Module 7 of 20

Memory, ownership and RAII

Storage durations and the stack-frame picture, new/delete and the ownership contract, RAII and why C++ needs no finally, std::unique_ptr and std::shared_ptr with std::weak_ptr, and the memory-error catalogue with the sanitizers that find it.

C++ has no garbage collector, and that is not a gap to work around but the reason the language can promise what it promises: every object lives somewhere definite for a span you can point to in the source, and its destructor runs at that moment — not later, not on a background thread, not maybe. This module is where that promise is made precise. It draws the stack of frames that automatic objects live in and the heap that `new` carves blocks from, separates a container's small handle from the elements it owns, and fixes the rule the rest of the track depends on: automatic objects are destroyed at the closing brace, in reverse order, on every path out — including the exception path.

The six lessons build up from the machine to the idiom. The memory model names the four storage durations and the lifetime rules. `new` and `delete` set out the ownership contract — one owner, one `delete`, every path — and the four ways a `delete` gets lost or doubled. RAII turns that contract into a type: acquire in the constructor, release in the destructor, and the language runs the cleanup for you, which is why C++ has no `finally` and needs none. `std::unique_ptr` is that type written once for everyone, move-only so a second owner is a compile error; `std::shared_ptr` counts owners through a control block, leaks on a cycle, and needs `std::weak_ptr` for every back-pointer. The last lesson catalogues the seven memory errors and reads AddressSanitizer, UBSan and Valgrind reports, then maps each error to the rule that prevents it.

Every exercise is a whole program whose destructor messages are in a fully defined order, so lifetimes are something you predict and then watch: a tracer that shows early returns and a function-local static, a recursion whose frames announce themselves, a buffer and a linked list that own raw memory with `new[]` / `delete[]` and `new` / `delete` and leak nothing, a scope logger that keeps printing through an exception, a commit-or-rollback transaction guard, a `unique_ptr` factory with a sink and a stack of owning nodes, an album whose tracks point back through `weak_ptr` while you print `use_count()`, an observer list that prunes expired listeners, a leak detector that audits an allocation ledger, and a bounds-checked buffer. The checkpoint adds a growable stack over raw storage, a ledger of shared handles, and a task pipeline where a scope guard and a factory's `unique_ptr` both meet the exception path.

CONTENTS

1	The memory model — where objects live and when they die	14 min
2	new and delete — dynamic memory and the ownership contract	14 min
3	RAII — a resource's lifetime is an object's lifetime	15 min
4	std::unique_ptr — exclusive ownership, enforced by the type	15 min
5	std::shared_ptr and std::weak_ptr — counted ownership and its cycles	15 min
6	Memory errors and the tools that find them	15 min
✓	Checkpoint — Memory, ownership and RAII	25 min

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

LESSON 1 · 14 MIN

The memory model — where objects live and when they die

Every object in a C++ program lives somewhere for a definite span of time, and the language tells you exactly where and exactly how long: there is no garbage collector deciding later. Module 4 lesson 6 named the four storage durations; this lesson draws the machine underneath them — the stack of frames that automatic objects live in, the heap that `new` carves blocks from, the split between a container's *handle* and its *elements* — and fixes the one rule everything else in this module rests on: an object's destructor runs at a moment you can point to in the source.

A constructor and a destructor, minimally

Module 8 teaches classes in full. This module needs two pieces early, so here they are in the smallest form:

```
#include <iostream>
#include <string>

struct Tracer {
    std::string name;
    explicit Tracer(const std::string& n) : name(n) { std::cout << "ctor " <<
    ~Tracer() { std::cout << "dtor " << name << '\n'; }
};
```

`Tracer(const std::string&)` is a **constructor**: the function that runs when an object is created, here copying the argument into the member through the `: name(n)` initialiser. `~Tracer()` is the **destructor**: the function that runs, without being called by anyone, at the moment the object's lifetime ends. Printing from both makes lifetimes visible, and because C++ defines the order in which destructors run, the printed sequence is not a snapshot of one compiler's mood but a fact about the program. Every exercise in this module uses a class of this shape.

The four storage durations

Duration	Created by	Lives until	Where
automatic	a local variable or parameter	the end of its block	the stack frame
static	a global, a namespace-scope variable, a function-local `static`	the program exits	a fixed data segment
thread	`thread_local`	the thread ends (Module 17)	per-thread storage
dynamic	`new`	`delete`	the heap

Automatic and dynamic are the two the rest of the module is about. Static appears in the first exercise, because its destructor runs after `main` has returned and the ordering is worth seeing once.

The stack: frames

A call pushes a **frame**: the parameters, every local the function declares, and the address to return to. The compiler computed the frame's size when it compiled the function, so allocating it is a subtraction from the stack pointer — effectively free — and returning pops the whole frame in one step. That is why locals cost nothing and why recursion (Module 4 lesson 4) has a depth limit: the main thread's stack is 8 MB on Linux, each frame consumes some of it, and the judge allows about 100 000 modest frames before the process dies with a segmentation fault, not an exception.

The pop is also why the address of a local is worthless after the function returns. The frame is gone, its bytes will be reused by the next call, and a pointer or reference to it *dangles* (Module 6 lesson 4). Nothing in the language stops you from returning `&local`; the compiler warns, and the behaviour is undefined.

```
void demo() {
    Tracer a("a");
    {
        Tracer b("b");
    }                       // dtor b
    Tracer c("c");
}                           // dtor c, then dtor a
```

Automatic objects are destroyed at the closing brace of their block, in **reverse order of declaration**: last constructed, first destroyed. Every path out of the block counts — `return`, `break`, an exception propagating through — which is the foundation of lesson 3.

The heap: blocks

`new` asks the process's allocator for a block big enough for the object, constructs the object in it and returns its address. The block stays until `delete` is called on that address; nothing else — not the pointer going out of scope, not the function returning — frees it. The allocator keeps free lists, headers and alignment padding, so a `new` costs tens to hundreds of instructions where a local costs none, and a program that allocates millions of small objects spends its time in the allocator rather than in its own logic. The heap is also the only place for objects whose size is known only at run time, or whose lifetime must outlast the function that created them.

Lesson 2 covers `new` and `delete` in detail. The point here is the contrast: an automatic object's end is a brace you can see; a dynamic object's end is a call somebody must remember to make.

Handle and elements

The most useful picture in this module is a `std::vector`:

```
void fill() {
    std::vector<int> v;      // the handle: three pointers, automatic, in fil
    v.push_back(1);          // the elements: a heap block the handle owns
    v.push_back(2);
}                            // v's destructor frees the block
```

The variable `v` is small and lives on the stack. Its elements live in a block on the heap that `v` allocated. When `v` goes out of scope its destructor releases the block, so *you* never do. `std::string` is the same shape, with the small-string optimisation keeping short text inside the handle (Module 5 lesson 1); `std::map` is a handle plus a tree of heap nodes. Copying the handle — `auto w = v;` — allocates a second block and copies the elements: value semantics, and Module 9 shows how it is implemented.

This picture explains the invalidation rule from Module 6: `int* p = &v[0]; v.push_back(3);` may move the elements to a bigger block, after which `p` points into freed memory.

Lifetime, precisely

An object's **lifetime** begins when its initialisation finishes and ends when its destructor begins. For an automatic object that is the declaration and the closing brace; for a dynamic object, the `new` expression and the `delete` expression; for a static, before first use and after `main` returns, statics destroyed in reverse order of their construction.

```
Tracer* make() {
    Tracer local("local");
    return new Tracer("heap");   // constructed here
}                                // dtor local — the heap Tracer lives on

int main() {
    Tracer* p = make();
    delete p;                    // dtor heap — only now
    return 0;
}
```

Output: `ctor local`, `ctor heap`, `dtor local`, `dtor heap`. Remove the `delete` and the last line never prints: the object is leaked, its destructor never runs, and the program cannot tell.

Function-local static

A `static` inside a function is constructed the first time control passes its declaration — once — and destroyed after `main` returns, after every automatic object in `main`, in reverse order relative to other statics. If control never reaches the declaration, it is never constructed and never destroyed. Declared after an automatic `a` in the same function, it prints `ctor a`, `ctor once` on the first call, and on every later call just `ctor a`; at exit its `dtor` line comes after `main`'s own locals have gone. The first exercise puts one beside automatics so the ordering is visible.

Pitfalls

Mistake	What happens
Returning `&local` or a local by reference	Dangling; reading it is undefined behaviour.
Holding `&v[i]` across a `push_back`	The block may move; the pointer dangles.
`new` because "the object must persist"	Usually a `std::vector` or a returned value does it; a `new` without a matching `delete` is a leak.
Expecting a `new`ed object's destructor to run	It runs only at `delete`.
`static` for a value each caller needs its own copy of	Every caller shares it.

Key takeaways

- Four storage durations: automatic (the frame), static (the program), thread, dynamic (`new` to `delete`).
- A frame is pushed on call and popped on return; addresses into it are invalid afterwards.
- Automatic objects are destroyed at the closing brace in reverse order of declaration, on every exit path.
- A container is a small handle that owns a heap block; the handle's destructor frees the elements.
- A dynamic object lives until `delete`; forgetting it leaks, and nothing warns you.

EXERCISE 1

Three ways out of a function

Make lifetimes visible. Complete `Tracer` so that its constructor prints `ctor <name>` and its destructor prints `dtor <name>`, then write `classify(int x)` exactly in this order:

- declare `Tracer a("a");`
- declare a function-local `static Tracer once("once");`
- if `x < 0`, return `"negative"`;
- declare `Tracer b("b");`

if `x == 0`, return `"zero"`;

declare `Tracer c("c")`;

return `"positive"`.

`main` (given) reads `n`, declares `Tracer m("main")`, then reads `n` integers and prints `classify(x)` for each. Do not change `main`. The output is the trace the language defines: automatics die at the return in reverse order, the static is constructed once and destroyed after `main`'s own locals.

Input: `n`, then `n` integers. **Output:** the trace lines and the `n` classification lines, in the order they happen.

```
2
-3
5
```

prints

```
ctor main
ctor a
ctor once
dtor a
negative
ctor a
ctor b
ctor c
dtor c
dtor b
dtor a
positive
dtor main
dtor once
```

INPUT 1

```
2
-3
5
```

OUTPUT 1

```
ctor main
ctor a
ctor once
dtor a
negative
ctor a
ctor b
ctor c
dtor c
dtor b
dtor a
positive
dtor main
dtor once
```

INPUT 2

```
1
0
```

OUTPUT 2

```
ctor main
ctor a
ctor once
ctor b
dtor b
dtor a
zero
dtor main
dtor once
```

REFERENCE SOLUTION

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

struct Tracer {
    std::string name;
    explicit Tracer(const std::string& n) : name(n) {
        std::cout << "ctor " << name << '\n';
    }
    ~Tracer() {
        std::cout << "dtor " << name << '\n';
    }
};

std::string classify(int x) {
    Tracer a("a");
    static Tracer once("once");
    if (x < 0) return "negative";
    Tracer b("b");
    if (x == 0) return "zero";
    Tracer c("c");
    return "positive";
}

int main() {
    int n;
    std::cin >> n;
    Tracer m("main");
    for (int i = 0; i < n; ++i) {
        int x;
        std::cin >> x;
        std::cout << classify(x) << '\n';
    }
    return 0;
}
```

EXERCISE 2

Frames that announce themselves

Trace a recursion's stack frames with an automatic object. Write `long long fact(int n)` computing `n!` recursively (`fact(0)` is 1), and inside it declare a `Frame` whose constructor prints `enter fact(<n>)` and whose destructor prints `leave fact(<n>) = <result>`. To do that, declare the local `long long result` **before** the `Frame` and give the `Frame` a reference to it: because automatics are destroyed in reverse order of declaration, the frame is destroyed while `result` is still alive, so the destructor can print the value the function is about to return.

After the recursion `main` prints `<n>! = <value>`.

Input: one integer `n`, 0 ≤ n ≤ 20. **Output:** the enter/leave trace, then the result line.

```
3
```

prints

```
enter fact(3)
enter fact(2)
enter fact(1)
enter fact(0)
leave fact(0) = 1
leave fact(1) = 1
leave fact(2) = 2
leave fact(3) = 6
3! = 6
```

INPUT 1

```
3
```

OUTPUT 1

```
enter fact(3)
enter fact(2)
enter fact(1)
enter fact(0)
leave fact(0) = 1
leave fact(1) = 1
leave fact(2) = 2
leave fact(3) = 6
3! = 6
```

INPUT 2

```
0
```

OUTPUT 2

```
enter fact(0)
leave fact(0) = 1
0! = 1
```

REFERENCE SOLUTION

```cpp
#include <iostream>

struct Frame {
    int n;
    const long long& result;
    Frame(int n_, const long long& r) : n(n_), result(r) {
        std::cout << "enter fact(" << n << ")\n";
    }
    ~Frame() {
        std::cout << "leave fact(" << n << ") = " << result << '\n';
    }
};

long long fact(int n) {
    long long result = 0;
    Frame frame(n, result);
    if (n == 0) {
        result = 1;
    } else {
        result = n * fact(n - 1);
    }
    return result;
}

int main() {
    int n;
    std::cin >> n;
    long long value = fact(n);
    std::cout << n << "! = " << value << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
struct T {
    char c;
    T(char ch) : c(ch) { std::cout << c; }
    ~T() { std::cout << '~' << c; }
};
int main() {
    T a('a');
    { T b('b'); }
    T c('c');
}
```

A. `ab~bc~c~a`

B. `abc~a~b~c`

C. `ab~bc~a~c`

D. `abc~c~b~a`

2. Inside a function, `std::vector<int> v(1000);` is declared. Where do the 1000 elements live?

A. On the stack, in the same frame as `v`

B. On the heap, in a block that `v`'s handle owns and frees in its destructor

C. In static storage, because the size is a constant

D. On the stack until the vector grows, then on the heap

3. Which statement about a function-local `static Tracer t("s");` is correct?

A. It is constructed before `main` starts, like a global

B. It is constructed the first time control passes its declaration and destroyed after `main` returns

C. It is constructed on every call and destroyed on every return

D. It is never destroyed

4. What happens here?

