Module 9 of 20

Copies, moves and the rule of five

Copy construction versus assignment and the memberwise defaults, destructors and the rule of three, copy-and-swap, rvalue references and std::move, the moved-from state and noexcept, the generation table behind the rule of five and the rule of zero, and value categories with guaranteed elision.

C++ objects are values: `Probe b = a;` makes a second, independent object, and when either one dies it cleans up after itself at a moment the language defines exactly. That is what lets a class own a heap block, a file or a lock — and it means every class must answer a question a garbage-collected language never asks: what does it mean to copy one of these, to move one, to destroy one? The compiler answers memberwise, and its answer is right for a class built from `std::string`, `std::vector` and `std::unique_ptr` and catastrophically wrong for a class that owns a raw pointer. This module is where you learn to tell the two apart and to write the five special member functions when — and only when — a class needs them.

The lessons build the picture one function at a time. Copy semantics separates copy construction from copy assignment, shows the memberwise default and the shallow copy it produces for an owning pointer, and settles self-assignment and `= delete`. Destructors and the rule of three fixes the order things die, writes an owning buffer with deep copy and a self-safe assignment, and rebuilds that assignment with copy-and-swap. Move semantics explains rvalue references, why `std::move` moves nothing, the moved-from state that is valid but unspecified, and why `std::vector` copies your elements unless the move is `noexcept`. The rule of five and the rule of zero lays out the generation table — declaring a destructor switches the moves off — and the design rule that follows: own resources through members and declare nothing. Value categories and elision names lvalue, prvalue and xvalue in plain terms, states the C++17 guarantee that a prvalue is built in place, and settles `return local;` against `return std::move(local);`.

The exercises are built around one tool: classes that print their own `copy`, `move` and `destroy` events, so that every rule in the text becomes a line you can predict and then watch. Only operations whose counts the standard fixes are ever judged — guaranteed elision, explicit `std::move`, `std::swap`, and `std::vector` after a `reserve()` — never a named return value optimisation or a reallocation. You will trace identity against value through a copy-assignment loop,

watch a struct copy its members in order, write an owning buffer with the rule of three and again with copy-and-swap, extend it to the rule of five with a moved-from state of `size=0`, observe what a moved-from `std::vector` guarantees, build a rule-of-zero grid on `std::unique_ptr`, test the generation table against four classes, trace guaranteed elision and the implicit move of a returned parameter, and sink values into a container three ways. The checkpoint adds a command-driven stack with all five members, a longer trace to predict, and two shelves passing books through `std::unique_ptr` with no special members at all.

CONTENTS

1	Copy semantics — what a copy is, and when the default one is wrong	14 min
2	Destructors and the rule of three	14 min
3	Move semantics — rvalue references, std::move and noexcept	15 min
4	The rule of five and the rule of zero	14 min
5	Value categories, temporaries and copy elision	15 min
✓	Checkpoint — Copies, moves and the rule of five	28 min

Exercises in this module compile on Clang 18 · C++20. Reference solutions and the answer key follow each lesson.

LESSON 1 · 14 MIN

Copy semantics — what a copy is, and when the default one is wrong

C++ has value semantics: `Probe b = a;` makes a second, independent object, not a second name for the first as it would in Java. That is what lets a class own memory, a file or a lock and clean it up when it dies — and it means every class must answer a question Java classes never face: *what does it mean to copy one of these?* The compiler answers memberwise, and its answer is right for most classes and catastrophically wrong for the rest. This lesson settles which copy operations exist, what the generated ones do, when you must write your own, and how to say "this type cannot be copied at all".

Two operations, not one

Copying happens in two different situations, and C++ gives each its own special member function:

```
Probe a(1);
Probe b = a;      // copy construction: b did not exist; it is built from a
Probe c{a};       // copy construction, brace form
b = c;            // copy assignment: b already exists; its value is replaced
```

The **copy constructor** is `Probe(const Probe& other)`. It takes the source by `const` reference — by value would need a copy to make the copy, and the compiler rejects that recursion. The **copy assignment operator** is `Probe& operator=(const Probe& other)`; it returns `*this` so that `a = b = c` chains. Construction starts from raw storage; assignment starts from an object that already holds a value, which it has to give up cleanly — which is why every hand-written assignment is harder than the matching constructor.

What the compiler generates

Declare neither and the compiler generates both, **memberwise**: the copy constructor copy-constructs each member from the source's member, in declaration order; the copy assignment copy-assigns each member, in declaration order. Base classes come first (Module 10). For a class made of well-behaved members, that is exactly right:

```
struct Employee {
    std::string name;
    std::vector<int> ratings;
    int grade{};
};

Employee e1{"Ada", {5, 4, 5}, 3};
Employee e2 = e1;       // copies name (its own characters), ratings (its own
e2.ratings.push_back(1);
std::cout << e1.ratings.size() << '\n';   // 3 — e1 is untouched
```

`std::string` and `std::vector` know how to copy themselves — each allocates its own storage and copies the contents — so a struct built from them gets a deep copy for free. This is the first half of the *rule of zero* (lesson 4): when every member manages itself, the class needs no copy code at all. One property of the generated assignment is worth knowing now: it does not check for self-assignment. `e1 = e1` copy-assigns each member to itself — harmless for standard types, merely wasteful.

Deep versus shallow

The memberwise rule copies *members*, and a pointer member is a pointer. Copying it produces a second pointer to the same object — a **shallow** copy. Whether that is a bug depends on what the pointer means. A non-owning pointer (a cursor into a vector someone else owns) is a name for something elsewhere, and two names for the same thing is exactly what a copy should give. The trouble starts when the pointer *owns* what it points to:

```
struct Buffer {
    int* data;
    std::size_t size;
    explicit Buffer(std::size_t n) : data(new int[n]()), size(n) {}
    ~Buffer() { delete[] data; }
};

void demo() {
    Buffer a(4);
    Buffer b = a;      // generated: b.data == a.data
}                      // b's destructor deletes the block; a's destructor dele
```

Two objects each believe they own one block. The second `delete[]` is undefined behaviour — a crash if you are lucky, silent heap corruption if you are not — and before that, every write

through `b.data` shows up in `a`. The generated copy is wrong because the class holds a resource whose meaning the compiler cannot see. An owning raw pointer is the signal: a class that has one needs hand-written copy operations, or (usually better) a member such as `std::vector` or `std::unique_ptr` that already knows how to copy itself.

Writing a deep copy

The copy constructor allocates its own block and copies the elements:

```
Buffer(const Buffer& other)
    : data(new int[other.size]), size(other.size) {
    std::copy_n(other.data, size, data);
}
```

The copy assignment has three extra things to get right. It must release the storage it already holds; it must not release it *before* the copy succeeds — `new` can throw, and an object whose pointer has been deleted but not replaced is a time bomb; and it must survive being assigned to itself:

```
Buffer& operator=(const Buffer& other) {
    if (this == &other) return *this;          // self-assignment guard
    int* fresh = new int[other.size];          // acquire first
    std::copy_n(other.data, other.size, fresh);
    delete[] data;                             // release old only after succe
    data = fresh;
    size = other.size;
    return *this;
}
```

Acquire, copy, then release: the old state stays intact until the new state exists, which is the *strong exception guarantee* (Module 15) falling out of plain ordering.

Self-assignment is not hypothetical

`a = a` looks like a line nobody writes. It arrives through aliases: `v[i] = v[j]` when `i == j`, `*p = *q` when both point at the same object. An assignment that does `delete[] data` and then reads `other.data` — the same pointer — copies from freed memory. The guard `if`

`(this == &other)` compares addresses, the only reliable test of identity. The copy-and-swap idiom (lesson 2) makes the guard unnecessary by never touching `other` after the copy.

Making a type non-copyable

Some objects have no sensible copy: a lock that is held, an open socket, a `std::thread`, a `std::unique_ptr`. The type says so by deleting the operations:

```
class Connection {
public:
    Connection(const Connection&) = delete;
    Connection& operator=(const Connection&) = delete;
};
Connection c2 = c1;   // error: use of deleted function 'Connection::Connecti
```

`= delete` declares the function and forbids calling it, so the mistake is a clear compile-time message rather than a silent shallow copy. Deleting the copy constructor also leaves the class unmovable unless you say otherwise — declaring any copy operation suppresses the implicit move operations, a rule lesson 4 lays out in full. Non-copyable types can still be passed by reference, returned from factories, and held in containers once they are movable.

Seeing copies happen

The rest of this module relies on one tool: a class that reports its own special member functions. Give each object an id from a counter and print what happens to it:

```
struct Probe {
    static inline int next = 1;
    int id;
    int value;
    explicit Probe(int v) : id(next++), value(v) { std::cout << "construct #"
    Probe(const Probe& o) : id(next++), value(o.value) { std::cout << "copy #'
    Probe& operator=(const Probe& o) {
        if (this == &o) { std::cout << "self-assign #" << id << '\n'; return *
        value = o.value;
        std::cout << "copy-assign #" << id << " from #" << o.id << '\n';
        return *this;
    }
    ~Probe() { std::cout << "destroy #" << id << '\n'; }
};
```

The id is the object's *identity*, the value is its *value*, and the two events differ exactly there: copy construction creates a new identity holding an old value; assignment keeps an identity and replaces its value. When the trace surprises you, the surprise is the lesson.

Pitfalls

Mistake	What happens
Generated copy of a class with an owning raw pointer	Two owners, one block: double delete, and writes through one visible in the other
`Probe(Probe other)` as the copy constructor	Rejected: a by-value parameter would need the copy constructor to exist already
`void operator=(const Probe&)`	Compiles, but `a = b = c` no longer does; return `Probe&`
Assignment that deletes before copying	Self-assignment copies from freed memory; a throwing `new` leaves a dangling pointer
Trusting the generated assignment to skip self-assignment	It assigns every member to itself — fine for `std::string`, fatal for a raw owner

Key takeaways

Copy construction builds a new object from an old one; copy assignment replaces the value of an existing one. They are separate functions with separate rules.

The generated versions are memberwise, in declaration order, and correct whenever every member copies itself correctly.

A member that owns through a raw pointer makes the memberwise copy shallow and wrong — write the copy yourself, or own through `std::vector` / `std::unique_ptr` instead.

Hand-written assignment must guard against self-assignment and acquire before it releases.

`= delete` on the copy pair makes a type non-copyable with a clear error; it also suppresses implicit moves.

An instrumented class with ids shows identity and value separately — read its trace before you trust your intuition.

EXERCISE 1

Identity and value

Complete the instrumented `Probe` class so that every copy operation reports itself, then drive it through a loop of assignments. Each object gets an id from a static counter when it is constructed (its *identity*) and carries an `int` (its *value*). The events are:

`construct #<id>=<value>` — already written;

`copy #<id> from #<source id>` — the copy constructor: a new id, the source's value;

`copy-assign #<id> from #<source id>` — the copy assignment: the same id, the source's value; it must return `*this`;

`self-assign #<id>` — the copy assignment when the source is this very object; nothing else happens;

`destroy #<id>` — already written.

Read an integer `n` and then `n` integers. Construct `current` from the first. For each remaining value, in its own block, construct a `Probe` from it and copy-assign it to `current`, so the temporary dies at the end of the block. After the loop, assign `current` to itself through a reference (`Probe& same = current; current = same;`), copy-construct `snapshot` from `current`, and print `current=<value> snapshot=<value>`. The destructors then run in reverse order.

Input: `n` (at least 1), then `n` integers. **Output:** the event trace, the values line, and the two final destroy lines.

```
3
5 8 13
```

prints

```
construct #1=5
construct #2=8
copy-assign #1 from #2
destroy #2
construct #3=13
copy-assign #1 from #3
destroy #3
self-assign #1
copy #4 from #1
current=13 snapshot=13
destroy #4
destroy #1
```

INPUT 1

```
3
5 8 13
```

OUTPUT 1

```
construct #1=5
construct #2=8
copy-assign #1 from #2
destroy #2
construct #3=13
copy-assign #1 from #3
destroy #3
self-assign #1
copy #4 from #1
current=13 snapshot=13
destroy #4
destroy #1
```

INPUT 2

```
1
42
```

OUTPUT 2

```
construct #1=42
self-assign #1
copy #2 from #1
current=42 snapshot=42
destroy #2
destroy #1
```

REFERENCE SOLUTION

```cpp
#include <iostream>

struct Probe {
    static inline int next = 1;
    int id;
    int value;

    explicit Probe(int v) : id(next++), value(v) {
        std::cout << "construct #" << id << '=' << value << '\n';
    }
    Probe(const Probe& other) : id(next++), value(other.value) {
        std::cout << "copy #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(const Probe& other) {
        if (this == &other) {
            std::cout << "self-assign #" << id << '\n';
            return *this;
        }
        value = other.value;
        std::cout << "copy-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    ~Probe() {
        std::cout << "destroy #" << id << '\n';
    }
};

int main() {
    int n;
    std::cin >> n;
    int first;
    std::cin >> first;
    Probe current(first);
    for (int i = 1; i < n; ++i) {
        int v;
        std::cin >> v;
        Probe next(v);
        current = next;
    }
    Probe& same = current;
    current = same;
    Probe snapshot = current;
    std::cout << "current=" << current.value << " snapshot=" << snapshot.value << '\
    return 0;
}
```

EXERCISE 2

Memberwise, in order

The `Probe` class is complete this time. Declare `struct Segment { Probe start; Probe end; };` — two members and **no** special member functions — and write `int length(const Segment&)`, which returns `end.value - start.value`. Then predict the trace of this driver before you run it:

Read four integers `s1 e1 s2 e2`.

`Segment a{Probe(s1), Probe(e1)};` then `Segment b{Probe(s2), Probe(e2)};` — each member is built in place from a prvalue, so only `construct` events appear.

