CS 247

Software Engineering Principles

Chapter 3

Special Member Functions

Victoria Sakhnini

Table of Contents

Introduction

Every C++ class has a small set of operations that are so fundamental to using it - constructing, destroying, copying, moving, comparing - that the language gives them privileged status. These are the special member functions, and the compiler will silently provide default versions of most of them if you do not write your own. That convenience is both a gift and a hazard: when the default versions do the right thing, you get correct behaviour with no work; when they do not, you can ship subtle bugs that only show up when an object is passed by value, returned from a function, stored in a container, or in one of half a dozen other situations.

Few things are as central to effective C++ programming as knowing exactly when the compiler inserts these functions into your classes and what they do when it does. This chapter walks through all six special member functions, plus equality (which the compiler does not generate but belongs in the same conversation about value semantics). For each one, we cover the default behaviour, the cases where it goes wrong, and how to write a correct hand-rolled version.

The Six Special Member Functions

- There are six special member functions:
- Default constructor - constructs an object with no arguments. Generated only if you declare no constructor at all.
- Destructor - cleans up when the object goes out of scope. Always generated unless you provide your own.
- Copy constructor - constructs a new object from an existing one (used implicitly when an object is passed or returned by value, or initialized with another of its type).
- Copy assignment operator - assigns the value of one existing object to another existing object.
- Move constructor - constructs a new object by taking ownership of an rvalue's resources, leaving the source in an empty-but-destructible state.
- Move assignment operator - assigns from an rvalue by taking its resources rather than copying.

We will also cover operator== separately, since the compiler never generates an equality operator, and you are always on your own with it.

> 💡 **Tip:** The interlocking rules about when the compiler generates which special member function are easy to forget. Skim the compiler-default table at the end of this chapter before reading on - it will appear at first to be an unmotivated grid of "defaulted/not declared/deleted," but each cell follows from a small set of rules that the rest of the chapter explains.

A Running Example

To make the rules concrete, we will work through one class per chapter. Each class contains a representative sample of the kinds of members the special operations interact with: a base-class subobject (which has its own special members), a member object (likewise), a managed pointer to a heap-allocated object (which determines what copying and moving look like), and a plain integer (which is trivial to copy and move).

```
class Base { /* ... */ };       // some base class
class C    { /* ... */ };       // some member-object type

class MyClass: public Base {
private:
    C comp_;       // member object of type C
    std::unique_ptr<C> ptr_;        // managed pointer to a C
    int simple_;     // simple primitive

public:
    MyClass();
    ~MyClass();
    MyClass(const MyClass& other);
    MyClass& operator=(const MyClass& other);
    MyClass(MyClass&& other) noexcept;
    MyClass& operator=(MyClass&& other) noexcept;

    [[nodiscard]] friend bool operator==(const MyClass& lhs,
                                         const MyClass& rhs);
    [[nodiscard]] friend bool operator!=(const MyClass& lhs,
                                         const MyClass& rhs) {
        return !(lhs == rhs);
    }
};
```

Notice that we explicitly declared all six special member functions. In practice, well-written modern C++ aims to declare as few of them as possible (see the Rule of Zero later), but for teaching purposes, it is useful to see every signature in one place. Each section below covers one of these functions, explains what it does, and shows how to implement it correctly.

> 💡 **Tip:** The `[[nodiscard]]` attribute on `operator==` and `operator!=` asks the compiler to warn if the caller does not use the result. Writing `if (a == b)` is fine; writing `a == b;` on its own line - which would compute a comparison and throw away the answer - is almost always a bug, and the warning catches it.

We are also using `std::unique_ptr<C>` rather than a raw `C*` for the managed pointer. `std::unique_ptr` handles memory management automatically: the wrapped pointer is deleted when the `unique_ptr` goes out of

scope. This eliminates an entire category of bugs (forgetting to delete in the destructor) and simplifies several special member functions, since the `unique_ptr` already knows how to move correctly.

Constructors and the Default Constructor

A constructor initializes a new object. We covered constructors in detail in Chapter 2, but the relevant rule for this chapter is simple: every constructor must put every data member into a well-defined state. If a member is left uninitialized at the end of the constructor, reading it later results in undefined behaviour.

> 💡 **Best Practice:** Every constructor should ensure that every data member is initialized to an appropriate value. The member initialization list is the right tool: it constructs each member in place rather than default-constructing and then assigning. As an aside, brace-initializing an object - Rational r{}; - calls default constructors for the built-in members too. The result is that Rational r{}; gives 0/0, while Rational r; with the same class would leave the members uninitialized.

Compiler-Generated Default Constructor

If you declare no constructor for a class, the compiler synthesizes a default constructor for you using memberwise initialization:

- Simple data members (built-in types like int, double, pointers): uninitialized.
- Pointer members: uninitialized (just an arbitrary address).
- Member objects: initialized using each member type's default constructor.
- Inherited members: initialized using the base class's default constructor.

Notice the asymmetry: object-typed members are properly default-constructed (the compiler recursively calls their default constructors), but built-in primitives are left in whatever state the memory happens to be in. For our MyClass, the compiler-generated default constructor would build comp_ via C's default constructor, build ptr_ via unique_ptr's default constructor (which sets it to nullptr), and invoke Base's default constructor, but leave simple_ as garbage.

> ⚠️ The compiler only synthesizes a default constructor if you declare no constructors at all. The moment you write any constructor - even one with all arguments - the implicit default constructor disappears, and a line like MyClass x; stops compiling unless you also write MyClass() explicitly or write MyClass() = default to ask for the compiler's version back.

