T.

---

8. Move Semantics and State Control

Move Semantics is the performance optimization technique that defines Modern C++ development. It enables objects to transfer ownership of expensive internal resources—like large memory buffers or file handles—instead of wasting time performing a deep copy. This concept relies entirely on **Value Categories** (Chapter 7) to distinguish between temporary objects (safe to steal from) and named objects (must be copied).

8.1 Deep vs. Shallow Copy Review

When an object contains a **raw pointer** to dynamically allocated resources (e.g., a buffer on the heap), there are two ways to copy that object:

1. **Shallow Copy (The Default):** Only the object itself (and the raw pointer within it) is copied. Both the original and the copy point to the **same underlying resource**. This is what the compiler provides by default and is catastrophic for resource management.

- **Danger:** When the destructor is called on both the original and the copy, it results in **double deletion** of the shared resource, leading to a crash (Chapter 6.4).

2. **Deep Copy:** Both the object and the resource it points to are copied. A new, independent resource is allocated, and the data is copied into it. The original and the copy are now completely independent.
    - **Requirement:** If your class manages a raw resource (a raw pointer), you **must** implement deep copying yourself.

8.2 The Copy Constructor and Copy Assignment Operator

To implement deep copying, you must define the two special member functions that handle Lvalue-to-Lvalue (named object-to-named object) duplication:

1. The Copy Constructor (`T(const T& other)`)

Called when a new object is initialized from an existing object (e.g., `T b = a;` or `T b(a);` or when passing by value).

```
class ResourceWrapper {
private:
    int* data_ = nullptr;
    size_t size_ = 0;
public:
    // ... Constructor, Destructor ...

    // The Copy Constructor: Performs a deep copy
    ResourceWrapper(const ResourceWrapper& other) : size_(other.size_) {
        std::cout << "Copy Constructor (Deep Copy)\n";
        data_ = new int[size_]; // 1. Allocate a NEW resource
        std::copy(other.data_, other.data_ + size_, data_); // 2. Copy the data
    }
};
```

2. The Copy Assignment Operator (`T& operator=(const T& other)`)

Called when an existing object is assigned the value of another existing object (e.g., `T b; b = a;`). This requires care to manage the existing resource.

```
    // The Copy Assignment Operator: Manages existing resources
    ResourceWrapper& operator=(const ResourceWrapper& other) {
        std::cout << "Copy Assignment Operator\n";
        if (this != &other) { // 1. Check for self-assignment (a = a)
            delete[] data_; // 2. Release the existing resource
            size_ = other.size_;
            data_ = new int[size_]; // 3. Allocate new resource
            std::copy(other.data_, other.data_ + size_, data_); // 4. Copy data
        }
```

```
        return *this; // 5. Return reference to self
    }
```

8.3 The Move Constructor and Move Assignment Operator

Move Semantics is implemented via two special member functions that accept an **Rvalue reference** (`T&&`) as a parameter (Chapter 7).

Instead of allocating memory and copying data, they perform the **Move operation**:

1. **Theft:** Copy the internal resource pointer (the raw pointer) from the source object.
2. **Pacification:** Set the source object's internal resource pointer to `nullptr`.

By nulling out the source's pointer, the source's destructor (when called) will safely attempt to `delete[] nullptr;`, which is guaranteed to do nothing, preventing the resource from being destroyed twice.

1. The Move Constructor (`T(T&& other) noexcept`)

Called when a new object is initialized from an Rvalue (e.g., a function return, or an object explicitly cast with `std::move`).

```
    // The Move Constructor: Steals resources from an expiring Rvalue
    ResourceWrapper(ResourceWrapper&& other) noexcept
        : data_(other.data_), size_(other.size_) { // 1. Steal the resource

        std::cout << "Move Constructor (Theft)\n";
        other.data_ = nullptr; // 2. Pacify the source (null its pointer)
        other.size_ = 0;       // 3. Clear source size
    }
```

2. The Move Assignment Operator (`T& operator=(T&& other) noexcept`)

Called when an existing object is assigned the value of an Rvalue.

```
    // The Move Assignment Operator
    ResourceWrapper& operator=(ResourceWrapper&& other) noexcept {
        std::cout << "Move Assignment Operator\n";
        if (this != &other) { // 1. Check for self-assignment (optional for move)
            delete[] data_; // 2. Release the existing resource held by *this*

            // 3. Perform the theft
            data_ = other.data_;
            size_ = other.size_;

            // 4. Pacify the source
            other.data_ = nullptr;
            other.size_ = 0;
        }
```

```
        return *this;
    }
```

Note: Move operations are typically marked with `noexcept` to inform the compiler that they will not throw exceptions.