`Segment c = a;` — the generated copy constructor copies the members in declaration order.

`c.end = b.end;` — assigns one member.

Assign `b` to itself through a reference (`Segment& same = b; b = same;`) — the generated assignment does not check for self-assignment; it assigns each member, and each `Probe` reports what it saw.

Print `a=<length> b=<length> c=<length>`.

The three segments die in reverse order, each destroying `end` before `start`.

Input: one line, `s1 e1 s2 e2`. **Output:** the trace, the lengths line, and six destroy lines.

```
0 10 3 7
```

prints

```
construct #1=0
construct #2=10
construct #3=3
construct #4=7
copy #5 from #1
copy #6 from #2
copy-assign #6 from #4
self-assign #3
self-assign #4
a=10 b=4 c=7
destroy #6
destroy #5
destroy #4
destroy #3
destroy #2
destroy #1
```

INPUT 1

```
0 10 3 7
```

OUTPUT 1

```
construct #1=0
construct #2=10
construct #3=3
construct #4=7
copy #5 from #1
copy #6 from #2
copy-assign #6 from #4
self-assign #3
self-assign #4
a=10 b=4 c=7
destroy #6
destroy #5
destroy #4
destroy #3
destroy #2
destroy #1
```

INPUT 2

```
5 5 -2 2
```

OUTPUT 2

```
construct #1=5
construct #2=5
construct #3=-2
construct #4=2
copy #5 from #1
copy #6 from #2
copy-assign #6 from #4
self-assign #3
self-assign #4
a=0 b=4 c=-3
destroy #6
destroy #5
destroy #4
destroy #3
destroy #2
destroy #1
```

REFERENCE SOLUTION

```cpp
#include <iostream>

struct Probe {
    static inline int next = 1;
    int id;
    int value;

    explicit Probe(int v) : id(next++), value(v) {
        std::cout << "construct #" << id << '=' << value << '\n';
    }
    Probe(const Probe& other) : id(next++), value(other.value) {
        std::cout << "copy #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(const Probe& other) {
        if (this == &other) {
            std::cout << "self-assign #" << id << '\n';
            return *this;
        }
        value = other.value;
        std::cout << "copy-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    ~Probe() {
        std::cout << "destroy #" << id << '\n';
    }
};

struct Segment {
    Probe start;
    Probe end;
};

int length(const Segment& s) {
    return s.end.value - s.start.value;
}

int main() {
    int s1, e1, s2, e2;
    std::cin >> s1 >> e1 >> s2 >> e2;
    Segment a{Probe(s1), Probe(e1)};
    Segment b{Probe(s2), Probe(e2)};
    Segment c = a;
    c.end = b.end;
    Segment& same = b;
    b = same;
    std::cout << "a=" << length(a) << " b=" << length(b) << " c=" << length(c) << '\
    return 0;
}
```

QUIZ

1.

```
Probe a(1);
Probe b = a;
b = a;
```

How many copy constructions and copy assignments does this perform?

A. One copy construction and one copy assignment

B. Two copy constructions and no assignment

C. No copy construction and two assignments

D. One copy construction; the third line is a no-op because `b` already equals `a`

2. Which declaration is a copy constructor?

A. `Probe(Probe other);`

B. `Probe(const Probe& other);`

C. `Probe& operator=(const Probe& other);`

D. `Probe(Probe* other);`

3.

```
struct Employee {
    std::string name;
    std::vector<int> ratings;
};
Employee e2 = e1;
e2.ratings.push_back(1);
```

What happens to `e1`?

A. Nothing — the generated copy copies each member, and `std::vector` copies its own elements

B. `e1.ratings` also gains the element: the vector's buffer is shared

C. Compile error: a struct with a `std::vector` member has no copy constructor

D. Undefined behaviour when both objects are destroyed

4. A class holds `int* data` allocated with `new[]` and deletes it in its destructor, but declares no copy operations. What does `Buffer b = a;` lead to?

A. A deep copy, because the destructor tells the compiler the class owns the block

B. A compile error: the class is non-copyable

C. Both objects point at one block; the second destructor deletes it again — undefined behaviour

D. A shallow copy that is safe as long as `b` is destroyed first

5. What is the purpose of `if (this == &other) return *this;` at the top of a copy assignment operator?

A. To make the assignment faster for all callers

B. To stop the operator from freeing its own resource before copying from it when the source is the same object

C. To prevent assigning from a `const` object

D. It is required by the standard for every user-defined `operator=`

6. A class declares `T(const T&) = delete;` and nothing else. What does `T b = std::move(a);` do?

A. Moves `a` into `b`; deleting the copy has no effect on moves

B. Fails to compile: declaring a copy constructor suppresses the implicit move, so the deleted copy constructor is selected

C. Performs a shallow copy

D. Compiles but throws at run time

ANSWER KEY · COPY SEMANTICS — WHAT A COPY IS, AND WHEN THE DEFAULT ONE IS WRONG

1. A `Probe b = a;` is initialisation — `b` does not exist yet, so the copy constructor runs. `b = a;` replaces the value of an existing object, which is the copy assignment operator. Equality of values never suppresses an assignment.

2. B The copy constructor takes the source by reference to `const`. A by-value parameter would need a copy to be made before the copy constructor could run, which the compiler rejects; `operator=` is the assignment; a pointer parameter is an ordinary constructor.

3. A The memberwise copy calls `std::string`'s and `std::vector`'s copy constructors, each of which allocates its own storage. Sharing happens only when a member is a raw pointer that is copied as a pointer.

4. C The generated copy constructor copies the pointer value. Two objects now own one block, writes through either are visible in both, and the second `delete[]` is undefined behaviour regardless of destruction order.

5. B An operator that releases its block and then reads `other.data` — the same pointer — copies from freed memory when the two are one object. Comparing addresses is the reliable identity test; the guard is not required and copy-and-swap avoids it altogether.

6. B A user-declared copy constructor — deleted or not — stops the compiler generating the move constructor. Overload resolution then finds only the deleted copy constructor and reports its use as an error.

LESSON 2 · 14 MIN

Destructors and the rule of three

A destructor is the other half of ownership. The constructor acquires; the destructor releases; and because C++ runs the destructor at a moment the language defines exactly — not "eventually", as a garbage collector would — a class can hold a heap block, a file or a lock with the certainty that it will be given back. This lesson pins down when destructors run and in what order, states the rule of three and why the three functions travel together, builds an owning buffer class that gets every case right, and finishes with the copy-and-swap idiom that makes assignment correct by construction.

The destructor

```
class Buffer {
public:
    explicit Buffer(std::size_t n) : data_(new int[n]()), size_(n) {}
    ~Buffer() { delete[] data_; }      // no parameters, no return type, one p
private:
    int* data_;
    std::size_t size_;
};
```

`~Buffer()` runs when the object's lifetime ends: a local at the closing brace of its block, a member when its enclosing object is destroyed, an element when the container erases it or dies, a heap object at `delete`, a temporary at the end of the full-expression that created it (lesson 5). You never call it by name in ordinary code. The compiler generates an empty one when you declare none, and it destroys the members either way — the body you write runs *before* the members go, so a destructor body may still use them.

Destructors are implicitly `noexcept`: an exception escaping one calls `std::terminate`, and there is no sensible alternative, because a destructor running during stack unwinding (Module 15) would otherwise have two exceptions in flight. Release resources, log if you must, but never throw from a destructor.

The order things die

Automatic objects are destroyed in **reverse order of construction**: the last declared is the first destroyed. Members are destroyed after the destructor body, in reverse declaration order. Bases go after the derived part (Module 10). With the `Probe` class from lesson 1:

```
{
    Probe a(1);            // construct #1
    Probe b(2);            // construct #2
    Probe c = a;           // copy #3 from #1
}                          // destroy #3, destroy #2, destroy #1
```

The rule is what makes RAII compose (Module 7, lesson 3): an object declared later may depend on one declared earlier — a lock guard on its mutex, a parser on its file — and the dependant is always gone before the thing it depends on. Container elements are the exception worth remembering: `std::vector` destroys its elements in an order the standard does not specify, so a program that must observe element destruction order should `pop_back()` them itself.

The rule of three

If a class needs a user-written **destructor**, **copy constructor** or **copy assignment operator**, it almost certainly needs all three. The reasoning is one step long: you wrote a destructor because the class owns something the members do not know how to release; the very same something is what the generated copy operations do not know how to duplicate. `Buffer` above compiles with the generated copies and double-deletes on the first copy, exactly as lesson 1 showed. The rule is a heuristic with the force of a law: a class that declares one of the three and not the others is a bug report waiting to be filed.

The converse is the more useful direction. A class with no destructor of its own — because every member is a `std::string`, a `std::vector`, a `std::unique_ptr` — needs no copy operations either. That is the rule of zero, lesson 4.

An owning buffer, done right

```cpp
#include <algorithm>
#include <cstddef>
#include <utility>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n) : data_(new int[n]()), size_(n) {}

    IntBuffer(const IntBuffer& other)
        : data_(new int[other.size_]), size_(other.size_) {
        std::copy_n(other.data_, size_, data_);
    }

    IntBuffer& operator=(const IntBuffer& other) {
        if (this == &other) return *this;
        int* fresh = new int[other.size_];
        std::copy_n(other.data_, other.size_, fresh);
        delete[] data_;
        data_ = fresh;
        size_ = other.size_;
        return *this;
    }

    ~IntBuffer() { delete[] data_; }

    std::size_t size() const { return size_; }
    int& operator[](std::size_t i) { return data_[i]; }
    const int& operator[](std::size_t i) const { return data_[i]; }

private:
    int* data_;
    std::size_t size_;
};
```

Read the four functions as one contract. The constructor establishes the invariant: `data_` points to `size_` ints this object owns (`new int[n]()` value-initialises them to zero). The copy constructor establishes the same invariant for a new object from an existing one. The assignment operator takes an existing object from one valid state to another without ever being in an invalid one: self-check, acquire, copy, release, commit. The destructor ends the invariant. `delete[]` matches `new[]`; mixing the array and scalar forms is undefined behaviour (Module 7, lesson 2). `new int[0]` is legal and returns a unique pointer you must still `delete[]`, so a zero-size buffer needs no special case. The two `operator[]` overloads are Module 8 material; the `const` one is what lets a `const IntBuffer&` parameter be read.

Copy-and-swap

The assignment operator above is correct, and it is also the function every reviewer reads twice. The **copy-and-swap idiom** gets the same result from two functions that are each trivially right:

```
class IntBuffer {
public:
    // constructor, copy constructor and destructor as before

    friend void swap(IntBuffer& a, IntBuffer& b) noexcept {
        std::swap(a.data_, b.data_);
        std::swap(a.size_, b.size_);
    }

    IntBuffer& operator=(IntBuffer other) {   // by value: the copy has alread
        swap(*this, other);
        return *this;
    }
    // ...
};
```

The parameter is taken **by value**, so by the time the body runs the copy constructor has already built `other` as a complete, independent copy of the source. The body swaps the two pointers and sizes: `*this` now owns the new block, `other` owns the old one, and when `other` goes out of scope at the end of the function its destructor releases the old block. Nothing is deleted before the copy exists, so a throwing `new` leaves `*this` untouched — the strong guarantee for free. Self-assignment needs no guard: `a = a` copies `a` into `other`, swaps, and destroys the copy, leaving `a` with its own contents in a different block. And the same operator serves as the *move* assignment once the class has a move constructor (lesson 3): pass an rvalue and `other` is move-constructed instead of copied.

The `swap` is a `friend` defined inside the class so that unqualified `swap(x, y)` finds it (Module 8, lesson 5) and `noexcept` because swapping two pointers and two integers cannot fail. The cost of the idiom is that every assignment allocates, even when the target's block is already the right size; for most classes that is invisible, and for the one class where it is not, the explicit version above is the alternative.

The destructor and `= default`

Writing `~Widget() = default;` says "the generated one, on purpose" — useful when a class is a polymorphic base and needs `virtual ~Widget() = default;` so that `delete` through a base pointer runs the derived destructor (Module 10, lesson 4). Even a defaulted destructor is *user-declared* for the generation rules in lesson 4: it switches the implicit move operations off. A class with no reason to declare a destructor should not, not even to log.

Pitfalls

Mistake	What happens
Destructor without the copy pair	Every copy double-deletes; the rule of three exists for this
`delete` for a `new[]` block	Undefined behaviour; match the forms
Throwing from a destructor	`std::terminate`, immediately if the stack is already unwinding
Assignment that releases before it copies	Self-assignment reads freed memory; a throw leaves a dangling pointer
Declaring a destructor "just to log"	Silently disables the implicit move operations (lesson 4)

Key takeaways

- A destructor runs at a defined moment — scope exit, `delete`, container erase, end of a temporary's full-expression — and must not throw.
- Locals die in reverse order of construction; members die after the destructor body, in reverse declaration order.
- Rule of three: a user-written destructor, copy constructor or copy assignment means the class owns something, and all three must know about it.
- The hand-written assignment is: self-check, acquire, copy, release, commit — never release first.
- Copy-and-swap builds assignment from the copy constructor, a `noexcept` friend `swap` and the destructor; it is self-assignment safe and strongly exception safe with no guard at all.
- Do not declare a destructor you do not need; it changes what the compiler generates.

EXERCISE 1

An owning buffer, rule of three

`IntBuffer` owns `size_` ints in a block allocated with `new int[n]()`. The constructor, `operator[]`, `size()`, `print()` and `sum()` are written. Give it the rule of three:

- the **copy constructor** allocates its own block, copies the elements, and prints `deep-copy <size>`;
- the **copy assignment** prints `self-assign` and returns when the source is this object; otherwise it allocates a fresh block, copies into it, releases the old block, commits, prints `copy-assign <size>` and returns `*this`;
- the **destructor** prints `release <size>` and then deletes the block (the constructor already prints `allocate <size>`).