```
int* make() {
    int x = 7;
    return &x;
}
int main() {
    int* p = make();
    std::cout << *p;
}
```

A. Prints `7`

B. Prints `0`

C. Undefined behaviour — `p` points into a frame that no longer exists

D. Compile error

5. When does the destructor of an object created with `new` run?

A. When the pointer variable holding its address goes out of scope

B. When `delete` is applied to a pointer to it

C. Automatically at program exit

D. When the last raw pointer to it is overwritten

6. `v` is a `std::vector<int>` whose size equals its capacity. What is true of `int* p = &v[0]; v.push_back(9); std::cout << *p;`?

A. Prints the first element — `push_back` only adds at the end

B. Undefined behaviour — the elements may have moved to a new block and `p` points into the old one

C. Compile error

D. Prints `9`

ANSWER KEY · THE MEMORY MODEL — WHERE OBJECTS LIVE AND WHEN THEY DIE

1. A `b` lives only in the inner block, so `~b` prints at that block's closing brace, before `c` is even constructed. At `main`'s end the remaining automatics go in reverse order of declaration: `~c` then `~a`.

2. B The variable `v` is a small handle in the frame; the elements are a heap block it allocated. Destroying `v` frees the block, which is why you never `delete` a vector's elements. The size being a constant changes nothing about where the block is.

3. B A function-local static is initialised once, lazily, when the declaration is first reached — never at all if the function is never called — and its destructor runs during program exit, after `main`'s automatics are gone. A global is the one initialised before `main`.

4. C `x` lives in `make`'s frame, which is popped on return; `p` dangles. The compiler warns ("address of local variable returned") but compiles, and whatever prints is an accident of what reused the stack slot.

5. B A dynamic object's lifetime ends only at `delete`. Pointers going out of scope or being overwritten change nothing about the object — that is a leak, and the destructor then never runs, not even at exit.

6. B A full vector reallocates on `push_back`: a bigger block is obtained, the elements copied across and the old block freed. `p` still holds the old address. Keep an index across insertions, never a pointer or reference.

LESSON 2 · 14 MIN

new and delete — dynamic memory and the ownership contract

`new` is the one expression that creates an object nothing will destroy for you, and `delete` is the promise that somebody will. Between them is a contract — exactly one piece of code owns the object and is responsible for ending it — and every memory bug in lesson 6 is a broken version of that contract. This lesson covers the two forms of `new` and `delete`, the ways a `delete` gets skipped or doubled, and why modern code writes `new` almost never: not because it is dangerous by itself, but because the bookkeeping it demands is better done by a class.

The expressions

```
int* p = new int(42);          // one int, initialised to 42
int* q = new int;              // one int, uninitialised — reading it is unde
int* z = new int{};            // one int, value-initialised to 0
delete p;
delete q;
delete z;

double* arr = new double[n];   // n doubles, uninitialised
double* zs  = new double[n]{}; // n doubles, all 0.0
delete[] arr;                  // the array form
delete[] zs;
```

`new T(args)` allocates storage for a `T`, constructs it with `args`, and yields a `T*`. `delete p` runs the destructor of `*p`, then frees the storage. `new T[n]` constructs `n` objects in one block — `n` may be a run-time value, which is what built-in arrays could not do — and `delete[] p` destroys all `n` and frees the block. The `[]` is not decoration: the array form records the count so that every element's destructor runs, and using the wrong form is undefined behaviour, however it happens to behave today.

For class types, `new Widget(a, b)` calls the constructor and `delete` calls the destructor. For fundamental types there is no destructor, and `new int` without an initialiser leaves the bytes as they were, so `new int{}` and `new int[n]{}` are the habit worth keeping.

When there is no memory, `new` throws `std::bad_alloc` — it does not return null, unlike C's `malloc`. `new (std::nothrow) int[n]` returns `nullptr` instead, for code that must not throw. `delete nullptr` is a no-op, defined and safe.

Ownership as a contract

A raw `T*` says nothing about who must delete it. It could point at a local, an element of a vector, a string literal, or a `new`ed object; it could be one of five copies of the same address. So the ownership has to live in a rule that the code follows:

> Exactly one owner. It holds the pointer and calls `delete` — once, on every path.
>
> Everyone else borrows. A borrowed pointer is used and dropped; the borrower never deletes and never keeps it past the owner's lifetime.
>
> A transfer of ownership is explicit: a function that takes over a pointer says so in its documentation, and a function that returns a `new`ed object says the caller now owns it.

In a function-sized program the contract is easy to keep. In a program with fifty files it is kept by putting the owning pointer inside a class whose destructor does the `delete` — lesson 3 — or inside `std::unique_ptr`, which is exactly that class written once for everyone — lesson 4.

How a delete gets lost

A **leak** is a block whose owner has forgotten it. The four common shapes:

```
void a() {
    int* p = new int[1000];
    // … no delete
}                                   // p is gone; the block is not

int b(int x) {
    int* p = new int[1000];
    if (x < 0) return -1;           // the early return skips the delete
    delete[] p;
    return 0;
}

void c() {
    int* p = new int[1000];
    risky();                        // throws: the delete below never runs
    delete[] p;
}

void d() {
    int* p = new int[10];
    p = new int[20];                // the first block's only address is overw
    delete[] p;
}
```

A leaked block is not a crash. The operating system reclaims everything at exit, so a short program never notices; a server that leaks a kilobyte per request notices after a week. The leak's other cost is that the destructor never runs — a file not flushed, a lock not released — which can matter far more than the bytes.

How a delete gets doubled

```
int* p = new int(1);
int* q = p;          // two copies of one address, no rule about which owns it
delete p;
delete q;            // double delete: undefined behaviour
*p = 2;              // use after free: undefined behaviour
```

Deleting twice hands the allocator a block it already has, which typically corrupts its free lists and crashes somewhere unrelated, later. Using the object after `delete` reads storage that may already belong to another object. The idiom `delete p; p = nullptr;` protects against

the double delete *through* `p` — it does nothing for `q` — and hides the logic error that produced two owners. The real fix is one owner.

Mismatching the forms is the third undefined behaviour: `delete` on a `new[]` block, `delete[]` on a single object, or `free` on a `new`ed pointer. `malloc` and `free` are C's allocator, they call no constructors or destructors, and nothing from one family may be released with the other.

A resource class that keeps the contract

Putting the pointer in a class turns the contract into code:

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

    IntBuffer(const IntBuffer&) = delete;             // copying would make tw
    IntBuffer& operator=(const IntBuffer&) = delete;  // Module 9 defines what

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

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

The constructor acquires, the destructor releases, and the two `= delete` lines stop the compiler from generating a memberwise copy that would leave two `IntBuffer`s pointing at one block — the double delete above, written for you. Every `IntBuffer` is now correct on every path out of its scope, including early returns and exceptions, because the destructor is the one piece of cleanup the language guarantees to call. `class`, `public` / `private` and `= delete` are Module 8 and Module 9 material; read them here as "the members outsiders may use" and "this operation does not exist".

Why modern code almost never writes new

Count the `new`s in a well-written C++20 program and you find nearly none. `std::vector<T>` replaces `new T[n]` and grows as well; `std::string` replaces `new char[]`;

`std::make_unique<T>()` replaces `new T` when a single object must live on the heap; `std::map` and `std::list` allocate their nodes themselves. Each of those is a class like `IntBuffer`, tested for decades, whose destructor does the `delete` at the brace. Writing `new` and `delete` by hand is for implementing such a class, and the reason to learn them is to understand what the classes do — and to read the code, still common, that predates them.

Pitfalls

Mistake	Consequence
`delete` for a `new[]` block (or the reverse)	Undefined behaviour; some destructors never run.
`new int[n]` then reading before writing	Uninitialised read; use `new int[n]{}`.
A `return` or a `throw` between `new` and `delete`	Leak — unless the pointer is in a class.
Two raw pointers to one block, both "owning"	Double delete.
Deleting a pointer to a local or to an array element	Undefined behaviour: only what `new` returned may be deleted.
`delete p;` and continuing to read `*p`	Use after free.

Key takeaways

`new T` constructs on the heap and returns a pointer; `delete p` destroys and frees. Arrays use `new[]` / `delete[]`, and the forms must match.

Ownership is a contract: one owner deletes once; everyone else borrows and never outlives the owner.

Leaks come from forgotten, skipped (early return, exception) or overwritten deletes; double deletes come from two owners.

A class whose constructor acquires and whose destructor releases keeps the contract automatically — with copying deleted or defined.

Modern code lets `std::vector`, `std::string` and `std::make_unique` write the `new` and the `delete`.

EXERCISE 1

A buffer that frees itself

Complete `IntBuffer`, a class that owns `n` ints obtained with `new int[n]{}` and releases them with `delete[]` in its destructor, which then prints `released <n> ints`. Copying is deleted (two owners would be a double delete). `at(i)` returns a reference to element `i` and throws `std::out_of_range` when `i` is not below `size()`.

`main` reads `n` and then `n` integers into the buffer. If `n` is 0 print `empty` and return early — the destructor must still run. Otherwise print `sum=<sum> min=<min> max=<max>` (the sum in `long long`), then `reversed:` followed by the elements last to first, one space before each.

Input: `n`, then `n` integers. **Output:** two lines (or `empty`), then the destructor's line.

```
4
3 -1 7 2
```

prints

```
sum=11 min=-1 max=7
reversed: 2 7 -1 3
released 4 ints
```

INPUT 1

```
4
3 -1 7 2
```

OUTPUT 1

```
sum=11 min=-1 max=7
reversed: 2 7 -1 3
released 4 ints
```

INPUT 2

```
1
42
```

OUTPUT 2

```
sum=42 min=42 max=42
reversed: 42
released 1 ints
```

REFERENCE SOLUTION

```
#include <cstddef>
#include <iostream>
#include <stdexcept>

class IntBuffer {
public:
    explicit IntBuffer(std::size_t n) : size_(n), data_(new int[n]{}) {}
    ~IntBuffer() {
        delete[] data_;
        std::cout << "released " << size_ << " ints\n";
    }
    IntBuffer(const IntBuffer&) = delete;
    IntBuffer& operator=(const IntBuffer&) = delete;

    std::size_t size() const { return size_; }
    int& at(std::size_t i) {
        if (i >= size_) throw std::out_of_range("IntBuffer::at");
        return data_[i];
    }

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

int main() {
    std::size_t n;
    std::cin >> n;
    IntBuffer buf(n);
    for (std::size_t i = 0; i < buf.size(); ++i) std::cin >> buf.at(i);
    if (n == 0) {
        std::cout << "empty\n";
        return 0;
    }
    long long sum = 0;
    int lo = buf.at(0);
    int hi = buf.at(0);
    for (std::size_t i = 0; i < buf.size(); ++i) {
        const int x = buf.at(i);
        sum += x;
        if (x < lo) lo = x;
        if (x > hi) hi = x;
    }
    std::cout << "sum=" << sum << " min=" << lo << " max=" << hi << '\n';
    std::cout << "reversed:";
    for (std::size_t i = buf.size(); i-- > 0;) std::cout << ' ' << buf.at(i);
    std::cout << '\n';
```

```
    return 0;
}
```

EXERCISE 2

A list that frees its nodes

Build a singly linked list from raw `new` / `delete` and prove it leaks nothing. `IntList` owns `Node`s allocated with `new Node{value, nullptr}`; `push_back` appends, `remove(v)` deletes every node whose value is `v` and returns how many it deleted, `print` shows the list, and the destructor deletes every remaining node and prints `freed <count> nodes`.

Read a target value `t`, then integers until the input ends, appending each. Print the list as `list: 1 → 2 → 3` (`list: (empty)` when empty), then `removed <k> nodes` after removing every `t`, then the list again. The destructor's line comes last.

Input: `t`, then zero or more integers. **Output:** four lines.

```
2
5 2 7 2
```

prints

```
list: 5 → 2 → 7 → 2
removed 2 nodes
list: 5 → 7
freed 2 nodes
```

INPUT 1

```
2
5 2 7 2
```

OUTPUT 1

```
list: 5 → 2 → 7 → 2
removed 2 nodes
list: 5 → 7
freed 2 nodes
```

INPUT 2

```
9
1 2 3
```

OUTPUT 2

```
list: 1 → 2 → 3
removed 0 nodes
list: 1 → 2 → 3
freed 3 nodes
```

REFERENCE SOLUTION

```cpp
#include <iostream>

struct Node {
    int value;
    Node* next;
};

class IntList {
public:
    IntList() = default;
    ~IntList() {
        int freed = 0;
        Node* cur = head_;
        while (cur != nullptr) {
            Node* next = cur->next;   // read the link before the node is gone
            delete cur;
            cur = next;
            ++freed;
        }
        std::cout << "freed " << freed << " nodes\n";
    }
    IntList(const IntList&) = delete;
    IntList& operator=(const IntList&) = delete;

    void push_back(int v) {
        Node* node = new Node{v, nullptr};
        if (tail_ == nullptr) {
            head_ = node;
        } else {
            tail_->next = node;
        }
        tail_ = node;
    }

    int remove(int v) {
        int removed = 0;
        Node** link = &head_;             // the pointer that currently points at *l
        while (*link != nullptr) {
            if ((*link)->value == v) {
                Node* doomed = *link;
                *link = doomed->next;     // unlink first …
                delete doomed;            // … then free
                ++removed;
            } else {
                link = &(*link)->next;
            }
        }
        tail_ = nullptr;
        for (Node* cur = head_; cur != nullptr; cur = cur->next) tail_ = cur;
```

```
        return removed;
    }

    void print() const {
        std::cout << "list:";
        if (head_ == nullptr) {
            std::cout << " (empty)\n";
            return;
        }
        for (Node* cur = head_; cur != nullptr; cur = cur->next) {
            std::cout << ' ' << cur->value;
            if (cur->next != nullptr) std::cout << " ->";
        }
        std::cout << '\n';
    }

private:
    Node* head_ = nullptr;
    Node* tail_ = nullptr;
};

int main() {
    int target;
    std::cin >> target;
    IntList list;
    int v;
    while (std::cin >> v) list.push_back(v);
    list.print();
    std::cout << "removed " << list.remove(target) << " nodes\n";
    list.print();
    return 0;
}
```

QUIZ

1. Which allocation and release are correctly paired?

A. `int* p = new int[10];` … `delete p;`

B. `int* p = new int;` … `delete[] p;`

C. `int* p = new int[10];` … `delete[] p;`

D. `int* p = static_cast<int*>(std::malloc(40));` … `delete p;`

2. What does this do?