8.4 Compiler-Generated Defaults and Explicit Deletion (`= default`, `= delete`)

The C++ compiler automatically generates the four/five special member functions unless certain conditions are met (e.g., if you declare a destructor, the compiler will suppress the default move functions).

Modern C++ provides tools to control this generation explicitly:

Keyword	Use	Description
`= default`	`T() = default;`	Explicitly request the compiler to generate the standard default implementation for a special member function. Used to gain back a default implementation that was suppressed by other code.
`= delete`	`T(const T&) = delete;`	Explicitly prevent the compiler from generating or using a special member function.

Example: Making a Move-Only Class

If a resource cannot be copied (e.g., a unique file lock), you can enforce that by deleting the copy operations:

```
class UniqueResource {
public:
    UniqueResource() = default;

    // Explicitly delete copy operations
    UniqueResource(const UniqueResource&) = delete;
    UniqueResource& operator=(const UniqueResource&) = delete;

    // Use default move operations
    UniqueResource(UniqueResource&&) = default;
    UniqueResource& operator=(UniqueResource&&) = default;
};
```

This pattern is common; it forces users to use `std::move` if they need to transfer ownership, similar to how C#'s `StreamReader` often requires explicit closure or scope usage.

8.5 The Rule of Zero/Three/Five: Modern Class Design

The C++ community has codified the rules for managing the special member functions into clear design principles:

1. The Rule of Five (Pre-C++11/Legacy Design)

If you must manage a raw resource (like a raw pointer), you must deal with five specific operations:

1. **Destructor**
2. **Copy Constructor**
3. **Copy Assignment Operator**
4. **Move Constructor**
5. **Move Assignment Operator**

If you define *any one* of these, you must define them **all** to ensure proper resource management and prevent the disastrous effects of a shallow copy combined with a raw pointer.

2. The Rule of Three (Pre-C++11)

The rule of five's predecessor, which only mandated the first three (Destructor, Copy Constructor, Copy Assignment Operator) because move operations didn't exist yet.

3. The Rule of Zero (Modern C++ Best Practice)

The goal of Modern C++ is to avoid the complexity of the Rule of Five entirely by following the **Rule of Zero**:

> **A class that manages a resource should define zero custom special member functions.** Instead, it should delegate resource management to a member that already handles it.

This is achieved by storing raw resources inside smart wrappers (like `std::unique_ptr` or `std::vector`). Since these wrappers already implement the Rule of Five safely (they deep copy/move correctly), the compiler-generated default copy and move operations for the containing class will automatically call the safe, correct operations on the member wrappers.

Recommendation: Strive to follow the **Rule of Zero** by using containers and **Smart Pointers** (Chapter 9) for all resources. Only resort to the Rule of Five when implementing the resource manager itself (e.g., the smart pointer class).

Key Takeaways

- **Move vs. Copy: Copy Semantics** performs expensive deep duplication. **Move Semantics** performs cheap resource **theft** (shallow copy of pointers followed by nulling the source).
- **The Four Functions:** Move Semantics requires defining the **Copy Constructor** (`const T&`), **Copy Assignment Operator** (`const T&`), **Move Constructor** (`T&&`), and **Move Assignment Operator** (`T&&`).
- **The Move Operation:** The core of moving is taking the raw pointer from the Rvalue source and setting the source's pointer to `nullptr` to prevent double deletion.
- **Control Defaults:** Use `= default` to regain a compiler-generated function and `= delete` to suppress an unwanted operation (e.g., making a class move-only).
- **Rule of Zero:** The ultimate goal. Design your classes to manage no raw resources directly; instead, delegate resource management to members that handle it correctly (like smart pointers).