The driver is written: it reads `n` and `n` integers into `a`, copies `a` into `b`, applies `m` edits of the form `i v` (set `b[i] = v`) to the copy only, prints both buffers, assigns `b` to `a`, assigns `a` to itself through a reference, and prints `a` with its sum. If `a` changes when `b` is edited, the copy is shallow.

Input: `n`, then `n` integers, then `m`, then `m` lines of `i v`. **Output:** the trace and the three buffer lines, as below.

```
3
1 2 3
1
0 9
```

prints

```
allocate 3
deep-copy 3
a=[1 2 3]
b=[9 2 3]
copy-assign 3
self-assign
a=[9 2 3] sum=14
release 3
release 3
```

INPUT 1

```
3
1 2 3
1
0 9
```

OUTPUT 1

```
allocate 3
deep-copy 3
a=[1 2 3]
b=[9 2 3]
copy-assign 3
self-assign
a=[9 2 3] sum=14
release 3
release 3
```

INPUT 2

```
4
5 -5 10 0
2
1 5
3 1
```

OUTPUT 2

```
allocate 4
deep-copy 4
a=[5 -5 10 0]
b=[5 5 10 1]
copy-assign 4
self-assign
a=[5 5 10 1] sum=21
release 4
release 4
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cstddef>
#include <iostream>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n) : data_(new int[n]()), size_(n) {
        std::cout << "allocate " << size_ << '\n';
    }

    IntBuffer(const IntBuffer& other) : data_(new int[other.size_]), size_(other.siz
        std::copy_n(other.data_, size_, data_);
        std::cout << "deep-copy " << size_ << '\n';
    }

    IntBuffer& operator=(const IntBuffer& other) {
        if (this == &other) {
            std::cout << "self-assign\n";
            return *this;
        }
        int* fresh = new int[other.size_];
        std::copy_n(other.data_, other.size_, fresh);
        delete[] data_;
        data_ = fresh;
        size_ = other.size_;
        std::cout << "copy-assign " << size_ << '\n';
        return *this;
    }

    ~IntBuffer() {
        std::cout << "release " << size_ << '\n';
        delete[] data_;
    }

    std::size_t size() const { return size_; }
    int& operator[](std::size_t i) { return data_[i]; }
    const int& operator[](std::size_t i) const { return data_[i]; }

    void print(const char* name) const {
        std::cout << name << "=[";
        for (std::size_t i = 0; i < size_; ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << data_[i];
        }
        std::cout << ']';
    }

    long long sum() const {
        long long total = 0;
```

```
        for (std::size_t i = 0; i < size_; ++i) total += data_[i];
        return total;
    }

private:
    int* data_;
    std::size_t size_;
};

int main() {
    std::size_t n;
    std::cin >> n;
    IntBuffer a(n);
    for (std::size_t i = 0; i < n; ++i) std::cin >> a[i];

    IntBuffer b = a;
    std::size_t m;
    std::cin >> m;
    for (std::size_t k = 0; k < m; ++k) {
        std::size_t i;
        int v;
        std::cin >> i >> v;
        b[i] = v;
    }
    a.print("a");
    std::cout << '\n';
    b.print("b");
    std::cout << '\n';

    a = b;
    IntBuffer& same = a;
    a = same;
    a.print("a");
    std::cout << " sum=" << a.sum() << '\n';
    return 0;
}
```

EXERCISE 2

Copy-and-swap

Rebuild `IntBuffer`'s assignment with the copy-and-swap idiom. The constructor (`allocate <size>`), copy constructor (`deep-copy <size>`) and destructor (`release <size>`) are written. Add:

- `friend void swap(IntBuffer& x, IntBuffer& y) noexcept` — exchanges the two pointers and the two sizes with `std::swap`, then prints `swap`;

`IntBuffer& operator=(IntBuffer other)` — takes the source **by value**, swaps `*this` with the parameter and returns `*this`. Nothing else: no guard, no `delete`.

Because the parameter is a complete copy, the old contents of the target leave with the parameter and are released by its destructor. The driver reads `a` (`n` values) and `b` (`m` values), assigns `b` to `a`, sets `b[0]` to `v` when `m > 0`, assigns `a` to itself through a reference, swaps `a` and `b` with a plain `swap(a, b)` call, and prints both.

Input: `n`, `n` integers, `m`, `m` integers, then `v`. **Output:** the trace and the two buffer lines.

```
3
1 2 3
2
7 8
9
```

prints

```
allocate 3
allocate 2
deep-copy 2
swap
release 3
deep-copy 2
swap
release 2
swap
a=[9 8]
b=[7 8]
release 2
release 2
```

INPUT 1

```
3
1 2 3
2
7 8
9
```

OUTPUT 1

```
allocate 3
allocate 2
deep-copy 2
swap
release 3
deep-copy 2
swap
release 2
swap
a=[9 8]
b=[7 8]
release 2
release 2
```

INPUT 2

```
1
5
3
1 2 3
-1
```

OUTPUT 2

```
allocate 1
allocate 3
deep-copy 3
swap
release 1
deep-copy 3
swap
release 3
swap
a=[-1 2 3]
b=[1 2 3]
release 3
release 3
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <utility>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n) : data_(new int[n]()), size_(n) {
        std::cout << "allocate " << size_ << '\n';
    }

    IntBuffer(const IntBuffer& other) : data_(new int[other.size_]), size_(other.siz
        std::copy_n(other.data_, size_, data_);
        std::cout << "deep-copy " << size_ << '\n';
    }

    friend void swap(IntBuffer& x, IntBuffer& y) noexcept {
        std::swap(x.data_, y.data_);
        std::swap(x.size_, y.size_);
        std::cout << "swap\n";
    }

    IntBuffer& operator=(IntBuffer other) {
        swap(*this, other);
        return *this;
    }

    ~IntBuffer() {
        std::cout << "release " << size_ << '\n';
        delete[] data_;
    }

    std::size_t size() const { return size_; }
    int& operator[](std::size_t i) { return data_[i]; }
    const int& operator[](std::size_t i) const { return data_[i]; }

    void print(const char* name) const {
        std::cout << name << "=[";
        for (std::size_t i = 0; i < size_; ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << data_[i];
        }
        std::cout << "]\n";
    }

private:
    int* data_;
    std::size_t size_;
};
```

```
IntBuffer readBuffer() {
    std::size_t n;
    std::cin >> n;
    IntBuffer buf(n);
    for (std::size_t i = 0; i < n; ++i) std::cin >> buf[i];
    return buf;
}

int main() {
    IntBuffer a = readBuffer();
    IntBuffer b = readBuffer();
    int v;
    std::cin >> v;
    a = b;
    if (b.size() > 0) b[0] = v;
    IntBuffer& same = a;
    a = same;
    swap(a, b);
    a.print("a");
    b.print("b");
    return 0;
}
```

QUIZ

1.

```
{
    Probe a(1);
    Probe b(2);
    Probe c(3);
}
```

In what order are the three destroyed at the closing brace?

A. `a`, `b`, `c`

B. `c`, `b`, `a`

C. Unspecified — the compiler chooses

D. All at once; the order is unobservable

2. When are a class's data members destroyed relative to its destructor body?

A. Before the body, in declaration order

B. After the body, in reverse declaration order

C. After the body, in declaration order

D. Only when the destructor is user-defined

3. What does the rule of three say?

A. Every class must declare a constructor, a destructor and `operator=`

B. If a class needs a user-written destructor, copy constructor or copy assignment, it almost certainly needs all three

C. A class may have at most three constructors

D. Three special members are generated only when the class has no members

4.

```
IntBuffer& operator=(IntBuffer other) {
    swap(*this, other);
    return *this;
}
```

Why is this safe for `a = a` without an explicit guard?

A. The compiler removes self-assignments before calling `operator=`

B. `other` is a complete copy made before the body runs, so swapping with it and letting it die leaves `a` with its own contents in a different block

C. `swap` detects that both arguments are the same object

D. It is not safe; copy-and-swap still needs `if (this == &other)`

5. An exception propagates out of a destructor during normal scope exit. What happens?

A. The enclosing `try` block catches it like any other exception

B. `std::terminate` is called: destructors are implicitly `noexcept`

C. The exception is silently discarded

D. The object is leaked but the program continues

6.

```
struct D {
    const char* n;
    ~D() { std::cout << n; }
};
int main() {
    D a{"a"};
    { D b{"b"}; }
    D c{"c"};
}
```

What is printed?

A. `abc`

B. `cba`

C. `bca`

D. `bac`

ANSWER KEY · DESTRUCTORS AND THE RULE OF THREE

1. B Automatic objects are destroyed in reverse order of construction, so the last declared goes first. The order is fully specified and observable — RAII depends on it.

2. B The body runs first — it may still use the members — and then the members are destroyed last-declared first. This happens whether the destructor is generated or written by hand.

3. B Needing one of the three means the class manages a resource the generated versions do not understand; the same resource needs the other two to be written as well. The rule of five extends it with the two move operations.

4. B The by-value parameter is copy-constructed from the source before anything in the target is touched. After the swap the parameter holds the old block and destroys it on return; nothing is ever freed and then read.

5. B Destructors are `noexcept` by default, and an exception leaving a `noexcept` function terminates the program. The rule exists because a destructor running during stack unwinding would otherwise put two exceptions in flight.

6. C `b` dies at its inner block's closing brace, before `c` is even constructed. At the end of `main`, `c` and `a` are destroyed in reverse order of construction: `c` then `a`.

LESSON 3 · 15 MIN

Move semantics — rvalue references, std::move and noexcept

A copy duplicates a value; a move *transfers* it, leaving the source empty but valid. C++11 added moves because copying was the wrong default for the most common thing programs do with big objects — build one in a function and hand it back — and because a temporary that is about to die has nothing to lose by giving its contents away. This lesson explains what an rvalue reference binds to, why `std::move` moves nothing, how to write a move constructor and move assignment, what state a moved-from object is allowed to be in, and why `noexcept` on a move is the difference between `std::vector` moving your elements and copying them.

Why moves exist

```
std::vector<int> primesBelow(int n);          // builds a million-element vec

std::vector<int> primes = primesBelow(1'000'000);
```

Before C++11, initialising `primes` from the returned vector meant copying a million ints and freeing the original — a full allocation and a full loop, to end up with exactly what the function already had. The same waste happened whenever a temporary was passed to a function or inserted into a container. A **move** replaces that with three pointer assignments: the new vector takes the old one's buffer, and the old one is left pointing at nothing. The value is transferred, not duplicated, and the cost is constant whatever the size.

The language needed two things to make this safe: a way to tell that the source is expendable, and a way for a class to say what "transfer" means for it. The first is the rvalue reference; the second is the move constructor and move assignment.

Rvalue references

An **lvalue** is an expression that names an object — a variable, `*p`, `v[i]`. An **rvalue** is an expression whose object is a temporary or is otherwise about to be discarded — `Probe(3)`, `a + b`, a function call returning by value. Lesson 5 gives the full taxonomy; this is enough for now.

`T&` binds only to lvalues, `const T&` binds to anything, and the new reference type `T&&` — an **rvalue reference** — binds only to rvalues:

```
void take(const Probe& p);   // #1: anything
void take(Probe&& p);        // #2: rvalues only

Probe a(1);
take(a);            // #1: a is an lvalue
take(Probe(2));     // #2: the temporary is an rvalue — and #2 is the better r
```

Overload resolution prefers `T&&` for an rvalue, which is the whole mechanism: a function that receives a `Probe&&` knows the argument will not be used again by the caller, so it may take the contents. Inside `take`, though, the *name* `p` is an lvalue — it has a name — so passing it on requires `std::move(p)` again. A named rvalue reference is an lvalue.

`std::move` is a cast

```
Probe a(1);
Probe b = std::move(a);      // move construction: a's contents are now in b
```

`std::move` moves nothing. It is `static_cast<T&&>(x)`: it takes an lvalue and returns an rvalue reference to the same object, which makes overload resolution pick the move constructor instead of the copy constructor. The move happens inside that constructor, or not at all — for a type with no move constructor, `std::move` selects the copy constructor and silently copies (lesson 4). After the line, `a` still exists and will still be destroyed at the end of its scope; what it contains is up to the class. The name `std::move` is the standard's most-criticised piece of naming; read it as "may be moved from".

The move constructor and move assignment

For `IntBuffer` from lesson 2, a move steals the pointer and leaves the source owning nothing:

```cpp
IntBuffer(IntBuffer&& other) noexcept
    : data_(other.data_), size_(other.size_) {
    other.data_ = nullptr;      // the source must not delete what we now own
    other.size_ = 0;
}

IntBuffer& operator=(IntBuffer&& other) noexcept {
    if (this == &other) return *this;
    delete[] data_;             // release what we hold
    data_ = other.data_;        // take theirs
    size_ = other.size_;
    other.data_ = nullptr;
    other.size_ = 0;
    return *this;
}
```

Three points. The parameter is `IntBuffer&&`, not `const IntBuffer&&` — moving *modifies* the source, so it cannot be `const`. The source is left in a state its destructor accepts: `delete[] nullptr` is a no-op, and `size_ == 0` keeps `size()` honest. And both are `noexcept`: pointer assignments cannot throw, and saying so matters more than any other `noexcept` in the class (below). `std::exchange(other.data_, nullptr)` from `<utility>` is the idiomatic one-liner for "take the value and leave this behind".

The self-check in the move assignment is cheap insurance: `a = std::move(a)` is rare but legal, and without the check the object would delete its block and then adopt the same, now dangling, pointer.