The Destructor

A destructor is the inverse of a constructor: it runs when an object's lifetime ends and is responsible for releasing any resources the object holds (closing files, freeing heap memory, releasing locks, and so on). Destructors are called automatically - when a local variable goes out of scope, when a temporary expires, when a heap object is deleted, or when a container shrinks - and they cannot be called with arguments because there is exactly one way to destroy an object.

Compiler-Generated Destructor

Unless you write your own destructor, the compiler generates one. It performs memberwise destruction:

- Simple data members: deallocated (i.e., the memory they occupy on the stack is freed when the enclosing scope ends).
- Pointer members: the pointer itself is deallocated, but the pointee is NOT deleted. The compiler has no way of knowing whether the pointer owns its target or is just a weak reference.
- Member objects: cleaned up by each member type's destructor.
- Inherited members: cleaned up by the base class's destructor.

This last point about raw pointers is exactly why we prefer `std::unique_ptr<C>` to a `raw C*`. The compiler-generated destructor would correctly destroy a `unique_ptr` (which in turn deletes the C it owns); but with a raw pointer, the compiler-generated destructor would leak. Using a smart pointer means the default destructor does the right thing, so we can often skip writing a destructor altogether.

When You Need a Custom Destructor

A class needs a custom destructor when it acquires a resource that the compiler-generated destructor would not release. Typical signs:

- A constructor or mutator allocates memory with new, and the class is responsible for deleting it.
- A constructor opens a file, acquires a lock, opens a socket, or grabs any other resource that needs explicit release.
- You want to add logging or invariant checks at the moment of destruction.
- You want to override the destructor in a derived class, in which case the base destructor must be declared virtual.

For polymorphic class hierarchies, the base-class destructor must be virtual so that deleting a derived object through a base pointer calls the correct destructor. If you want the compiler-generated body but still need the function to be virtual, use the = default syntax:

```
class Animal {
public:
    virtual ~Animal() = default;   // virtual, but body is compiler-generated
};
```

The Copy Constructor

A copy constructor constructs a new object whose value is equal to that of an existing object. The signature has a specific form:

```
MyClass(const MyClass& other);
```

The argument is passed by reference because passing the source by value would require a copy, and the copy constructor is exactly the function we are defining. References to the rescue: a const reference lets us read from the source without copying it. The const matters too, since we never need to modify the source while copying from it.

Whenever the compiler needs to copy an object, it calls the copy constructor. The common cases:

- Passing an object to a function by value.
- Returning an object from a function by value (subject to copy/move elision - covered later).
- Initializing one object from another of the same type: `MyClass b = a;` or `MyClass b(a);`

Consider this little Money example. How many copies are made?

```
class Money;
Money operator+(Money m, Money n);   // takes BOTH operands by value

int main() {
    Money m;
    Money n{m};      // (1) copy constructor - constructs n from m
    Money p = m;     // (1) copy constructor - same idea, different syntax
    p = p + n;       // copy ctors fire for the by-value parameters of operator+
}
```

On the line where `p = p + n;` runs, the by-value parameters of `operator+` each require a copy constructor call. Returning the sum by value typically does NOT require a copy in modern C++ because of return-value optimization (RVO) - the compiler builds the returned object directly in p's location rather than constructing a temporary and copying it.

> 💡 **Tip:** You only need to think about copy constructors if your class is going to be passed by value, returned by value, or stored in something that copies it (like std::vector growing). If those never happen - for example, if the class is non-copyable for design reasons - you do not need to write a copy constructor. The default version is harmless when it never runs.

Shallow Copy vs. Deep Copy

When a class contains a pointer to a heap-allocated object, the question "how should copying work?" has two reasonable answers, and the wrong one is dangerous.

Shallow copy. Copy the object's fields verbatim, including pointer addresses. After the copy, the original and the copy share the same pointer value - they refer to the same underlying object. This is what the compiler does by default for raw pointers.

Deep copy. Copy the object's fields, but for each pointer, allocate a fresh copy of the pointee and store the new pointer. After the copy, the original and the copy refer to distinct underlying objects that happen to have the same value.

Shallow copy is fast, but creates a sharing relationship the program did not ask for. Two `MyClass` objects that share a `ptr_` have an interlocking lifetime - neither can independently destroy the pointed-to object without breaking the other. Worse, if both eventually call delete on the shared pointer in their destructors, you have a double-free, which is undefined behaviour and often a security vulnerability.

Deep copy is the safer default for any class that conceptually owns its pointed-to data. The two copies become independent, so each can mutate, destroy, or outlive the other without affecting the other.

> 💡 **Tip:** The choice between shallow and deep copy is part of the design of the ADT, not a low-level implementation detail. Entity classes that should never be copied at all use neither - they delete the copy constructor outright. Value classes that wrap heap data want deep copies. Smart pointers (std::unique_ptr, std::shared_ptr) build different sharing semantics into their type, so you do not have to write them by hand.

Compiler-Generated Copy Constructor

If you do not provide a copy constructor (and have not declared any of the other "user-disabling" special members), the compiler generates one using memberwise copy:

- Simple data members: bitwise copy.
- Pointer members: bitwise copy of the pointer (i.e., a shallow copy).
- Member objects: copied using each member type's copy constructor (which is itself recursively memberwise unless that member writes its own).
- Inherited members: copied using the base class's copy constructor.

The recursive structure is elegant: the compiler trusts that each member type knows how to copy itself correctly and applies the copy operation member by member. At the leaves - for built-in types and raw pointers - the operation is a bitwise copy. That is what makes raw pointers dangerous: their default copy is shallow, and the compiler will happily generate a shallow-copy copy constructor for any class that contains one.