```
int* p = new int;
std::cout << *p;
delete p;
```

A. Prints `0` — heap memory is zeroed

B. Compile error: `p` is uninitialised

C. Undefined behaviour — `new int` without an initialiser leaves the value indeterminate

D. Prints a random number, then crashes at `delete`

3. What does `new` do when the allocator cannot supply the memory?

A. Returns `nullptr`

B. Throws `std::bad_alloc`

C. Returns a pointer to zeroed memory of a smaller size

D. Calls `std::terminate` immediately

4. When does this function leak?

```
void f(int n) {
    int* a = new int[n]{};
    if (n == 0) return;
    delete[] a;
}
```

A. Never — `new int[0]` allocates nothing

B. Only when `n == 0`: the early return skips the `delete[]`

C. Always — `new int[n]{}` cannot be released with `delete[]`

D. Only when `n > 0`

5. What is `int* p = new int(1); int* q = p; delete p; delete q;` ?

A. Fine — each pointer is released once

B. A double delete: one block freed twice, undefined behaviour

C. A leak: `q` was never allocated

D. A compile error, because `q` is not the owner

6. Why is `delete p; p = nullptr;` an incomplete fix for a double-delete bug?

A. `delete nullptr` is itself undefined behaviour

B. It only protects deletes through `p`; other copies of the address still delete the block, and the second owner is left undiagnosed

C. `nullptr` cannot be assigned to an `int*`

D. It turns the double delete into a leak

7. Which is the modern replacement for `int* a = new int[n]; … delete[] a;` when `n` is read at run time?

A. `std::vector<int> a(n);`

B. `int a[n];`

C. `std::array<int, n> a;`

D. `int* a = static_cast<int*>(std::malloc(n * sizeof(int)));`

1. C The array form must be released with `delete[]`, the single-object form with `delete`, and `malloc` memory only with `free`. Every other pairing is undefined behaviour, whatever it appears to do today.

2. C `new int` (no initialiser) does not initialise; reading the value is undefined. `new int{}` or `new int()` gives 0. The `delete` itself is fine — the bug is the read, and it need not crash or look random.

3. B Plain `new` throws `std::bad_alloc`; only the `new (std::nothrow)` form returns null. Checking a plain `new` result against `nullptr` is dead code inherited from C's `malloc`.

4. B `new int[0]` is a real allocation that must be released, and the `return` before `delete[]` skips it. A block of zero ints is still a block; putting the pointer in a class with a destructor is what makes the early return safe.

5. B `p` and `q` hold the same address; the compiler has no notion of which owns it. The second `delete` hands the allocator a block it already has. Two copies of an owning pointer is exactly the situation `unique_ptr` makes impossible.

6. B `delete nullptr` is a defined no-op, so nulling `p` does guard *that* variable — but the bug was two owners, and the other one is untouched. Fix the ownership; don't paper over the symptom.

7. A `std::vector` owns a heap block of run-time size and frees it in its destructor. `int a[n]` is a variable-length array, a GCC extension that is not C++; `std::array` needs a compile-time size; `malloc` is the C version of the same manual contract.

LESSON 3 · 15 MIN

RAII — a resource's lifetime is an object's lifetime

C++ has no `finally`, no `using` block, no `with` statement and no garbage collector, and it needs none of them, because it has something those languages lack: a destructor that runs at a moment the compiler can see. **Resource Acquisition Is Initialisation** — RAII, an awkward name for a simple rule — says: acquire the resource in a constructor, release it in the destructor, and let the object's lifetime carry the resource's. Every path out of a scope then releases it, including the paths you did not write. This lesson shows the rule, the guarantees it rests on, the resources it is used for, and why it is the idiom that makes exceptions usable at all.

The rule

A resource is anything that must be given back: heap memory, an open file, a mutex, a socket, a counter that must be decremented, a database transaction. Manual code acquires it, uses it, releases it — and must release it on every exit:

```
bool save(const std::string& path, const std::vector<int>& data) {
    FILE* f = std::fopen(path.c_str(), "w");
    if (!f) return false;
    for (int x : data) {
        if (std::fprintf(f, "%d\n", x) < 0) { std::fclose(f); return false; }
    }
    std::fclose(f);
    return true;
}
```

Two `fclose` calls for one file, and a third would be needed if anything in the loop could throw. With RAII the release is written once, in a destructor:

```
bool save(const std::string& path, const std::vector<int>& data) {
    std::ofstream out(path);            // acquires: opens the file
    if (!out) return false;
    for (int x : data) out << x << '\n';
    return static_cast<bool>(out);
}                                       // releases: out's destructor closes t
```

`std::ofstream` is an RAII class: its constructor opens, its destructor flushes and closes. The function has no cleanup code because the cleanup is a property of the type, not of the function.

What the language guarantees

Three rules make RAII work, and they are guarantees, not conventions:

A destructor runs when an automatic object's block is left — by falling off the end, by `return`, `break`, `continue` or `goto`, or by an exception propagating through.

Destruction is in reverse order of construction. In one block, the last declared object is destroyed first; a class's members are destroyed after its destructor body, in reverse declaration order (Module 9 lesson 2). Reverse order is what makes dependencies safe: a lock acquired after a file is released before it.

During stack unwinding — the walk from a `throw` to the `catch` that handles it — every automatic object in every frame being abandoned is destroyed before the handler runs. Module 15 covers exceptions; the fact that matters here is that RAII cleanup happens *even then*.

```
struct Log {
    std::string name;
    explicit Log(std::string n) : name(std::move(n)) { std::cout << "begin " <
    ~Log() { std::cout << "end " << name << '\n'; }
};

void inner() {
    Log l("inner");
    throw std::runtime_error("boom");
}

int main() {
    try {
        Log l("outer");
        inner();
    } catch (const std::runtime_error& e) {
        std::cout << "caught " << e.what() << '\n';
    }
}
```

Output: `begin outer`, `begin inner`, `end inner`, `end outer`, `caught boom`. Both `end` lines print *before* `caught`, because unwinding runs destructors on the way to the handler. The first exercise builds a logger like this and drives it from input, exception included.

A scope guard

Sometimes the cleanup is a piece of code rather than a resource with a class of its own. A **scope guard** holds a callable and runs it in its destructor:

```
#include <functional>

class ScopeExit {
public:
    explicit ScopeExit(std::function<void()> f) : action_(std::move(f)) {}
    ~ScopeExit() { action_(); }
    ScopeExit(const ScopeExit&) = delete;
    ScopeExit& operator=(const ScopeExit&) = delete;
private:
    std::function<void()> action_;
};

void process(int& depth) {
    ++depth;
    ScopeExit restore([&] { --depth; std::cout << "restored\n"; });
    // … work that may return early or throw …
}   // restored
```

The lambda captures by reference (Module 4 lesson 5) and runs at the brace whatever happened above it. Module 12 shows the template version that avoids `std::function`; the shape is the same. Copying is deleted because a copied guard would run the action twice.

One shape, many resources

Resource	RAII type	Constructor	Destructor
heap memory	`std::vector`, `std::string`, `std::unique_ptr`	allocates	frees
a file	`std::ofstream`, `std::ifstream`	opens	flushes and closes
a mutex	`std::lock_guard`, `std::scoped_lock` (Module 17)	locks	unlocks
a counter, a nesting depth, an "in progress" flag	a small class of your own	increments, sets	decrements, clears
a transaction	a commit-or-rollback guard	snapshots	rolls back unless committed
elapsed time	a scope timer	reads the clock	prints the difference

The last row never appears in a judged exercise — wall-clock output is not deterministic — but it is the everyday use: `Timer t("parse");` at the top of a function, and the duration prints at every exit.

The transaction guard

A guard that undoes work unless told the work succeeded is RAII's strongest trick:

```
class Transaction {
public:
    explicit Transaction(long long& balance) : balance_(balance), saved_(balar
    ~Transaction() { if (!committed_) balance_ = saved_; }
    void commit() { committed_ = true; }
private:
    long long& balance_;
    long long saved_;
    bool committed_ = false;
};
```

Apply the changes, and if every step succeeded call `commit()`; any early return or exception leaves `committed_` false, and the destructor restores the snapshot. The caller writes no rollback code, and cannot forget to. The second exercise builds exactly this.

Why there is no finally

`finally` attaches cleanup to a *block*. RAII attaches it to a *type*, and the difference is who has to remember. With `finally`, every function that opens a file must write the close; forget it once and the leak is in that function. With RAII, `std::ofstream` closes itself in every function that will ever use it, written once by the library author. RAII also composes: three resources are three declarations, destroyed in reverse order, where `finally` would be three nested `try` blocks. Stroustrup's argument was that `finally` would be used instead of RAII by people who did not yet know the better tool, and the committee has declined it every time it has been proposed.

The rule has two conditions. A destructor must not throw — an exception leaving a destructor during unwinding terminates the program (Module 15 lesson 2) — so a release that can fail reports through another channel, or swallows the failure. And a class that owns a resource must say what copying means: deleted, deep, or transferred (Module 9). Both are consequences of "the destructor is the cleanup": it must always succeed, and it must run exactly once per resource.

Pitfalls

Mistake	What happens
`ScopeExit([&] { … });` with no variable name	A temporary: constructed and destroyed on the same line, so the action runs immediately.
`new ScopeExit(…)`	A heap guard has no scope; the action never runs.
Copying a guard	The action runs twice (delete the copy operations).
Throwing from a destructor	`std::terminate` when it happens during unwinding.
A resource acquired outside any object "just this once"	The early return you add next month leaks it.

Key takeaways

RAII: acquire in the constructor, release in the destructor; the object's scope is the resource's lifetime.

Destructors run on every exit path, in reverse order of construction, and during exception unwinding before the handler.

`std::vector`, `std::string`, `std::ofstream`, `std::lock_guard` and `std::unique_ptr` are RAII types; a scope guard makes any cleanup one.

A commit-or-rollback guard undoes work unless `commit()` is called.

No `finally`, because the type rather than the block owns the cleanup; destructors must not throw, and owning types must define copying.

EXERCISE 1

A scope logger that survives exceptions

Write `ScopeLog`, an RAII logger: its constructor prints `begin <name>` and its destructor prints `end <name>`, each indented by two spaces per nesting depth. Drive it from input with a recursive `run(depth)` that reads commands:

- `enter <name>` — declare a `ScopeLog` for `name` at the current depth and call `run(depth + 1)` **inside the same block**, so the log ends when the nested run returns;
- `leave` — return from the current `run`;
- `say <word>` — print `word` indented by two spaces per depth;
- `fail <reason>` — `throw std::runtime_error(reason)`.

`main` (given) calls `run(0)` inside a `try` and prints `caught: <reason>` if the exception arrives. A `leave` at depth 0 ends the run; so does the end of input, and every open scope must still print its `end` line — by returning, or by unwinding. Do not write any `end` line by hand: the destructors do it.

Input: commands, one per line. **Output:** the log.

```
enter load
say reading
enter parse
say tokens
leave
enter validate
fail bad-header
leave
say never
```

prints

```
begin load
  reading
  begin parse
    tokens
  end parse
  begin validate
  end validate
end load
caught: bad-header
```

INPUT 1

```
enter load
say reading
enter parse
say tokens
leave
enter validate
fail bad-header
leave
say never
```

OUTPUT 1

```
begin load
  reading
  begin parse
    tokens
  end parse
  begin validate
  end validate
end load
caught: bad-header
```

INPUT 2

```
enter a
enter b
say deep
```

OUTPUT 2

```
begin a
  begin b
    deep
  end b
end a
```

REFERENCE SOLUTION

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

struct ScopeLog {
    std::string name;
    int depth;
    ScopeLog(const std::string& n, int d) : name(n), depth(d) {
        std::cout << std::string(2 * depth, ' ') << "begin " << name << '\n';
    }
    ~ScopeLog() {
        std::cout << std::string(2 * depth, ' ') << "end " << name << '\n';
    }
};

void run(int depth) {
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "enter") {
            std::string name;
            std::cin >> name;
            ScopeLog log(name, depth);
            run(depth + 1);
        } else if (cmd == "leave") {
            return;
        } else if (cmd == "say") {
            std::string word;
            std::cin >> word;
            std::cout << std::string(2 * depth, ' ') << word << '\n';
        } else if (cmd == "fail") {
            std::string reason;
            std::cin >> reason;
            throw std::runtime_error(reason);
        }
    }
}

int main() {
    try {
        run(0);
    } catch (const std::runtime_error& e) {
        std::cout << "caught: " << e.what() << '\n';
    }
    return 0;
}
```

EXERCISE 2

Commit or roll back

Write `Transaction`, a guard that snapshots a `long long& balance` in its constructor and restores the snapshot in its destructor unless `commit()` was called. Then process a ledger with it.

The first line is the starting balance. Each following line is a transaction: zero or more integers (deposits and withdrawals) optionally followed by `commit` or `abort`. For each line, declare a `Transaction` and apply the integers in order; if the balance ever goes below zero, `throw std::runtime_error("overdrawn")` — the starter's `catch` reports it. Call `commit()` when the `commit` word is reached. After the line has been processed and the guard has died, print one of:

- `committed: balance=<b>` — `commit` was reached;
- `aborted: balance=<b>` — the line ended with `abort`, or with no keyword at all;
- `rejected: balance=<b>` — the overdraft exception was thrown.

You must not write any rollback code: the destructor does it on every path, including the exception path.