The moved-from state

After a move, the source is **valid but unspecified**: it is a real object on which every operation without preconditions must still work — destroying it, assigning to it, calling `size()` or `empty()` — but what those calls return is up to the class. For your own types, say what it is: "a moved-from `IntBuffer` has size 0". For the standard library the answer differs by type:

Type	After `T b = std::move(a);`
`std::vector`	`a` is guaranteed empty
`std::unique_ptr`	`a` is guaranteed null
`std::string`	valid but unspecified — usually empty, never rely on it
`int`, pointers	unchanged: built-in types copy

The safe pattern is to not read a moved-from object at all until it has been given a new value. `a = std::vector<int>{}` or `a.clear()` afterwards is fine; `std::cout << a.size()` after moving a `std::string` is a bug that happens to print `0` on most compilers. Note too that `std::move` the *cast* is different from `std::move` the *algorithm* in `<algorithm>`, which moves elements one by one from a range and leaves the source container the same size with moved-from elements inside it.

`noexcept` on moves, and why `std::vector` cares

When a `std::vector<T>` grows, it allocates a bigger block and transfers every element across. Moving them is cheap, but if a move constructor could throw halfway through, the vector would have some elements in the new block, some in the old, and no way to restore either — it would lose the *strong exception guarantee* that its `push_back` promises. So the vector asks: is `T`'s move constructor `noexcept`? If yes, it moves. If not, it **copies** every element, silently, on every reallocation. The check is `std::move_if_noexcept`, and the outcome for a class with a throwing (or merely unmarked) move constructor is that it is copied in the one place moves matter most.

Mark move operations `noexcept` whenever they only shuffle pointers and integers, which is nearly always. The standard containers' own moves are `noexcept`, so a rule-of-zero class built from them gets a `noexcept` move for free.

Pitfalls

Mistake	What happens
Using a moved-from `std::string`	Valid but unspecified: reading it is a bug that usually looks fine
Forgetting to null the source's pointer	Two owners again; the source's destructor frees the block you now hold
`const T&&` parameter	Cannot modify the source, so it cannot move; falls back to copying
Move operations without `noexcept`	`std::vector` copies your elements on every reallocation
Passing a named `T&&` on without `std::move`	It is an lvalue; the callee copies
`std::move` on something you use afterwards	The later use reads a transferred-from object

Key takeaways

A move transfers a value in constant time and leaves the source valid but empty; it exists so returning, passing and inserting big objects is cheap.

`T&&` binds only to rvalues; overload resolution picks it over `const T&` for temporaries and for anything wrapped in `std::move`.

`std::move` is a cast to `T&&`; it moves nothing itself and, for a type without a move constructor, copies.

A move constructor steals the members and nulls the source; move assignment releases its own first, then steals; both are `noexcept`.

The moved-from state is valid but unspecified — define it for your types, trust `empty()` for `std::vector`, do not read a moved-from `std::string`.

Without `noexcept` on the move constructor, `std::vector` copies elements when it reallocates.

EXERCISE 1

Rule of five buffer

Extend the rule-of-three `IntBuffer` (constructor, copy constructor, copy assignment and destructor are written, with the events `allocate`, `deep-copy`, `copy-assign`, `self-assign` and `release`) with the two move operations, both `noexcept`:

- the **move constructor** takes the source's pointer and size, leaves the source with a null pointer and size 0, and prints `move <size>`;
- the **move assignment** prints `self-assign` and returns when the source is this object; otherwise it releases its own block, takes the source's, leaves the source empty, prints `move-assign <size>` and returns `*this`.

A moved-from `IntBuffer` is defined to print as `size=0 []`. The driver reads `n` and `n` integers into `a`, then: move-constructs `b` from `a` and prints both; move-assigns `b` back into `a` and prints both; move-assigns `a` to itself through a reference (`IntBuffer& same = a; a = std::move(same);`); copy-constructs `c` from `a`; copy-assigns `c` to `b`; prints `sum a=<> b=<> c=<>`. Buffers print as `<name>: size=<n> [<values>]`.

Input: `n`, then `n` integers. **Output:** the trace, the four buffer lines and the sums, as below.

```
3
1 2 3
```

prints

```
allocate 3
move 3
a: size=0 []
b: size=3 [1 2 3]
move-assign 3
a: size=3 [1 2 3]
b: size=0 []
self-assign
deep-copy 3
copy-assign 3
sum a=6 b=6 c=6
release 3
release 3
release 3
```

INPUT 1

```
3
1 2 3
```

OUTPUT 1

```
allocate 3
move 3
a: size=0 []
b: size=3 [1 2 3]
move-assign 3
a: size=3 [1 2 3]
b: size=0 []
self-assign
deep-copy 3
copy-assign 3
sum a=6 b=6 c=6
release 3
release 3
release 3
```

INPUT 2

```
2
-9 4
```

OUTPUT 2

```
allocate 2
move 2
a: size=0 []
b: size=2 [-9 4]
move-assign 2
a: size=2 [-9 4]
b: size=0 []
self-assign
deep-copy 2
copy-assign 2
sum a=-5 b=-5 c=-5
release 2
release 2
release 2
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <utility>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n) : data_(new int[n]()), size_(n) {
        std::cout << "allocate " << size_ << '\n';
    }

    IntBuffer(const IntBuffer& other) : data_(new int[other.size_]), size_(other.siz
        std::copy_n(other.data_, size_, data_);
        std::cout << "deep-copy " << size_ << '\n';
    }

    IntBuffer& operator=(const IntBuffer& other) {
        if (this == &other) {
            std::cout << "self-assign\n";
            return *this;
        }
        int* fresh = new int[other.size_];
        std::copy_n(other.data_, other.size_, fresh);
        delete[] data_;
        data_ = fresh;
        size_ = other.size_;
        std::cout << "copy-assign " << size_ << '\n';
        return *this;
    }

    IntBuffer(IntBuffer&& other) noexcept
        : data_(std::exchange(other.data_, nullptr)), size_(std::exchange(other.size
        std::cout << "move " << size_ << '\n';
    }

    IntBuffer& operator=(IntBuffer&& other) noexcept {
        if (this == &other) {
            std::cout << "self-assign\n";
            return *this;
        }
        delete[] data_;
        data_ = std::exchange(other.data_, nullptr);
        size_ = std::exchange(other.size_, 0);
        std::cout << "move-assign " << size_ << '\n';
        return *this;
    }

    ~IntBuffer() {
        std::cout << "release " << size_ << '\n';
```

```
        delete[] data_;
    }

    std::size_t size() const { return size_; }
    int& operator[](std::size_t i) { return data_[i]; }
    const int& operator[](std::size_t i) const { return data_[i]; }

    void print(const char* name) const {
        std::cout << name << ": size=" << size_ << " [";
        for (std::size_t i = 0; i < size_; ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << data_[i];
        }
        std::cout << "]\n";
    }

    long long sum() const {
        long long total = 0;
        for (std::size_t i = 0; i < size_; ++i) total += data_[i];
        return total;
    }

private:
    int* data_;
    std::size_t size_;
};

int main() {
    std::size_t n;
    std::cin >> n;
    IntBuffer a(n);
    for (std::size_t i = 0; i < n; ++i) std::cin >> a[i];

    IntBuffer b = std::move(a);
    a.print("a");
    b.print("b");

    a = std::move(b);
    a.print("a");
    b.print("b");

    IntBuffer& same = a;
    a = std::move(same);

    IntBuffer c = a;
    b = c;
    std::cout << "sum a=" << a.sum() << " b=" << b.sum() << " c=" << c.sum() << '\n'
    return 0;
}
```

EXERCISE 2

Moved-from, observed

Observe what the standard library promises about moved-from objects — and only that. Read a one-word `label`, then `n` and `n` integers into a vector `staging`, then `m` and `m` more integers.

`std::string title = std::move(label);` and print `title=<title>`. Do **not** print `label` — a moved-from `std::string` is valid but unspecified. Instead assign it `"reused"` and print `label=reused`: assigning to a moved-from object is always allowed.

`std::vector<int> committed = std::move(staging);` and print `commit 1: committed=<size> staging=<size>`. The move constructor guarantees the source is empty, so `staging` prints 0.

Read the `m` integers into `staging` with `push_back` — a moved-from vector can be filled again — then move its *elements* into `committed` with the algorithm `std::move(staging.begin(), staging.end(), std::back_inserter(committed))`. Print `commit 2: committed=<size> staging=<size>`: the algorithm moves elements, not the container, so `staging` still has `m` of them.

`staging.clear();` and print `after clear: staging=0`.

Print `committed=[<values separated by spaces>]` and `sum=<total>` (a `long long`).

Input: `label`, `n`, `n` integers, `m`, `m` integers. **Output:** seven lines.

```
alpha
3
1 2 3
2
4 5
```

prints

```
title=alpha
label=reused
commit 1: committed=3 staging=0
commit 2: committed=5 staging=2
after clear: staging=0
committed=[1 2 3 4 5]
sum=15
```

INPUT 1

```
alpha
3
1 2 3
2
4 5
```

OUTPUT 1

```
title=alpha
label=reused
commit 1: committed=3 staging=0
commit 2: committed=5 staging=2
after clear: staging=0
committed=[1 2 3 4 5]
sum=15
```

INPUT 2

```
batch-7
1
-10
3
1 2 3
```

OUTPUT 2

```
title=batch-7
label=reused
commit 1: committed=1 staging=0
commit 2: committed=4 staging=3
after clear: staging=0
committed=[-10 1 2 3]
sum=-4
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <iostream>
#include <iterator>
#include <string>
#include <utility>
#include <vector>

int main() {
    std::string label;
    std::cin >> label;
    std::string title = std::move(label);
    std::cout << "title=" << title << '\n';
    label = "reused";
    std::cout << "label=" << label << '\n';

    std::size_t n;
    std::cin >> n;
    std::vector<int> staging;
    for (std::size_t i = 0; i < n; ++i) {
        int v;
        std::cin >> v;
        staging.push_back(v);
    }
    std::vector<int> committed = std::move(staging);
    std::cout << "commit 1: committed=" << committed.size() << " staging=" << stagir

    std::size_t m;
    std::cin >> m;
    for (std::size_t i = 0; i < m; ++i) {
        int v;
        std::cin >> v;
        staging.push_back(v);
    }
    std::move(staging.begin(), staging.end(), std::back_inserter(committed));
    std::cout << "commit 2: committed=" << committed.size() << " staging=" << stagir

    staging.clear();
    std::cout << "after clear: staging=" << staging.size() << '\n';

    std::cout << "committed=[";
    for (std::size_t i = 0; i < committed.size(); ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << committed[i];
    }
    std::cout << "]\n";
    long long sum = 0;
    for (int v : committed) sum += v;
    std::cout << "sum=" << sum << '\n';
```

```
    return 0;
}
```

QUIZ

1. What does `std::move(x)` do?

A. Moves `x`'s contents into a temporary and returns it

B. Casts `x` to an rvalue reference so that overload resolution can pick a move constructor or move assignment; it moves nothing itself

C. Destroys `x` after transferring its contents

D. Marks `x` as unusable until it is reassigned

2. After `std::vector<int> b = std::move(a);`, what does the standard guarantee about `a`?

A. `a` is unchanged; vectors always copy

B. `a` is empty

C. Any use of `a` is undefined behaviour

D. `a` has the same size as `b` but unspecified contents

3. After `std::string t = std::move(s);` you print `s`. Which statement is correct?

A. It always prints an empty string, by the standard

B. It is undefined behaviour

C. `s` is valid but its contents are unspecified — usually empty, but code must not rely on it

D. It prints the original text; strings cannot be moved

4. Why should a move constructor be declared `noexcept`?

A. The compiler refuses to generate a `std::vector` of the type otherwise

B. `std::vector` moves elements during reallocation only when the move constructor is `noexcept`; otherwise it copies them to keep its strong exception guarantee

C. `noexcept` makes the move constructor run faster

D. Without it, `std::move` selects the copy constructor

5.

```
void g(const Probe&);   // #1
void g(Probe&&);        // #2
void f(Probe&& p) { g(p); }
```

Which `g` does `f` call?

A. #2, because `p` is declared as an rvalue reference

B. #1, because the expression `p` has a name and is therefore an lvalue

C. Ambiguous — compile error

D. #2 for temporaries and #1 for named objects, decided at run time

6.

```
Probe a(1);
Probe b = std::move(a);
Probe c = b;
```

With a `Probe` that prints each special member, which events follow `construct`?

A. `move`, `copy`

B. `move`, `move`

C. `copy`, `copy`

D. `construct`, `copy`

ANSWER KEY · MOVE SEMANTICS — RVALUE REFERENCES, STD::MOVE AND NOEXCEPT

1. B `std::move` is `static_cast<T&&>(x)`. The transfer, if any, happens inside the move constructor or assignment that the cast lets overload resolution choose — and for a type without one, `std::move` selects the copy.

2. B The vector move constructor is specified to leave the source empty. A moved-from object is always valid; for `std::vector` the state is also *specified*, which is not true of `std::string`.

3. C A moved-from standard object is in a valid but unspecified state. Printing it is well-defined but its value is not promised, so a test that expects the empty string is testing the implementation, not the language.

4. B A move that could throw halfway through a reallocation would leave the vector unrecoverable, so `std::move_if_noexcept` falls back to copying. The cost is silent: the program works and is slower than it looks.

5. B A named rvalue reference is an lvalue inside the function, so it binds to `const Probe&`. Forwarding the move on requires `g(std::move(p))`.

6. A `std::move(a)` is an rvalue, so `b` is move-constructed. `b` on its own is an lvalue, so `c` is copy-constructed; nothing about `b` having been the target of a move makes it an rvalue.