⚠️ For a class that owns heap memory through a raw pointer, the compiler-generated copy constructor is almost certainly wrong. It produces aliased pointers (both objects pointing at the same target), which will lead to double-free crashes when both objects are eventually destructed. This is the canonical reason for hand-writing a copy constructor: to prevent the compiler-generated shallow copy from being used.

Writing a Deep Copy Constructor

A correct deep copy constructor for `MyClass` copies the base subobject, the member object, and the simple integer, and - critically - allocates a new C and copies the value it points to:

```
MyClass(const MyClass& other)
    : Base(other),                                          // 1
      comp_(other.comp_),                                   // 2
      ptr_(other.ptr_ ? std::make_unique<C>(*other.ptr_)    // 3
                      : nullptr),
      simple_(other.simple_)                                // 4
{
    std::cout << "MyClass Copy Constructor called\n";
}
```

Reading the initializer list line by line:

1. `Base(other)` invokes the base class's copy constructor on the base subobject of other. Inheritance and slicing work correctly because of this call.
2. `comp_(other.comp_)` invokes C's copy constructor with `other.comp_` as its source.
3. `ptr_` uses a ternary to handle the `nullptr` case. If the source pointer is null, we leave the new pointer null; otherwise we allocate a fresh C on the heap, initialized as a copy of whatever `other.ptr_` pointed to. `std::make_unique<C>(*other.ptr_)` does both - it allocates and copy-constructs in a single call, and the result is wrapped in a `unique_ptr` that takes ownership.
4. `simple_(other.simple_)` does an integer copy.

💡 **Best Practice:** Copy simple data members and object members via their copy constructors invoked in the initialization list, not by assignment in the body. The initialization-list version is more efficient (no default-construct-then-assign) and more consistent with construction in general. Reserve the body for actions that cannot be expressed as initialization - logging, validation, allocation of resources whose constructors take complex arguments.

The Copy Assignment Operator

Copy assignment is similar to the copy constructor, but with one key difference: the destination of the assignment already exists. The signature is `operator=`.

```
MyClass& operator=(const MyClass& other);
```

Returning a reference to `*this` lets clients chain assignments: `a = b = c` works by computing `b = c` first (which returns a reference to `b`), and then `a =` (that result). Without the return reference, the second assignment would have nothing to assign from.

Compiler-Generated Copy Assignment

If you do not provide a copy or move constructor or copy or move assignment, the compiler generates a copy assignment operator using memberwise assignment:

- Simple data members: bitwise copy.
- Pointer members: bitwise copy (shallow assignment).
- Member objects: assigned using each member type's assignment operator.
- Inherited members: assigned using the base class's assignment operator.

The default has two problems, exactly parallel to the copy constructor case:

Shallow pointer assignment. The compiler-generated version copies pointer values rather than what they refer to, producing aliased pointers. Same diagnosis as before, same fix: hand-write the operator if your class owns heap memory through a raw pointer.

No cleanup of the old value. When you assign `x = y`, the destination `x` already had a value. If that value owned heap memory through a raw pointer, simply overwriting the pointer leaks the old memory. The compiler-generated assignment operator does not call delete on anything - it just overwrites. Even more reason to either use smart pointers (which clean up automatically when reassigned) or hand-write the operator.

Standard Format of operator=

A correct hand-written copy assignment operator follows a standard pattern. Here is the version for `MyClass`:

```
MyClass& operator=(const MyClass& other) {
    if (this == &other)              // 1. self-assignment check
        return *this;
```

```
    Base::operator=(other);          // 2. assign the base subobject
    comp_   = other.comp_;           // 3. assign each member by value
    simple_ = other.simple_;
    ptr_    = other.ptr_                                        // 4.
              ? std::make_unique<C>(*other.ptr_)
              : nullptr;

    std::cout << "MyClass Copy Assignment Operator called\n";
    return *this;                    // 5. return reference to self
}
```

Five things deserve comment.

1. Self-assignment check. A line like `x = x` looks pathological, but it can happen indirectly - for example, when two pointers alias the same object and one is assigned through the other. Without the check, the assignment can delete the source's data before copying it (because the source and destination share state), corrupting the object. The early-out for self-assignment short-circuits this. It is also a small optimization for the legitimate self-assignment case.

2. Base class assignment. The base subobject has its own assignment operator that updates its state. Calling `Base::operator=(other)` routes through it, ensuring that base-class invariants are preserved.

3. Member-wise assignment. For each member object and simple data member, use that member type's assignment operator (or just integer assignment for primitives).

4. Deep assignment for the pointer. The right-hand side is a ternary that allocates a new C if the source is non-null. Assigning to ptr_ (which is a unique_ptr) automatically deletes whatever it previously owned - this is one of the reasons unique_ptr is preferred over raw pointers. With a raw pointer, you would need an explicit delete ptr_; before the new allocation, and you would also have to deal with the exception-safety concern of what happens if the new allocation throws.

5. Return *this. Required for the chained-assignment idiom to work.

> 💡 **Tip:** When you assign to a unique_ptr, the old pointer is automatically deleted before the new value is stored. This is one of the headline benefits of using smart pointers: the bookkeeping you would otherwise have to do by hand (and easily forget) is built into the type.

> 💡 **Tip:** std::make_unique either succeeds completely (returning a unique_ptr that owns a new object) or throws without leaking memory. This exception safety is essential: if you had instead written new C(*other.ptr_) and the C constructor threw, the freshly allocated memory would leak. make_unique was specifically designed to close that gap.

The Copy-and-Swap Idiom

Hand-written copy constructors and copy assignment operators tend to duplicate a lot of work - both need to copy every member, both need to handle the pointer-allocation case, and both need to handle base classes. Worse, they need to handle that logic correctly in both places, and bugs creep in when you fix one but not the other.