Input: the balance, then transaction lines. **Output:** one line per transaction.

```
100
10 -5 commit
-200 commit
20 abort
-100 commit
```

prints

```
committed: balance=105
rejected: balance=105
aborted: balance=105
committed: balance=5
```

INPUT 1

```
100
10 -5 commit
-200 commit
20 abort
-100 commit
```

OUTPUT 1

```
committed: balance=105
rejected: balance=105
aborted: balance=105
committed: balance=5
```

INPUT 2

```
0
50 commit
-50 commit
-1 commit
```

OUTPUT 2

```
committed: balance=50
committed: balance=0
rejected: balance=0
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <sstream>
#include <stdexcept>
#include <string>

class Transaction {
public:
    explicit Transaction(long long& balance) : balance_(balance), saved_(balance) {}
    ~Transaction() {
        if (!committed_) balance_ = saved_;
    }
    void commit() { committed_ = true; }

private:
    long long& balance_;
    long long saved_;
    bool committed_ = false;
};

int main() {
    std::string line;
    std::getline(std::cin, line);
    long long balance = std::stoll(line);
    while (std::getline(std::cin, line)) {
        if (line.empty()) continue;
        std::istringstream ss(line);
        std::string outcome = "aborted";
        try {
            Transaction tx(balance);
            std::string token;
            while (ss >> token) {
                if (token == "commit") {
                    tx.commit();
                    outcome = "committed";
                } else if (token == "abort") {
                    outcome = "aborted";
                } else {
                    balance += std::stoll(token);
                    if (balance < 0) throw std::runtime_error("overdrawn");
                }
            }
        } catch (const std::runtime_error&) {
            outcome = "rejected";
        }
        std::cout << outcome << ": balance=" << balance << '\n';
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
struct G {
    const char* n;
    G(const char* s) : n(s) { std::cout << n; }
    ~G() { std::cout << '~' << n; }
};
void f() { G a("a"); G b("b"); throw 1; }
int main() {
    try { f(); } catch (int) { std::cout << 'c'; }
}
```

A. `ab~b~ac`

B. `abc~b~a`

C. `abc`

D. `ab~a~bc`

2. Why does C++ have no `finally`?

A. Exceptions are rare in C++, so cleanup is not needed

B. RAII attaches cleanup to the type rather than to each block, so every use of the type is covered and several resources compose without nesting

C. It was removed in C++11 because it was slow

D. Destructors cannot run during stack unwinding, so `finally` would be unsafe

3. What is wrong with this use of a scope guard?

```
ScopeExit([&] { std::cout << "done\n"; });
work();
```

A. Nothing — `done` prints after `work()` returns

B. The guard is a temporary with no name, destroyed at the end of its own statement, so `done` prints *before* `work()`

C. The lambda must be declared `mutable`

D. It does not compile: a lambda cannot be stored in a `std::function`

4. Which statement about destruction order is correct?

A. Automatic objects in a block are destroyed in the order they were declared

B. A class's members are destroyed before its destructor body runs

C. Automatic objects in a block are destroyed in reverse order of construction, and members after the destructor body in reverse declaration order

D. The order is unspecified and must not be relied on

5. Why must a destructor not throw?

A. It is a compile error

B. If it throws while another exception is unwinding the stack, the program calls `std::terminate`

C. Destructors have no return type, so they cannot signal errors

D. A throwing destructor deadlocks the allocator

6. Which of these is NOT an RAII type?

A. `std::lock_guard<std::mutex>`

B. `std::ofstream`

C. `FILE*` from `std::fopen`

D. `std::vector<int>`

ANSWER KEY · RAII — A RESOURCE'S LIFETIME IS AN OBJECT'S LIFETIME

1. A Unwinding destroys `f`'s automatics — in reverse order, `b` then `a` — before the handler runs, so `~b~a` precedes `c`. Skipping the destructors (`abc`) is what a language without RAII would do.

2. B With `finally` every function that opens a file must remember the close; with RAII the file type closes itself everywhere, once, written by the library author. Three resources are three declarations destroyed in reverse order, not three nested `try` blocks.

3. B A temporary lives until the end of the full expression that created it. Without a variable name the guard is constructed and destroyed on that line. Name it — `ScopeExit guard([&] { … });` — and it lives to the closing brace.

4. C Reverse order is guaranteed, which is what lets a later object depend on an earlier one. Members are destroyed after the body — the body can still use them — in reverse of their declaration. Nothing here is unspecified.

5. B Destructors run during unwinding; a second exception in flight at that point cannot be handled, and the runtime terminates. Destructors are implicitly `noexcept` for this reason. A release that can fail must report some other way, or swallow the failure.

6. C A `FILE*` is a raw handle: nothing closes it when it goes out of scope. The other three acquire in their constructor and release in their destructor — the lock, the file and the heap block respectively.

LESSON 4 · 15 MIN

std::unique_ptr — exclusive ownership, enforced by the type

`std::unique_ptr<T>` is the `IntBuffer` of lesson 2 written once, for every type: a small object that holds a pointer, deletes it in its destructor, and cannot be copied. That last property is the point. A raw owning pointer relies on a rule that one owner deletes; a `unique_ptr` makes a second owner a compile error. It costs nothing over the raw pointer — one word of storage with the default deleter, the same indirection — so it is the default way to hold a heap object in modern C++, and the `new`-and-remember-to-`delete` of lesson 2 becomes `std::make_unique`.

Creating and using

```
#include <memory>

auto p = std::make_unique<std::string>("hello");   // std::unique_ptr<std::str
std::cout << *p << ' ' << p->size() << '\n';       // hello 5
if (p) std::cout << "owns something\n";            // converts to bool: non-nu
```

`std::make_unique<T>(args…)` performs `new T(args…)` and wraps the result. Prefer it over `std::unique_ptr<T>(new T(…))`: it never names `new`, it cannot leak when it appears as one of several arguments to a function (an evaluation-order hazard before C++17), and `std::make_unique<T[]>(n)` value-initialises an array. `*p` and `p->` work as for a raw pointer; `p` in a condition is true when it holds an object.

When `p` goes out of scope the string is deleted. There is nothing else to write.

Move-only

```
auto a = std::make_unique<int>(1);
auto b = a;                 // error: the copy constructor is deleted
auto c = std::move(a);      // ownership transfers; a is now null
std::cout << (a ? "a owns" : "a is empty") << '\n';   // a is empty
```

The copy operations are deleted, so the only way to hand the object to another `unique_ptr` is `std::move`, which leaves the source holding `nullptr` — a defined, testable state, not a valid-but-unspecified one. Module 9 explains rvalue references and what `std::move` does in general; for `unique_ptr` the contract is simple: after `std::move(a)`, `a` is null and `c` owns.

The three escape hatches

Call	Effect	Ownership afterwards
`p.get()`	returns the raw pointer	still `p`'s; the raw pointer is a borrow
`p.release()`	returns the raw pointer and sets `p` to null	yours — you must `delete` it or hand it to another owner
`p.reset()` / `p.reset(q)`	deletes the current object, then holds `q` (or nothing)	`p` owns `q`

`get()` is for calling code that takes a `T*` and does not keep it. `release()` is rare — passing to a C API that takes ownership, or re-wrapping in a different smart pointer — and a `release()` whose result is dropped is a leak. `reset()` is how you destroy early, at a point you choose rather than at the closing brace.

Passing and returning

The parameter type is the ownership statement:

```
void inspect(const Document& d);              // lends: the caller keeps owni
void annotate(Document& d);                   // lends, may modify
void maybe_use(Document* d);                  // lends, may be null
void archive(std::unique_ptr<Document> d);    // takes ownership: call with s
void reseat(std::unique_ptr<Document>& d);    // may replace what the caller

std::unique_ptr<Document> open(const std::string& path);   // factory: the ca
```

Call a sink with `archive(std::move(doc))`; afterwards `doc` is null, and the parameter inside `archive` dies at its closing brace, taking the document with it. Lend with `inspect(*doc)` or `maybe_use(doc.get())`. Return from a factory by value — `return doc;` moves automatically, and C++17 guarantees no copy — and the caller writes `auto doc = open("a.txt");`. Passing

`const std::unique_ptr<T>&` to a function that merely reads the object is a common mistake: it forces the caller to own a `unique_ptr`, where `const T&` would accept any object at all.

Arrays and custom deleters

```
auto buf = std::make_unique<int[]>(n);        // n ints, zero-initialised; del
buf[3] = 7;                                   // operator[], no bounds check

std::unique_ptr<FILE, decltype(&std::fclose)> file(std::fopen("x.txt", "r"), &
```

The `T[]` specialisation calls `delete[]` and provides `[]` — the right tool when the size is fixed at creation and a `std::vector` would be overkill, which is rarer than it sounds; `std::vector` is usually the answer. A second template argument names a **deleter**: any callable invoked instead of `delete`, which is how a `unique_ptr` closes a `FILE*`, frees a C library handle, or returns a connection to a pool. A stateless lambda or function object as the deleter keeps the pointer one word wide; a function pointer adds a second.

Linked structures

```
struct Node {
    int value;
    std::unique_ptr<Node> next;
    Node(int v, std::unique_ptr<Node> n) : value(v), next(std::move(n)) {}
};

std::unique_ptr<Node> head;
head = std::make_unique<Node>(1, std::move(head));   // push front: the new no

std::unique_ptr<Node> old = std::move(head);         // pop front: detach the
head = std::move(old->next);                         // … take its successor …
                                                     // … and old deletes the
```

Each node owns the next, so the chain is freed by whichever `unique_ptr` holds the head. The pop is written in two steps on purpose: the tempting one-liner `head = std::move(head->next)` deletes the old node inside the assignment while still referring to a member of it, and the standard's wording for that assignment leaves the result undefined. Detach, then take.

Destroying a long chain is recursive — `head`'s destructor deletes the first node, whose `next` deletes the second, one destructor frame per node — and overflows the stack for a list of a few hundred thousand. The fix is a destructor that pops from the front in a loop until the chain is empty; the second exercise writes it.

In containers and hierarchies

`std::vector<std::unique_ptr<Shape>>` is the standard way to own a collection of polymorphic objects (Module 10): `shapes.push_back(std::make_unique<Circle>(2.0));`, and the vector's destructor deletes every shape. A `unique_ptr` member makes a class own a heap object without writing a destructor at all — the rule of zero, Module 9 lesson 4.

Pitfalls

Mistake	What happens
`std::unique_ptr<T> a(raw);` `std::unique_ptr<T> b(raw);`	Two owners of one object: double delete.
Keeping `p.get()` after `p` is destroyed or reset	Dangling raw pointer.
`p.release();` with the result discarded	Leak — nobody deletes.
`f(std::unique_ptr<T>(new T), g())` in C++14	If `g()` throws between the `new` and the wrap, a leak; `make_unique` closes the gap.
Using `a` after `std::move(a)` as if it still owned	It is null; `*a` is undefined behaviour.
`std::unique_ptr<T> p = new T;`	Does not compile: the constructor is `explicit`. Write `make_unique`.

Key takeaways

`std::unique_ptr<T>` owns one heap object, deletes it in its destructor, and costs the same as a raw pointer.

Create with `std::make_unique`; copying is a compile error; `std::move` transfers and leaves the source null.

`get()` borrows, `release()` gives up ownership to you, `reset()` deletes now.

By value = give (call with `std::move`); `T&`, `const T&` or `T*` = lend; factories return by value.

`T[]` and custom deleters cover arrays and non-memory resources; a `unique_ptr` chain needs a two-step pop and an iterative destructor.

EXERCISE 1

Factory and sink

Practise the three ways a `std::unique_ptr` moves through a program: a **factory** that returns one, a **lender** that takes `const Document&`, and a **sink** that takes the pointer by value.

`Document` holds a title and a word count; its destructor prints `destroyed <title>`. Write:

- `std::unique_ptr<Document> make_document(const std::string& title, int words)` — `std::make_unique`, print `created <title>`, return it;
- `void inspect(const Document& doc)` — print `<title> has <words> words`;
- `void archive(std::unique_ptr<Document> doc)` — print `archived <title>`; the document dies when the parameter does.

`main` reads `n` documents. For each: create it, inspect it (pass `*doc`), then if it has fewer than 100 words call `doc.reset()` and print `discarded <title>`, otherwise call `archive(std::move(doc))`. Then print `owner empty: true` or `false` (`std::boolalpha`) — after either path the owner in `main` is null. Finally print `processed <n>`.

Input: `n`, then `n` lines of `<title> <words>` (the title is one word). **Output:** five lines per document, then the count.

```
2
report 120
memo 40
```

prints

```
created report
report has 120 words
archived report
destroyed report
owner empty: true
created memo
memo has 40 words
destroyed memo
discarded memo
owner empty: true
processed 2
```

INPUT 1

```
2
report 120
memo 40
```

OUTPUT 1

```
created report
report has 120 words
archived report
destroyed report
owner empty: true
created memo
memo has 40 words
destroyed memo
discarded memo
owner empty: true
processed 2
```

INPUT 2

```
1
thesis 100
```

OUTPUT 2

```
created thesis
thesis has 100 words
archived thesis
destroyed thesis
owner empty: true
processed 1
```

REFERENCE SOLUTION

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

struct Document {
    std::string title;
    int words;
    Document(const std::string& t, int w) : title(t), words(w) {}
    ~Document() { std::cout << "destroyed " << title << '\n'; }
};

std::unique_ptr<Document> make_document(const std::string& title, int words) {
    auto doc = std::make_unique<Document>(title, words);
    std::cout << "created " << doc->title << '\n';
    return doc;
}

void inspect(const Document& doc) {
    std::cout << doc.title << " has " << doc.words << " words\n";
}

void archive(std::unique_ptr<Document> doc) {
    std::cout << "archived " << doc->title << '\n';
}

int main() {
    std::cout << std::boolalpha;
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string title;
        int words;
        std::cin >> title >> words;
        std::unique_ptr<Document> doc = make_document(title, words);
        inspect(*doc);
        if (words < 100) {
            doc.reset();
            std::cout << "discarded " << title << '\n';
        } else {
            archive(std::move(doc));
        }
        std::cout << "owner empty: " << (doc == nullptr) << '\n';
    }
    std::cout << "processed " << n << '\n';
    return 0;
}
```

EXERCISE 2

A stack that owns its nodes

Implement a stack as a chain of nodes where each node owns the next through a `std::unique_ptr<Node>`, so that no `delete` appears anywhere in the program.

Commands, one per line: `push <v>`, `pop` (print `popped <v>` or `empty`), `peek` (print `top=<v>` or `empty`), `size` (print `size=<k>`), `print` (print `stack: <top> … <bottom>` or `stack: (empty)`).

Push must be `head_ = std::make_unique<Node>(v, std::move(head_))`. Pop must be the two-step form from the lesson — detach the head into a local `unique_ptr`, then move its `next` into `head_` — and the destructor must pop in a loop rather than let the chain destroy itself recursively.

Input: commands. **Output:** one line per command other than `push`.