LESSON 4 · 14 MIN

The rule of five and the rule of zero

With moves in the language, a class has five special member functions beyond the default constructor: the destructor, the copy constructor, the copy assignment, the move constructor and the move assignment. The compiler generates each of them under rules that depend on which of the others you declared, and those rules have a consequence that surprises everyone once: writing a destructor switches the moves off. This lesson lays out the generation table, states the rule of five and the rule of zero, and gives the one-line decision for which of them a class should follow.

The five

Special member	Signature	Generated version does
Destructor	`~T()`	destroys members in reverse order
Copy constructor	`T(const T&)`	copy-constructs each member
Copy assignment	`T& operator=(const T&)`	copy-assigns each member
Move constructor	`T(T&&) noexcept`	move-constructs each member
Move assignment	`T& operator=(T&&) noexcept`	move-assigns each member

Plus the default constructor `T()`, which the compiler generates only when the class declares no constructor at all — the rule Module 8 lesson 2 gave. The generated move operations are memberwise moves: each member is moved with `std::move`, so a member that is a `std::vector` is transferred and a member that is an `int` is copied. A generated move is `noexcept` when every member's move is.

What the compiler generates, and when

The full rule fits in four lines:

If you declare **nothing**, all five are generated.

If you declare a **move constructor or move assignment**, the copy constructor and copy assignment are **deleted**. A type that knows how to move but says nothing about copying is presumed non-copyable — `std::unique_ptr` is exactly this shape.

If you declare a **copy constructor, copy assignment or destructor**, the move constructor and move assignment are **not declared** at all. They are not deleted: `std::move(x)` still compiles, overload resolution finds only `const T&`, and the object is *copied*.

`= default` and `= delete` both count as declaring.

Rule 3 is the trap. The reasoning behind it is sound — a class that wrote a destructor or copy is managing something, and a generated memberwise move might transfer that something incorrectly — but the failure mode is silent. The class still compiles, still runs, and is slower than it looks:

```
struct Logged {
    std::vector<int> data;
    ~Logged() { std::cout << "gone\n"; }     // user-declared destructor: move
};

Logged a;
Logged b = std::move(a);      // copies data — a million ints if there are a m
```

Nothing warns. The compiler was told the class has special needs and did the conservative thing. The fix is one line, `Logged(Logged&&) = default;` together with its assignment — or, better, not writing the destructor.

The rules are why "does this class have a move constructor?" is a real interview question with a three-part answer: yes if it declares nothing or declares the moves; no if it declares any of the copy operations or the destructor and does not also default the moves.

The rule of five

The rule of three, extended: **if a class declares any one of the five, it should consider all five.** A rule-of-three class from before C++11 is correct after C++11 — copies work, moves fall back to copies — but it leaves performance behind, because everything that could have been a move is now a copy. Adding the two move operations completes it:

```
class IntBuffer {
public:
    explicit IntBuffer(std::size_t n);
    IntBuffer(const IntBuffer& other);                 // deep copy
    IntBuffer& operator=(const IntBuffer& other);      // self-safe deep copy
    IntBuffer(IntBuffer&& other) noexcept;             // steal, null the sou
    IntBuffer& operator=(IntBuffer&& other) noexcept;  // release, steal, nul
    ~IntBuffer();                                      // delete[]
    // ...
};
```

With copy-and-swap the count drops: a by-value `operator=(IntBuffer other)` serves both assignments, chosen by whether the argument is an lvalue (copy-constructed parameter) or an rvalue (move-constructed parameter). That version is four functions plus a `swap`.

"Consider" is deliberate. Some classes should declare the moves as `= delete` — a type that must never leave its address, such as a mutex — and some should declare the copies deleted and only the moves, like a file handle. The rule is that the decision must be made for all five, not that all five must be written.

The rule of zero

The better rule is to never be in that position. **A class that declares none of the five** gets correct copies, correct moves and a correct destructor from its members — provided every member owns itself:

```
class Document {
public:
    Document(std::string title, std::vector<std::string> lines)
        : title_(std::move(title)), lines_(std::move(lines)) {}
    // no destructor, no copy, no move: std::string and std::vector do it all
private:
    std::string title_;
    std::vector<std::string> lines_;
};
```

`Document` is copyable (both members are), movable (both members are, `noexcept`), and destroys itself. It cannot double-delete, cannot leak, cannot be shallow-copied, because there is no code in it that could get those things wrong. The same holds when a member is `std::unique_ptr`:

```
class Grid {
public:
    Grid(std::size_t rows, std::size_t cols)
        : rows_(rows), cols_(cols), cells_(std::make_unique<int[]>(rows * cols
    bool empty() const { return !cells_; }
private:
    std::size_t rows_, cols_;
    std::unique_ptr<int[]> cells_;
};

Grid g(2, 3);
Grid h = std::move(g);     // moves cells_, copies rows_ and cols_; g.empty()
Grid k = h;                // error: unique_ptr's copy constructor is deleted,
```

The class is movable and non-copyable by construction, and the compiler says so at the point of use. Note what the moved-from `g` looks like: `rows_` and `cols_` still hold `2` and `3`, because ints copy, and `cells_` is null. A rule-of-zero class's moved-from state is the sum of its members' moved-from states, which is why `empty()` tests the pointer and not the sizes.

The name means what it says: the ideal number of special member functions in a class is zero. Ownership of raw resources belongs in small, single-purpose classes — `std::unique_ptr`, `std::vector`, a `FileHandle` you write once — and everything else composes them.

Which rule, when

The class...	Rule	Declare
owns nothing directly; members are strings, vectors, smart pointers	zero	nothing
owns one raw resource (a `new[]` block, a file descriptor, a C handle)	five	all five, `noexcept` on the moves
must not be copied but may be moved	five	copies `= delete`, moves written or defaulted
must not be copied or moved (a mutex, a singleton)	five	copies `= delete`; the moves are then not generated, so `std::move` fails to compile too
is a polymorphic base	five	`virtual ~T() = default;` plus the four defaulted, usually `protected` (Module 10)

Two habits follow. First, one raw resource per class: a class that owns two raw pointers has a copy assignment that must handle one allocation succeeding and the other failing, which is exactly the code nobody gets right. Second, `= default` is documentation: `Widget(Widget&&) = default;` in a class that needed a destructor says "I know the rule, and the memberwise move is correct here".

Pitfalls

Mistake	What happens
A destructor added for logging or debugging	The moves vanish; every `std::move` becomes a copy
Copy constructor written, move forgotten	Correct but slow: `std::move` and returns copy
Move constructor written, copies not mentioned	The copies are deleted; `T b = a;` fails to compile
Two raw owning pointers in one class	An assignment operator with a failure case between two allocations
Defaulting the moves in a class with a raw owning pointer	The memberwise move copies the pointer and leaves the source owning it too

---

Key takeaways

- Five special members: destructor, copy constructor, copy assignment, move constructor, move assignment — all generated memberwise when you declare none.
- Declaring a copy operation or a destructor stops the moves being generated; `std::move` then silently copies. Declaring a move deletes the copies.
- Rule of five: declare one, decide about all five. Rule of zero: own resources through members and declare none.
- A rule-of-zero class's moved-from state is its members' moved-from states — a `std::unique_ptr` becomes null, an `int` stays.
- Keep raw ownership in small single-resource classes and compose them; `= default` on a move documents that the memberwise version is right.

EXERCISE 1

A rule-of-zero grid

Write `Grid`, a rows × columns table of `int` that owns its cells through `std::unique_ptr<int[]>` and declares **none** of the five special member functions — the rule of zero. The compiler makes it movable and non-copyable because `std::unique_ptr` is.

- `Grid(std::size_t rows, std::size_t cols)` allocates `rows * cols` zeroed cells with `std::make_unique<int[]>`.
- `int& at(std::size_t r, std::size_t c)` and a `const` overload, row-major: `cells_[r * cols_ + c]`.
- `bool empty() const` returns whether the pointer is null — the moved-from state of a rule-of-zero class is its members' moved-from states, and a moved-from `std::unique_ptr` is null while the two sizes are simply copied.
- `Grid transpose() const` returns a new columns × rows grid.
- `void print() const` prints each row on its own line, values separated by single spaces.

Read `R C` and then `R` rows of `C` integers into `g`. Compute `Grid t = g.transpose();`, then `Grid moved = std::move(g);`. Print `grid RxC:` followed by `moved`'s rows, `transposed CxR:` followed by `t`'s rows, then `source empty=true` and `moved empty=false` from the two `empty()` calls (print `true` / `false`). If you are curious, add `Grid copy = moved;` and read the compiler's error before removing it.

Input: `R C`, then `R` lines of `C` integers. **Output:** as below.

```
2 3
1 2 3
4 5 6
```

prints

```
grid 2x3:
1 2 3
4 5 6
transposed 3x2:
1 4
2 5
3 6
source empty=true
moved empty=false
```

INPUT 1

```
2 3
1 2 3
4 5 6
```

OUTPUT 1

```
grid 2x3:
1 2 3
4 5 6
transposed 3x2:
1 4
2 5
3 6
source empty=true
moved empty=false
```

INPUT 2

```
1 4
7 8 9 10
```

OUTPUT 2

```
grid 1x4:
7 8 9 10
transposed 4x1:
7
8
9
10
source empty=true
moved empty=false
```

REFERENCE SOLUTION

```cpp
#include <cstddef>
#include <iostream>
#include <memory>
#include <utility>

class Grid {
public:
    Grid(std::size_t rows, std::size_t cols)
        : rows_(rows), cols_(cols), cells_(std::make_unique<int[]>(rows * cols)) {}

    // No copy, no move, no destructor: the unique_ptr member owns the cells.

    std::size_t rows() const { return rows_; }
    std::size_t cols() const { return cols_; }

    int& at(std::size_t r, std::size_t c) { return cells_[r * cols_ + c]; }
    const int& at(std::size_t r, std::size_t c) const { return cells_[r * cols_ + c]

    bool empty() const { return !cells_; }

    Grid transpose() const {
        Grid t(cols_, rows_);
        for (std::size_t r = 0; r < rows_; ++r)
            for (std::size_t c = 0; c < cols_; ++c)
                t.at(c, r) = at(r, c);
        return t;
    }

    void print() const {
        for (std::size_t r = 0; r < rows_; ++r) {
            for (std::size_t c = 0; c < cols_; ++c) {
                if (c > 0) std::cout << ' ';
                std::cout << at(r, c);
            }
            std::cout << '\n';
        }
    }

private:
    std::size_t rows_;
    std::size_t cols_;
    std::unique_ptr<int[]> cells_;
};

int main() {
    std::size_t r, c;
    std::cin >> r >> c;
    Grid g(r, c);
    for (std::size_t i = 0; i < r; ++i)
```

```
        for (std::size_t j = 0; j < c; ++j)
            std::cin >> g.at(i, j);

    Grid t = g.transpose();
    Grid moved = std::move(g);

    std::cout << "grid " << moved.rows() << 'x' << moved.cols() << ":\n";
    moved.print();
    std::cout << "transposed " << t.rows() << 'x' << t.cols() << ":\n";
    t.print();
    std::cout << "source empty=" << (g.empty() ? "true" : "false") << '\n';
    std::cout << "moved empty=" << (moved.empty() ? "true" : "false") << '\n';
    return 0;
}
```

EXERCISE 2

What std::move actually does

Test the generation table. `Tracker` is a member type that prints `copy`, `move`, `copy-assign` or `move-assign` (two leading spaces) whenever one of its special members runs. Complete four wrapper structs, each holding one `Tracker t;`:

- `Zero` declares nothing;
- `DtorOnly` declares only `~DtorOnly() = default;`;
- `CopyOnly` declares `CopyOnly() = default;`, a defaulted copy constructor and a defaulted copy assignment;
- `Five` declares a defaulted default constructor and all five special members, defaulted.

Read words until the input ends. For each word print it on its own line, then run `X a; X b = std::move(a); a = std::move(b);` for the matching type, so `Tracker` prints what the compiler generated: a move when the move operations exist, a copy when they were suppressed. An unknown word prints `unknown`. Predict the output before you run it: which of the four declared something that stops the moves being generated?

Input: words separated by whitespace. **Output:** three lines per word.

```
zero dtor
```

prints

```
zero
  move
  move-assign
dtor
  copy
  copy-assign
```

INPUT 1

```
zero dtor
```

OUTPUT 1

```
zero
  move
  move-assign
dtor
  copy
  copy-assign
```

INPUT 2

```
copy five
```

OUTPUT 2

```
copy
  copy
  copy-assign
five
  move
  move-assign
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>
#include <utility>

struct Tracker {
    Tracker() = default;
    Tracker(const Tracker&) { std::cout << "  copy\n"; }
    Tracker(Tracker&&) noexcept { std::cout << "  move\n"; }
    Tracker& operator=(const Tracker&) { std::cout << "  copy-assign\n"; return *thi
    Tracker& operator=(Tracker&&) noexcept { std::cout << "  move-assign\n"; return
};

struct Zero {
    Tracker t;
};

struct DtorOnly {
    Tracker t;
    ~DtorOnly() = default;
};

struct CopyOnly {
    Tracker t;
    CopyOnly() = default;
    CopyOnly(const CopyOnly&) = default;
    CopyOnly& operator=(const CopyOnly&) = default;
};

struct Five {
    Tracker t;
    Five() = default;
    Five(const Five&) = default;
    Five& operator=(const Five&) = default;
    Five(Five&&) = default;
    Five& operator=(Five&&) = default;
    ~Five() = default;
};

template <typename X>
void exercise() {
    X a;
    X b = std::move(a);
    a = std::move(b);
}

int main() {
    std::string word;
    while (std::cin >> word) {
        std::cout << word << '\n';
```

```
        if (word == "zero") exercise<Zero>();
        else if (word == "dtor") exercise<DtorOnly>();
        else if (word == "copy") exercise<CopyOnly>();
        else if (word == "five") exercise<Five>();
        else std::cout << "  unknown\n";
    }
    return 0;
}
```

QUIZ

1. A class declares only a destructor. What does `T b = std::move(a);` do?

A. Move-constructs `b`; the destructor is irrelevant

B. Copy-constructs `b`: a user-declared destructor stops the move operations being generated, and `std::move` falls back to the copy constructor

C. Fails to compile

D. Move-constructs `b` but then calls the destructor on `a` immediately

2. A class declares only a move constructor. What happens to `T b = a;` where `a` is an lvalue?

A. Copies `a`, using the generated copy constructor

B. Moves `a`, since that is the only constructor

C. Fails to compile: declaring a move operation deletes the copy constructor and copy assignment

D. Copies `a` with a warning

3. Does `~Widget() = default;` count as declaring a destructor for the generation rules?

A. No — `= default` asks for the generated one, so nothing changes

B. Yes — it is user-declared, and the implicit move operations are no longer generated

C. Only if the class also has a virtual function

D. Only when the destructor is defined outside the class

4. Which class follows the rule of zero?

A. One that declares all five special members as `= default`

B. One whose members are `std::string`, `std::vector<int>` and `std::unique_ptr<Node>`, and declares none of the five

C. One with a raw owning pointer and a destructor that deletes it

D. One with no data members at all

5.

```
class Grid {
    std::size_t rows_, cols_;
    std::unique_ptr<int[]> cells_;
};
Grid h = std::move(g);
```

What is the state of `g` afterwards?