The copy-and-swap idiom unifies them. The idea: when assigning `x = y`, do not write out the assignment logic from scratch. Instead, (1) build a temporary `MyClass copy{m}` using the copy constructor, (2) swap the contents of `*this` with the temporary, (3) return `*this`, and (4) let the temporary go out of scope, taking the old contents with it.

```
MyClass& MyClass::operator=(const MyClass& m) {
    Base::operator=(m);
    simple_ = m.simple_;       // 1. (re)use member's operator=
    comp_   = m.comp_;

    MyClass copy{m};           // 2. MyClass copy constructor
    C* temp;

    temp        = copy.ptr_;   // 3. swap ptr_ data members
    copy.ptr_   = ptr_;
    ptr_        = temp;

    return *this;              // 4. return reference to self
}                              //    (copy destructor cleans up the old data)
```

When the copy goes out of scope at the closing brace, its destructor runs - and because we swapped the pointers, the destructor cleans up what was originally our old data. No manual delete, no leaked memory. The copy constructor was already responsible for implementing the deep-copy logic, so we get that logic "for free" in operator= without rewriting it.

> 💡 **Tip:** The pattern shown above swaps only the pointer member, while the simple data and member objects are assigned directly. A more idiomatic version swaps all of them by either using a swap function or calling std::swap on *this and copy. The simpler version above is enough to illustrate the technique, but a full implementation would extract the swap into its own method.

Copy-and-swap has clear advantages - less duplication, automatic strong exception safety (if the copy constructor throws, `*this` is untouched), and a single place to maintain the copy logic. But it also has a cost.

⚠ Copy-and-swap is highly inefficient if all you want to do is overwrite a few fields. To assign, you copy-construct the whole source object (which may include allocating fresh heap memory for every pointer member), swap the pointers, then immediately destroy the temporary (which frees the just-allocated memory that you no longer want). For container-heavy types, informal benchmarks have shown the copy-and-swap implementation running up to 8× slower than a hand-written assignment operator. Use it when correctness and code clarity matter more than the last few percent of performance.

Historically, the copy-and-swap idiom was the "right answer" to a question students often asked: "Can we implement `operator=` using the copy constructor?" Before C++11, the answer was no - the copy constructor produced a temporary that could not be returned. With swap, we get the same effect: a temporary built by the copy constructor takes the old data with it when it dies. Modern C++ adds move semantics, which give an alternative that handles many of the same problems without the unconditional copy. We turn to that next.

What you would actually use in production code is:

```
void swap(MyClass& a, MyClass& b) {
   using std::swap;
   swap(static_cast<Base&>(a), static_cast<Base&>(b));
   swap(a.simple_, b.simple_);
   swap(a.comp_, b.comp_); swap(a.ptr_, b.ptr_);
}

MyClass& operator=(MyClass m) {
     swap(*this, m);
     return *this;
}
```

lvalue and rvalue

Before we can talk about move semantics, we need vocabulary for the two kinds of expressions C++ distinguishes between. The names are historical and slightly confusing, but the concepts are not difficult.

lvalue. Anything with an identity in the program - a variable, a function parameter, a dereferenced pointer, an array element. The defining property is that it has an address: you can take `&`expression and get a pointer to it. The name "lvalue" comes from "left-hand value," because lvalues are exactly the things that can appear on the left side of an assignment (you can store into them).

rvalue. An expression with no identity beyond its current value - a literal, the result of an arithmetic expression, the return of a function that returns by value. The defining property is that it has no address: trying to take `&(x + 5)` is a compile error. The name comes from "right-hand value," because rvalues are the things you can read from but not write to.

A few concrete examples make this clear:

```
int x = 5;
f(x);           // x is an lvalue - it has an address (&x is valid)
f(5);           // 5 is an rvalue - &5 is a compile error

std::string f() { return "CS247"; }
std::string s = f();    // f() is an rvalue - the returned string is
                        //   a temporary that disappears at the end of the line
```

Some expressions you might expect to be lvalues are actually rvalues. The result of `x + y` is an rvalue even if `x` and `y` are both lvalues, because the sum is a fresh value that exists only as a transient computation. By contrast, `++x` produces an lvalue (the variable `x` has been incremented; you can take its address afterward), while `x++` produces an rvalue (the pre-increment value is conceptually a snapshot that no longer corresponds to any storage).

References to lvalues and rvalues

C++ has two kinds of reference, one for each value category.

lvalue reference (T&). What we have been calling a "reference" all along - an alias for an existing lvalue, declared with one ampersand. An lvalue reference must be initialized to an lvalue at construction time; trying to bind one to an rvalue is a compile error:

```
int x = 5;
int& y = x;     // OK - y is an lvalue reference to x
int& z = 5;     // ERROR - cannot bind non-const lvalue reference to rvalue 5
```

Exception: a const lvalue reference can bind to an rvalue. The compiler creates a temporary with the rvalue's value, extends its lifetime to match the reference, and binds the reference to it. This is the rule that lets us pass literals and temporaries to functions that take by const reference:

```
void f(int& x)        { /* ... */ }    // takes non-const ref
void g(const int& x)  { /* ... */ }    // takes const ref

f(5);    // ERROR - 5 is an rvalue, cannot bind to int&
g(5);    // OK - compiler creates a temporary holding 5,
         //   extends its lifetime, binds the const reference to it
```

You can extend the lifetime of an rvalue even further by binding it to a named const reference. The temporary lives for as long as the reference does:

```
std::string f() { return "Hello, World"; }

const std::string& s = f();    // s is a const reference to the temporary
                               // returned by f(); the temporary is kept
                               // alive for as long as s exists
```

rvalue reference (T&&). A reference that binds to an rvalue (and only to an rvalue). Declared with two ampersands. The defining property: a function that takes a T&& parameter knows the argument is a temporary that the caller no longer needs. That permission to "steal" the contents of the argument is what makes move semantics possible.

```
int&& a = 5;        // OK - rvalue reference can bind to rvalue 5
int x = 10;
int&& b = x;        // ERROR - cannot bind rvalue reference to lvalue x
int&& c = std::move(x);   // OK - std::move(x) casts x to an rvalue
```

std::move does not move anything by itself - it is just a cast that tells the compiler, “Treat this lvalue as an rvalue from here on." After the cast, the expression can bind to T&& references and trigger move constructors or move assignment operators. The "actual" moving occurs inside the constructor or assignment that is called with the rvalue.