```
push 1
push 2
push 3
print
pop
peek
size
pop
pop
pop
print
```

prints

```
stack: 3 2 1
popped 3
top=2
size=2
popped 2
popped 1
empty
stack: (empty)
```

INPUT 1

```
push 1
push 2
push 3
print
pop
peek
size
pop
pop
pop
print
```

OUTPUT 1

```
stack: 3 2 1
popped 3
top=2
size=2
popped 2
popped 1
empty
stack: (empty)
```

INPUT 2

```
peek
pop
size
print
```

OUTPUT 2

```
empty
empty
size=0
stack: (empty)
```

REFERENCE SOLUTION

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

struct Node {
    int value;
    std::unique_ptr<Node> next;
    Node(int v, std::unique_ptr<Node> n) : value(v), next(std::move(n)) {}
};

class Stack {
public:
    ~Stack() {
        while (head_) {
            std::unique_ptr<Node> old = std::move(head_);
            head_ = std::move(old->next);
        }   // old deletes one node per iteration: no recursion, however long the ch
    }

    void push(int v) {
        head_ = std::make_unique<Node>(v, std::move(head_));
        ++size_;
    }

    bool pop(int& out) {
        if (!head_) return false;
        std::unique_ptr<Node> old = std::move(head_);   // detach
        head_ = std::move(old->next);                   // take the successor
        out = old->value;
        --size_;
        return true;                                    // old dies here and frees t
    }

    bool peek(int& out) const {
        if (!head_) return false;
        out = head_->value;
        return true;
    }

    std::size_t size() const { return size_; }

    void print() const {
        std::cout << "stack:";
        if (!head_) {
            std::cout << " (empty)\n";
            return;
        }
```

```
        for (const Node* cur = head_.get(); cur != nullptr; cur = cur->next.get()) {
            std::cout << ' ' << cur->value;
        }
        std::cout << '\n';
    }

private:
    std::unique_ptr<Node> head_;
    std::size_t size_ = 0;
};

int main() {
    Stack stack;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "push") {
            int v;
            std::cin >> v;
            stack.push(v);
        } else if (cmd == "pop") {
            int v;
            if (stack.pop(v)) std::cout << "popped " << v << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "peek") {
            int v;
            if (stack.peek(v)) std::cout << "top=" << v << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "size") {
            std::cout << "size=" << stack.size() << '\n';
        } else if (cmd == "print") {
            stack.print();
        }
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
auto a = std::make_unique<int>(5);
auto b = std::move(a);
std::cout << (a ? "a" : "-") << (b ? "b" : "-") << *b;
```

A. `ab5`

B. `-b5`

C. `a-5`

D. Undefined behaviour

2. Which parameter type says "this function takes ownership of the object"?

A. `void f(const Widget&)`

B. `void f(Widget*)`

C. `void f(std::unique_ptr<Widget>)`

D. `void f(const std::unique_ptr<Widget>&)`

3. `std::unique_ptr<int> a = std::make_unique<int>(1); auto b = a;` — what happens?

A. Compile error: the copy constructor is deleted

B. `b` becomes null and `a` keeps ownership

C. Both own the object and it is deleted twice

D. A run-time exception

4. What does `p.release()` do to a `std::unique_ptr<T> p`?

A. Deletes the object and sets `p` to null

B. Returns the raw pointer and sets `p` to null without deleting — the caller now owns it

C. Decrements a reference count

D. Returns the raw pointer while `p` keeps ownership

5. A `std::unique_ptr<Node>` heads a chain of one million nodes, each owning the next. The head goes out of scope with the default destructor. What happens?

A. Nothing special — the nodes are freed in a loop

B. Each node's destructor destroys the next: a million-deep recursion that may overflow the stack

C. A leak: only the first node is freed

D. Undefined behaviour

6. Why prefer `std::make_unique<T>(x)` over `std::unique_ptr<T>(new T(x))`?

A. It is measurably faster at run time

B. No naked `new`, and it cannot leak between the allocation and the wrap when used as one of several function arguments

C. It allows the pointer to be copied

D. `make_unique` bounds-checks the object

7. How much larger is `std::unique_ptr<T>` than a raw `T*`, with the default deleter?

A. Twice as large — it stores a deleter pointer

B. The same size

C. Larger by a control block

D. It depends on `T`

ANSWER KEY · STD::UNIQUE_PTR — EXCLUSIVE OWNERSHIP, ENFORCED BY THE TYPE

1. B `std::move` transfers ownership: `b` holds the int and `a` is null — a defined state you may test. Only dereferencing `a` afterwards would be undefined; testing it is exactly what the null state is for.

2. C A `unique_ptr` by value can only be passed with `std::move`, which is the transfer made visible at the call site. References and raw pointers lend; a `const unique_ptr&` lends awkwardly, forcing the caller to own a `unique_ptr` for no reason.

3. A Exclusive ownership is enforced at compile time — a copy would create the second owner that a raw pointer cannot prevent. Transfer needs an explicit `std::move`.

4. B `release()` gives ownership away; discarding its result is a leak. `reset()` is the one that deletes; `get()` is the one that borrows without changing ownership.

5. B The default destructor of `Node` destroys its `next`, whose destructor destroys *its* `next`, one frame per node. Write a destructor that pops from the front in a loop for long chains.

6. B Before C++17, `f(std::unique_ptr<T>(new T), g())` could evaluate `new T`, then `g()` — which throws — before the wrap, leaking the `T`. `make_unique` is one call. The run-time cost is identical.

7. B The default deleter is an empty class stored with the empty-base optimisation, so the smart pointer is one pointer wide and the indirection is identical. A function-pointer deleter would add a word; `shared_ptr` is the one with a control block.

LESSON 5 · 15 MIN

std::shared_ptr and std::weak_ptr — counted ownership and its cycles

Sometimes an object has no single owner: a cache entry used by several requests, a node in a graph reached from two parents, a callback that must outlive the code that registered it. `std::shared_ptr<T>` handles that with a **reference count**: every copy of the pointer is one owner, the count rises and falls as copies are made and destroyed, and the object is deleted when the last owner goes. It is the smart pointer beginners reach for first and experts reach for last, because counting has a cost, back-pointers need an escape from the count, and most objects that seem shared turn out to have an owner after all. This lesson covers the machinery, the cycle that leaks, `std::weak_ptr`, and when shared ownership is actually the right answer.

The control block

```
#include <memory>

auto a = std::make_shared<std::string>("shared");   // count 1
{
    auto b = a;                                      // count 2: a copy is a
    std::cout << a.use_count() << '\n';              // 2
}                                                    // b destroyed: count 1
std::cout << a.use_count() << '\n';                  // 1
a.reset();                                           // count 0: the string
```

A `shared_ptr` is two pointers wide: one to the object, one to a **control block** that holds the strong count, the weak count and the deleter. `std::make_shared<T>(args…)` allocates the object and the control block in a single allocation, which is faster and keeps them adjacent; `std::shared_ptr<T>(new T)` makes two allocations and, like its `unique_ptr` cousin, should not appear in modern code. Copying increments the strong count, destroying a copy decrements it, and when it reaches zero the object is destroyed — at that moment, deterministically, inside whichever `shared_ptr`'s destructor or `reset()` made the final decrement.

`use_count()` reports the strong count. In a single-threaded program it is exact, which is why the exercises print it; in a multithreaded one it is a snapshot. The increments themselves are

atomic (Module 17), so copying a `shared_ptr` costs a synchronised read-modify-write — many times an ordinary increment — and passing one by value through ten function calls pays that ten times for no reason.

Passing a shared_ptr

The rule from lesson 4 carries over. A function that merely uses the object takes `const T&` or `T*`; only a function that will *keep* an owner — store it in a member, a container, a callback — takes `std::shared_ptr<T>` by value, and the caller pays one increment for the copy that is then kept. Passing `const std::shared_ptr<T>&` is a half-measure: it avoids the increment but still demands that the caller own a `shared_ptr`.

The cycle that leaks

```
struct Node {
    std::shared_ptr<Node> other;
    ~Node() { std::cout << "destroyed\n"; }
};

int main() {
    auto x = std::make_shared<Node>();
    auto y = std::make_shared<Node>();
    x->other = y;        // y's count: 2
    y->other = x;        // x's count: 2
}                        // x and y go out of scope: both counts drop to 1. N
```

Each node is kept alive by the other. After `main`'s two variables die, each object still has one owner — the other object — and neither count reaches zero. Reference counting cannot collect cycles; a tracing garbage collector can, and this is the one place C++'s model is weaker. Every parent-and-children, doubly linked, observer-and-subject structure built with `shared_ptr` in both directions has this leak.

---

weak_ptr: a pointer that does not own

```cpp
struct Album;
struct Track {
    std::string title;
    std::weak_ptr<Album> album;                  // back-pointer: does not kee
};
struct Album {
    std::string title;
    std::vector<std::shared_ptr<Track>> tracks;  // the album owns its tracks
};

auto album = std::make_shared<Album>();
auto t = std::make_shared<Track>();
t->album = album;                                // weak: album.use_count() st
album->tracks.push_back(t);

if (auto a = t->album.lock()) {                  // lock(): a shared_ptr, or e
    std::cout << a->title << ' ' << a.use_count() << '\n';   // 2 while a exis
}
album.reset();                                   // the album and its tracks v
std::cout << t->album.expired() << '\n';         // 1 (true)
```

A `std::weak_ptr<T>` observes an object owned by `shared_ptr`s without counting as an owner: it holds the control block (bumping the weak count, which keeps only the block alive) and can tell whether the object still exists. `expired()` answers yes or no; `lock()` returns a `shared_ptr` that owns the object for as long as you hold it, or an empty one if the object is gone — and `lock()` is the one to use, because between an `expired()` check and the access that follows the object could die. Ownership points one way (album to tracks), observation points back (track to album), and the cycle is broken.

The same shape handles caches (the cache holds `weak_ptr`s and hands out `shared_ptr`s; entries die when unused) and observer lists (the subject holds `weak_ptr`s to listeners that may have been destroyed — the second exercise).

---

enable_shared_from_this, in brief

A member function that needs a `shared_ptr` to its own object must not write `std::shared_ptr<T>(this)` — that creates a *second* control block for the same object, and the object is deleted twice. Deriving from `std::enable_shared_from_this<T>` gives the class a `shared_from_this()` that returns a `shared_ptr` sharing the existing control block. It works

only on an object already owned by a `shared_ptr`, so never inside a constructor. Module 10 covers the inheritance it relies on.

When shared ownership is right

Rarely, and specifically:

> The object genuinely has several owners of independent lifetimes — a cached resource, a graph node with several parents, an immutable configuration snapshot handed to many threads.
>
> A callback or an asynchronous task must keep its target alive until it runs.
>
> The lifetime cannot be tied to any one scope.

"I am not sure who should own it" is not on the list; it is the smell that `shared_ptr` is being used to avoid a design decision. The default is a value; a `unique_ptr` when the object must live on the heap; a `shared_ptr` when one of the reasons above applies; a raw pointer or reference for every non-owning use in between. C++20 added `std::shared_ptr<T[]>` for shared arrays; a `std::vector` inside a `shared_ptr` is usually clearer.

Pitfalls

Mistake	What happens
`std::shared_ptr<T> a(raw);` `std::shared_ptr<T> b(raw);`	Two control blocks: double delete.
Cycles of `shared_ptr`	Leak; use `weak_ptr` for the back edge.
Dereferencing `lock()` without testing it	It may be empty; test the result.
`shared_ptr` by value everywhere	Atomic churn; pass `const T&` to code that merely uses the object.
`std::shared_ptr<T>(this)` inside a member	A second control block; use `enable_shared_from_this`.
Assuming the count makes the object thread-safe	Only the count is atomic; the object needs its own lock.

Key takeaways

- `shared_ptr` owns through a control block: copies increment, destruction decrements, zero deletes; `make_shared` allocates both at once.
- `use_count()` is exact in one thread; copying costs an atomic increment — pass `const T&` to functions that merely use the object.
- Two `shared_ptr`s pointing at each other never reach zero: a cycle leaks.
- `weak_ptr` observes without owning; `lock()` gives a temporary `shared_ptr` or an empty one; `expired()` asks whether the object is gone.
- Prefer values and `unique_ptr`; use `shared_ptr` only when several owners with independent lifetimes truly exist.

EXERCISE 1

An album and its tracks

Build the ownership graph from the lesson and print the counts it defines. An `Album` owns its tracks through `std::vector<std::shared_ptr<Track>>`; each `Track` points back with a `std::weak_ptr<Album>`. The album's destructor prints `album <title> released`.

Read the album title (a whole line), then `n`, then `n` track titles (whole lines). Create the album with `std::make_shared`, then each track with `std::make_shared`, set its back-pointer and push it into the album. Then print:

- `album <title>: <n> tracks, owners=<album.use_count()>`;
- for each track, iterating by `const auto&`: `lock()` the back-pointer and print `<track> → <album title> (track owners=<track.use_count()>, album owners=<locked.use_count()>)`;
- if there are no tracks, print `no tracks`, reset the album and stop;
- otherwise copy the first track into a `std::shared_ptr<Track> first` and print `first track owners=<first.use_count()>`;
- reset the album pointer in `main` (the destructor line prints here);
- print `first track owners=<first.use_count()>` again, then `album expired=<first.expired()>` with `std::boolalpha`.

Every number is fixed by the rules: a `weak_ptr` is not an owner, a `const auto&` loop makes no copies, and `lock()` is an owner for as long as it lives.