A. `rows_` and `cols_` are 0 and `cells_` is null

B. `rows_` and `cols_` keep their values and `cells_` is null

C. `g` is destroyed

D. Unchanged: a class with no move constructor cannot be moved

6. Which of these types can be copied?

A. `struct A { std::unique_ptr<int> p; };`

B. `struct B { std::vector<int> v; std::string s; };`

C. `struct C { C(C&&) = default; };`

D. `struct D { D(const D&) = delete; };`

ANSWER KEY · THE RULE OF FIVE AND THE RULE OF ZERO

1. B Declaring any of destructor, copy constructor or copy assignment means the implicit moves are not declared. They are not deleted, so overload resolution quietly picks `const T&`. The program compiles and copies.

2. C A type that says how to move and nothing about copying is presumed non-copyable, so the copy operations are defined as deleted. `std::unique_ptr` has exactly this shape.

3. B `= default` and `= delete` both declare the function. A defaulted destructor is exactly what a polymorphic base needs, and exactly why such a base should also default its copy and move operations explicitly.

4. B The rule of zero is about ownership: every member owns itself, so the class needs no special member functions and the memberwise defaults are correct. Defaulting all five is a rule-of-five class that happens to have easy members.

5. B The generated move constructor moves each member: `std::unique_ptr`'s move leaves the source null, and an integer's "move" is a copy. A rule-of-zero class's moved-from state is the sum of its members' moved-from states.

6. B `B`'s members are all copyable, so its generated copy constructor is fine. `A`'s generated copy is deleted because a member's is; `C` declared a move, which deletes the copies; `D` deleted the copy explicitly.

LESSON 5 · 15 MIN

Value categories, temporaries and copy elision

Every C++ expression has a type and a **value category**, and the category is what decides whether an initialisation copies, moves or does neither. The three categories have unlovely names — lvalue, prvalue, xvalue — but the ideas behind them are plain: does the expression name an object that persists, is it a pure value that has not been given a home yet, or is it an object whose contents may be taken? This lesson gives the taxonomy in those terms, explains when a temporary is created and when it dies, states the C++17 guarantee that a prvalue initialises its target with no copy at all, and settles the two return-statement questions every C++ interviewer asks.

Three categories in plain terms

Category	Meaning	Examples
lvalue	names an object that outlives the expression	`a`, `*p`, `v[i]`, `s.name`, a call returning `T&`
prvalue	a pure value with no object yet	`42`, `a + b`, `Probe(3)`, a call returning `T`
xvalue	an object whose contents may be reused	`std::move(a)`, a call returning `T&&`, `Probe(3).value`

Two groupings matter. An lvalue or xvalue has *identity* — it refers to an actual object — so together they are **glvalues**. A prvalue or xvalue may be *moved from*, so together they are **rvalues**. When lesson 3 said `T&&` binds to rvalues, it meant both kinds: a temporary from a prvalue, or an lvalue that `std::move` has turned into an xvalue.

The category is a property of the *expression*, not the variable. `Probe&& r = std::move(a);` declares `r` as an rvalue reference, but the expression `r` is an lvalue — it has a name — so `Probe b = r;` copies. That is why a forwarding function has to write `std::move(r)` again.

Which reference binds to which

```
void f(Probe& p);          // lvalues only
void g(const Probe& p);    // anything
void h(Probe&& p);         // rvalues only

Probe a(1);
f(a);  g(a);  h(std::move(a));   // h(a) does not compile: a is an lvalue
g(Probe(2));  h(Probe(2));       // f(Probe(2)) does not compile: a temporary
```

With `g` and `h` both present, an rvalue argument picks `h` — the more specific match — which is how the copy constructor and move constructor are chosen. With only `g` present, an rvalue argument binds to `const Probe&` and is copied. That last case is what "declaring a copy suppresses the move" from lesson 4 looks like at the call site.

Temporaries and their lifetime

A prvalue becomes an object — is **materialised** into a temporary — only when something needs an object: binding it to a reference, calling a member function on it, passing it to a by-reference parameter. The temporary lives until the end of the **full-expression** that created it, usually the semicolon:

```
std::size_t n = (std::string("hello") + " world").size();   // the temporary s
```

Binding a temporary directly to a `const T&` or `T&&` local extends its lifetime to the reference's:

```
const std::string& greeting = makeGreeting();   // the temporary lives as long
std::string&& owned = makeGreeting();           // same, and owned may be modi
```

Extension applies only to the direct binding. A reference obtained *through* the temporary — `const std::string& first = makePair().first;` extends, but `const char* p = makeGreeting().c_str();` leaves `p` dangling the moment the statement ends. Module 5, lesson 6 showed the same trap with `std::string_view`.

Guaranteed copy elision

Since C++17, initialising an object from a prvalue of the same type does not copy and does not move: the prvalue's value is constructed **directly in the target**. There is no temporary to

elide, which is why the standard calls this "unmaterialised value passing" rather than an optimisation.

```
Probe make(int v) { return Probe(v); }       // prvalue returned

Probe a = Probe(1);         // construct #1 — one event, no copy, no move
Probe b = make(2);          // construct #2 — the object is built in b
take(Probe(3));             // construct #3 — built in the parameter
Probe c = flag ? Probe(4) : Probe(5);       // both arms are prvalues: still
```

Because no copy or move is *performed*, none is *required*: a factory `static Widget create() { return Widget(...); }` works for a type whose copy and move constructors are deleted. This is what makes returning by value the right default — the call `make(2)` costs exactly one construction wherever its result ends up.

Named return value optimisation

`return Probe(v);` is a prvalue and always elides. `return local;` is different: `local` is an lvalue with a name, and the compiler is *permitted*, not required, to build it directly in the caller's object. This is **NRVO**, named return value optimisation. Every major compiler does it whenever the function has one local that every `return` statement returns; it cannot when different paths return different locals, when the returned object is a parameter, or when the returned expression is not plain `local`.

When NRVO does not happen, the fallback is still not a copy. A `return` statement that names a local variable (or a by-value parameter) treats it as an rvalue first — **implicit move** — so the return object is move-constructed from it. Returning a parameter is the fully deterministic case: NRVO is never applied to parameters, so `Probe pass(Probe p) { return p; }` always performs exactly one move construction. In a trace, that shows as one `move` event; `return Probe(v)` shows none.

Two rules for `return`

You write	What happens	Verdict
`return local;`	NRVO (usually), else implicit move	right
`return Probe(v);`	guaranteed elision	right
`return std::move(local);`	NRVO disabled; a move always	pessimisation; GCC and Clang warn with `-Wpessimizing-move`
`return std::move(param);`	one move, same as `return param;`	redundant; `-Wredundant-move`
`const Probe& f() { Probe p; return p; }`	reference to a destroyed local	undefined behaviour

The pessimisation is the most common misuse of `std::move` in real code. `std::move(local)` is an xvalue, not a name, so it no longer qualifies for NRVO; the compiler must do the move it was previously allowed to skip. Return by value, return the name, and let the compiler choose.

`std::move` on a `const` object copies

```
const Probe k(7);
Probe m = std::move(k);        // copy #… from k: quietly
```

`std::move(k)` yields `const Probe&&`. The move constructor takes `Probe&&`, which cannot bind to a `const` object, so overload resolution falls through to `const Probe&` — the copy constructor. No error, no warning by default, just a copy where a move was intended. The same happens for a `const` *member*: a class with a `const std::string name;` can never move that member, so its generated move constructor copies it. The lesson is that `const` and moving are opposites, which is why a class's data members are almost never `const`.

Pitfalls

Mistake	What happens
`return std::move(local);`	Disables NRVO and forces a move
`const char* p = makeString().c_str();`	Dangling as soon as the statement ends
Treating a named `T&&` as an rvalue	It is an lvalue; the callee copies unless you `std::move` it again
`std::move` on a `const` object or member	Selects the copy constructor without a diagnostic
Assuming NRVO in a trace or a test	Permitted, not guaranteed; only prvalue elision and parameter returns are certain
Returning a reference to a local	Undefined behaviour; return by value instead

Key takeaways

lvalue: has a name and persists; prvalue: a pure value not yet stored; xvalue: an object whose contents may be taken. Rvalues are prvalues and xvalues; `T&&` binds to those.

A named rvalue reference is an lvalue; forwarding it on needs another `std::move`.

A temporary dies at the end of its full-expression unless bound directly to a `const T&` or `T&&`, which extends it — not through a member function's return.

C++17 guarantees that initialising from a prvalue of the same type performs no copy or move at all; return `T(...)` and the object is built in place.

`return local;` gets NRVO or an implicit move; `return std::move(local);` gets only the move. Returning a by-value parameter is always exactly one move.

`std::move` on a `const` object or member silently copies.

EXERCISE 1

Elision, traced

Complete the five special members of `Probe` — copy constructor (`copy #<id> from #<source>`), copy assignment (`copy-assign …`), move constructor (`move …`, and the source's value becomes 0), move assignment (`move-assign …`, source zeroed) — and write `Probe make(int v)`, which returns

`Probe(v)`, and `Probe pass(Probe p)`, which returns `p`. Then predict every line of this driver before running it:

Read `v1 v2 v3 v4`.

`Probe a = Probe(v1);` — a prvalue initialiser: guaranteed elision, one `construct`.

`Probe b = make(v2);` — a prvalue returned: still one `construct`.

`Probe c = pass(b);` — `b` is copied into the parameter, the parameter is returned with an implicit move (never NRVO — it is a parameter), then the parameter is destroyed.

`c = Probe(v3);` — a temporary is constructed, move-assigned, destroyed.

`const Probe k(v4);` then `Probe d = std::move(k);` — a `const` object cannot be moved from: watch which event appears.

`Probe e = std::move(a);`

Print `values a=<> b=<> c=<> d=<> e=<>`.

Everything dies in reverse order.

Input: four integers. **Output:** the trace, the values line, and six destroy lines.

```
1 2 3 7
```

prints

```
construct #1=1
construct #2=2
copy #3 from #2
move #4 from #3
destroy #3
construct #5=3
move-assign #4 from #5
destroy #5
construct #6=7
copy #7 from #6
move #8 from #1
values a=0 b=2 c=3 d=7 e=1
destroy #8
destroy #7
destroy #6
destroy #4
destroy #2
destroy #1
```

INPUT 1

```
1 2 3 7
```

OUTPUT 1

```
construct #1=1
construct #2=2
copy #3 from #2
move #4 from #3
destroy #3
construct #5=3
move-assign #4 from #5
destroy #5
construct #6=7
copy #7 from #6
move #8 from #1
values a=0 b=2 c=3 d=7 e=1
destroy #8
destroy #7
destroy #6
destroy #4
destroy #2
destroy #1
```

INPUT 2

```
10 20 30 40
```

OUTPUT 2

```
construct #1=10
construct #2=20
copy #3 from #2
move #4 from #3
destroy #3
construct #5=30
move-assign #4 from #5
destroy #5
construct #6=40
copy #7 from #6
move #8 from #1
values a=0 b=20 c=30 d=40 e=10
destroy #8
destroy #7
destroy #6
destroy #4
destroy #2
destroy #1
```

REFERENCE SOLUTION

```
#include <iostream>
#include <utility>

struct Probe {
    static inline int next = 1;
    int id;
    int value;

    explicit Probe(int v) : id(next++), value(v) {
        std::cout << "construct #" << id << '=' << value << '\n';
    }
    Probe(const Probe& other) : id(next++), value(other.value) {
        std::cout << "copy #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(const Probe& other) {
        value = other.value;
        std::cout << "copy-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    Probe(Probe&& other) noexcept : id(next++), value(std::exchange(other.value, 0))
        std::cout << "move #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(Probe&& other) noexcept {
        value = std::exchange(other.value, 0);
        std::cout << "move-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    ~Probe() {
        std::cout << "destroy #" << id << '\n';
    }
};

Probe make(int v) {
    return Probe(v);
}

Probe pass(Probe p) {
    return p;
}

int main() {
    int v1, v2, v3, v4;
    std::cin >> v1 >> v2 >> v3 >> v4;

    Probe a = Probe(v1);
    Probe b = make(v2);
    Probe c = pass(b);
    c = Probe(v3);
    const Probe k(v4);
```

```
    Probe d = std::move(k);
    Probe e = std::move(a);

    std::cout << "values a=" << a.value << " b=" << b.value << " c=" << c.value
              << " d=" << d.value << " e=" << e.value << '\n';
    return 0;
}
```

EXERCISE 2

Sink parameters

`Inbox` stores `Probe` objects in a `std::vector` and takes new ones through a **sink** parameter — `void add(Probe p)` — which it moves into the vector with `items_.push_back(std::move(p))`. The constructor reserves room for every item up front, so no reallocation ever happens and every event is fixed by the standard. Complete the five-special-member `Probe`, `add`, `drain` (which pops items from the back until the vector is empty, so the destruction order is defined) and `total`.