> 💡 **Tip:** After std::move(x), the value of x is unspecified - most operations on it are still legal (you can assign a new value, you can destroy it), but reading its old value gives whatever the move operation chose to leave behind. Treat moved-from objects as "empty but destructible." If you need to use x again afterwards, re-initialize it first.

Overloading on Value Category

Functions can be overloaded on lvalue versus rvalue reference parameters, and the compiler picks the right one based on the value category of the argument:

```
void f(const std::string&  s) { std::cout << "1\n"; }   // lvalue ref
void f(std::string&&       s) { std::cout << "2\n"; }   // rvalue ref

std::string s{"CS247"};
f(s);                          // prints 1 - s is an lvalue
f(std::string{"CS247"});       // prints 2 - temporary is an rvalue
```

This overload pattern is exactly how `std::vector::push_back` differentiates between "copy the argument in" (lvalue version) and "move the argument in" (rvalue version): the same call site dispatches to whichever overload preserves greater efficiency given the argument's type. The user does not have to think about it at all - the right overload is chosen automatically.

The Move Constructor

A move constructor constructs a new object whose value is equal to an existing object but does not preserve the value of the existing object. The signature uses an rvalue reference:

```
MyClass(MyClass&& other) noexcept;
```

The key insight: the source object is an rvalue, so the caller has signalled that they no longer need it. We can therefore "steal" the source's heap-allocated resources rather than allocating fresh copies. Instead of allocating a new C and copying the value from `*other.ptr_` (as the copy constructor did), we just take other.ptr_'s pointer for ourselves and replace it with `nullptr` in the source. No allocation, no copy of pointed-to data; just a pointer swap.

> 💡 **Tip:** Move operations should be marked noexcept whenever possible. Some standard-library containers (notably std::vector) use moves to relocate elements during reallocation, but only if the move operations are guaranteed not to throw. If your moves can throw, the container falls back to copying, defeating the optimization. Marking moves noexcept is essentially free for well-written move operations - pointer swaps and primitive copies cannot throw.

std::move and std::exchange

Inside the move constructor, we need to move each member individually. Member objects, base classes, and managed pointers each have their own move constructors that we should invoke via `std::move`; simple data members can simply be exchanged.

```
MyClass(MyClass&& other) noexcept
    : Base(std::move(other)),
      comp_(std::move(other.comp_)),
      ptr_(std::move(other.ptr_)),
      simple_(std::exchange(other.simple_, 0))
{
    std::cout << "MyClass Move Constructor called\n";
}
```

`std::move` on each member invokes the member type's move constructor: `Base(std::move(other))` moves the base `subobject` (which the compiler then forwards to `Base`'s move constructor), `comp_(std::move(other.comp_))` moves the C, and `ptr_(std::move(other.ptr_))` moves the unique_ptr (which is a quick pointer swap, leaving the source unique_ptr as nullptr).

For the simple integer, we use `std::exchange`. The call `std::exchange(other.simple_, 0)` atomically sets `other.simple_` to 0 AND returns its previous value. This is exactly what a move of a primitive should do: transfer the value to the destination, leave the source in a defined empty state.

> 💡 **Tip:** std::exchange is a small addition to the standard library (C++14) that turns "move and reset" into a one-liner. The hand-written equivalent - separate steps to read the old value, store the new value, and return the old - is verbose, more error-prone, and not obviously atomic.

For comparison, here is the same move constructor without `std::exchange`:

```
// Before std::exchange:
MyClass(MyClass&& other) noexcept
    : ptr_(other.ptr_),
      simple_(other.simple_)
{
    other.ptr_    = nullptr;
    other.simple_ = 0;
}

// With std::exchange:
MyClass(MyClass&& other) noexcept
    : ptr_(std::exchange(other.ptr_, nullptr)),
      simple_(std::exchange(other.simple_, 0))
{}
```

Compiler-Generated Move Constructor

If you do not provide a default constructor or copy/move constructor or copy/move assignment, the compiler generates a move constructor based on memberwise move:

- Simple data members: bitwise copy (a primitive does not really "move" - copying an int leaves the source perfectly usable, but the conceptual semantics still apply).
- Pointer members: bitwise copy of the pointer (shallow - same issue as the copy ctor).
- Member objects: moved using each member type's move (or copy, if move is not available) constructor.
- Inherited members: moved using the base class's move constructor.

The conditions for the compiler to generate a move constructor are more restrictive than those for the other special members. The compiler-default table at the end of the chapter shows exactly when. Briefly: the compiler will NOT generate a move constructor if you have user-declared a destructor, a copy constructor, or a copy assignment operator. The reasoning is conservative: if you cared enough to write any of those, your class probably has invariants the compiler does not understand, and a memberwise move might break them.

