Module 4 of 20

Functions

Declarations and definitions, the three ways to pass a parameter, overloading and default arguments, recursion with memoisation, a first look at lambdas, and scope, lifetime and linkage.

A C++ program is a set of functions calling each other, and every decision about a function — what it takes, what it returns, where it may be called from, how long the things it names stay alive — is a decision the compiler enforces and the linker checks. Get the shape of a function right and the rest of the language has somewhere to live: classes are functions with state, templates are functions with type parameters, algorithms are functions that take functions.

This module builds that shape. It starts with the vocabulary — signature, prototype, definition, the one-definition rule as it applies to functions — and the layout rules that let `main` come first. Passing parameters follows: by value, by reference and by const reference, the cost model that decides between them, and why returning a value beats an output parameter. Overloading and default arguments come next, with the resolution ranks that explain why `f(3L)` is ambiguous and why a string literal picks a `bool` overload. Recursion covers the base case, the call stack and its limits, memoisation with a `std::vector`, and the fill and permutation shapes interviews ask for. Lambdas get their first look — captures by value and reference, `mutable`, generic parameters, a comparator for `std::sort` — and the module closes with scope, lifetime and linkage: function-local `static`, anonymous namespaces, `extern`, `inline`, and what a dangling reference is.

Every exercise is a whole program that names the functions it wants: prototypes above `main` and definitions below, an out-parameter by reference, a struct returned by value, an overload set and a defaulted parameter, a memoised recursion and a flood fill, lambdas handed to `std::sort` and `std::count_if`, a function-local `static` counter and a helper hidden in an anonymous namespace. The checkpoint's three programs — permutations, a grade report and a leaderboard — draw on all of it.

CONTENTS

1	Declarations and definitions — the shape of a function	12 min
2	Passing parameters — by value, by reference, by const reference	14 min
3	Overloading and default arguments	13 min

4	Recursion — base cases, the call stack and memoisation	15 min
5	Lambdas — a first look	14 min
6	Scope, lifetime and linkage	14 min
✓	Checkpoint — Functions	25 min

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

LESSON 1 · 12 MIN

Declarations and definitions — the shape of a function

A C++ program is a set of functions calling each other, and `main` is only the one the runtime calls first. This lesson settles the vocabulary (signature, parameter, argument, prototype), the difference between *declaring* a function and *defining* it, why the order of functions in a file matters and how a forward declaration lifts the constraint, what `void` and the implicit `return 0` of `main` mean, and why small functions are the unit that testing, reuse and reading are built on.

Anatomy of a function

```
long long cube(long long x) {   // head: return type, name, parameter list
    return x * x * x;           // body: statements, ending in a return
}
```

The **return type** says what the call evaluates to. The **parameters** are the local variables the call initialises from the caller's **arguments** — `cube(3)` copies `3` into `x`: the parameter is the variable in the definition, the argument is the value at the call site.

A function's **signature** is its name plus the types of its parameters — `cube(long long)`. The return type is not part of it, which is why two functions cannot differ only in what they return (Lesson 3 returns to this when overloading). Everything after the head is the **body**, one block whose statements run top to bottom until a `return` or the closing brace.

Declaration versus definition

A **declaration** tells the compiler a name exists and what type it has. A **definition** provides the thing itself. For a function the declaration is the head followed by a semicolon — a **prototype** — and the definition is the head followed by a body:

```
long long cube(long long x);   // declaration: "there is a function with this
long long cube(long long);     // the same declaration — parameter names are o

long long cube(long long x) {  // definition: the code
    return x * x * x;
}
```

A definition is also a declaration. A program may declare a function as many times as it likes, in as many translation units as it likes, but must define it exactly once across the whole program — the one-definition rule from Module 1, lesson 2. The two halves fail in different places when they are missing:

```
main.cpp:7:18: error: 'cube' was not declared in this scope
```

is the *compiler* refusing a call to a name it has not seen — a missing declaration.

```
/usr/bin/ld: main.o: in function `main': undefined reference to `cube(long lon
```

is the *linker* failing to find the code — a declaration was seen, a definition never was. The `(long long)` in the message is the signature; a definition of `cube(int)` would not satisfy it, which is the most common way a "but I did define it" turns out to be a mismatched parameter type.

Order in the file

The compiler reads a translation unit from top to bottom and must have seen a declaration of a function before the first call to it. That leaves two layouts. Bottom-up puts every helper above `main`, definitions only, so a reader meets the pieces before the whole. Top-down puts prototypes at the top and `main` first, so the file reads like a table of contents followed by the chapters:

```cpp
#include <iostream>

bool isEven(int n);      // the interface, at a glance
int halve(int n);

int main() {
    int n = 0;
    std::cin >> n;
    if (isEven(n)) std::cout << halve(n) << '\n';
    else std::cout << "odd\n";
}

bool isEven(int n) { return n % 2 == 0; }
int halve(int n) { return n / 2; }
```

The prototype form scales: a header file is nothing but a list of prototypes any source file can `#include` (Module 8, lesson 6 builds one), and two functions that call each other — `isEven` written in terms of `isOdd` and back — need a forward declaration, because whichever comes first needs the other. The exercises in this module use this layout: the reading scaffold in `main` stays at the top and the functions you write sit below it.

`void`, `return`, and the exit code of `main`

A function that produces no value has return type `void`. Its body may end without a `return`, or use a bare `return;` to leave early:

```cpp
void printBanner(const std::string& title) {
    if (title.empty()) return;          // nothing to print
    std::cout << "== " << title << " ==\n";
}
```

A non-`void` function must return a value on every path. Falling off the end of one is not a compile error — it is **undefined behaviour**, and the optimiser may assume it never happens. `-Wall` reports it as `control reaches end of non-void function`; treat that warning as an error.

`main` is the one exception: if control reaches its closing brace, the program returns `0`. That value is the process's **exit code**; zero means success, anything else failure, and the study judge marks a non-zero exit as a runtime error however correct the output looked. Write `return 0;` or leave it out — both are fine — but never `return 1;` on a path that succeeded.

`[[nodiscard]]`

Some results are the whole point of the call. A function that reports success, computes a value, or hands back an error state should not be callable as a bare statement by mistake:

```
[[nodiscard]] bool tryParse(const std::string& text, int& out);

tryParse(line, value);            // warning: ignoring return value of 'tryPar
if (tryParse(line, value)) { /* use value */ }
```

The attribute goes before the return type on the declaration and turns a silently ignored result into a warning. `std::vector::empty()` is `[[nodiscard]]` since C++20 because `v.empty();` as a statement is nearly always someone who meant `v.clear()`. Put it on predicates and pure computations; leave it off functions called for their side effect.

Functions are the unit of testing

A function with one job, whose result depends only on its parameters, can be checked in isolation: call it with a known input, compare with the known answer. That is what "unit" in unit test means, and it is why the exercises in this track name the functions they want: a `clamp(value, lo, hi)` can be tested with three calls; a `main` that does the clamping inline in a loop cannot be tested without running the whole program.

```
bool check(const std::string& name, int actual, int expected) {
    if (actual == expected) { std::cout << "PASS " << name << '\n'; return tru
    std::cout << "FAIL " << name << ": expected " << expected << " got " << ac
    return false;
}
```

Ten lines like this are a test harness. Prefer functions that are **pure** — no globals, no printing, no hidden state — because they are the ones a harness can check; keep the I/O in `main`. Name a function for what it does or answers: verbs for actions (`readGrid`), `is` / `has` for predicates (`isPrime`), nouns for computations (`mean`); a function that needs "and" in its name is two functions.

Pitfalls

Semicolon after the head of a definition. `int f(int x); { return x; }` declares `f` and then presents a stray block; the compiler reports the `return` as being outside any function.

Calling before declaring. The error names the function as "not declared in this scope" even though it is defined twenty lines lower. Move it up or add a prototype.

Definition inside a function. C++ has no nested functions; the closest thing is a lambda (Lesson 5).

Missing return. Undefined behaviour, not an error. Compile with `-Wall` and read the warnings.

Key takeaways

A signature is the name plus the parameter types; the return type is not part of it.

Declare as often as you like; define exactly once. "Not declared in this scope" is a compiler error, "undefined reference" is a linker error.

A function must be declared before the point of use — prototypes at the top let `main` come first and make mutual recursion possible.

A non-`void` function must return on every path (falling off the end is undefined behaviour); `main` alone returns `0` implicitly, and that number is the exit code.

`[[nodiscard]]` turns an ignored result into a warning; small pure functions are what unit tests check.

EXERCISE 1

Prototypes above, definitions below

Read an integer `n`, then `n` integers `x` (each between −1 000 000 and 1 000 000, one per line). For each, print `<x>: cube=<x³> prime=<yes|no>`.