Read `n`, then `n` lines of `<form> <value>`, and call `add` in the matching way:

- `copy v`: in a block, `Probe p(v); inbox.add(p);` — the lvalue is copied into the parameter, the parameter is moved into the vector, the parameter dies, then `p` dies at the block's end;
- `move v`: in a block, `Probe p(v); inbox.add(std::move(p));` — moved into the parameter, moved into the vector;
- `temp v`: `inbox.add(Probe(v));` — the prvalue is built directly in the parameter (guaranteed elision), then moved into the vector.

After the loop print `total=<sum of the stored values>`, then `inbox.drain()`.

Input: `n`, then `n` lines. **Output:** the trace, the total, and the `n` destroy lines from `drain`.

```
3
copy 5
move 6
temp 7
```

prints

```
construct #1=5
copy #2 from #1
move #3 from #2
destroy #2
destroy #1
construct #4=6
move #5 from #4
move #6 from #5
destroy #5
destroy #4
construct #7=7
move #8 from #7
destroy #7
total=18
destroy #8
destroy #6
destroy #3
```

INPUT 1

```
3
copy 5
move 6
temp 7
```

OUTPUT 1

```
construct #1=5
copy #2 from #1
move #3 from #2
destroy #2
destroy #1
construct #4=6
move #5 from #4
move #6 from #5
destroy #5
destroy #4
construct #7=7
move #8 from #7
destroy #7
total=18
destroy #8
destroy #6
destroy #3
```

INPUT 2

```
2
temp -4
temp 10
```

OUTPUT 2

```
construct #1=-4
move #2 from #1
destroy #1
construct #3=10
move #4 from #3
destroy #3
total=6
destroy #4
destroy #2
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <string>
#include <utility>
#include <vector>

struct Probe {
    static inline int next = 1;
    int id;
    int value;

    explicit Probe(int v) : id(next++), value(v) {
        std::cout << "construct #" << id << '=' << value << '\n';
    }
    Probe(const Probe& other) : id(next++), value(other.value) {
        std::cout << "copy #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(const Probe& other) {
        value = other.value;
        std::cout << "copy-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    Probe(Probe&& other) noexcept : id(next++), value(std::exchange(other.value, 0))
        std::cout << "move #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(Probe&& other) noexcept {
        value = std::exchange(other.value, 0);
        std::cout << "move-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    ~Probe() {
        std::cout << "destroy #" << id << '\n';
    }
};

class Inbox {
public:
    explicit Inbox(std::size_t capacity) { items_.reserve(capacity); }

    void add(Probe p) {
        items_.push_back(std::move(p));
    }

    void drain() {
        while (!items_.empty()) items_.pop_back();
    }

    long long total() const {
        long long sum = 0;
        for (const Probe& p : items_) sum += p.value;
```

```
        return sum;
    }

private:
    std::vector<Probe> items_;
};

int main() {
    std::size_t n;
    std::cin >> n;
    Inbox inbox(n);
    for (std::size_t i = 0; i < n; ++i) {
        std::string form;
        int v;
        std::cin >> form >> v;
        if (form == "copy") {
            Probe p(v);
            inbox.add(p);
        } else if (form == "move") {
            Probe p(v);
            inbox.add(std::move(p));
        } else {
            inbox.add(Probe(v));
        }
    }
    std::cout << "total=" << inbox.total() << '\n';
    inbox.drain();
    return 0;
}
```

QUIZ

1. What is the value category of the expression `std::move(x)`?

A. lvalue

B. prvalue

C. xvalue

D. It has no category; it is a function call

2. Which of these expressions is a prvalue?

A. `a`

B. `*p`

C. `Probe(3)`

D. `std::move(a)`

3.

```
Probe make() { return Probe(1); }
Probe b = make();
```

How many copy or move constructions happen in C++17 and later?

A. Two: one for the return, one for `b`

B. One: the return is elided, then `b` is moved

C. Zero: initialising from a prvalue builds the object directly in `b`, guaranteed

D. Zero on most compilers, but only as an optimisation

4. What is wrong with `return std::move(local);` for a local variable of the function's return type?

A. Nothing; it is the recommended form

B. It disables named return value optimisation and forces a move that `return local;` could have avoided entirely

C. It copies instead of moving

D. It returns a dangling reference

5.

```
const Probe k(1);
Probe m = std::move(k);
```

Which constructor runs?

A. The move constructor

B. The copy constructor: `std::move` on a `const` object yields `const Probe&&`, which cannot bind to `Probe&&`

C. Neither — compile error

D. The move constructor, after an implicit `const_cast`

6.

```
std::string makeGreeting();
const std::string& r = makeGreeting();
std::cout << r;
```

Is this safe?

A. No — the temporary dies at the semicolon and `r` dangles

B. Yes — binding a temporary directly to a `const&` extends its lifetime to the reference's scope

C. Yes, but only because `std::string` is small

D. No — a `const&` cannot bind to a temporary

7.

```
Probe pass(Probe p) { return p; }
Probe c = pass(b);
```

Which events does the `return` statement itself produce?

A. None — NRVO builds `p` directly in `c`

B. Exactly one move construction: a returned parameter is treated as an rvalue and never elided

C. One copy construction, since `p` is an lvalue

D. One move and one copy

ANSWER KEY · VALUE CATEGORIES, TEMPORARIES AND COPY ELISION

1. C `std::move` returns `T&&`, and a function call returning an rvalue reference is an xvalue — an expression with identity whose resources may be reused. A call returning `T` by value would be a prvalue.

2. C `Probe(3)` is a pure value with no object yet — it gets one only when something needs it. `a` and `*p` name objects (lvalues); `std::move(a)` is an xvalue.

3. C Since C++17 a prvalue initialises its target without any temporary; this is mandated, not optional, and it works even when the copy and move constructors are deleted. NRVO for a *named* local is the case that remains optional.

4. B `std::move(local)` is an xvalue, not a name, so NRVO no longer applies. `return local;` already treats the local as an rvalue if it cannot be elided, so the explicit `std::move` can only make things worse; GCC and Clang warn with `-Wpessimizing-move`.

5. B A move must modify its source, so the move constructor takes a non-`const` rvalue reference. `const Probe&&` binds only to `const Probe&`, and overload resolution quietly picks the copy, with no diagnostic.

6. B Lifetime extension applies to a temporary bound directly to a `const T&` or `T&&`. It does not reach through a member function: `const char* p = makeGreeting().c_str();` dangles immediately.

7. B NRVO is never applied to function parameters, and a `return` naming a by-value parameter performs the implicit move. The result is deterministic: one `move`, then the parameter's `destroy`. The initialisation of `c` from the call is a prvalue and elides.

CHECKPOINT · 28 MIN

Checkpoint — Copies, moves and the rule of five

This checkpoint covers the whole module: copy construction versus copy assignment, destructors and the order things die, the rule of three and copy-and-swap, rvalue references and `std::move`, the moved-from state and `noexcept`, the generation table behind the rule of five and the rule of zero, and value categories with guaranteed elision.

How it works. Fourteen questions and three programs. You need 70% on the questions and every program accepted to clear the module. You can retake it as often as you like; your best score counts.

Before you start, make sure you can answer these from memory:

- Why does the generated copy of a class with an owning raw pointer double-delete?
- In what order are three locals, and the members of one object, destroyed?
- Why does copy-and-swap need no self-assignment check?
- What does `std::move` do, and what does it not do?
- Which declarations stop the compiler generating the move operations?
- What state is a `std::vector` in after being moved *from*, and a `std::string`?
- Why is `return std::move(local);` worse than `return local;`?

The three programs are a fixed-capacity stack with all five special members driven by commands, an event trace through a struct of two probes that you predict before you run, and two shelves passing books between them through `std::unique_ptr` with no special members at all.

EXERCISE 1

A stack with the rule of five

Write `IntStack`, a fixed-capacity stack of `int` that owns its storage through a raw pointer and implements all five special member functions. Events: the constructor prints `allocate <capacity>`; the copy constructor `deep-copy <size>`; the copy assignment `copy-assign <size>` (or `self-assign` for itself, doing nothing else); the move constructor `move <size>`; the move assignment `move-assign <size>` (or `self-assign`); the destructor `release <capacity>`. A copy takes the source's capacity and elements. A moved-from stack has a null pointer, capacity 0 and size 0 — so a push into it reports `full`, and it releases 0.

Read `cap`, then `k`, then `k` commands operating on two stacks `a` and `b`, both constructed with capacity `cap` (in that order):

- `push X v` — pushes; prints `X: full` if there is no room, nothing otherwise;
- `pop X` — prints `X: popped <v>` or `X: empty`;
- `print X` — prints `X: size=<n> [<values bottom to top>]`;
- `copy X Y` — `X = Y` (copy assignment; when `X` and `Y` are the same name it is a self-assignment);
- `move X Y` — `X = std::move(Y)`;
- `clone X` — in a block, copy-constructs a stack from `X` and prints it as `clone: size=<n> [<values>]`; the clone dies at the block's end.

At the end of `main` the two stacks are destroyed in reverse order.

Input: `cap`, `k`, then `k` command lines. **Output:** the trace, as below.

```
2
8
push a 1
push a 2
push a 3
copy b a
pop b
move a b
print a
print b
```

prints

```
allocate 2
allocate 2
a: full
copy-assign 2
b: popped 2
move-assign 1
a: size=1 [1]
b: size=0 []
release 0
release 2
```

INPUT 1

```
2
8
push a 1
push a 2
push a 3
copy b a
pop b
move a b
print a
print b
```

OUTPUT 1

```
allocate 2
allocate 2
a: full
copy-assign 2
b: popped 2
move-assign 1
a: size=1 [1]
b: size=0 []
release 0
release 2
```

INPUT 2

```
3
7
push a 4
push a 5
clone a
copy a a
pop b
move a a
print a
```

OUTPUT 2

```
allocate 3
allocate 3
deep-copy 2
clone: size=2 [4 5]
release 3
self-assign
b: empty
self-assign
a: size=2 [4 5]
release 3
release 3
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cstddef>
#include <iostream>
#include <string>
#include <utility>

class IntStack {
public:
    explicit IntStack(std::size_t capacity)
        : data_(new int[capacity]()), capacity_(capacity), size_(0) {
        std::cout << "allocate " << capacity_ << '\n';
    }

    IntStack(const IntStack& other)
        : data_(new int[other.capacity_]()), capacity_(other.capacity_), size_(other
        std::copy_n(other.data_, size_, data_);
        std::cout << "deep-copy " << size_ << '\n';
    }

    IntStack& operator=(const IntStack& other) {
        if (this == &other) {
            std::cout << "self-assign\n";
            return *this;
        }
        int* fresh = new int[other.capacity_]();
        std::copy_n(other.data_, other.size_, fresh);
        delete[] data_;
        data_ = fresh;
        capacity_ = other.capacity_;
        size_ = other.size_;
        std::cout << "copy-assign " << size_ << '\n';
        return *this;
    }

    IntStack(IntStack&& other) noexcept
        : data_(std::exchange(other.data_, nullptr)),
          capacity_(std::exchange(other.capacity_, 0)),
          size_(std::exchange(other.size_, 0)) {
        std::cout << "move " << size_ << '\n';
    }

    IntStack& operator=(IntStack&& other) noexcept {
        if (this == &other) {
            std::cout << "self-assign\n";
            return *this;
        }
        delete[] data_;
        data_ = std::exchange(other.data_, nullptr);
        capacity_ = std::exchange(other.capacity_, 0);
```

```
        size_ = std::exchange(other.size_, 0);
        std::cout << "move-assign " << size_ << '\n';
        return *this;
    }

    ~IntStack() {
        std::cout << "release " << capacity_ << '\n';
        delete[] data_;
    }

    bool push(int v) {
        if (size_ == capacity_) return false;
        data_[size_++] = v;
        return true;
    }
    bool pop(int& out) {
        if (size_ == 0) return false;
        out = data_[--size_];
        return true;
    }
    void print(const std::string& name) const {
        std::cout << name << ": size=" << size_ << " [";
        for (std::size_t i = 0; i < size_; ++i) {
            if (i > 0) std::cout << ' ';
            std::cout << data_[i];
        }
        std::cout << "]\n";
    }

private:
    int* data_;
    std::size_t capacity_;
    std::size_t size_;
};

int main() {
    std::size_t cap;
    std::size_t k;
    std::cin >> cap >> k;
    IntStack a(cap);
    IntStack b(cap);
    for (std::size_t i = 0; i < k; ++i) {
        std::string cmd, x;
        std::cin >> cmd >> x;
        IntStack& target = (x == "a") ? a : b;
        if (cmd == "push") {
            int v;
            std::cin >> v;
            if (!target.push(v)) std::cout << x << ": full\n";
```

```
        } else if (cmd == "pop") {
            int v;
            if (target.pop(v)) std::cout << x << ": popped " << v << '\n';
            else std::cout << x << ": empty\n";
        } else if (cmd == "print") {
            target.print(x);
        } else if (cmd == "copy") {
            std::string y;
            std::cin >> y;
            IntStack& source = (y == "a") ? a : b;
            target = source;
        } else if (cmd == "move") {
            std::string y;
            std::cin >> y;
            IntStack& source = (y == "a") ? a : b;
            target = std::move(source);
        } else if (cmd == "clone") {
            IntStack c = target;
            c.print("clone");
        }
    }
    return 0;
}
```

EXERCISE 2

Predict the trace

Write the five-special-member `Probe` from lesson 5 (events `construct`, `copy`, `copy-assign`, `move`, `move-assign`, `destroy`; a moved-from value becomes 0), `struct Pair { Probe first; Probe second; };` with no special members, `Pair makePair(int x, int y)` returning `Pair{Probe(x), Probe(y)}`, and `Probe pick(const Pair& p, bool first)` returning `first ? p.first : p.second` by value. Then write this driver and predict its trace **before** running it:

Read `x y z`.