Input: the album title line, `n`, then `n` title lines. **Output:** as above.

```
Abbey Road
2
Come Together
Something
```

prints

```
album Abbey Road: 2 tracks, owners=1
Come Together → Abbey Road (track owners=1, album owners=2)
Something → Abbey Road (track owners=1, album owners=2)
first track owners=2
album Abbey Road released
first track owners=1
album expired=true
```

INPUT 1

```
Abbey Road
2
Come Together
Something
```

OUTPUT 1

```
album Abbey Road: 2 tracks, owners=1
Come Together → Abbey Road (track
owners=1, album owners=2)
Something → Abbey Road (track owners=1,
album owners=2)
first track owners=2
album Abbey Road released
first track owners=1
album expired=true
```

INPUT 2

```
Solo
1
Only Song
```

OUTPUT 2

```
album Solo: 1 tracks, owners=1
Only Song → Solo (track owners=1, album
owners=2)
first track owners=2
album Solo released
first track owners=1
album expired=true
```

REFERENCE SOLUTION

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

struct Album;

struct Track {
    std::string title;
    std::weak_ptr<Album> album;
};

struct Album {
    std::string title;
    std::vector<std::shared_ptr<Track>> tracks;
    ~Album() { std::cout << "album " << title << " released\n"; }
};

int main() {
    std::cout << std::boolalpha;
    std::string title;
    std::getline(std::cin, title);
    int n;
    std::cin >> n;
    std::cin.ignore();

    auto album = std::make_shared<Album>();
    album->title = title;
    for (int i = 0; i < n; ++i) {
        auto track = std::make_shared<Track>();
        std::getline(std::cin, track->title);
        track->album = album;                // weak: the album's count stays 1
        album->tracks.push_back(track);
    }                                        // the loop's track dies: the vector i

    std::cout << "album " << album->title << ": " << album->tracks.size()
              << " tracks, owners=" << album.use_count() << '\n';
    for (const auto& track : album->tracks) {
        if (auto owner = track->album.lock()) {
            std::cout << track->title << " -> " << owner->title
                      << " (track owners=" << track.use_count()
                      << ", album owners=" << owner.use_count() << ")\n";
        }
    }
    if (album->tracks.empty()) {
        std::cout << "no tracks\n";
        album.reset();
        return 0;
    }
```

```
    std::shared_ptr<Track> first = album->tracks.front();
    std::cout << "first track owners=" << first.use_count() << '\n';
    album.reset();                              // last owner: the album and its vecto
    std::cout << "first track owners=" << first.use_count() << '\n';
    std::cout << "album expired=" << first->album.expired() << '\n';
    return 0;
}
```

EXERCISE 2

Observers that may vanish

A `Subject` keeps a list of listeners it does not own — `std::vector<std::weak_ptr<Listener>>` — so that a listener that has been destroyed elsewhere simply stops receiving. Owners live in a `std::map<std::string, std::shared_ptr<Listener>>` in `main`.

Commands, one per line:

- `add <name>` — if the name exists print `exists <name>`; otherwise create the listener with `std::make_shared`, store it in the map, subscribe it to the subject and print `added <name> (owners=<use_count>)` — which must be 1, because the subject holds only a `weak_ptr`;
- `drop <name>` — erase it from the map (print `dropped <name>`), or `unknown <name>` if absent;
- `notify <word>` — for each weak pointer in subscription order, `lock()` it; a live listener prints `<name> received <word>`, a dead one is counted; then remove the expired entries with `std::erase_if` and print `delivered=<d> pruned=<p>`.

Input: commands. **Output:** one line per `add` / `drop`, and the notification lines.

```
add ann
add bob
notify hello
drop ann
notify again
add ann
notify third
```

prints

```
added ann (owners=1)
added bob (owners=1)
ann received hello
bob received hello
delivered=2 pruned=0
dropped ann
bob received again
delivered=1 pruned=1
added ann (owners=1)
bob received third
ann received third
delivered=2 pruned=0
```

INPUT 1

```
add ann
add bob
notify hello
drop ann
notify again
add ann
notify third
```

OUTPUT 1

```
added ann (owners=1)
added bob (owners=1)
ann received hello
bob received hello
delivered=2 pruned=0
dropped ann
bob received again
delivered=1 pruned=1
added ann (owners=1)
bob received third
ann received third
delivered=2 pruned=0
```

INPUT 2

```
notify empty
add x
add x
drop y
```

OUTPUT 2

```
delivered=0 pruned=0
added x (owners=1)
exists x
unknown y
```

REFERENCE SOLUTION

```cpp
#include <iostream>
#include <map>
#include <memory>
#include <string>
#include <vector>

struct Listener {
    std::string name;
};

class Subject {
public:
    void subscribe(const std::shared_ptr<Listener>& listener) {
        listeners_.push_back(listener);       // shared_ptr converts to weak_ptr: no
    }

    void notify(const std::string& message) {
        int delivered = 0;
        int pruned = 0;
        for (const auto& weak : listeners_) {
            if (auto listener = weak.lock()) {
                std::cout << listener->name << " received " << message << '\n';
                ++delivered;
            } else {
                ++pruned;
            }
        }
        std::erase_if(listeners_, [](const std::weak_ptr<Listener>& w) { return w.ex
        std::cout << "delivered=" << delivered << " pruned=" << pruned << '\n';
    }

private:
    std::vector<std::weak_ptr<Listener>> listeners_;
};

int main() {
    std::map<std::string, std::shared_ptr<Listener>> owners;
    Subject subject;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "add") {
            std::string name;
            std::cin >> name;
            if (owners.contains(name)) {
                std::cout << "exists " << name << '\n';
                continue;
            }
            auto& slot = owners[name];
            slot = std::make_shared<Listener>();
```

```
            slot->name = name;
            subject.subscribe(slot);
            std::cout << "added " << name << " (owners=" << slot.use_count() << ")\r
        } else if (cmd == "drop") {
            std::string name;
            std::cin >> name;
            if (owners.erase(name) == 1) std::cout << "dropped " << name << '\n';
            else std::cout << "unknown " << name << '\n';
        } else if (cmd == "notify") {
            std::string word;
            std::cin >> word;
            subject.notify(word);
        }
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
auto a = std::make_shared<int>(1);
std::weak_ptr<int> w = a;
auto b = a;
std::cout << a.use_count();
b.reset();
std::cout << a.use_count() << w.expired();
a.reset();
std::cout << w.expired();
```

A. `2101`

B. `3101`

C. `2100`

D. `2111`

2. Two objects hold `std::shared_ptr`s to each other: `x→other = y; y→other = x;`. After both `x` and `y` in `main` go out of scope…

A. Both objects are destroyed in reverse order

B. Neither is destroyed: each count drops from 2 to 1 and stays there — a leak

C. Only `x` is destroyed

D. The program terminates with `std::bad_weak_ptr`

3. What does `std::weak_ptr<T>::lock()` return when the object has already been destroyed?

A. It throws `std::bad_weak_ptr`

B. An empty `std::shared_ptr<T>`

C. A dangling `std::shared_ptr<T>` — undefined to use

D. A raw `nullptr` of type `T*`

4. Why is `std::make_shared<T>(…)` preferred over `std::shared_ptr<T>(new T(…))`?

A. One allocation holds both the object and the control block, and no naked `new` appears

B. Only `make_shared` objects can be observed by a `weak_ptr`

C. It makes `use_count()` exact

D. It avoids atomic operations on the count

5. Inside a member function of an object already owned by a `shared_ptr`, what is wrong with `std::shared_ptr<T> self(this);`?

A. Nothing — it shares the existing count

B. It creates a second, independent control block, so the object is deleted twice

C. It increments the count twice

D. It does not compile

6. A function only reads a `Widget` that its caller happens to own through a `std::shared_ptr`. Which signature is right?

A. `void f(std::shared_ptr<Widget>)`

B. `void f(const std::shared_ptr<Widget>&)`

C. `void f(const Widget&)`

D. `void f(std::weak_ptr<Widget>)`

7. Which part of a `std::shared_ptr` is thread-safe without further locking?

A. The pointed-to object

B. The reference count updates

C. Both the count and the object

D. Neither

ANSWER KEY · STD::SHARED_PTR AND STD::WEAK_PTR — COUNTED OWNERSHIP AND ITS CYCLES

1. A The weak pointer is not an owner, so the count is 2 with `a` and `b`, 1 after `b.reset()`, and `w` is not yet expired (`0`). After `a.reset()` the last owner is gone, the int is deleted, and `w.expired()` is `1`.

2. B Each object is still owned by the other, so neither count reaches zero. Reference counting cannot collect cycles; one direction must be a `weak_ptr`.

3. B `lock()` never throws; it yields an empty `shared_ptr` you test with `if (auto p = w.lock())`. Constructing a `shared_ptr` *directly* from an expired `weak_ptr` is the call that throws `bad_weak_ptr`.

4. A `make_shared` co-allocates object and control block — faster, better locality, exception safe. `weak_ptr`, exactness and atomics are the same either way.

5. B A `shared_ptr` built from a raw pointer knows nothing about other owners; two control blocks each delete the object once. `std::enable_shared_from_this` gives `shared_from_this()`, which reuses the existing block.

6. C Ownership is not the function's concern, so it should not appear in the signature. `const Widget&` works for any owner — stack, `unique_ptr`, `shared_ptr` — and costs no atomic increment. Take a `shared_ptr` by value only when you will keep a copy.

7. B Copying and destroying `shared_ptr`s from several threads is safe because the count is atomic. The object itself has no such protection: concurrent writes through the pointer need a mutex (Module 17).

LESSON 6 · 15 MIN

Memory errors and the tools that find them

Every memory error in C++ is the same event in different clothing: the program reads or writes storage it does not own at that moment. The compiler cannot see it, the language calls it undefined behaviour, and the optimiser assumes it never happens, so the symptom — a crash, a wrong number, a test that passes on one machine — appears somewhere far from the cause. This lesson catalogues the seven errors, shows what each looks like in source and in the report of the tool that catches it, and closes with the ownership rules from this module that make each one impossible to write.

The catalogue

Error	The code	The symptom
leak	`new` without `delete`; a `shared_ptr` cycle	memory grows; destructors never run
dangling pointer or reference	`&local` returned; a pointer into a vector after `push_back`	reads garbage or another object's value
use after free	`delete p; *p`	wrong values now, corruption later
double free	`delete p; delete p;` — two owners	an allocator crash, often much later
buffer overflow	`a[n]` on an array of `n`; `strcpy` into a short buffer	overwrites a neighbour; a security hole
uninitialised read	`int x; if (x > 0)`; `new int` read before written	differs between runs and optimisation levels
invalidated iterator	`v.erase(it)` in a loop over `v`; `it` held across `push_back`	skipped elements, crashes

Each is undefined behaviour: the standard places no requirement on what happens next. In practice the optimiser reasons from "this cannot happen" — it may delete a null check that follows a dereference, keep in a register a value that memory says has changed, or fold a

read of uninitialised storage into a constant. The judge's `-O2` build of a program with a dangling read once printed `0` and passed; the same program under another compiler printed a neighbouring variable. Neither is right; the program has no meaning.

Reading the source

```
std::vector<int> v{1, 2, 3};
int& first = v[0];
v.push_back(4);              // may reallocate: first now refers to freed stor
std::cout << first << '\n';  // use after free through a dangling reference

int* make() {
    int local = 5;
    return &local;           // dangling: the frame is gone when the caller re
}

char name[8];
std::strcpy(name, "a long name");   // stack buffer overflow: 12 bytes into 8
```

The common feature is a pointer or reference that outlived what it pointed at, or a count that was not checked. None of them is caught by the compiler, though `-Wall` reports `return &local` and `-Wextra` some uninitialised reads. Finding the rest is the job of run-time tools.

AddressSanitizer

ASan instruments every load and store with a check against a shadow map of which bytes are valid. Compile with `-fsanitize=address -g` (add `-fno-omit-frame-pointer` for better stack traces), run normally, and it aborts at the first bad access with a report naming the kind of error, the access, and where the memory came from:

```
$ g++ -std=c++20 -g -fsanitize=address main.cpp -o main && ./main
==4242==ERROR: AddressSanitizer: heap-use-after-free on address 0x60200000001(
READ of size 4 at 0x602000000010 thread T0
    #0 0x55d0c4 in main main.cpp:7
0x602000000010 is located 0 bytes inside of 12-byte region
freed by thread T0 here:
    #0 0x7f3a12 in operator delete(void*)
    #1 0x55d0a1 in std::vector<int>::push_back main.cpp:6
previously allocated by thread T0 here:
    #1 0x55d033 in main main.cpp:4
```

Three stacks: where the bad access is, where the block was freed, where it was allocated — which is the whole story of a use-after-free. ASan also catches heap and stack buffer overflows, double frees and (through the LeakSanitizer it includes on Linux) leaks, reported at exit as `Direct leak of 400 byte(s) in 1 object(s) allocated from:` with the allocating stack. The cost is around 2× run time and 2–3× memory: a development flag, never a release one.

UndefinedBehaviorSanitizer

UBSan checks for undefined behaviour that is not a memory access: signed overflow, shifts by too much, null dereference, misaligned access, out-of-range enum values, and out-of-bounds indexing into arrays whose bound the compiler knows. Compile with `-fsanitize=undefined`, usually together with ASan, and it prints one line per violation and continues:

```
main.cpp:12:15: runtime error: signed integer overflow: 2147483647 + 1 cannot
main.cpp:20:9: runtime error: index 8 out of bounds for type 'int [8]'
```

ASan and UBSan together are the standard local test configuration for any program that uses pointers; Module 19 returns to them for the wider undefined-behaviour catalogue.

Valgrind

Valgrind's `memcheck` runs the *unmodified* binary on a synthetic CPU and tracks every byte's validity and every allocation. No recompilation, so it works on code you cannot rebuild, and it catches uninitialised reads that ASan does not. The price is a 20–50× slowdown.