Write two free functions, **declared as prototypes above `main` and defined below it** (the starter has the layout):

`long long cube(long long x);` — returns x × x × x.

`bool isPrime(long long x);` — true when x is a prime; numbers below 2 (including 0, 1 and every negative) are not prime.

Input: `n`, then `n` integers. **Output:** `n` lines.

Example: input `4`, `2`, `7`, `10`, `-3` →

```
2: cube=8 prime=yes
7: cube=343 prime=yes
10: cube=1000 prime=no
-3: cube=-27 prime=no
```

INPUT 1

```
4
2
7
10
-3
```

OUTPUT 1

```
2: cube=8 prime=yes
7: cube=343 prime=yes
10: cube=1000 prime=no
-3: cube=-27 prime=no
```

INPUT 2

```
3
1
0
97
```

OUTPUT 2

```
1: cube=1 prime=no
0: cube=0 prime=no
97: cube=912673 prime=yes
```

REFERENCE SOLUTION

```cpp
#include <iostream>

long long cube(long long x);
bool isPrime(long long x);

int main() {
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        long long x = 0;
        std::cin >> x;
        std::cout << x << ": cube=" << cube(x) << " prime=" << (isPrime(x) ? "yes" :
    }
    return 0;
}

long long cube(long long x) {
    return x * x * x;
}

bool isPrime(long long x) {
    if (x < 2) return false;
    for (long long d = 2; d * d ⩽ x; ++d) {
        if (x % d == 0) return false;
    }
    return true;
}
```

EXERCISE 2

A tiny test harness

Write a `[[nodiscard]] int clamp(int value, int lo, int hi)` that returns `lo` when `value < lo`, `hi` when `value > hi`, and `value` otherwise — and a `bool check(const std::string& name, int actual, int expected)` that prints `PASS <name>` when they are equal or `FAIL <name>: expected <expected> got <actual>` when they are not, and returns whether they matched.

Read `n`, then `n` lines `value lo hi expected`. For line `i` (counting from 1) call `check("case" + std::to_string(i), clamp(value, lo, hi), expected)` and count the results. Finally print `passed=<p> failed=<f>`.

Input: `n`, then `n` lines of four integers. **Output:** `n` PASS/FAIL lines, then the summary line.

Example: input `2`, `15 0 10 15`, `3 3 3 3` →

```
FAIL case1: expected 15 got 10
PASS case2
passed=1 failed=1
```

INPUT 1

```
3
5 0 10 5
-4 0 10 0
99 0 10 10
```

OUTPUT 1

```
PASS case1
PASS case2
PASS case3
passed=3 failed=0
```

INPUT 2

```
2
15 0 10 15
3 3 3 3
```

OUTPUT 2

```
FAIL case1: expected 15 got 10
PASS case2
passed=1 failed=1
```

REFERENCE SOLUTION

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

[[nodiscard]] int clamp(int value, int lo, int hi);
bool check(const std::string& name, int actual, int expected);

int main() {
    int n = 0;
    std::cin >> n;
    int passed = 0;
    int failed = 0;
    for (int i = 1; i <= n; ++i) {
        int value = 0, lo = 0, hi = 0, expected = 0;
        std::cin >> value >> lo >> hi >> expected;
        if (check("case" + std::to_string(i), clamp(value, lo, hi), expected)) {
            ++passed;
        } else {
            ++failed;
        }
    }
    std::cout << "passed=" << passed << " failed=" << failed << '\n';
    return 0;
}

int clamp(int value, int lo, int hi) {
    if (value < lo) return lo;
    if (value > hi) return hi;
    return value;
}

bool check(const std::string& name, int actual, int expected) {
    if (actual == expected) {
        std::cout << "PASS " << name << '\n';
        return true;
    }
    std::cout << "FAIL " << name << ": expected " << expected << " got " << actual <
    return false;
}
```

QUIZ

1. The whole program is:

```
#include <iostream>
int twice(int x);
int main() { std::cout << twice(21) << '\n'; }
```

What happens?

A. It prints 42

B. Compile error: 'twice' was not declared in this scope

C. Linker error: undefined reference to `twice(int)`

D. It compiles and links but crashes at run time

2. Which of these declares the **same** function as `int total(int count, double rate);` rather than a different one?

A. `int total(int, double);`

B. `double total(int count, double rate);`

C. `int total(double rate, int count);`

D. `int total(int count);`

3. A non-`void` function has a path on which control reaches the closing brace without a `return`. What does C++ say about that?

A. It is always a compile error

B. The function returns 0 on that path

C. It is undefined behaviour; `-Wall` warns "control reaches end of non-void function"

D. It throws an exception at run time

4. `main` finishes by reaching its closing brace with no `return` statement. What exit code does the process have, and what does the study judge make of it?

A. An unspecified value; the judge may report a runtime error

B. 0 — success; the judge is satisfied

C. 1 — failure; the judge reports a runtime error

D. The program does not compile

5. Compiled as C++20 with `-Wall`, what does the statement `v.empty();` on a `std::vector<int> v` produce?

A. Nothing — the call is valid and does nothing

B. A warning: ignoring return value declared with attribute `nodiscard`

C. A compile error

D. It clears the vector

6. `isOdd(n)` is written as `!isEven(n)` and `isEven(n)` as `n == 0 || isOdd(n - 1)`. Which layout compiles?

A. Define `isEven` above `isOdd`

B. Define `isOdd` above `isEven`

C. Put a prototype of one of them before the definition of the other

D. Mutual recursion cannot be written in C++

7. What is the strongest reason to split a long `main` into small functions?

A. The compiler generates faster code for short functions

B. Each function can be tested on its own with known inputs and expected outputs

C. It reduces the size of the executable

D. C++ limits a function to 100 lines

ANSWER KEY · DECLARATIONS AND DEFINITIONS — THE SHAPE OF A FUNCTION

1. C The prototype satisfies the compiler — the call type-checks — but no definition exists anywhere, so the **linker** cannot find the code. "Not declared in this scope" would appear only if the prototype were missing too.

2. A Parameter names are not part of the signature, so dropping them changes nothing. A different return type with the same parameters is an error (not an overload), and a different parameter order or count is a different function.

3. C The language makes it undefined behaviour rather than an error, and the optimiser may assume the path is never taken. The warning is the only safety net, which is why `-Wall` is always on in this track. Only `main` gets an implicit `return 0`.

4. B `main` is the one function with an implicit `return 0;`. The judge treats any non-zero exit as a runtime error, so a `return 1;` on a successful path would fail a case even with correct output.

5. B `empty()` is `[[nodiscard]]` since C++20 precisely because this statement is almost always a mistaken `clear()`. The attribute turns an ignored result into a warning, not an error.

6. C Whichever is defined first calls the other before it has been declared. A forward declaration (prototype) of the second function above the first breaks the cycle; the order of the definitions then does not matter.

7. B A function whose result depends only on its parameters can be checked in isolation; logic inlined in `main` can only be tested by running the whole program. Speed and size are unaffected, and there is no line limit.

LESSON 2 · 14 MIN

Passing parameters — by value, by reference, by const reference

How an argument reaches a parameter decides two things: whether the function can change the caller's object, and how much the call costs. C++ gives you the choice explicitly, and the wrong default is the most common performance mistake in beginner code — a `std::vector` of a million elements copied on every call because the parameter said `std::vector<int>` instead of `const std::vector<int>&`. This lesson settles the three ways to pass, the cost model behind the choice, why returning a value beats filling an output parameter, and how to return several things at once.

Three ways to pass

```
void byValue(std::string s);                  // s is a copy; the caller's str
void byReference(std::string& s);             // s is the caller's string unde
void byConstReference(const std::string& s);  // the caller's string, read-onl
```

By value: a copy

The parameter is a fresh object initialised from the argument. Changes inside the function are changes to the copy:

```
void bump(int n) { n += 1; }

int main() {
    int x = 5;
    bump(x);
    std::cout << x << '\n';   // 5 — bump changed its own n
}
```

This is **value semantics** — the same rule that makes `int b = a;` give `b` its own 5. For `int`, `double`, `char`, `bool` and pointers the copy is one register and costs nothing. For a `std::string` or a `std::vector` the copy allocates and copies every element, and the function pays that price on every call whether it needed a copy or not.

By reference: an alias

`T&` declares the parameter as another name for the caller's object. There is no copy; writes go straight through:

```
void swapValues(int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}

int x = 1, y = 2;
swapValues(x, y);        // x is 2, y is 1
```

A reference parameter must be given an object it can alias — an **lvalue**. `swapValues(1, 2)` does not compile: there is no variable for `a` to name. That restriction is a feature: a `T&` parameter announces "I will modify what you pass", and the compiler stops you passing something that cannot be modified.

By const reference: an alias you may only read

`const T&` is the alias without the write access. No copy, no modification, and — because nothing can be changed through it — it also binds to temporaries and literals:

```
std::size_t countVowels(const std::string& text) {   // no copy, however long
    std::size_t n = 0;
    for (char c : text) {
        if (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u') ++n;
    }
    return n;
}

countVowels(line);             // aliases line
countVowels("temporary");      // a std::string is built from the literal and
```

The cost model

Parameter form	What happens at the call	Use it for
`T`	a copy is made	`int`, `double`, `char`, `bool`, pointers, iterators, `std::string_view` — anything that fits in a register or two, or when you need your own copy anyway
`const T&`	nothing is copied; read only	everything else you only read: `std::string`, `std::vector`, structs, class objects
`T&`	nothing is copied; writes go through	the function must change the caller's object
`T*`	a pointer is copied	optional arguments (`nullptr` allowed) and C-style APIs — Module 6

The rule of thumb: **by value for small things, `const T&` for big things, `T&` only when you mean to modify.** "Small" means about the size of two pointers; `std::string_view` (Module 5, lesson 6) and `std::span` are small by design so they can go by value. Module 19, lesson 2 counts these costs precisely; for now the shape is enough: a copy of a container is O(n) work and an allocation, a reference is a single address.

Returning by value

The instinct after learning references is to avoid copies on the way out too — to take a `std::vector<int>& out` and fill it. Modern C++ does not need that:

```
std::vector<int> squares(int n) {
    std::vector<int> result;
    result.reserve(n);
    for (int i = 1; i ⩽ n; ++i) result.push_back(i * i);
    return result;            // no copy: elided or moved
}

auto v = squares(5);
```

Since C++17 a returned temporary is constructed directly in the caller's variable (guaranteed copy elision), and a returned local is at worst *moved* — its buffer handed over, not copied — through the machinery Module 9 explains. Returning by value is the right default for any type, including containers and strings.

Returning the answer beats writing it into a parameter for three reasons: the call site reads as an expression (`auto v = squares(5)` rather than `std::vector<int> v; squares(5, v);`), the result can be `const`, and the caller cannot forget to look at it. With an output parameter, input and result travel through the same door, and readers must check the body to learn which is which.

Returning more than one thing

When a function has two answers, return a **pair** or a **struct**:

```
#include <utility>

std::pair<int, int> minMax(const std::vector<int>& v) {
    int lo = v.front(), hi = v.front();
    for (int x : v) { lo = std::min(lo, x); hi = std::max(hi, x); }
    return {lo, hi};
}

struct Stats { long long sum; int min; int max; };

Stats summarise(const std::vector<int>& v) {
    Stats s{0, v.front(), v.front()};
    for (int x : v) { s.sum += x; s.min = std::min(s.min, x); s.max = std::ma
    return s;
}

auto [lo, hi] = minMax(values);          // structured bindings, C++17
Stats st = summarise(values);
std::cout << st.sum << ' ' << st.min << '\n';
```

`std::pair` is right when the two halves are of equal standing and short-lived — `first` and `second` say nothing about meaning. A named struct is right the moment a reader would have to guess which member is which; three lines of `struct` cost nothing and document the result forever.

When an output parameter is still right

A result and a success flag together. `bool tryParse(const std::string& text, int& out)` returns whether it worked and leaves the value in `out`. The modern alternative is

`std::optional<int>` (Module 15, lesson 3), but the reference form is everywhere in existing code.

Reusing a buffer. `std::getline(std::cin, line)` writes into a string the caller owns so that reading a million lines does not allocate a million strings.

In-out parameters. A function that *updates* something — `void normalise(std::vector<double>& v)` — is not returning a value at all; the reference is the point.

Outside those cases, return the value.

Pitfalls

`std::vector<int> v` where `const std::vector<int>& v` was meant. Compiles, runs, gives the right answer — and copies the container on every call, without a warning.

Forgetting the `&` on an output parameter. `void tryDivide(int a, int b, int q, int r)` assigns to its own copies; the caller's variables never change, and nothing reports it.

A non-`const` reference to a temporary. `void f(int& x); f(5);` is an error: there is nothing for `x` to alias. Either take `const int&`, or `int` by value.

Returning a reference to a local. `const std::string& f() { std::string s; return s; }` hands back a name for an object that no longer exists — Lesson 6 covers the lifetime rule.

Key takeaways

By value copies (right for small types), `T&` aliases and may write (right for in-out and output parameters), `const T&` aliases and may only read (right for everything else).

A copy of a container or string is O(n) and allocates; a reference is one address. `const T&` is the default for anything larger than a couple of pointers.

Return by value: copy elision and moves make it free, and the call site reads as an expression.

Two results: `std::pair` for equal, anonymous halves; a small `struct` whenever the members have names worth writing.

Output parameters remain right for success-plus-value, buffer reuse and genuine in-out updates.

EXERCISE 1

Divide with out-parameters

Write `bool tryDivide(int dividend, int divisor, int& quotient, int& remainder)`: when `divisor` is 0 it returns `false` and leaves both out-parameters untouched; otherwise it stores the truncating quotient and the remainder (plain `/` and `%`) through the references and returns `true`.

Read `n`, then `n` pairs `a b`. For each, print `<a> / <b> = <q> remainder <r>` or `<a> / <b>: division by zero`.

Input: `n`, then `n` lines of two integers. **Output:** `n` lines.

Example: input `3`, `17 5`, `-17 5`, `8 0` →

```
17 / 5 = 3 remainder 2
-17 / 5 = -3 remainder -2
8 / 0: division by zero
```

The two results travel back through `int&` parameters; the return value is the success flag — the pattern the lesson calls "a result and a success flag together".

INPUT 1

```
3
17 5
-17 5
8 0
```

OUTPUT 1

```
17 / 5 = 3 remainder 2
-17 / 5 = -3 remainder -2
8 / 0: division by zero
```

INPUT 2

```
2
100 10
7 -2
```

OUTPUT 2

```
100 / 10 = 10 remainder 0
7 / -2 = -3 remainder 1
```

REFERENCE SOLUTION

```cpp
#include <iostream>

bool tryDivide(int dividend, int divisor, int& quotient, int& remainder);

int main() {
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int a = 0, b = 0;
        std::cin >> a >> b;
        int q = 0, r = 0;
        if (tryDivide(a, b, q, r)) {
            std::cout << a << " / " << b << " = " << q << " remainder " << r << '\n'
        } else {
            std::cout << a << " / " << b << ": division by zero\n";
        }
    }
    return 0;
}

bool tryDivide(int dividend, int divisor, int& quotient, int& remainder) {
    if (divisor == 0) return false;
    quotient = dividend / divisor;
    remainder = dividend % divisor;
    return true;
}
```

EXERCISE 2

Summarise by const reference

Define `struct Stats { long long sum; int min; int max; };` and write two functions that take the data **by const reference** — neither may copy the vector:

- `Stats summarise(const std::vector<int>& values);` — returns the sum, the smallest and the largest.
- `std::string join(const std::vector<int>& values, const std::string& separator);` — the values as text with the separator between them.

Read `n` (at least 1), then `n` integers. Print `values: <joined with ", ">`, then `sum=<sum> min=<min> max=<max> mean=<sum ÷ n, fixed to 2 decimals>`.

Input: `n`, then `n` integers. **Output:** two lines.

Example: input `4`, `3 -1 7 2` →

```
values: 3, -1, 7, 2
sum=11 min=-1 max=7 mean=2.75
```

INPUT 1

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

OUTPUT 1

```
values: 3, -1, 7, 2
sum=11 min=-1 max=7 mean=2.75
```

INPUT 2

```
1
42
```

OUTPUT 2

```
values: 42
sum=42 min=42 max=42 mean=42.00
```

REFERENCE SOLUTION

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

struct Stats {
    long long sum;
    int min;
    int max;
};

Stats summarise(const std::vector<int>& values);
std::string join(const std::vector<int>& values, const std::string& separator);

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

    Stats s = summarise(values);
    double mean = static_cast<double>(s.sum) / static_cast<double>(n);
    std::cout << "values: " << join(values, ", ") << '\n';
    std::cout << "sum=" << s.sum << " min=" << s.min << " max=" << s.max
              << " mean=" << std::fixed << std::setprecision(2) << mean << '\n';
    return 0;
}

Stats summarise(const std::vector<int>& values) {
    Stats s{0, values.front(), values.front()};
    for (int x : values) {
        s.sum += x;
        if (x < s.min) s.min = x;
        if (x > s.max) s.max = x;
    }
    return s;
}

std::string join(const std::vector<int>& values, const std::string& separator) {
    std::string out;
    for (std::size_t i = 0; i < values.size(); ++i) {
        if (i > 0) out += separator;
        out += std::to_string(values[i]);
    }
    return out;
}
```

QUIZ

1. What does this print?

```
void grow(std::vector<int> v) { v.push_back(1); }
int main() {
    std::vector<int> a;
    grow(a);
    std::cout << a.size() << '\n';
}
```

A. 0

B. 1

C. Compile error

D. Undefined behaviour

2. What does this print?

```
void setTo(int& x, int value) { x = value; }
int main() {
    int n = 3;
    setTo(n, 9);
    setTo(n, n + 1);
    std::cout << n << '\n';
}
```

A. 3

B. 9

C. 10

D. Compile error

3. A function only *reads* a `std::string` that may be very long. Which parameter type is right?

A. `std::string s`

B. `std::string& s`

C. `const std::string& s`

D. `const std::string s`

4. `void bump(int& x) { ++x; }` is declared. What does the call `bump(5);` do?