`Pair p = makePair(x, y);`

`Probe q = pick(p, x > y);`

`Pair r = p;`

`std::swap(p.first, p.second);` — `std::swap` is one move construction and two move assignments through a temporary, which then dies.

`r = std::move(p);` — the generated move assignment moves the members in declaration order.

`q = Probe(z);`

Print `p=(<first>,<second>) q=<value> r=(<first>,<second>)` using the `value` members.

Everything dies in reverse order of construction.

Input: three integers. **Output:** the trace, the values line, and five destroy lines.

```
1 3 5
```

prints

```
construct #1=1
construct #2=3
copy #3 from #2
copy #4 from #1
copy #5 from #2
move #6 from #1
move-assign #1 from #2
move-assign #2 from #6
destroy #6
move-assign #4 from #1
move-assign #5 from #2
construct #7=5
move-assign #3 from #7
destroy #7
p=(0,0) q=5 r=(3,1)
destroy #5
destroy #4
destroy #3
destroy #2
destroy #1
```

INPUT 1

```
1 3 5
```

OUTPUT 1

```
construct #1=1
construct #2=3
copy #3 from #2
copy #4 from #1
copy #5 from #2
move #6 from #1
move-assign #1 from #2
move-assign #2 from #6
destroy #6
move-assign #4 from #1
move-assign #5 from #2
construct #7=5
move-assign #3 from #7
destroy #7
p=(0,0) q=5 r=(3,1)
destroy #5
destroy #4
destroy #3
destroy #2
destroy #1
```

INPUT 2

```
9 2 0
```

OUTPUT 2

```
construct #1=9
construct #2=2
copy #3 from #1
copy #4 from #1
copy #5 from #2
move #6 from #1
move-assign #1 from #2
move-assign #2 from #6
destroy #6
move-assign #4 from #1
move-assign #5 from #2
construct #7=0
move-assign #3 from #7
destroy #7
p=(0,0) q=0 r=(2,9)
destroy #5
destroy #4
destroy #3
destroy #2
destroy #1
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <utility>

struct Probe {
    static inline int next = 1;
    int id;
    int value;

    explicit Probe(int v) : id(next++), value(v) {
        std::cout << "construct #" << id << '=' << value << '\n';
    }
    Probe(const Probe& other) : id(next++), value(other.value) {
        std::cout << "copy #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(const Probe& other) {
        value = other.value;
        std::cout << "copy-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    Probe(Probe&& other) noexcept : id(next++), value(std::exchange(other.value, 0))
        std::cout << "move #" << id << " from #" << other.id << '\n';
    }
    Probe& operator=(Probe&& other) noexcept {
        value = std::exchange(other.value, 0);
        std::cout << "move-assign #" << id << " from #" << other.id << '\n';
        return *this;
    }
    ~Probe() {
        std::cout << "destroy #" << id << '\n';
    }
};

struct Pair {
    Probe first;
    Probe second;
};

Pair makePair(int x, int y) {
    return Pair{Probe(x), Probe(y)};
}

Probe pick(const Pair& p, bool first) {
    return first ? p.first : p.second;
}

int main() {
    int x, y, z;
    std::cin >> x >> y >> z;
```

```
    Pair p = makePair(x, y);
    Probe q = pick(p, x > y);
    Pair r = p;
    std::swap(p.first, p.second);
    r = std::move(p);
    q = Probe(z);

    std::cout << "p=(" << p.first.value << ',' << p.second.value << ")"
              << " q=" << q.value
              << " r=(" << r.first.value << ',' << r.second.value << ")\n";
    return 0;
}
```

EXERCISE 3

Shelves and unique_ptr

Model two bookshelves that pass books between them, with **no special member functions anywhere** — the rule of zero. `struct Book { std::string title; int pages; }` with a two-argument constructor; `class Shelf` holds a name and a `std::vector<std::unique_ptr<Book>>`, and offers:

- `void add(std::string title, int pages)` — appends a book made with `std::make_unique`;
- `std::unique_ptr<Book> take(std::size_t index)` — removes and returns the book at `index` (0-based), or a null pointer when there is no such book;
- `void give(std::unique_ptr<Book> book)` — appends a book received by value (the caller must `std::move` it in);
- `void print() const` — `<name>: books=<n> pages=<total>`, then one line `<title> (<pages>)` per book in shelf order.

The shelves are `left` and `right`. Read `k` and then `k` command lines:

- `add <shelf> <pages> <title>` — the title is the rest of the line and may contain spaces;
- `move <from> <to> <index>` — takes the book from one shelf and gives it to the other, printing `moved <title> to <to>`, or `no book at <index>` when `take` returns null;
- `print <shelf>`.

Input: `k`, then `k` lines. **Output:** as below.

```
6
add left 412 Dune
add left 88 Emma
add right 300 The Hobbit
move left right 0
print left
print right
```

prints

```
moved Dune to right
left: books=1 pages=88
  Emma (88)
right: books=2 pages=712
  The Hobbit (300)
  Dune (412)
```

INPUT 1

```
6
add left 412 Dune
add left 88 Emma
add right 300 The Hobbit
move left right 0
print left
print right
```

OUTPUT 1

```
moved Dune to right
left: books=1 pages=88
  Emma (88)
right: books=2 pages=712
  The Hobbit (300)
  Dune (412)
```

INPUT 2

```
4
add right 120 A Short Book
move right left 0
move right left 0
print left
```

OUTPUT 2

```
moved A Short Book to left
no book at 0
left: books=1 pages=120
  A Short Book (120)
```

REFERENCE SOLUTION

```
#include <iostream>
#include <memory>
#include <sstream>
#include <string>
#include <utility>
#include <vector>

struct Book {
    std::string title;
    int pages;
    Book(std::string t, int p) : title(std::move(t)), pages(p) {}
};

class Shelf {
public:
    explicit Shelf(std::string name) : name_(std::move(name)) {}

    void add(std::string title, int pages) {
        books_.push_back(std::make_unique<Book>(std::move(title), pages));
    }

    std::unique_ptr<Book> take(std::size_t index) {
        if (index ⩾ books_.size()) return nullptr;
        std::unique_ptr<Book> book = std::move(books_[index]);
        books_.erase(books_.begin() + static_cast<std::ptrdiff_t>(index));
        return book;
    }

    void give(std::unique_ptr<Book> book) {
        books_.push_back(std::move(book));
    }

    void print() const {
        long long total = 0;
        for (const auto& book : books_) total += book->pages;
        std::cout << name_ << ": books=" << books_.size() << " pages=" << total << '
        for (const auto& book : books_) {
            std::cout << "  " << book->title << " (" << book->pages << ")\n";
        }
    }

private:
    std::string name_;
    std::vector<std::unique_ptr<Book>> books_;
};

int main() {
    std::size_t k;
    std::cin >> k;
```

```
    std::string rest;
    std::getline(std::cin, rest);
    Shelf left("left");
    Shelf right("right");
    auto shelf = [&](const std::string& name) -> Shelf& { return name == "left" ? le

    for (std::size_t i = 0; i < k; ++i) {
        std::string line;
        std::getline(std::cin, line);
        std::istringstream in(line);
        std::string cmd;
        in >> cmd;
        if (cmd == "add") {
            std::string name;
            int pages;
            in >> name >> pages >> std::ws;
            std::string title;
            std::getline(in, title);
            shelf(name).add(title, pages);
        } else if (cmd == "move") {
            std::string from, to;
            std::size_t index;
            in >> from >> to >> index;
            std::unique_ptr<Book> book = shelf(from).take(index);
            if (!book) {
                std::cout << "no book at " << index << '\n';
            } else {
                std::cout << "moved " << book->title << " to " << to << '\n';
                shelf(to).give(std::move(book));
            }
        } else if (cmd == "print") {
            std::string name;
            in >> name;
            shelf(name).print();
        }
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. `Probe b = a;` and `b = a;` — which special member functions do these two lines call, in order?

A. Copy assignment, then copy assignment

B. Copy constructor, then copy assignment

C. Copy constructor, then copy constructor

D. Default constructor, then copy assignment

2. In what order does the generated copy constructor copy a class's data members?

A. Alphabetically

B. In declaration order

C. In reverse declaration order

D. In an order the compiler chooses per build

3. A class owns a `new[]` block through a raw pointer and declares only a destructor. What is wrong?

A. Nothing; the destructor frees the block

B. Copies are shallow: two objects own one block and the second destructor double-deletes it

C. The class cannot be constructed

D. The destructor will not be called for heap objects

4.

```
struct S {
    Probe a;
    Probe b;
};
```

When an `S` is destroyed, in which order are `a` and `b` destroyed, and when?

A. `a` then `b`, before the destructor body

B. `b` then `a`, after the destructor body

C. `a` then `b`, after the destructor body

D. Unspecified

5. Why does `IntBuffer& operator=(IntBuffer other)` in the copy-and-swap idiom take its parameter by value?

A. By-value parameters are faster than references

B. So that the copy (or, for an rvalue argument, the move) happens in the parameter before the body runs, and the old contents leave with it

C. Because `operator=` cannot take a reference

D. To avoid needing a copy constructor

6. Which statement about `std::move` is correct?

A. It transfers the contents of its argument to a new object

B. It is a cast to `T&&` and performs no move; a move happens only if a move constructor or assignment is then selected

C. It empties its argument

D. It is only valid on temporaries

7. After `std::vector<int> b = std::move(a);` and `std::string t = std::move(s);`, which is guaranteed?

A. Both `a` and `s` are empty

B. `a` is empty; `s` is valid but unspecified

C. `s` is empty; `a` is valid but unspecified

D. Neither may be used again

8. A move constructor is not marked `noexcept`. What is the practical consequence?

A. The class cannot be stored in a `std::vector`

B. `std::vector` copies the elements when it reallocates instead of moving them

C. The move constructor is never called

D. The compiler emits an error

9. Which declaration stops the compiler generating the move constructor and move assignment?

A. A user-declared default constructor

B. A user-declared destructor, copy constructor or copy assignment

C. A `static` data member

D. A `const` member function

10. A class declares `T(T&&) = default;` and nothing else about copying. What is `T b = a;`?

A. A copy

B. A move

C. A compile error: declaring a move deletes the copy operations

D. A shallow copy with a warning

11. What is the rule of zero?

A. A class should have zero data members

B. A class should declare none of the five special members and let members such as `std::string`, `std::vector` and `std::unique_ptr` own their resources

C. A class should default all five special members to zero cost

D. Every constructor should zero-initialise its members

12. What is the value category of a function call that returns `Probe` by value?

A. lvalue

B. prvalue

C. xvalue

D. glvalue

13. A type has its copy and move constructors deleted. Which line still compiles?

A. `T b = a;`

B. `T b = std::move(a);`

C. `T b = T();`

D. None of them

14.

```
Probe f() {
    Probe local(1);
    return std::move(local);
}
```

What is the effect of the `std::move`?

A. It enables the move that `return local;` could not do

B. It prevents NRVO and forces a move construction that `return local;` would usually have elided entirely

C. It causes a copy

D. It has no effect either way

ANSWER KEY · CHECKPOINT — COPIES, MOVES AND THE RULE OF FIVE

1. B The first line initialises a new object (copy constructor, despite the `=`); the second replaces the value of an existing one (copy assignment).

2. B All generated memberwise operations follow declaration order; destruction is the reverse. The order is observable and fully specified.

3. B The rule of three: a class that needs a destructor for a resource needs copy operations that understand the same resource. The generated ones copy the pointer.

4. B Members are destroyed after the destructor body runs, in reverse declaration order — the mirror image of construction.

5. B The parameter is a complete, independent object built by the copy or move constructor. The body only swaps, so the old block is released by the parameter's destructor and the operator never frees before it copies.

6. B `std::move` changes the value category of an expression, nothing more. For a type with no move operations it selects the copy.

7. B The vector move constructor is specified to leave the source empty. `std::string` promises only a valid state. Both may be assigned to and reused.

8. B To keep the strong exception guarantee, `std::vector` uses `std::move_if_noexcept`, which falls back to copying for a move that might throw.

9. B Any of the classic three switches the implicit moves off — including `= default` versions. The moves are not deleted, so `std::move` silently copies.

10. C Declaring a move constructor or move assignment makes the copy constructor and copy assignment deleted; a type that only moves is presumed non-copyable.

11. B Ownership lives in members that already know how to copy, move and destroy themselves; the class then gets correct memberwise operations without writing any.

12. B A call returning by value is a pure value with no object yet — a prvalue. Returning `T&` gives an lvalue; returning `T&&` gives an xvalue.

13. C Initialising from a prvalue of the same type is guaranteed to construct the object in place since C++17, so no copy or move constructor is needed at all. The other two lines need one.

14. B `return local;` already treats the local as an rvalue when it is not elided, and is eligible for NRVO when it is. `std::move(local)` is not a name, so NRVO is off and the move is mandatory; compilers warn with `-Wpessimizing-move`.