```
$ valgrind --leak-check=full ./main
==5151== Invalid read of size 4
==5151==    at 0x1091A3: main (main.cpp:7)
==5151==  Address 0x4dbec80 is 0 bytes inside a block of size 12 free'd
==5151== Conditional jump or move depends on uninitialised value(s)
==5151==    at 0x1091C8: main (main.cpp:15)
==5151== LEAK SUMMARY:
==5151==    definitely lost: 400 bytes in 1 blocks
==5151==    indirectly lost: 0 bytes in 0 blocks
```

"Definitely lost" is a block nothing points at any more; "indirectly lost" is a block reachable only from a lost block (the children of a leaked tree); "still reachable" is memory a global still points at when the program exits, which is usually not a bug. Compile with `-g` for line numbers and `-O0` or `-O1` so that the lines mean what they say.

Cheaper checks

`-D_GLIBCXX_ASSERTIONS` makes libstdc++'s `operator[]` on `std::vector` and `std::string` check bounds and abort with a message — nearly free, and a good default for every debug build. `-fanalyzer` (GCC) is a static analysis that finds some leaks, double frees and null dereferences at compile time. `-Wall -Wextra` remain the first line. And the judge has none of these: a submission that passes has not been shown correct, only not shown wrong.

The rules that prevent them

Each error in the table maps to a rule from the earlier lessons:

No naked `new` / `delete` outside a class that owns the result. Leaks and double frees come from manual ownership; `std::vector`, `std::string`, `std::make_unique` and `std::make_shared` do it correctly.

One owner; everyone else borrows, and a borrow never outlives the owner. Raw pointers and references are non-owning: never store one past the scope that guarantees the owner is alive.

A container may move its elements. Do not keep pointers, references or iterators into a `std::vector` or `std::string` across an insertion; keep the index instead.

Initialise everything. `int x{};`, `new int[n]{}`, member initialisers. There is no reason to declare an uninitialised variable in C++20.

Bounds come from the container. Loop to `v.size()`, use `.at()` when the index comes from outside, and use `std::array` or `std::vector` rather than a raw array and a separate count.

Break every cycle with `weak_ptr`.

Test with ASan and UBSan before you trust a passing run.

The two exercises are detectors written in well-defined code: a leak detector that reads a ledger of allocations and frees and reports leaks, double frees and invalid frees, and a bounds-checked buffer that answers `out of range` where an unchecked one would corrupt its neighbour — the tools' logic, modelled, since a real memory error cannot be judged.

Pitfalls

Mistake	Consequence
Trusting a test that passes without sanitizers	Undefined behaviour that happened to be benign on one build.
Running ASan at `-O3` without frame pointers	Stack traces lose frames; use `-O1 -g -fno-omit-frame-pointer`.
Ignoring "indirectly lost"	The leaked children of the block you did see.
Silencing a warning about a returned local's address	The bug is real.
Fixing a double free with `p = nullptr`	Hides the second owner; find it.

Key takeaways

Seven errors, one cause: accessing storage the program does not own at that moment; all are undefined behaviour the optimiser may hide.

AddressSanitizer (`-fsanitize=address -g`) reports use-after-free, overflows, double frees and leaks with three stacks: access, free, allocation.

UBSan (`-fsanitize=undefined`) adds overflow, bad shifts, null dereference and known-bound indexing; Valgrind needs no recompile and finds uninitialised reads.

`-D_GLIBCXX_ASSERTIONS` checks `[]` on standard containers almost for free.

The ownership rules prevent the errors: no naked `new`, one owner, no pointers across container growth, initialise everything, `weak_ptr` for cycles.

EXERCISE 1

A leak detector

A real leak detector watches every allocation and free; yours audits a ledger of them, since the judge cannot observe real memory errors. Read commands until the input ends:

`alloc <id> <bytes>` — a new block. If `id` is currently live, print `already allocated: <id>` and ignore the command.

`free <id>` — release the block. If `id` was never allocated print `invalid free: <id>`; if it was allocated but is already freed print `double free: <id>`.

An id may be allocated again after it has been freed; that is a new block. When the input ends, list every block still live in order of allocation as `leaked: <id> (<bytes> bytes)` — or a single `no leaks` line — followed by `summary: allocs=<a> frees=<f> leaked=<k> blocks / <total> bytes, errors=<e>`, where `allocs` and `frees` count successful commands and `errors` counts the three error kinds. Bytes are summed in `long long`.

Input: commands. **Output:** error lines as they occur, then the leak list and the summary.

```
alloc a 64
alloc b 32
free b
free b
free z
alloc c 16
```

prints

```
double free: b
invalid free: z
leaked: a (64 bytes)
leaked: c (16 bytes)
summary: allocs=3 frees=1 leaked=2 blocks / 80 bytes, errors=2
```

INPUT 1

```
alloc a 64
alloc b 32
free b
free b
free z
alloc c 16
```

OUTPUT 1

```
double free: b
invalid free: z
leaked: a (64 bytes)
leaked: c (16 bytes)
summary: allocs=3 frees=1 leaked=2
blocks / 80 bytes, errors=2
```

INPUT 2

```
alloc x 8
free x
alloc x 24
free x
```

OUTPUT 2

```
no leaks
summary: allocs=2 frees=2 leaked=0
blocks / 0 bytes, errors=0
```

REFERENCE SOLUTION

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

struct Block {
    long long bytes = 0;
    bool live = false;
    int seq = 0;      // allocation order, for the leak listing
};

int main() {
    std::map<std::string, Block> blocks;
    int allocs = 0;
    int frees = 0;
    int errors = 0;
    int seq = 0;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "alloc") {
            std::string id;
            long long bytes;
            std::cin >> id >> bytes;
            auto it = blocks.find(id);
            if (it != blocks.end() && it->second.live) {
                std::cout << "already allocated: " << id << '\n';
                ++errors;
                continue;
            }
            blocks[id] = Block{bytes, true, seq++};
            ++allocs;
        } else if (cmd == "free") {
            std::string id;
            std::cin >> id;
            auto it = blocks.find(id);
            if (it == blocks.end()) {
                std::cout << "invalid free: " << id << '\n';
                ++errors;
            } else if (!it->second.live) {
                std::cout << "double free: " << id << '\n';
                ++errors;
            } else {
                it->second.live = false;
                ++frees;
            }
        }
    }
```

```
    std::vector<std::pair<int, std::string>> leaked;   // (seq, id)
    for (const auto& entry : blocks) {
        if (entry.second.live) leaked.emplace_back(entry.second.seq, entry.first);
    }
    std::sort(leaked.begin(), leaked.end());
    long long total = 0;
    if (leaked.empty()) std::cout << "no leaks\n";
    for (const auto& entry : leaked) {
        const Block& b = blocks[entry.second];
        total += b.bytes;
        std::cout << "leaked: " << entry.second << " (" << b.bytes << " bytes)\n";
    }
    std::cout << "summary: allocs=" << allocs << " frees=" << frees
              << " leaked=" << leaked.size() << " blocks / " << total
              << " bytes, errors=" << errors << '\n';
    return 0;
}
```

EXERCISE 2

A buffer that refuses to overflow

An unchecked write past the end of a buffer silently corrupts whatever lives next to it; a checked one reports. Read the buffer size `n` and create a `std::vector<int>` of `n` zeros, then process commands:

- `set <i> <v>` — store `v` at index `i` through `.at()`;
- `get <i>` — print `buf[<i>]=<value>`;
- `sum` — print `sum=<total>` (`long long`).

Read `i` as a `long long` and index with `static_cast<std::size_t>(i)`; a negative `i` becomes a huge unsigned value, which `.at()` rejects like any other out-of-range index. Catch the `std::out_of_range` that `.at()` throws and print `out of range: <i>` with the index as it was read. Never let the exception escape `main`.

Input: `n`, then commands. **Output:** one line per `get` / `sum`, and one per rejected index.

```
5
set 0 10
set 4 20
set 5 30
get 4
get -1
sum
```

prints

```
out of range: 5
buf[4]=20
out of range: -1
sum=30
```

INPUT 1

```
5
set 0 10
set 4 20
set 5 30
get 4
get -1
sum
```

OUTPUT 1

```
out of range: 5
buf[4]=20
out of range: -1
sum=30
```

INPUT 2

```
3
set 1 -7
get 1
get 0
sum
```

OUTPUT 2

```
buf[1]=-7
buf[0]=0
sum=-7
```

REFERENCE SOLUTION

```cpp
#include <cstddef>
#include <iostream>
#include <stdexcept>
#include <string>
#include <vector>

int main() {
    std::size_t n;
    std::cin >> n;
    std::vector<int> buf(n);
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "set") {
            long long i;
            int v;
            std::cin >> i >> v;
            try {
                buf.at(static_cast<std::size_t>(i)) = v;
            } catch (const std::out_of_range&) {
                std::cout << "out of range: " << i << '\n';
            }
        } else if (cmd == "get") {
            long long i;
            std::cin >> i;
            try {
                const int value = buf.at(static_cast<std::size_t>(i));   // read fir
                std::cout << "buf[" << i << "]=" << value << '\n';
            } catch (const std::out_of_range&) {
                std::cout << "out of range: " << i << '\n';
            }
        } else if (cmd == "sum") {
            long long total = 0;
            for (int x : buf) total += x;
            std::cout << "sum=" << total << '\n';
        }
    }
    return 0;
}
```

QUIZ

1. Which tool reports a read of an uninitialised heap `int` without recompiling the program?

A. AddressSanitizer

B. UndefinedBehaviorSanitizer

C. Valgrind memcheck

D. `-Wall -Wextra`

2. An AddressSanitizer `heap-use-after-free` report shows three stack traces. Which three?

A. Where the bad access happened, where the block was freed, where it was allocated

B. Compile, link and run

C. `main`, the current thread and the signal handler

D. Only the allocation — the rest is inferred

3. Classify this:

```
std::vector<int> v{1, 2, 3};
auto it = v.begin();
v.push_back(4);
std::cout << *it;
```

A. Fine — `it` still refers to the first element

B. An invalidated iterator: `push_back` may reallocate, and `*it` is undefined behaviour

C. A leak

D. A double free

4. Which flag makes `v[i]` on a libstdc++ `std::vector` abort with a message when `i` is out of range?

A. `-O2`

B. `-D_GLIBCXX_ASSERTIONS`

C. `-Wall`

D. `-fsanitize=undefined`

5. A program reads through a dangling pointer, prints the expected value and passes the judge. What is the correct description?

A. The judge ignores memory errors, so the program is correct

B. The behaviour is undefined; the optimised binary happened to produce the expected value, and nothing about the program is established

C. The judge runs with AddressSanitizer, which repaired the read

D. A dangling read always returns the old value

6. In a Valgrind leak summary, what distinguishes "definitely lost" from "still reachable"?

A. Both are leaks of equal severity

B. Definitely lost has no pointer to it anywhere; still reachable is pointed at by something (a global, say) at exit and is usually not a bug

C. Still reachable is the more serious of the two

D. They are two names for the same category

7. Which rule prevents use-after-free through an element of a `std::vector`?

A. Always use `.at()` instead of `[]`

B. Never keep a pointer, reference or iterator into the vector across an insertion; keep an index

C. Store the elements as `std::shared_ptr`

D. Call `reserve(0)` before every loop

ANSWER KEY · MEMORY ERRORS AND THE TOOLS THAT FIND THEM

1. C Valgrind runs the unmodified binary and tracks every byte's initialisation state. ASan and UBSan are compiled in and neither tracks uninitialised heap values; `-Wall` sees only what is visible at compile time.

2. A Access, free, allocation: the complete story of a use-after-free, which is why the report usually points straight at the fix.

3. B Insertion into a `std::vector` can move every element to a new block; iterators, pointers and references into the old block dangle. Take the iterator after the insertion, or keep an index.

4. B `_GLIBCXX_ASSERTIONS` turns on the library's own bounds checks for `operator[]`, nearly free. UBSan checks arrays whose bound the compiler knows, not a vector's heap block; ASan would catch the access only if it left the allocated block.

5. B Undefined behaviour has no required outcome, and the judge's `-O2` build has no sanitizers. A passing run is not evidence; test with ASan and UBSan before trusting it.

6. B A block nothing points at can never be freed — a real leak. A block a global still references at exit was never lost; the process end reclaims it. "Indirectly lost" is the third kind: reachable only from a lost block.

7. B The danger is reallocation moving the elements; an index survives it, a pointer does not. `.at()` checks the index, not whether an old pointer is still valid.

CHECKPOINT · 25 MIN

Checkpoint — Memory, ownership and RAII

This checkpoint covers the whole module: the four storage durations and the stack-frame picture, `new` / `delete` and the ownership contract, RAII and the guarantees behind it, `std::unique_ptr`, `std::shared_ptr` with `std::weak_ptr`, and the memory-error catalogue with the tools that find it.

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:

- In what order are three automatic objects declared in one block destroyed, and does an early `return` change it?
- What is undefined about `delete` on a block that came from `new[]`?
- What runs between a `throw` and the `catch` that handles it?
- What does `std::move` leave in a `std::unique_ptr`, and can you test it?
- Why does a `weak_ptr` break a `shared_ptr` cycle, and what does `lock()` return when the object is gone?
- Which three stacks does an AddressSanitizer use-after-free report show?

The three programs are a growable stack that owns its array with `new[]` / `delete[]`, a ledger of `std::shared_ptr` handles whose `use_count` you print, and a task pipeline where a scope guard and a factory's `unique_ptr` must both behave on the exception path. Every destructor message is in a defined order — trace it on paper before you run.

EXERCISE 1

A growable stack over raw storage

Write `IntStack`, a stack that owns an `int` array obtained with `new int[capacity]`, starting at capacity 2. When a push finds the array full, allocate one of double the capacity, copy the elements across, release the old block with `delete[]`, and print `grow <old> → <new>`. The destructor releases the block and prints `released capacity <capacity>`. Copying is deleted.

Commands, one per line: `push <v>`; `pop` (print `popped <v>` or `empty`); `top` (print `top=<v>` or `empty`); `size` (print `size=<k>`); `print` (print `stack: <bottom> … <top>` or `stack: (empty)`). Popping never shrinks the array.