A. Prints nothing and increments a temporary

B. Compile error: a non-const lvalue reference cannot bind to an rvalue

C. Undefined behaviour

D. Increments the literal 5 for the rest of the program

5. Why is returning a `std::vector<int>` by value from a function *not* an expensive copy in C++17 and later?

A. Vectors are always small enough to copy cheaply

B. Guaranteed copy elision constructs the result directly in the caller's variable, and a returned local is moved at worst

C. The compiler silently rewrites it to use an output parameter

D. It is expensive — output parameters should always be preferred

6. A function computes both the smallest and the largest element of a range. Which return type documents the result best?

A. `std::pair<int, int>` — `first` is the min, `second` the max

B. A `std::vector<int>` holding two elements

C. A `struct MinMax { int min; int max; };`

D. Two `int&` output parameters

7. What does this print?

```
void divide(int a, int b, int q, int r) { q = a / b; r = a % b; }
int main() {
    int q = 0, r = 0;
    divide(7, 2, q, r);
    std::cout << q << ' ' << r << '\n';
}
```

A. 3 1

B. 0 0

C. Compile error

D. Undefined behaviour

ANSWER KEY · PASSING PARAMETERS — BY VALUE, BY REFERENCE, BY CONST REFERENCE

1. A `v` is a by-value parameter — a copy of `a`. The `push_back` grows the copy, which is destroyed when `grow` returns. To modify the caller's vector the parameter would need to be `std::vector<int>&`.

2. C `x` aliases `n`, so the first call makes `n` 9. The argument `n + 1` is evaluated before the second call as 10 and copied into `value`; the assignment through `x` then sets `n` to 10.

3. C `const std::string&` neither copies nor allows modification, and it binds to literals and temporaries too. By value (with or without `const`) copies the whole string; a non-`const` reference advertises modification and rejects temporaries.

4. B A `T&` parameter needs an lvalue — a named object to alias. `5` is a temporary, and the compiler refuses: `cannot bind non-const lvalue reference of type 'int&' to an rvalue of type 'int'`. A `const int&` or a by-value `int` would accept it.

5. B A returned prvalue is built in place with no copy at all since C++17, and a returned named local hands over its buffer through a move (Module 9). Return by value is the modern default; output parameters are for success flags, buffer reuse and in-out updates.

6. C Named members say which value is which at every use site; `first` / `second` and element indices leave the reader guessing, and out-parameters force the caller to declare variables before the call. `std::pair` is fine when the halves genuinely have no distinct meaning.

7. B `q` and `r` inside `divide` are by-value copies; the assignments change the copies and nothing in `main`. The missing `&` is a silent bug — the program compiles and prints the untouched zeros.

LESSON 3 · 13 MIN

Overloading and default arguments

Two functions may share a name if their parameter lists differ; a parameter may have a value the caller can leave out. Both features exist to make call sites read naturally — `area(4)`, `area(3, 5)`, `frame("hi")` — and both have rules that surprise people who have only seen them work. This lesson settles how the compiler picks an overload (and when it refuses to), the string and pointer cases that pick the wrong one, where a default argument may be written, and when each feature is the clearer choice.

What overloading is

```
long long area(long long side);                       // a square
long long area(long long width, long long height);    // a rectangle
double area(double radius);                           // a circle
```

Three functions, one name, told apart by their **signatures** — the parameter count and types. The return type plays no part: `int f()` and `double f()` in the same scope is an error, "ambiguating new declaration", because a call `f();` could not choose between them.

Overloading exists so that one *idea* gets one name. `std::to_string` has nine overloads, `std::abs` has one per numeric type, `std::sort` has two — with and without a comparator. A good overload set does the same thing for every parameter list; a set whose members do different things is a naming problem dressed as a language feature.

How the compiler chooses

At a call, the compiler collects every function with that name that is visible (the **candidates**), discards the ones the arguments cannot be converted to (leaving the **viable** ones), and ranks the survivors by how good the conversion for each argument is:

Exact match — the argument's type, or a trivial adjustment (adding `const` to what a reference names, array to pointer).

Promotion — `char`, `short`, `bool` and small enums to `int`; `float` to `double`.

Standard conversion — any other built-in conversion: `int` to `double`, `double` to `int`, `long` to `int`, integer to `bool`, a pointer to `bool`, `nullptr` to a pointer.

User-defined conversion — a constructor or conversion operator, such as `const char*` to `std::string`.

The best viable function must be strictly better on at least one argument and no worse on the rest. If two tie, the call is **ambiguous** and does not compile — the compiler will not guess.

```
void f(int);
void f(double);

f(3);      // int: exact
f(3.0);    // double: exact
f(3.0f);   // double: float to double is a promotion, float to int a conversi
f('a');    // int: char to int is a promotion
f(3L);     // error: long to int and long to double are both conversions — amb
```

The last line is the one to remember. With the `area` set above, `area(4)` is ambiguous for the same reason: `int` to `long long` and `int` to `double` are both standard conversions of the same rank. Give the variable the exact type (`long long side = 4;`) or add an `int` overload, but do not rely on the compiler "obviously" preferring the integer one — it does not.

The string and pointer traps

```
void show(const std::string& s);
void show(bool b);

show("hello");     // calls show(bool)
```

`"hello"` is a `const char[6]`. Decaying to `const char*` and then to `bool` is a *standard* conversion; building a `std::string` is a *user-defined* one; the standard conversion wins. The cure is an overload that is exact for a literal — `void show(const char* s)` — or a `std::string_view` parameter, which a literal converts to cheaply (Module 5, lesson 6). The same rank rule makes `f(0)` prefer `f(int)` over `f(const char*)`, and it is why `nullptr` exists: `nullptr` converts to any pointer and never to an integer, so `f(nullptr)` picks the pointer overload unambiguously where `f(NULL)` could not.

Signed and unsigned bite the same way: with `g(int)` and `g(unsigned)`, `g(v.size())` passes an `unsigned long`, whose conversions to `unsigned` and to `int` rank equally — ambiguous. Keep overload sets small and their parameter types far apart.

Default arguments

```
std::string frame(const std::string& text, char border = '*', int padding = 1)

frame("hi");            // * hi *
frame("hi", '#');       // # hi #
frame("hi", '#', 3);    // ### hi ###
```

A default is a value the compiler inserts at the call when the argument is missing. The rules:

Trailing only. Once a parameter has a default, every parameter after it must have one. `f(int a = 1, int b)` is an error, and there is no way to "skip" the middle argument at a call.

Specified once, on the first declaration the caller sees. Write it on the prototype; write the definition *without* it. Repeating it on the definition is an error ("default argument given for parameter 2 after previous specification"), and putting it only on a definition that comes after the calls means the calls never saw it.

Evaluated at each call, in the caller's scope. A default may be an expression — `int retries = maxRetries()` — and it is re-evaluated every time it is used.

Bound at compile time by the static type. With virtual functions (Module 10, lesson 2) the default from the *base* declaration is used even when the derived override runs — one of the reasons to keep defaults out of virtual functions.

A default is not part of the signature. `frame(const std::string&, char, int)` is the one function whether the caller supplied one, two or three arguments.

Overloading versus defaults

Use a **default** when the parameter is genuinely optional and the behaviour with and without it is the same function: a separator, a padding width, a log level. Use **overloads** when the parameter *types* differ, when the behaviours differ enough that one body would be full of `if (hasValue)`, or when the "optional" argument changes the return type.

```
void log(const std::string& msg, int level = 1);              // one functio
std::string describe(int v);
std::string describe(double v);                               // overloads: t
```

When both would work, the default is simpler — one definition, one place to maintain — and the overload is more flexible. The wrong choice is neither; it is a single function with a flag parameter whose two halves share nothing.