The Move Assignment Operator

Move assignment is to the move constructor what copy assignment is to the copy constructor: the same idea, but the destination already exists. Signature:

```
MyClass& operator=(MyClass&& other) noexcept;
```

The body looks almost identical to the move constructor, with two additional considerations: the destination holds an old value that needs cleaning up, and we should guard against self-move-assignment for the same reason we guarded copy assignment:

```
MyClass& operator=(MyClass&& other) noexcept {
    if (this == &other)
        return *this;

    Base::operator=(std::move(other));
    comp_   = std::move(other.comp_);
    ptr_    = std::move(other.ptr_);
    simple_ = std::exchange(other.simple_, 0);

    std::cout << "MyClass Move Assignment Operator called\n";
    return *this;
}
```

The `unique_ptr` again does the right thing automatically: when we move-assign `other.ptr_` into `ptr_`, the unique_ptr's move-assignment operator first deletes whatever ptr_ used to own, then takes the pointer from other.ptr_ and leaves `other.ptr_` as `nullptr`. The same is true for member objects with proper move-assignment operators - each one knows how to clean up its old state before installing the new.

> 💡 **Tip:** The moved-from object must satisfy just two conditions: it must be safe to destroy, and it must be safe to assign a new value to. Any other operation on a moved-from object is not guaranteed to behave sensibly - reading state, calling member functions, comparing - unless the class explicitly documents otherwise. If you want stronger guarantees, document them and write the move functions to uphold them.

Copy and Move Elision

The compiler is allowed (and in some cases required) to skip copy and move operations even when the source language requires them. This optimization is called copy elision (or copy/move elision, since it applies to both).

```
Vec makeAVec() { return {0, 0}; }    // returns a fresh Vec

Vec v = makeAVec();    // What runs? Copy ctor? Move ctor? Basic ctor?
```

Before C++17, the compiler was permitted to skip the move (with the move constructor still required to exist as a fallback). Since C++17, elision is mandatory in this exact pattern: the compiler builds the returned object directly into v's storage, never constructing a separate temporary in the function and then transferring it out. Neither the copy constructor nor the move constructor runs - only the basic constructor that builds {0, 0} fires, and it writes directly into v's memory.

> 💡 **Tip:** You are not expected to memorize exactly when elision must happen, only that it does happen. Specifically, when a function returns a "pure rvalue" expression (a temporary built directly with a constructor call or a brace-init list, not a named local variable), C++17 mandates that no copy or move occurs. When you compile with g++ and only see the basic constructor message but no copy or move messages, you have observed elision in action.

Returning a named local variable from a function (return value optimization, RVO) is a related but slightly weaker form: the compiler usually elides the copy or move here too, but it is allowed not to. Either way, you cannot rely on the copy or move constructor being called for return values - and you should write your special members so that they remain correct whether they are called or elided.

Rule of Zero, Three, and Five

The interactions among the six special members are governed by a small set of rules that together determine which functions the compiler generates. Modern C++ summarizes these in three idioms: the Rule of Zero, the Rule of Three (pre-C++11), and the Rule of Five (C++11 and later). Internalizing them is the difference between writing classes that "just work" and writing classes that compile but corrupt their data on assignment.

The Compiler-Default Table

The table below summarizes what the compiler generates for each special member, based on what you have declared. Read across each row: "If I declared this, what does the compiler do for everything else?"

User Declares	Default Ctor	Destructor	Copy Ctor	Copy Assign	Move Ctor	Move Assign
Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
Any constructor	not decl.	defaulted	defaulted	defaulted	defaulted	defaulted
Default ctor	user	defaulted	defaulted	defaulted	defaulted	defaulted
Destructor	defaulted	user	defaulted	defaulted	not decl.	not decl.
Copy ctor	not decl.	defaulted	user	defaulted	not decl.	not decl.
Copy assign	defaulted	defaulted	defaulted	user	not decl.	not decl.
Move ctor	not decl.	defaulted	deleted	deleted	user	not decl.
Move assign	defaulted	defaulted	deleted	deleted	not decl.	user

Reading the cells: "defaulted" means the compiler synthesizes the member according to the memberwise rules; "user" means you declared it yourself; "not decl." means the compiler does NOT generate the member, and attempting to use it produces an error indicating it is absent; "deleted" means the compiler explicitly marks it as = delete, so attempting to use it produces an error indicating the function is unavailable.

The Rules in Plain English

The table is dense, but the underlying rules are short:

1. Declare nothing → the compiler gives you all six special members, defaulted.
2. Declare any constructor (including a non-default one) → the compiler does NOT give you a default constructor.
3. Declare a destructor → move operations are NOT generated. Copy operations still are, but the language standard deprecates them.
4. Declare a copy constructor or copy assignment → move operations are NOT generated.
5. Declare a move constructor or move assignment → copy operations are DELETED (i.e., the class becomes non-copyable, on the assumption that if you cared enough to write moves, the defaults for copy are probably wrong).

The conservatism of rules 3, 4, and 5 reflects a hard lesson: if a class needs a custom destructor, copy constructor, or copy assignment, it almost certainly has invariants that the compiler-generated versions of the other special members would violate. Rather than silently generating dangerous defaults, the language disables the related members and forces you to confront them.

> 💡 **Rule of Three:** If you write a destructor, a copy constructor, or a copy assignment operator, you almost certainly need to write all three. The class manages a resource that the destructor releases, and the other two must

agree on how to share or duplicate it. This is a pre-C++11 idiom, but it remains a useful sanity check: any one of these three suggests the other two need careful thought.

💡 **Rule of Five:** In C++11 and later, the same logic extends to the move constructor and move assignment. If you need any one of {destructor, copy ctor, copy assign, move ctor, move assign}, you almost certainly need to think about all five. Either write all five explicitly, or use RAII wrappers (smart pointers, standard containers) that have correct versions baked in.