Input: commands. **Output:** the command output, the `grow` lines as they happen, and the destructor's line last.

```
push 1
push 2
push 3
size
print
pop
pop
top
pop
pop
```

prints

```
grow 2 → 4
size=3
stack: 1 2 3
popped 3
popped 2
top=1
popped 1
empty
released capacity 4
```

INPUT 1

```
push 1
push 2
push 3
size
print
pop
pop
top
pop
pop
```

OUTPUT 1

```
grow 2 → 4
size=3
stack: 1 2 3
popped 3
popped 2
top=1
popped 1
empty
released capacity 4
```

INPUT 2

```
top
pop
print
size
```

OUTPUT 2

```
empty
empty
stack: (empty)
size=0
released capacity 2
```

REFERENCE SOLUTION

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

class IntStack {
public:
    IntStack() : capacity_(2), size_(0), data_(new int[2]) {}
    ~IntStack() {
        delete[] data_;
        std::cout << "released capacity " << capacity_ << '\n';
    }
    IntStack(const IntStack&) = delete;
    IntStack& operator=(const IntStack&) = delete;

    void push(int v) {
        if (size_ == capacity_) grow();
        data_[size_] = v;
        ++size_;
    }
    bool pop(int& out) {
        if (size_ == 0) return false;
        --size_;
        out = data_[size_];
        return true;
    }
    bool top(int& out) const {
        if (size_ == 0) return false;
        out = data_[size_ - 1];
        return true;
    }
    std::size_t size() const { return size_; }
    void print() const {
        std::cout << "stack:";
        if (size_ == 0) {
            std::cout << " (empty)\n";
            return;
        }
        for (std::size_t i = 0; i < size_; ++i) std::cout << ' ' << data_[i];
        std::cout << '\n';
    }

private:
    void grow() {
        const std::size_t bigger = capacity_ * 2;
        int* fresh = new int[bigger];
        std::copy(data_, data_ + size_, fresh);   // copy before the old block goes
        delete[] data_;
        data_ = fresh;
```

```
        std::cout << "grow " << capacity_ << " → " << bigger << '\n';
        capacity_ = bigger;
    }

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

int main() {
    IntStack stack;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "push") {
            int v;
            std::cin >> v;
            stack.push(v);
        } else if (cmd == "pop") {
            int v;
            if (stack.pop(v)) std::cout << "popped " << v << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "top") {
            int v;
            if (stack.top(v)) std::cout << "top=" << v << '\n';
            else std::cout << "empty\n";
        } else if (cmd == "size") {
            std::cout << "size=" << stack.size() << '\n';
        } else if (cmd == "print") {
            stack.print();
        }
    }
    return 0;
}
```

EXERCISE 2

A ledger of shared handles

Named handles own resources through `std::shared_ptr`; several handles may share one resource. `Resource` has a name and a destructor that prints `freed <name>`. Keep the handles in a `std::map<std::string, std::shared_ptr<Resource>>` and process commands:

- `create <handle> <resource>` — if the handle exists print `handle <handle> in use`; otherwise `std::make_shared` the resource and print `<resource> owners=<use_count>`;
- `alias <new> <existing>` — if `existing` is unknown print `unknown handle <existing>`; if `new` is taken print `handle <new> in use`; otherwise copy the pointer and print `<resource> owners=`

`<use_count>`;

`release <handle>` — if unknown print `unknown handle <handle>`; otherwise erase it: when other owners remain print `<resource> owners=<remaining>`, and when it was the last owner the destructor's `freed` line is the only output.

When the input ends, release every remaining handle **in alphabetical order of handle name**, with exactly the output an explicit `release` would give — so every resource is freed before `main` returns and the order is defined.

Input: commands. **Output:** one line per command, then the end-of-input releases.

```
create a disk
alias b a
create c net
release a
release b
release c
```

prints

```
disk owners=1
disk owners=2
net owners=1
disk owners=1
freed disk
freed net
```

INPUT 1

```
create a disk
alias b a
create c net
release a
release b
release c
```

OUTPUT 1

```
disk owners=1
disk owners=2
net owners=1
disk owners=1
freed disk
freed net
```

INPUT 2

```
create x cache
alias y x
alias z x
release y
```

OUTPUT 2

```
cache owners=1
cache owners=2
cache owners=3
cache owners=2
cache owners=1
freed cache
```

REFERENCE SOLUTION

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

struct Resource {
    std::string name;
    explicit Resource(const std::string& n) : name(n) {}
    ~Resource() { std::cout << "freed " << name << '\n'; }
};

using Handles = std::map<std::string, std::shared_ptr<Resource>>;

void release(Handles& handles, const std::string& handle) {
    auto it = handles.find(handle);
    if (it == handles.end()) {
        std::cout << "unknown handle " << handle << '\n';
        return;
    }
    const std::string name = it->second->name;          // copy: the resource may di
    const long remaining = it->second.use_count() - 1;
    handles.erase(it);                                   // last owner: "freed <name
    if (remaining > 0) std::cout << name << " owners=" << remaining << '\n';
}

int main() {
    Handles handles;
    std::string cmd;
    while (std::cin >> cmd) {
        if (cmd == "create") {
            std::string handle, name;
            std::cin >> handle >> name;
            if (handles.contains(handle)) {
                std::cout << "handle " << handle << " in use\n";
                continue;
            }
            auto& slot = handles[handle];
            slot = std::make_shared<Resource>(name);
            std::cout << name << " owners=" << slot.use_count() << '\n';
        } else if (cmd == "alias") {
            std::string fresh, existing;
            std::cin >> fresh >> existing;
            auto it = handles.find(existing);
            if (it == handles.end()) {
                std::cout << "unknown handle " << existing << '\n';
                continue;
            }
            if (handles.contains(fresh)) {
```

```
                std::cout << "handle " << fresh << " in use\n";
                continue;
            }
            handles[fresh] = it->second;                  // one more owner
            std::cout << it->second->name << " owners=" << it->second.use_count() <<
        } else if (cmd == "release") {
            std::string handle;
            std::cin >> handle;
            release(handles, handle);
        }
    }
    while (!handles.empty()) {
        const std::string first = handles.begin()->first;   // copy: erase invalidat
        release(handles, first);
    }
    return 0;
}
```

EXERCISE 3

A pipeline on the exception path

Combine a scope guard with a `unique_ptr` factory and watch both on the exception path. `ScopeGuard` prints `start <id>` in its constructor and `cleanup <id>` in its destructor. Write:

- `std::unique_ptr<Result> run_task(const std::string& id, long long value)` — declare a `ScopeGuard` for `id`; if `value` is negative `throw std::invalid_argument("negative value for " + id)`; otherwise return a `Result` made with `std::make_unique` holding `id` and `value * value`;
- `long long store(std::unique_ptr<Result> result)` — a sink: print `stored <id> -> <square>` and return the square.

`main` reads `n` tasks. For each, inside a `try`, call `run_task` and pass the result to `store` with `std::move`, adding the square to a total; `catch` the `std::invalid_argument` and print `error: <what()>`. Finally print `results=<stored> total=<sum>`. The guard's `cleanup` line must come before both `stored` (the guard dies when `run_task` returns) and `error` (unwinding runs it before the handler).

Input: `n`, then `n` lines of `<id> <value>` (|value| ≤ 1 000 000). **Output:** three lines per task, then the summary.

```
3
a 3
b -2
c 10
```

prints

```
start a
cleanup a
stored a → 9
start b
cleanup b
error: negative value for b
start c
cleanup c
stored c → 100
results=2 total=109
```

INPUT 1

```
3
a 3
b -2
c 10
```

OUTPUT 1

```
start a
cleanup a
stored a → 9
start b
cleanup b
error: negative value for b
start c
cleanup c
stored c → 100
results=2 total=109
```

INPUT 2

```
1
zero 0
```

OUTPUT 2

```
start zero
cleanup zero
stored zero → 0
results=1 total=0
```

REFERENCE SOLUTION

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

struct Result {
    std::string id;
    long long square = 0;
};

class ScopeGuard {
public:
    explicit ScopeGuard(const std::string& id) : id_(id) {
        std::cout << "start " << id_ << '\n';
    }
    ~ScopeGuard() {
        std::cout << "cleanup " << id_ << '\n';
    }
    ScopeGuard(const ScopeGuard&) = delete;
    ScopeGuard& operator=(const ScopeGuard&) = delete;

private:
    std::string id_;
};

std::unique_ptr<Result> run_task(const std::string& id, long long value) {
    ScopeGuard guard(id);
    if (value < 0) throw std::invalid_argument("negative value for " + id);
    auto result = std::make_unique<Result>();
    result->id = id;
    result->square = value * value;
    return result;                                  // guard dies after the return
}

long long store(std::unique_ptr<Result> result) {
    std::cout << "stored " << result->id << " -> " << result->square << '\n';
    return result->square;                          // result dies at the brace
}

int main() {
    int n;
    std::cin >> n;
    int stored = 0;
    long long total = 0;
    for (int i = 0; i < n; ++i) {
        std::string id;
        long long value;
        std::cin >> id >> value;
```

```
        try {
            auto result = run_task(id, value);
            total += store(std::move(result));
            ++stored;
        } catch (const std::invalid_argument& e) {
            std::cout << "error: " << e.what() << '\n';
        }
    }
    std::cout << "results=" << stored << " total=" << total << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What is the storage duration of a function parameter?

A. Static

B. Automatic — it lives in the call's frame and dies when the call returns

C. Dynamic

D. Thread

2. What does this print?

```
struct T {
    char c;
    T(char ch) : c(ch) { std::cout << c; }
    ~T() { std::cout << '~' << c; }
};
void f(bool early) {
    T a('a');
    if (early) return;
    T b('b');
}
int main() { f(true); f(false); }
```

A. `a~aab~b~a`

B. `a~aab~a~b`

C. `ab~b~aab~b~a`

D. `a~ab~b`

3. `int* p = new int[8]; delete p;` — what is the problem?

A. None — `delete` frees any pointer from `new`

B. Mismatched forms: a `new[]` block must be released with `delete[]`; this is undefined behaviour

C. A leak of seven ints

D. A compile error

4. Which `new` returns `nullptr` instead of throwing when memory is exhausted?

A. `new int[n]`

B. `new (std::nothrow) int[n]`

C. `new int[n]{}`

D. `std::make_unique<int[]>(n)`

5. What does RAII give a function that manual acquire-and-release code cannot?

A. Faster allocation

B. The release runs on every exit path — `return`, `break`, and an exception unwinding through — without any code in the function

C. Automatic reference counting

D. Protection against uninitialised reads

6. During stack unwinding from a `throw` to its `catch`, when do the destructors of automatic objects in the abandoned frames run?

A. After the handler finishes

B. Before the handler body runs

C. Only if the handler rethrows

D. They do not run — unwinding skips destructors

7. After `auto b = std::move(a);` where `a` is a `std::unique_ptr<T>`, what is `a`?

A. Unchanged — `std::move` copies

B. Null — a defined state you can test with `if (a)`

C. Valid but unspecified

D. Destroyed; using its name is a compile error

8. A function reads a `Document` and never stores it. The caller owns it through a `std::unique_ptr<Document>`. The parameter should be…

A. `std::unique_ptr<Document>`

B. `std::unique_ptr<Document>&`

C. `const Document&`

D. `std::shared_ptr<Document>`

9. What is special about `std::unique_ptr<int[]>`?

A. It bounds-checks `operator[]`

B. It releases with `delete[]` and provides `operator[]`

C. It cannot be created with `make_unique`

D. It is the same as `std::vector<int>`

10. What does this print?

```cpp
auto a = std::make_shared<int>(7);
auto b = a;
{
    auto c = b;
    std::cout << a.use_count();
}
std::cout << a.use_count();
```

A. `32`

B. `21`

C. `33`

D. `22`

11. What is `std::weak_ptr` for?

A. A faster `shared_ptr` without atomics

B. Observing an object owned by `shared_ptr`s without keeping it alive — breaking cycles, caches, observer lists

C. A `unique_ptr` that can be copied

D. A pointer that is automatically null-checked

12. Why does a class use `std::enable_shared_from_this`?

A. To let a member function obtain a `shared_ptr` to `*this` that shares the existing control block

B. To make its constructor return a `shared_ptr`

C. To disable copying

D. To allow `weak_ptr`s to point at it

13. Which compiler flags enable AddressSanitizer with usable line numbers?

A. `-O2 -Wall`

B. `-fsanitize=address -g`

C. `-D_GLIBCXX_ASSERTIONS`

D. `-fanalyzer`

14. Which of these is NOT undefined behaviour?

A. `delete nullptr;`

B. `delete p; delete p;`

C. `delete` on a pointer from `new[]`

D. Reading `*p` after `delete p`

ANSWER KEY · CHECKPOINT — MEMORY, OWNERSHIP AND RAII

1. B Parameters are locals of the call: constructed on entry, destroyed on return, in the frame. Nothing about them survives the call, which is why returning a reference to one dangles.

2. A On the early path only `a` exists, so `a~a`. On the full path `b` is constructed after `a` and destroyed before it: `ab~b~a`. Objects declared after a `return` that is taken are never constructed.

3. B The array form records the element count and destroys each element; plain `delete` does neither. The standard makes the mismatch undefined, however it behaves on one allocator.

4. B Only the `std::nothrow` placement form returns null. Every other form, including `make_unique`, throws `std::bad_alloc`.

5. B The destructor is the one cleanup the language guarantees to call, at every exit. Manual code must repeat the release at every `return` and cannot run it at all when an exception passes through.

6. B Every frame between the throw and the handler is destroyed, its automatics in reverse order, before the handler's first statement. That is what makes RAII cleanup reliable on the exception path.

7. B `unique_ptr`'s move constructor takes the pointer and leaves the source empty. Unlike a moved-from `std::string`, the state is specified: exactly `nullptr`.

8. C Lending is `const T&`; the caller passes `*doc`. By value would demand a `std::move` and take the document away; a `unique_ptr&` is for reseating; `shared_ptr` would force a change of ownership model for a read.

9. B The array specialisation matches the array form of `new`. `std::make_unique<int[]>(n)` creates it, value-initialised. There is no bounds check, and no size, growth or iteration — which is why `std::vector` is usually the better choice.

10. A Three owners while `c` exists, two after it is destroyed at the inner brace. Every copy is an owner; every destroyed copy decrements.

11. B A `weak_ptr` holds the control block, not an ownership share. `lock()` produces a temporary owner when the object is still alive; a parent-child cycle with the back-pointer weak can reach zero and be freed.

12. A `std::shared_ptr<T>(this)` would create a second control block and a double delete; `shared_from_this()` reuses the block of the `shared_ptr` that already owns the object. It works only once such an owner exists — not in the constructor.

13. B `-fsanitize=address` instruments the binary; `-g` gives the report file and line numbers. `_GLIBCXX_ASSERTIONS` checks container indexing only; `-fanalyzer` is a compile-time analysis.

14. A Deleting a null pointer is a defined no-op — the reason `delete` needs no null check. The other three are a double free, a mismatched form and a use after free.

CodeKairo · C++ study plan · Module 7: Memory, ownership and RAII. Solve the exercises online to track your progress and earn XP.