What counts as a different signature

Differs in	Makes an overload?
number of parameters	yes
parameter types (`int` vs `double` , `int&` vs `int`)	yes
`const` on what a pointer or reference names (`const T&` vs `T&`)	yes
return type	no — error
parameter names	no — same function
top-level `const` on a by-value parameter (`int` vs `const int`)	no — same function
default arguments	no — same function

Pitfalls

Ambiguity you did not expect. Mixed integer widths and mixed integer/floating overloads are the usual cause; the fix is an exact-type variable or an added overload, not a cast at every call.

`show("literal")` reaching a `bool` overload. Add a `const char*` overload or take `std::string_view` .

Default on the definition, prototype without it. Legal only if no call comes before the definition — and the day someone reorders the file, the calls stop compiling.

Overloads differing only in top-level `const` . `void f(int)` and `void f(const int)` are a redefinition error, not two functions.

Key takeaways

Overloads share a name and differ in parameter count or types; the return type never distinguishes them.

Resolution ranks exact match, promotion, standard conversion, then user-defined conversion; a tie is a compile error, and `int` versus `long long` / `double` overloads tie constantly.

A string literal is a pointer: it prefers `bool` over `std::string`. Use `nullptr`, not `0`, for pointer overloads.

Default arguments are trailing, written once on the first declaration, and evaluated at every call.

Defaults for an optional value of the same kind; overloads when the types or behaviours differ.

EXERCISE 1

Three areas, one name

Write an overload set named `area`:

- `long long area(long long side);` — a square.
- `long long area(long long width, long long height);` — a rectangle.
- `double area(double radius);` — a circle, π × r² using `std::numbers::pi`.

Read `n`, then `n` lines: `square <side>`, `rect <width> <height>` or `circle <radius>`. Print `square: <area>`, `rect: <area>` or `circle: <area fixed to 2 decimals>`.

Input: `n`, then `n` shape lines (integer sides up to 100 000; the radius is a decimal). **Output:** `n` lines.

Example: input `3`, `square 4`, `rect 3 5`, `circle 2.5` →

```
square: 16
rect: 15
circle: 19.63
```

Read the sides into `long long` variables and the radius into a `double` so every call is an exact match: `area(4)` with an `int` argument would be ambiguous between the `long long` and `double` overloads.

INPUT 1

```
3
square 4
rect 3 5
circle 2.5
```

OUTPUT 1

```
square: 16
rect: 15
circle: 19.63
```

INPUT 2

```
2
circle 1
square 12
```

OUTPUT 2

```
circle: 3.14
square: 144
```

REFERENCE SOLUTION

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

long long area(long long side);
long long area(long long width, long long height);
double area(double radius);

int main() {
    int n = 0;
    std::cin >> n;
    std::cout << std::fixed << std::setprecision(2);
    for (int i = 0; i < n; ++i) {
        std::string shape;
        std::cin >> shape;
        if (shape == "square") {
            long long side = 0;
            std::cin >> side;
            std::cout << "square: " << area(side) << '\n';
        } else if (shape == "rect") {
            long long w = 0, h = 0;
            std::cin >> w >> h;
            std::cout << "rect: " << area(w, h) << '\n';
        } else {
            double radius = 0.0;
            std::cin >> radius;
            std::cout << "circle: " << area(radius) << '\n';
        }
    }
    return 0;
}

long long area(long long side) {
    return side * side;
}

long long area(long long width, long long height) {
    return width * height;
}

double area(double radius) {
    return std::numbers::pi * radius * radius;
}
```

EXERCISE 2

Frame it, with defaults

Write `std::string frame(const std::string& text, char border = '*', int padding = 1);` — it returns the border character repeated `padding` times, a space, the text, a space, and the border repeated again. **Put the defaults on the prototype above `main` and write the definition below without them.**

Read `n`, then `n` lines each holding `text` (one word) optionally followed by a border character and optionally a padding count. Call `frame` with one, two or three arguments — exactly as many as the line supplied — and print the result.

Input: `n`, then `n` lines. **Output:** `n` lines.

Example: input `3`, `hi`, `hi #`, `hi # 3` →

```
* hi *
# hi #
### hi ###
```

The starter reads each line into a `std::istringstream`; extracting the border and the padding fails cleanly when they are absent, which tells you how many arguments to pass.

INPUT 1

```
3
hi
hi #
hi # 3
```

OUTPUT 1

```
* hi *
# hi #
### hi ###
```

INPUT 2

```
2
report = 2
x
```

OUTPUT 2

```
== report ==
* x *
```

REFERENCE SOLUTION

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

std::string frame(const std::string& text, char border = '*', int padding = 1);

int main() {
    int n = 0;
    std::cin >> n;
    std::cin.ignore();
    for (int i = 0; i < n; ++i) {
        std::string line;
        std::getline(std::cin, line);
        std::istringstream in(line);
        std::string text;
        char border = '\0';
        int padding = 0;
        in >> text;
        bool hasBorder = static_cast<bool>(in >> border);
        bool hasPadding = hasBorder && static_cast<bool>(in >> padding);
        if (hasPadding) {
            std::cout << frame(text, border, padding) << '\n';
        } else if (hasBorder) {
            std::cout << frame(text, border) << '\n';
        } else {
            std::cout << frame(text) << '\n';
        }
    }
    return 0;
}

std::string frame(const std::string& text, char border, int padding) {
    std::string edge(static_cast<std::size_t>(padding), border);
    return edge + " " + text + " " + edge;
}
```

QUIZ

1. `void f(int); void f(double);` are declared. What does the call `f(3L)` do?

A. Calls `f(int)` — integers prefer integers

B. Calls `f(double)` — `long` is wider than `int`

C. Compile error: the call is ambiguous

D. Calls whichever overload was declared first

2. What does this print?

```
void show(const std::string& s) { std::cout << "string\n"; }
void show(bool b) { std::cout << "bool\n"; }
int main() { show("hi"); }
```

A. `string`

B. `bool`

C. Compile error: ambiguous

D. Undefined behaviour

3. Which declaration is ill-formed?

A. `int f(int a, int b = 2);`

B. `int f(int a = 1, int b = 2);`

C. `int f(int a = 1, int b);`

D. `int f(int a, int b, int c = 3);`

4. Does this compile?

```
int pad(int n, int width = 4);
int pad(int n, int width = 4) { return n + width; }
```

A. Yes — the two defaults agree, so it is fine

B. No — a default argument may be specified only once; the definition repeats it

C. Yes, with a warning

D. No — a definition may never have a default argument

5. Which statement about overloading on the return type alone (`int f();` and `double f();`) is correct?

A. Allowed — the compiler picks by the type the result is assigned to

B. Not allowed — the return type is not part of the signature

C. Allowed only for `constexpr` functions

D. Allowed since C++20

6. What does this print?

```
void f(int x)  { std::cout << "int "; }
void f(char x) { std::cout << "char "; }
int main() {
    f('a');
    f(97);
    short s = 1;
    f(s);
}
```

A. `char int int`

B. `char int char`

C. `int int int`

D. `char char char`

7. `log(msg)` and `log(msg, level)` do the same thing, with `level` treated as 1 when omitted. Which is the cleaner tool?

A. One function with a default argument `int level = 1`

B. Two overloads, one forwarding to the other

C. A global `currentLevel` the caller sets first

D. A template on the number of arguments

ANSWER KEY · OVERLOADING AND DEFAULT ARGUMENTS

1. C `long` to `int` and `long` to `double` are both *standard conversions* of the same rank, so neither overload is better and the compiler refuses to choose. Declaration order never breaks a tie.

2. B `"hi"` is a `const char[3]`; decaying to a pointer and converting to `bool` is a standard conversion, while building a `std::string` is a user-defined conversion, which ranks lower. Add a `const char*` overload or take `std::string_view` to fix it.

3. C Default arguments must be trailing: once a parameter has one, every later parameter must too. There is no way to supply `b` while omitting `a` at a call, so the language rejects the declaration.

4. B A default is given on the first declaration and must not be repeated, even with the same value: "default argument given for parameter 2 of 'int pad(int, int)' after previous specification". A definition may carry defaults only when it is the first declaration the calls see.

5. B A call `f();` could not choose, so the second declaration is an error ("ambiguating new declaration"). Overloads must differ in the number or types of their parameters.

6. A `'a'` is an exact match for `char`; `97` is an exact match for `int`; `short` to `int` is a *promotion*, which beats the `short` to `char` *conversion*, so the last call goes to `f(int)` as well.

7. A The parameter is optional and the behaviour is one function, which is exactly what a default expresses — one definition, one place to maintain. Overloads are for different *types* or different *behaviours*; a global adds hidden state.