Exercises

1. **Shallow Copy Failure:** Implement the `ResourceWrapper` class from the chapter, including a constructor that allocates an integer array and a destructor that calls `delete[]`. **Do not** write a custom copy constructor or assignment operator.
    - *Task:* In `main()`, create two wrappers, `ResourceWrapper a;` and `ResourceWrapper b = a;`. Run the program and observe the crash. Explain why the compiler's default shallow copy failed.
    - *Hint:* The program will crash due to **double deletion** of the same heap resource.
2. **Enforcing Move-Only:** Take a simple class `Logger` and delete its copy constructor and copy assignment operator using the `= delete` syntax.
    - *Task:* Try to pass an instance of `Logger` to a function by value. The compiler should fail. Explain why this design choice is useful for objects like file handles or network connections.
    - *Hint:* Copying resources like file handles is often illogical; they should be unique or transferable.
3. **Manual Move Implementation:** Complete the implementation of the `ResourceWrapper` class by adding the **Move Constructor** and setting the source pointer to `nullptr`.
    - *Task:* In `main()`, initialize a new object using `ResourceWrapper b = std::move(a);`. Verify that `a.data_` is now `nullptr` and no copy was performed.
    - *Hint:* The move constructor should execute, and the copy constructor should not.
4. **The Rule of Zero Design:** Consider a class `Employee` that needs to manage a pointer to a `Passport` object.
    - *Task:* Sketch the definition of the `Employee` class by making the `Passport*` member into an `std::unique_ptr<Passport>`. Explain why this new design adheres to the **Rule of Zero** and requires no custom copy/move/destructor code.
    - *Hint:* The `unique_ptr` handles the destructor and move semantics automatically, and it deletes copy semantics, which the compiler-generated defaults inherit.

---

9. Smart Pointers and RAII

The journey through raw pointers, move semantics, and the dangers of manual memory management (Chapters 6, 7, 8) culminates here. **Smart Pointers** are the primary mechanism for implementing the core C++ memory safety paradigm known as RAII. **You should use smart pointers for virtually all heap allocations in modern C++.**

9.1 The RAII Principle: Resource Acquisition Is Initialization

RAII (Resource Acquisition Is Initialization) is the single most important idiom in C++ for safe resource management. It is the C++ answer to the problem solved by garbage collection (GC), but it achieves deterministic, rather than probabilistic, cleanup.

The principle states that **resource ownership must be tied to the lifetime of a stack-based object**.

The RAII Mechanism (The C++ Safety Net)

1. **Acquisition:** The resource (e.g., heap memory, a file handle, a network lock) is acquired in the **constructor** of a stack-allocated wrapper object.

2. **Initialization:** The wrapper object is initialized on the **Stack** (Automatic Storage Duration).
3. **Guaranteed Release:** When the stack-allocated wrapper object goes out of scope (e.g., the function returns or a block exits), its **destructor** is guaranteed to be called **deterministically** (Chapter 5.3).
4. **Release:** The destructor contains the code to safely release the resource (e.g., calling `delete`, closing the file, or releasing the lock).

Smart pointers are RAII wrapper classes designed to manage heap memory, ensuring that `delete` is called automatically when the pointer object leaves scope, eliminating the possibility of memory leaks from forgetting `delete`.

9.2 `std::unique_ptr`: Exclusive, Transferable Ownership

`std::unique_ptr` is the preferred and most efficient smart pointer, designed for situations where an object on the heap has **only one owner**.

Characteristics

- **Exclusive Ownership:** Only one `unique_ptr` can point to the resource at a time.
- **Zero Overhead:** At runtime, a `std::unique_ptr` is exactly the same size and speed as a raw pointer. It has **no runtime overhead** compared to manual `new/delete`.
- **Move-Only Semantics:** It cannot be copied (it deletes its copy constructor, Chapter 8.4). Its ownership can only be **transferred** using move semantics (`std::move` or returning from a function).

```
#include <memory>
#include <iostream>

// 1. Exclusive Ownership: only 'data' owns the object
std::unique_ptr<int> data = std::make_unique<int>(10);

// 2. Transfer Ownership (Move)
// The object pointed to by 'data' is now owned by 'new_owner'.
std::unique_ptr<int> new_owner = std::move(data);

// 3. Guaranteed Deletion
// When 'new_owner' goes out of scope, the destructor calls delete on the integer.
```

The ability to return a `unique_ptr` from a function is key: it returns the resource by value, which implicitly invokes the fast **move constructor** (Chapter 8.3) to transfer ownership to the caller.

