Chapter 9: Value Semantics and Special Members

Why this matters: This chapter explores a core C++ design philosophy: **value semantics**. The goal is to make your custom classes behave just like built-in types like `int`. When you write `int y = x;`, you get a completely new, independent `int y` with the same value as `x`. Modifying `y` has no effect on `x`. Value semantics means your objects should have this same independent quality. This is in contrast to **reference semantics** (common in languages like Java or C#), where copying an object often just copies a reference, and both variables point to the *same* underlying object.

To achieve value semantics for classes that manage resources (like memory), you must understand and correctly implement a special set of member functions.

9.1 Copy Constructor and Assignment

The Problem: If your class manages a resource via a raw pointer, the default copy operations generated by the compiler are **always wrong**. The compiler performs a **shallow copy**, which just copies the value of the pointer (the memory address). This leads to two objects pointing to the same resource.

```cpp
// A class that manages a resource (a C-style string)
class MyString {
private:
    char* data_;
    size_t size_;
};

MyString s1("hello");
MyString s2 = s1; // DANGER! Shallow copy by default.
                  // s1.data_ and s2.data_ now hold the SAME memory address.
```

This leads to disaster: **the double-free bug**. When `s2` goes out of scope, its destructor frees the memory. When `s1` goes out of scope, its destructor tries to free that *exact same memory again*, causing a crash.

The Solution: Deep Copy To fix this, you must perform a **deep copy**: allocate new memory for the copy and copy the *contents* of the original resource.

The Rule of Three (The Classic Rule)

If your class needs a custom destructor, a custom copy constructor, or a custom copy assignment operator, it almost certainly needs all three.

1. **Destructor:** Releases the resource. `cpp     ~MyString() { delete[] data_; }`
2. **Copy Constructor:** Creates a new object as a copy of an existing one. `cpp     // Called for: MyString s2 = s1; or MyString s2(s1);     MyString(const MyString& other) {         size_ = other.size_;         data_ = new char[size_ + 1]; // 1. Allocate new memory         std::memcpy(data_, other.data_, size_ + 1); // 2. Copy the data     }`
3. **Copy Assignment Operator:** Overwrites an existing object with a copy of another.

   ```cpp
   // Called for: s2 = s1;
   MyString& operator=(const MyString& other) {
       // 1. Self-assignment check: crucial to prevent self-destruction!
       if (this == &other) {
           return *this;
       }

       // 2. Free the old resource this object was holding
       delete[] data_;

       // 3. Allocate new memory and copy the data from the other object
       size_ = other.size_;
       data_ = new char[size_ + 1];
       std::memcpy(data_, other.data_, size_ + 1);

       // 4. Return a reference to this object to allow chaining (a = b = c)
       return *this;
   ```

```
}
```

9.2 The Rule of Five/Zero

Why the change? C++11 introduced move semantics (Chapter 13), which added two new special members. The rules evolved because defining a copy constructor tells the compiler you're doing special resource management, which in turn **disables the automatic generation of the new, more efficient move operations**.

The Five Special Member Functions: 1. Destructor 2. Copy Constructor 3. Copy Assignment Operator 4. **Move Constructor (C++11)** 5. **Move Assignment Operator (C++11)**

This leads to the modern **Rule of Five**: If you declare any of the five special member functions, you should define or `=default` all of them to ensure your class behaves correctly and efficiently.

The Rule of Zero (The Modern Ideal)

What is it? The best and safest approach is to **not declare any of the special member functions yourself**.

How is this possible? You achieve this by composing your class from other well-behaved types that already manage their own resources correctly. Use `std::string` instead of `char*`. Use `std::vector` instead of `T*`. Use `std::unique_ptr` for dynamically allocated objects. These classes already implement all the special member functions correctly.

When your class contains members like these, the compiler-generated special members will automatically do the right thing by calling the corresponding special member on each of your class's members.

```
#include <string>
#include <vector>

// This class follows the Rule of Zero.
// It is correct, copyable, movable, and destructible without any custom code.
class Person {
private:
    std::string name_;
    std::vector<int> favorite_numbers_;
    // The compiler-generated destructor calls ~vector() then ~string().
    // The compiler-generated copy constructor calls the string copy constructor then the vector copy constructor.
    // ...and so on for all five special members.
};
```

Using `=default` and `=delete`

C++11 gives you explicit control over the generation of special member functions.

- **`= default`**: Explicitly tells the compiler to generate the default implementation. This is useful if you need to define one special member (e.g., a custom constructor) but want the compiler to generate the rest as normal.
- **`= delete`**: Forbids the compiler from generating a function. This is a powerful tool to create types that cannot be copied or moved. This is essential for classes that represent a unique resource, like a file handle or a network connection, where copying doesn't make logical sense.

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

    // Forbid copying by deleting the copy constructor and copy assignment operator
    UniqueFile(const UniqueFile&) = delete;
    UniqueFile& operator=(const UniqueFile&) = delete;

    // (Move operations would typically be defined here)
private:
    // ... file handle ...
};

UniqueFile f1;
// UniqueFile f2 = f1; // COMPILE ERROR: use of deleted function
```

Projects for Chapter 9

Project 1: The Deep-Copy `IntVector` Class

- **Problem Statement:** Implement a simple `IntVector` class that acts like a dynamic array of integers. It should manage a raw `int*` pointer to heap-allocated memory. You must correctly implement the **Rule of Three**: a destructor, a copy constructor (for deep copies), and a copy assignment operator (for deep copies, including the self-assignment check). Provide methods to `push_back` an element and `get_at(index)`.
- **Core Concepts to Apply:** Rule of Three, deep vs. shallow copy, copy constructor, copy assignment operator, manual memory management with `new[]` and `delete[]`.

Project 2: The Rule-of-Zero `Course`

- **Problem Statement:** Create a `Course` class that follows the **Rule of Zero**. It should contain a `std::string` for the course title and a `std::vector<std::string>` to store the names of enrolled students. Write a `void enroll(const std::string& student_name)` method. In `main`, create a `Course` object, enroll several students, and then create a copy of the course. Modify the original course (enroll another student). Verify that the copy remains unchanged, proving that the compiler-generated copy constructor performed a deep copy for you.
- **Core Concepts to Apply:** Rule of Zero, composition with `std::string` and `std::vector`.

Project 3: The Non-Copyable `DatabaseConnection`

- **Problem Statement:** A database connection should be unique; it doesn't make sense to copy it. Create a `DatabaseConnection` class. The constructor should take a connection string and print `"Connection established to [string]"`. The destructor should print `"Connection closed."`. Use the `= delete` syntax to make the class non-copyable. In `main`, create an instance of your class. Then, add code that tries to copy it (`DatabaseConnection conn2 = conn1;`) and verify that it produces a compile-time error.
- **Core Concepts to Apply:** `= delete`, creating non-copyable types, RAII.