LESSON 4 · 15 MIN

Recursion — base cases, the call stack and memoisation

A function may call itself. Used well, recursion turns a problem into a smaller copy of the same problem plus one step of work, and the code reads like the definition — a factorial, a tree walk, a flood fill, a permutation generator. Used carelessly, it is a stack overflow or an exponential run time. This lesson settles what every recursive function needs, what the call stack does and how deep it may go, when a loop is the better form, how a memo turns an exponential recursion into a linear one, and the recursion shapes interviews ask for.

The two cases

```
long long factorial(int n) {
    if (n ⩽ 1) return 1;                // base case: an answer with no furth
    return n * factorial(n - 1);        // recursive case: a smaller problem,
}
```

Every recursive function has a **base case** — an input answered directly — and a **recursive case** that calls the function on an input strictly closer to the base. Miss either and the recursion never ends. The base case here is `n ⩽ 1` rather than `n == 1` so that `factorial(0)` and a negative `n` both stop instead of running away towards `INT_MIN`.

The recursive call is an ordinary call. `factorial(4)` evaluates `4 * factorial(3)`, which waits for `factorial(3)`, which waits for `factorial(2)`, and so on down to `factorial(1)`; the products are then computed on the way back up:

```
factorial(4)
  factorial(3)
    factorial(2)
      factorial(1) → 1
    → 2 * 1 = 2
  → 3 * 2 = 6
→ 4 * 6 = 24
```

The call stack

Each call gets a **frame**: its parameters, its local variables and the address to return to, pushed on the stack and popped on return (Module 7, lesson 1 draws the picture). In the trace above four frames of `factorial` exist at once, each with its own `n` — that is why recursion works, and it is what limits it.

The stack is finite — 8 MB on the judge and on a typical Linux machine; a frame for a small function is a few dozen bytes, so a hundred thousand frames is fine and a hundred million is not. Overflowing it is not an exception you can catch — the process dies with a segmentation fault. You get there by a missing or unreachable base case (infinite recursion), which is a bug, or by a correct recursion on an input too large for it (a list of ten million nodes walked recursively), which is a signal to write a loop.

Recursive or iterative?

Any recursion can be rewritten as a loop, sometimes with an explicit stack of your own. For **linear** problems — sum a range, count digits, compute a factorial or a Fibonacci number — the loop (`for (int i = 2; i ⩽ n; ++i) result *= i;`) is shorter, faster and cannot overflow the stack.

Recursion earns its place on **branching** problems, where each call spawns several: walking a tree or directory, generating permutations and subsets, flood-filling a region, divide and conquer (merge sort, binary search), parsing nested expressions. There a loop needs an explicit stack that reproduces what the call stack gave you for free.

Exponential blow-up and memoisation

```
long long fib(int n) {
    if (n ⩽ 1) return n;
    return fib(n - 1) + fib(n - 2);
}
```

This is a correct definition and a terrible program. `fib(n)` calls `fib(n - 1)` and `fib(n - 2)` , each of which repeats most of the other's work; the number of calls is 2 × fib(n + 1) − 1, so `fib(40)` makes over 330 million calls and `fib(50)` would take hours — the same subproblems computed again and again.

Memoisation stores each result the first time it is computed:

```cpp
long long fib(int n, std::vector<long long>& memo) {
    if (n <= 1) return n;
    if (memo[n] != -1) return memo[n];          // already known
    memo[n] = fib(n - 1, memo) + fib(n - 2, memo);
    return memo[n];
}

std::vector<long long> memo(91, -1);            // fib(90) is the largest that
std::cout << fib(90, memo) << '\n';
```

Now every `n` is computed once: linear time, `n` frames deep at most. The memo is passed by reference — by value it would be copied on every call and the results written into a copy that is thrown away. A `std::vector` is the memo when the argument is a small non-negative integer; when the argument space is sparse, negative, or a pair of numbers, use `std::map<Key, long long>` (Module 13, lesson 4) with the same look-up-else-compute-and-store shape. Filling the table bottom-up with a loop is the same idea under the name dynamic programming.

Tail calls

A call in **tail position** is the last thing the function does, with nothing left to compute afterwards:

```cpp
long long sumTo(int n, long long acc) {
    if (n == 0) return acc;
    return sumTo(n - 1, acc + n);        // tail call: nothing to do after it
}
```

A compiler *may* turn this into a jump that reuses the frame, and GCC and Clang at `-O2` often do; the C++ standard does not require it, and a debug build will not do it. A recursion that only works because the optimiser removed the frames is a stack overflow waiting for a different flag. Write the loop.

Recursion over a structure

The shapes worth knowing by heart:

A number's digits. `digitSum(n)` is `n % 10 + digitSum(n / 10)`, with base case `n == 0`.

Permutations, by swapping into position. Fix the character at position `k` by trying each of the remaining ones there, recurse on `k + 1`, then swap back so the next choice starts from the same state:

```
void permute(std::string& s, std::size_t k, std::vector<std::string>& out) {
    if (k == s.size()) { out.push_back(s); return; }
    for (std::size_t i = k; i < s.size(); ++i) {
        std::swap(s[k], s[i]);
        permute(s, k + 1, out);
        std::swap(s[k], s[i]);           // undo, or the later iterations see
    }
}
```

Flood fill on a grid. Count the cells of a region by marking the current cell and adding the results of the four neighbours; bounds and "already visited" are the base cases:

```
int fill(std::vector<std::string>& g, int r, int c) {
    if (r < 0 || c < 0 || r ⩾ static_cast<int>(g.size()) || c ⩾ static_cast<
    if (g[r][c] ≠ '.') return 0;         // wall, or already filled
    g[r][c] = '#';                       // mark BEFORE recursing
    return 1 + fill(g, r + 1, c) + fill(g, r - 1, c) + fill(g, r, c + 1) + fi
}
```

The depth of a flood fill is the size of the region in the worst case (a snake-shaped corridor), so a 1 000 × 1 000 open grid can reach a million frames — an explicit queue (breadth-first search, Module 13, lesson 3) is the production form. On a 30 × 30 grid the recursion is exactly right.

Pitfalls

No base case, or one the input never reaches — `factorial(-1)` with `n == 0` as the only stop. Segmentation fault.

Marking after recursing in a fill or a graph walk: the neighbour visits you back before you are marked, forever.

Overflow, not recursion, is the limit. `factorial(21)` and `fib(93)` overflow `long long` long before the stack runs out; signed overflow is undefined behaviour, not a wrong answer you can detect afterwards.

The memo passed by value. It compiles, it runs, it is exponential again — and slower than before, because now it copies a vector per call.

Key takeaways

A recursive function needs a base case and a recursive case that moves strictly towards it.

Each call is a frame on a finite stack; a missing base case or a too-deep linear recursion is a crash, not an exception.

Loops for linear problems; recursion for branching ones — trees, permutations, fills, divide and conquer.

Memoise overlapping subproblems with a `std::vector` (dense integer keys) or `std::map` (sparse or composite keys), passed by reference.

Tail-call elimination is an optimisation, not a guarantee; never let correctness depend on it.

EXERCISE 1

Count the ways, with a memo

A staircase of `n` steps can be climbed one, two or three steps at a time. Write `long long ways(int n, std::vector<long long>& memo)` — recursive, memoised — returning the number of distinct ways: `ways(0)` is 1 (stand still), a negative `n` contributes 0, and otherwise `ways(n) = ways(n - 1) + ways(n - 2) + ways(n - 3)`.

Read `q`, then `q` values of `n` ($0 \le n \le 60$). Allocate **one** memo of size 61 filled with −1 before the queries and pass it by reference to every call; print `ways(<n>)=<value>` for each query.

Input: `q`, then `q` integers. **Output:** `q` lines.

Example: input `4`, `3`, `4`, `10`, `0` →

```
ways(3)=4
ways(4)=7
ways(10)=274
ways(0)=1
```

Without the memo, `ways(60)` would make more calls than the universe has time for; with it, each `n` is computed once.