💡 **Rule of Zero:** The cleanest design avoids special members entirely. If every data member is itself a well-behaved type (a primitive, a standard container, a smart pointer), the compiler-generated versions of all six special members are correct, and you write none of them. This is the Rule of Zero: prefer to write zero special members, by delegating resource management to types that already manage their resources correctly.

= default and = delete

Two pieces of syntax let you ask the compiler precisely what you want.

= default after a special member declaration tells the compiler, "generate the default version of this function, even though my declaration would normally suppress it." This is useful when you need to declare a function for some reason (for example, because it must be virtual, or because you also declared one of its siblings and that suppressed it) but want the body the compiler would have written:

```
class Animal {
public:
    virtual ~Animal() = default;    // virtual but body is compiler-generated

    Animal() = default;             // explicitly request the default ctor
                                    // even after declaring other constructors
};
```

= delete does the opposite: it tells the compiler "this function is unavailable; refuse to call it." Trying to call a deleted function produces a clear error message at the call site, which is much friendlier than the "function is private" hack used before C++11:

```
class Singleton {
public:
    Singleton(const Singleton&)            = delete;  // non-copyable
    Singleton& operator=(const Singleton&) = delete;
    Singleton(Singleton&&)                 = delete;  // and non-movable
    Singleton& operator=(Singleton&&)      = delete;
};
```

Combining = default and = delete with the rules above gives precise control over which operations a class supports. A class that wants to be copyable but not movable can say so. A class that wants the default copy semantics, but a hand-written destructor can = default the copy operations to keep them. The pattern of declaring all five special members (some defaulted, some deleted, some user-written) makes it explicit to anyone reading the header what the class's value semantics are.

> 💡 **Best Practice:** Either declare all five special member functions explicitly (giving each one of: user-written, = default, or = delete) or declare none and rely on the Rule of Zero. The intermediate cases - declaring some but not others - are where the conservative compiler behaviour bites you. Containers like std::vector<MyClass> may silently fall back to copying when they could have moved, or fail to compile at all, depending on which special members you declared without thinking.

Equality

Operator== is conspicuously absent from the list of special member functions: the compiler does NOT generate a default equality operator. If you want `operator==` for your class, you must write it yourself.

> 💡 **Tip:** C++20 added the spaceship operator <=> and a way to ask for default comparison operators via = default, which generates lexicographic comparison from the members. For C++17 and earlier, however, equality is always hand-written.

Just like copying, equality has two reasonable interpretations:

Deep equality. Two objects are equal if their underlying values are equal - including following pointer members to compare what they point to, rather than just comparing the pointer addresses. This is what value-based ADTs typically want.

Shallow equality. Two objects are equal if their bit patterns are equal - including comparing pointer addresses rather than pointees. This is rarely what you want for value semantics, but it is fast and correct for entity types (where it amounts to "are these the same identity?").

For our `MyClass`, the natural choice is deep equality, since we have been treating it as a value-style class with copy and move semantics. A correct implementation:

```
[[nodiscard]] friend bool operator==(const MyClass& lhs,
                                     const MyClass& rhs) {
    auto ptr_equal = (!lhs.ptr_ && !rhs.ptr_) ||
                     (lhs.ptr_  && rhs.ptr_  && *lhs.ptr_ == *rhs.ptr_);

    std::cout << "MyClass equality Operator called\n";
    return lhs.comp_  == rhs.comp_
        && lhs.simple_ == rhs.simple_
        && ptr_equal;
}

[[nodiscard]] friend bool operator!=(const MyClass& lhs,
                                     const MyClass& rhs) {
    return !(lhs == rhs);
}
```

The pointer comparison handles three cases: both null (equal), both non-null with equal pointees (equal), or otherwise (not equal). The same approach works for raw pointers and smart pointers. make_unique's internal state never affects equality; only what the `unique_ptr` owns matters.

> 💡 **Tip:** The [[nodiscard]] attribute on operator== generates a compiler warning if the caller discards the result. Code like a == b; on its own line - which computes a comparison and then throws away the answer - is almost certainly a bug (often a typo where someone meant a = b;), and the warning catches it.

Notice that `operator!=` is implemented in terms of `operator==`, returning `!(lhs == rhs)`. Defining it this way guarantees that == and != stay consistent: any change to == automatically propagates to !=. The alternative - copying the comparison logic in both - is a common source of bugs in real code.

Summary

The six special member functions are the heart of C++ value semantics. The compiler will generate most of them for you most of the time, but knowing exactly when and what they do is the difference between writing classes that compose cleanly and writing classes that corrupt their data the first time they are passed by value.

- Constructors put each data member into a known state. The default constructor is only generated when you declare no constructor at all.
- The destructor cleans up resources. The compiler-generated destructor performs memberwise cleanup but does NOT delete what raw pointers point to, which is why smart pointers are the modern default for owned heap data.
- The copy constructor and copy assignment build/overwrite from an existing object. The compiler-generated versions do memberwise copy, which is a shallow copy for raw pointers. Hand-write them when your class owns heap resources, and follow the standard format (self-assignment check, base assignment, member assignment, deep pointer copy, return *this).
- The copy-and-swap idiom expresses operator= via the copy constructor and a swap, at the cost of unconditional copying.
- lvalues have addresses; rvalues do not. lvalue references bind to lvalues; rvalue references bind to rvalues; const lvalue references bind to either. std::move casts an lvalue to an rvalue without doing anything else.
- The move constructor and move assignment take an rvalue source and "steal" its resources, leaving it empty but destructible. std::move and std::exchange are the standard tools. Move operations should be marked noexcept.
- Copy and move elision let the compiler skip these operations when it can. In C++17, certain elisions are mandatory.
- The Rule of Zero, Three, and Five captures the inter-dependencies: special members are rarely independent, and declaring one usually obliges you to think about the others. Either declare all five explicitly (with = default and = delete where appropriate) or declare none and use RAII types.
- Equality is NOT compiler-generated. Choose deep or shallow equality based on whether your class is value-style or entity-style, define operator==, and implement operator!= as !(==).