9.3 `std::make_unique` vs. `new unique_ptr`

While you *can* construct a `unique_ptr` directly using `new`, the standard factory function `std::make_unique` (C++14) is strongly preferred for safety and efficiency.

Exception Safety

Using `std::make_unique` ensures **exception safety**. Consider the unsafe construction:

```
// UNSAFE: Potential leak if some_func() throws an exception
process(std::unique_ptr<T>(new T()), some_func());
```

In the unsafe line, the compiler might perform the steps out of order:

1. `new T()`: Memory is allocated.
2. `some_func()`: This function runs.
3. `std::unique_ptr<T>()`: The smart pointer is constructed.

If `some_func()` throws an exception *after* the raw memory is allocated but *before* the smart pointer is constructed, the raw memory is never wrapped and is permanently leaked.

`std::make_unique` performs the memory allocation and the smart pointer construction in a single, atomic step, guaranteeing that if an exception occurs, no memory is leaked.

Best Practice: Always use **`std::make_unique<T>(args)`** for creating `unique_ptr` objects.

9.4 `std::shared_ptr`: Shared Ownership and Reference Counting

`std::shared_ptr` is designed for complex scenarios where multiple, non-exclusive owners need to share a resource on the heap.

Characteristics and Overhead

- **Shared Ownership:** Multiple `shared_ptr` objects can point to the same resource.
- **Reference Counting:** The resource is only deleted when the last `shared_ptr` pointing to it is destroyed or reset. This requires **runtime overhead** in the form of a **control block**.
- **Control Block:** A separate, small allocation alongside the object that contains the **reference count** and the **weak count**.

```
// Two shared pointers point to the same resource. Count is 2.
std::shared_ptr<int> p1 = std::make_shared<int>(50);
std::shared_ptr<int> p2 = p1;

// When p2 is destroyed, the count drops to 1.
// When p1 is destroyed, the count drops to 0, and the destructor calls delete.
```

Because of the reference counting overhead, `std::unique_ptr` is preferred unless true shared ownership is necessary.

9.5 `std::make_shared` for Performance and Safety

Similar to `unique_ptr`, `std::make_shared` is the safe factory for `shared_ptr`, but it provides a critical performance benefit: **single-allocation optimization**.

Single Allocation

When using `new` for `shared_ptr`:

1. Allocation 1: The object itself (`new T()`).
2. Allocation 2: The control block (reference count, weak count).

When using `std::make_shared`:

- **Single Allocation:** Both the object and the control block are allocated in a single block of contiguous memory.

This reduces the total memory required, improves memory locality (leading to better cache performance), and is exception-safe.

Best Practice: Always use `std::make_shared<T>(args)` for creating `shared_ptr` objects.

9.6 `std::weak_ptr`: Observer Pointers and Breaking Circular References

A `std::weak_ptr` is a non-owning **observer** pointer designed to monitor a resource managed by a `std::shared_ptr`.

Characteristics and Use

- **Non-Owning:** A `weak_ptr` does not affect the `shared_ptr`'s reference count.
- **Safe Access:** You cannot directly dereference a `weak_ptr`. To safely access the resource, you must call its `lock()` method, which returns a temporary `std::shared_ptr` if the resource still exists. If the resource has been deleted, `lock()` returns `nullptr`.

The primary use case for `weak_ptr` is preventing **circular references**, which are the only way to cause a memory leak with `shared_ptr`.

Breaking Circular References

A circular reference occurs when two objects manage each other via `shared_ptr`.

```
class Parent; // Needs a forward declaration
class Child {
public:
    // Child owns Parent (shared_ptr)
    std::shared_ptr<Parent> parent_ptr;
};

class Parent {
public:
    // Parent should NOT own Child to avoid circular reference.
    std::shared_ptr<Child> child_ptr; // Problematic
};
```

If `p` owns `c` and `c` owns `p`, the reference count for both will never drop to zero (it will stay at 1), and neither object's memory will ever be freed.