INPUT 1

```
4
3
4
10
0
```

OUTPUT 1

```
ways(3)=4
ways(4)=7
ways(10)=274
ways(0)=1
```

INPUT 2

```
2
30
1
```

OUTPUT 2

```
ways(30)=53798080
ways(1)=1
```

REFERENCE SOLUTION

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

long long ways(int n, std::vector<long long>& memo);

int main() {
    int q = 0;
    std::cin >> q;
    std::vector<long long> memo(61, -1);
    for (int i = 0; i < q; ++i) {
        int n = 0;
        std::cin >> n;
        std::cout << "ways(" << n << ")=" << ways(n, memo) << '\n';
    }
    return 0;
}

long long ways(int n, std::vector<long long>& memo) {
    if (n < 0) return 0;
    if (n == 0) return 1;
    if (memo[n] != -1) return memo[n];
    memo[n] = ways(n - 1, memo) + ways(n - 2, memo) + ways(n - 3, memo);
    return memo[n];
}
```

EXERCISE 2

Flood fill the map

Read `R C` and then `R` rows of `C` characters: `.` is open ground and `#` is wall. Two open cells belong to the same region when they share an edge (up, down, left, right).

Write `int fill(std::vector<std::string>& grid, int r, int c)` — recursive — that returns 0 when `(r, c)` is outside the grid or not open, otherwise marks the cell `#` and returns 1 plus the results of the four neighbours. In `main`, scan every cell; each time you meet an open one, call `fill` and count a region. Print `regions=<count> largest=<size of the biggest region>`.

Input: `R C` (each at most 30), then `R` rows. **Output:** one line.

Example:

```
3 4
..#.
.##.
##..
```

→ `regions=2 largest=4`

INPUT 1

```
3 4
..#.
.##.
##..
```

OUTPUT 1

```
regions=2 largest=4
```

INPUT 2

```
2 2
##
##
```

OUTPUT 2

```
regions=0 largest=0
```

REFERENCE SOLUTION

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

int fill(std::vector<std::string>& grid, int r, int c);

int main() {
    int rows = 0, cols = 0;
    std::cin >> rows >> cols;
    std::vector<std::string> grid(rows);
    for (int r = 0; r < rows; ++r) std::cin >> grid[r];

    int regions = 0;
    int largest = 0;
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            if (grid[r][c] == '.') {
                int size = fill(grid, r, c);
                ++regions;
                if (size > largest) largest = size;
            }
        }
    }
    std::cout << "regions=" << regions << " largest=" << largest << '\n';
    return 0;
}

int fill(std::vector<std::string>& grid, int r, int c) {
    if (r < 0 || c < 0 || r >= static_cast<int>(grid.size()) || c >= static_cast<int
    if (grid[r][c] != '.') return 0;
    grid[r][c] = '#';
    return 1 + fill(grid, r + 1, c) + fill(grid, r - 1, c) + fill(grid, r, c + 1) +
}
```

QUIZ

1. What does this print?

```cpp
int f(int n) {
    if (n == 0) return 0;
    return n + f(n - 1);
}
int main() { std::cout << f(4) << '\n'; }
```

A. 10

B. 4

C. 24

D. Stack overflow

2. `int f(int n) { return n + f(n - 1); }` is called as `f(3)`. What happens?

A. Compile error: no base case

B. It returns 6

C. Infinite recursion until the stack is exhausted — a segmentation fault

D. It throws `std::overflow_error`

3. Roughly how many calls does the naive `fib(n) = fib(n-1) + fib(n-2)` make to compute `fib(40)`?

A. 40

B. About 1 600 (40^2)

C. About 330 million

D. Exactly 80

4. `long long fib(int n, std::vector<long long> memo)` — the memo is passed **by value**. What is the consequence?

A. None — it still memoises correctly

B. Compile error

C. Every call works on its own copy, so results are never shared and the run time is exponential again — plus a vector copy per call

D. The memo is shared because vectors are reference types

5. Which statement about tail calls in C++ is correct?

A. The standard guarantees that a tail-recursive call reuses the frame

B. Compilers may eliminate tail calls at `-O2`, but nothing guarantees it — write a loop when the depth matters

C. Tail calls are eliminated only in `void` functions

D. Every recursion is converted to a loop by the compiler

6. A flood fill recurses into the four neighbours **first** and marks the current cell `#` afterwards. What goes wrong?

A. Nothing — the order does not matter

B. The region count is off by one

C. A neighbour recurses back into the still-unmarked cell, and the recursion never terminates

D. Only the first row is filled

7. What does this print?

```
int digitSum(int n) {
    return n == 0 ? 0 : n % 10 + digitSum(n / 10);
}
int main() { std::cout << digitSum(4071) << '\n'; }
```

A. 12

B. 4071

C. 1

D. 7

ANSWER KEY · RECURSION — BASE CASES, THE CALL STACK AND MEMOISATION

1. A `f(4)` is 4 + 3 + 2 + 1 + 0 = 10: four frames deep, then the sums are formed on the way back. The base case `n == 0` is reached because every call moves `n` down by one.

2. C Nothing stops the recursion, so frames pile up until the stack runs out and the process is killed. The compiler cannot detect a missing base case, and a stack overflow is not an exception.

3. C The call count is 2 × fib(n + 1) − 1, and fib(41) is about 165 million — the same subproblems are recomputed over and over. Memoisation reduces it to about 40 calls.

4. C A by-value parameter is a fresh copy; the results written into it are thrown away when the call returns. The program is correct but slower than the unmemoised version. The memo must be `std::vector<long long>&`.

5. B Tail-call elimination is an optimisation, not a language rule; a debug build or a slightly different function shape can leave every frame in place. Correctness must never depend on it.

6. C The mark is the base case that stops re-entry. Without it in place before the recursive calls, cell A visits B, B sees A still open and visits A, and so on until the stack overflows.

7. A 4071 % 10 is 1, then 407 % 10 is 7, then 40 % 10 is 0, then 4, then the base case: 1 + 7 + 0 + 4 = 12.

LESSON 5 · 14 MIN

Lambdas — a first look

Some functions are too small and too local to deserve a name at the top of the file: the comparison that sorts these words by length, the test that counts the values above this threshold. A **lambda expression** writes such a function in place, right where it is used, and can carry values from the surrounding scope into its body. This lesson settles the syntax, what a lambda actually is, capturing by value and by reference, `mutable`, return-type deduction, generic lambdas with `auto` parameters, and how a lambda travels into `std::sort` and `std::count_if`. Module 14, lesson 4 goes deeper — capture defaults, `std::function`, recursive lambdas — once the algorithm library is in play.

A function without a name

```
auto square = [](int x) { return x * x; };
std::cout << square(7) << '\n';      // 49
```

The shape is `[captures](parameters) -> return_type { body }`. The square brackets introduce the lambda and list what it captures (nothing here). The parameters and body are those of any function. The return type is usually omitted: the compiler deduces it from the `return` statements, which must all agree — write `-> double` when a body returns `1` on one path and `x / 2.0` on another, or when you want the conversion to be explicit.

A lambda expression creates an object — a **closure** — of a type the compiler invents on the spot, with an `operator()` whose body is yours. No two lambdas have the same type, and the type has no name, which is why `auto` is how a lambda is stored. Calling it is calling any function object: `square(7)`.

Passing a lambda to an algorithm

The usual reason to write one is that a standard algorithm wants a callable:

```cpp
#include <algorithm>
#include <string>
#include <vector>

std::vector<std::string> words = {"fig", "banana", "kiwi", "apple"};

std::sort(words.begin(), words.end(), [](const std::string& a, const std::str
    if (a.size() != b.size()) return a.size() < b.size();   // shorter first..
    return a < b;                                            // ...then alphal
});

auto longWords = std::count_if(words.begin(), words.end(), [](const std::strir
    return w.size() > 4;
});
```

`std::sort` takes a comparator answering "does `a` go before `b`?" and requires a **strict weak ordering**: `<`-like, never `≤`-like. A comparator that returns `true` for equal elements breaks the sort — undefined behaviour, and on some inputs a crash. `std::count_if` takes a **predicate**, a callable returning `bool`. The parameters of the lambda take `const std::string&` for the same reason a function would: no copy per comparison.

Captures

A lambda's body sees its own parameters and anything with static lifetime. To use a local variable of the enclosing function it must **capture** it:

```cpp
int threshold = 10;
long long sum = 0;

auto above = [threshold](int x) { return x > threshold; };   // by value: a co
auto add   = [&sum](int x) { sum += x; };                    // by reference:

for (int x : values) if (above(x)) add(x);
```

`[threshold]` copies the variable *into* the closure at the moment the lambda is created; `[&sum]` stores a reference to the caller's variable. The difference is the same value-versus-reference distinction as parameters, with one twist that catches everyone once — a by-value capture is taken when the lambda is **created**, not when it is called:

```
int n = 1;
auto f = [n] { return n; };      // copies 1 now
n = 2;
std::cout << f() << '\n';        // 1
```

Several captures are listed with commas: `[threshold, &sum]`. The defaults `[=]` (everything used, by value) and `[&]` (everything used, by reference) exist and Module 14 explains when they are safe; until then list captures by name, so the reader sees what the lambda depends on.

A reference capture must not outlive the variable it names. Returning a lambda that captured a local by `&`, or storing it somewhere that runs after the enclosing function returned, leaves a dangling reference — the same undefined behaviour as any other reference to a dead object (Lesson 6).