Practice Problems

1.

Complete the running example.

Complete the definitions of class Base, class C, and class `MyClass` so that the following `main()` produces the expected output below. Every special member of every class should print a line identifying itself.

```
int main() {
    std::cout << "step1\n";
    MyClass obj1;
    std::cout << "step2\n";
    MyClass obj2 = obj1;
    std::cout << "step3\n";
    MyClass obj3 = std::move(obj1);
    std::cout << "step4\n";
    obj2 = obj3;
    std::cout << "step5\n";
    obj3 = std::move(obj2);
    std::cout << "step6\n";
    return 0;
}
```

Expected output begins:

step1
Base Constructor called
C Constructor called
MyClass Default Constructor called
step2
Base Copy Constructor called
C Copy Constructor called
MyClass Copy Constructor called
step3
Base Move Constructor called
...
(continuing through step 6 and the chain of destructors at the end of main)

For each step, predict the exact order of the constructor/destructor/operator= messages before running your program. Use the chapter's standard format for each special member.

2.

Diagnose the leak.

The following class compiles but leaks memory under several circumstances. Identify every leak and fix the class. Use raw pointers (not smart pointers) - the exercise is about understanding why smart pointers are the modern default.

```
class Buffer {
    char*  data_;
    size_t size_;
public:
    Buffer(size_t n) : data_{new char[n]}, size_{n} {}
    ~Buffer() { delete[] data_; }
    // Copy ctor, assignment, move ctor, move assignment are all compiler-
generated.
};

void f(Buffer b);              // takes by value
Buffer g();                    // returns by value

int main() {
    Buffer a{1024};
    Buffer b{2048};
    f(a);                      // (i)
    a = b;                     // (ii)
    Buffer c = g();            // (iii)
    return 0;                  // (iv)
}
```

(a) For each of (i)-(iv), say what the compiler-generated special members will do at that line, and identify whether memory is leaked, double-freed, or correctly managed.

(b) Write a complete Rule-of-Five version of Buffer: hand-written destructor, copy ctor, copy assignment, move ctor, move assignment. Use the standard format from this chapter.

(c) Rewrite Buffer using the Rule of Zero - replace the raw pointer with a member of an appropriate standard-library type, so that all five special members can be left compiler-generated. Show that your version still compiles with all of `main()` above.

3.

Move semantics and containers.

You are given a class Image that owns a heap-allocated pixel buffer. The class is non-trivial - copying it allocates a fresh buffer and copies every byte. Suppose Image has a correct copy constructor and copy assignment, but no move operations.

(a) When the user writes `std::vector<Image> v; v.push_back(myImage);` will the `push_back` copy or move `myImage`? Explain why.

(b) When `std::vector<Image>` later runs out of capacity and reallocates its internal storage, what happens to existing elements - are they copied or moved into the new buffer? Why?

(c) Now suppose we add move operations to Image, but forget to mark them `noexcept`. Does the behaviour in (b) change? Explain.

(d) Write the `noexcept` move constructor and move assignment for Image. Show that with both move operations marked `noexcept`, the vector reallocation behaviour in (b) changes to moves.

(e) Suppose someone refactors Image so that the destructor logs every destruction with `std::cout`. Does that destructor automatically become `noexcept`? What happens to the move operations? Hint: think about what `noexcept` means and what `cout` can do.

4.

Self-assignment and exception safety.

The following copy assignment operator looks reasonable, but has two bugs that surface in specific edge cases.

```
class Stash {
    int* data_;
    size_t size_;
public:
    Stash& operator=(const Stash& other) {
        delete[] data_;
        size_ = other.size_;
        data_ = new int[size_];
        for (size_t i = 0; i < size_; ++i)
            data_[i] = other.data_[i];
        return *this;
    }
    // ... rest of the class omitted ...
};
```

(a) Explain what goes wrong when a client writes `s = s;` (self-assignment). Show the exact sequence of operations that produces a problem.

(b) Suppose the `new int[size_]` line throws `std::bad_alloc` (out of memory). What state is `*this` in afterwards? What is the consequence the next time the destructor runs?

(c) Fix both bugs by rewriting `operator=` using the copy-and-swap idiom, with a friend swap function. Explain how copy-and-swap automatically handles both problems.

(d) The copy-and-swap version is slower for the common case where allocation succeeds. Write a third version that handles both bugs without copy-and-swap (using a self-assignment check and careful ordering of operations) - and explain the trade-off.

5.

Lifetime-extension puzzles.

For each of the following code snippets, say whether the code is well-defined or undefined behaviour. If well-defined, predict what the program prints. If undefined, explain why.

(a)
```
const std::string& s = std::string{"hello"};
    std::cout << s << '\n';
```

(b)
```
std::string make() { return "made"; }
    const std::string& s = make();
    std::cout << s << '\n';
```

(c)
```
const std::string& f(const std::string& s) { return s; }
    const std::string& s = f(std::string{"temp"});
    std::cout << s << '\n';
```

(d)
```
struct Wrap { const std::string& r_; Wrap(const std::string& s) : r_{s} {} };
    Wrap w{std::string{"trap"}};
    std::cout << w.r_ << '\n';
```

(e)
```
std::string&& s = std::string{"goodbye"};
    std::cout << s << '\n';
    s = "goodbye again";   // is this legal?
    std::cout << s << '\n';
```

Reflect on what these tell you about when temporaries' lifetimes are extended (and when they are not). This is a real-world source of dangling references.