Solution: The weaker link in the relationship (e.g., the back-reference from the child to the parent) must be changed to a `std::weak_ptr`:

```
class Parent { /* ... */ };
class Child {
public:
    // Parent should monitor Child, not own it.
    std::weak_ptr<Parent> parent_ptr; // Breaks the cycle
};
```

The `weak_ptr` allows the connection without maintaining the reference count, ensuring that when all external `shared_ptr`s are destroyed, the memory is safely cleaned up.

Key Takeaways

- **RAII is the C++ Safety Net: Resource Acquisition Is Initialization** is the core idiom. Smart pointers are stack-allocated wrapper objects whose destructors guarantee resource release.
- **`std::unique_ptr` is the Default:** Use `std::unique_ptr` for exclusive, single ownership. It is efficient, has **zero runtime overhead**, and is move-only.
- **`std::make_unique` is Standard:** Always use `std::make_unique` to create `unique_ptr` for exception safety.
- **`std::shared_ptr` Has Overhead:** Use `std::shared_ptr` only when true shared ownership is required, as it imposes overhead via the **reference counting control block**.
- **`std::make_shared` is Better:** Always use `std::make_shared` for `shared_ptr` to gain the performance and memory benefits of **single-allocation optimization**.
- **`std::weak_ptr` Breaks Leaks:** Use `std::weak_ptr` as a non-owning observer, primarily to **break circular references** between `shared_ptr` objects, preventing memory leaks.

Exercises

1. **Unique vs. Shared Ownership:** Write two blocks of code. In the first, try to copy a `std::unique_ptr` (`std::unique_ptr<int> p2 = p1;`). In the second, try to copy a `std::shared_ptr`.
    - *Task:* Explain the compiler's response in the first case and the runtime mechanism in the second case.
    - *Hint:* `unique_ptr` has deleted its copy constructor. `shared_ptr` increments its reference count.
2. **`std::unique_ptr` Resource Release:** Create a simple class with a destructor that prints its name. Write a function where you create an instance of this class on the heap using `std::make_unique`.
    - *Task:* Show that the destructor is called automatically when the function returns, proving the RAII principle. Then, call `ptr.release()` on the unique pointer before the function returns. Observe that the destructor is *not* called. Explain why `release()` defeats the RAII guarantee.
    - *Hint:* `release()` returns the raw pointer without deleting the resource, making the memory leak your responsibility.
3. **The `std::weak_ptr` Access:** Create a `std::shared_ptr<std::string>` named `data`. Create a `std::weak_ptr<std::string>` named `observer` that monitors `data`. Reset `data` to `nullptr`.
    - *Task:* Try to access the string's content directly through `observer`. Then, try to access it via `observer.lock()`. Explain why the direct access fails and why `lock()` is the only safe way.

- *Hint:* `lock()` converts the observer into a temporary `shared_ptr` that safely checks if the resource is still alive.

4. **Circular Reference Creation:** Model the `Parent` and `Child` classes from Section 9.6, initially using `std::shared_ptr` for both the forward and backward references.
    - *Task:* Create instances of both, assign them to each other, and let them go out of scope. (You'll need a way to detect the leak, like printing a message in their destructors). Explain why the destructors are never called.
    - *Hint:* The cycle means the reference count for each object is permanently held at 1 by the other object.

Where to go Next

- **Part I:: The C++ Ecosystem and Foundation:** This section establishes the philosophical and technical underpinnings of C++, focusing on compilation, linking, and the modern modularization system.
- **Part II: Core Constructs, Classes, and Basic I/O:** Here, we cover the essential C++ syntax, focusing on differences in data types, scoping, `const` **correctness**, and the function of **lvalue references**.
- **Part III: The C++ Memory Model and Resource Management:** The most critical section, which deeply explores raw pointers, value categories, **move semantics**, and the indispensable role of **smart pointers** and the **RAII** idiom.
- **Part IV: Classical OOP, Safety, and Type Manipulation:** This part addresses familiar object-oriented concepts like **inheritance** and **polymorphism**, emphasizing C++'s rules for **exception safety** and type-safe casting.
- **Part V: Genericity, Modern Idioms, and The Standard Library:** Finally, we explore the advanced capabilities of **templates**, **C++20 Concepts**, **lambda expressions**, and the power of the **Standard Library containers** and **Ranges** for highly generic and expressive code.
- **Appendix:** Supplementary materials including coding style guidelines, compiler flags, and further reading.