`mutable`

The copies a lambda holds are `const` inside the body. Trying to change one is an error — "increment of read-only variable" — because the closure's `operator()` is a `const` member function by default. `mutable` lifts that:

```
int line = 0;
auto nextLine = [line]() mutable { return ++line; };

nextLine(); nextLine();
std::cout << nextLine() << '\n';   // 3
std::cout << line << '\n';         // 0 — the lambda changed its own copy
```

The state lives in the closure object; each call sees what the previous one left. Copying the lambda copies its state. The same can be written with an **init-capture** that creates the variable inside the brackets — `[count = 0]() mutable { return ++count; }` — when there is no outer variable to copy from. A `mutable` lambda is a small stateful object; when the state grows past a counter, a class (Module 8) says the same thing more clearly.

Generic lambdas

Since C++14 a parameter may be `auto`, and the lambda then works for any type that supports the body:

```
auto add = [](auto a, auto b) { return a + b; };
add(1, 2);                                  // 3
add(1.5, 2.25);                             // 3.75
add(std::string("ab"), std::string("c"));   // "abc"
```

Under the hood the closure's `operator()` is a template, instantiated per argument-type combination — Module 12 shows the mechanism. Generic lambdas are what you reach for when an algorithm's element type is long to spell or you want one comparator for several containers.

Storing and passing lambdas

`auto` holds a lambda by its exact type — free, and the form to use inside a function.

A template parameter takes any callable: `template <typename F> void forEachLine(F f)`; the algorithms are written this way.

A function pointer can hold a lambda that captures nothing: `int (*fp)(int) = [](int x) { return x + 1; };` — useful for C APIs.

`std::function<int(int)>` holds any callable with that signature, captures and all, at the cost of an allocation and an indirect call. It is the type for "a callback stored in a struct" and the subject of Module 14, lesson 4.

When to use a lambda, and when a function

A lambda is right when the behaviour is used once, belongs beside the code that uses it, or needs local state from the enclosing scope. A named function is right when the behaviour is reused, is long enough to want its own tests, or is the kind of thing a reader would search for by name. A lambda longer than about ten lines is a named function that has not been extracted yet.

Pitfalls

Capturing by reference and outliving the variable. Dangling, undefined behaviour, and the compiler rarely notices.

Capturing by value and expecting later changes to show. The copy is taken at creation.

Modifying a value capture without `mutable`. Compile error, with a message that mentions a read-only variable.

A comparator with `<=`. Violates strict weak ordering; `std::sort` may run off the end of the range.

Inconsistent return types. `return 1;` on one path and `return 2.5;` on another is an error; add `-> double`.

`[=]` copying a large container silently. Name the captures until Module 14.

Key takeaways

`[captures](params) { body }` creates a closure object with a unique type; store it in `auto`.

Capture by value copies at creation; capture by reference aliases and must not outlive the variable.

`mutable` lets a lambda change its own copies; the state persists between calls and copies with the lambda.

`std::sort` needs a strict-weak-ordering comparator, `std::count_if` a predicate; both take the lambda directly.

`auto` parameters make a lambda generic; `std::function` stores any callable, at a cost — Module 14 takes both further.

EXERCISE 1

Sort and count with lambdas

Read `minLen`, then `n`, then `n` words (single tokens). Sort the words with **a lambda comparator passed to `std::sort`**: shorter words first, and words of equal length alphabetically. Print the sorted words one per line, then `long=<count>`, where the count comes from **`std::count_if` with a lambda that captures `minLen` by value** and is true for words of length at least `minLen`.

Input: `minLen`, `n`, then `n` words. **Output:** `n` lines, then the count line.

Example: input `4`, `5`, `banana fig apple kiwi date` →

```
fig
date
kiwi
apple
banana
long=4
```

INPUT 1

```
4
5
banana fig apple kiwi date
```

OUTPUT 1

```
fig
date
kiwi
apple
banana
long=4
```

INPUT 2

```
6
3
ab abc a
```

OUTPUT 2

```
a
ab
abc
long=0
```

REFERENCE SOLUTION

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

int main() {
    std::size_t minLen = 0;
    std::size_t n = 0;
    std::cin >> minLen >> n;
    std::vector<std::string> words(n);
    for (std::size_t i = 0; i < n; ++i) std::cin >> words[i];

    std::sort(words.begin(), words.end(), [](const std::string& a, const std::string
        if (a.size() != b.size()) return a.size() < b.size();
        return a < b;
    });

    auto longEnough = [minLen](const std::string& w) { return w.size() >= minLen; };
    long long longWords = std::count_if(words.begin(), words.end(), longEnough);

    for (const auto& w : words) std::cout << w << '\n';
    std::cout << "long=" << longWords << '\n';
    return 0;
}
```

EXERCISE 2

Captures: by value, by reference, mutable

Read `threshold`, then `n`, then `n` integers. Build three lambdas, each stored in an `auto` variable:

- `above` — captures `threshold` **by value** and returns whether `x > threshold`.
- `addToTotal` — captures a `long long total` **by reference** and adds `x` to it.
- `nextLine` — captures an `int line` (initially 0) **by value**, is `mutable`, and returns `++line` on each call.

Loop over the values: for each one `above` accepts, print `<nextLine()>: <x>` and call `addToTotal(x)`. Finally print `kept=<std::count_if with above> total=<total> outer=<main's line variable>` — the last number shows that the mutable lambda changed its own copy, not `main`'s.

Input: `threshold`, `n`, then `n` integers. **Output:** one line per accepted value, then the summary line.

Example: input `10`, `6`, `4 12 10 25 -3 11` →

```
1: 12
2: 25
3: 11
kept=3 total=48 outer=0
```

INPUT 1

```
10
6
4 12 10 25 -3 11
```

OUTPUT 1

```
1: 12
2: 25
3: 11
kept=3 total=48 outer=0
```

INPUT 2

```
0
3
-1 -2 -3
```

OUTPUT 2

```
kept=0 total=0 outer=0
```

REFERENCE SOLUTION

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

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

    long long total = 0;
    int line = 0;

    auto above = [threshold](int x) { return x > threshold; };
    auto addToTotal = [&total](int x) { total += x; };
    auto nextLine = [line]() mutable { return ++line; };

    for (int x : values) {
        if (above(x)) {
            std::cout << nextLine() << ": " << x << '\n';
            addToTotal(x);
        }
    }

    long long kept = std::count_if(values.begin(), values.end(), above);
    std::cout << "kept=" << kept << " total=" << total << " outer=" << line << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
int n = 5;
auto f = [n] { return n * 2; };
n = 100;
std::cout << f() << '\n';
```

A. 10

B. 200

C. Compile error

D. Undefined behaviour

2. What does this print?

```
int n = 5;
auto f = [&n] { n += 1; };
f();
f();
std::cout << n << '\n';
```

A. 5

B. 6

C. 7

D. Compile error

3. Does this compile?

```
int count = 0;
auto tick = [count] { return ++count; };
```

A. Yes — `tick()` returns 1, then 2

B. No — increment of read-only variable; the lambda needs `mutable`

C. Yes — `tick()` always returns 1

D. No — a lambda cannot capture an `int`

4. `std::sort(v.begin(), v.end(), [](int a, int b) { return a ⩽ b; });` — what is the consequence of the `⩽` ?

A. The vector is sorted ascending

B. The vector is sorted descending

C. Undefined behaviour: the comparator is not a strict weak ordering

D. Compile error

5. What is the right way to hold a lambda in a local variable?

A. `std::function` — it is the type of every lambda

B. `auto` — each lambda has a unique unnamed closure type

C. `void*`

D. `lambda` — the keyword names the type

6. What does this print?

```
auto add = [](auto a, auto b) { return a + b; };
std::cout << add(1, 2) << ' ' << add(std::string("x"), std::string("y")) << '\n
```

A. `3 xy`

B. `3 x+y`

C. Compile error: a lambda cannot be generic

D. `12 xy`

7. What does this print?

```
std::vector<int> v{1, 5, 8, 12};
int limit = 6;
std::cout << std::count_if(v.begin(), v.end(), [limit](int x) { return x > limi
```

A. 2

B. 3

C. 1

D. 0

ANSWER KEY · LAMBDAS — A FIRST LOOK

1. A A by-value capture copies `n` when the lambda is *created*, while it is still 5. The later assignment changes `main`'s `n`, not the closure's copy.

2. C `[&n]` stores a reference to `main`'s `n`, so each call increments the original: 5 → 6 → 7. No `mutable` is needed because the lambda modifies the referenced object, not a captured copy.

3. B By-value captures are `const` inside the body because the closure's `operator()` is a `const` member. `[count]() mutable { return ++count; }` makes the copy writable, and then the calls return 1, 2, 3 as the copy persists.

4. C A comparator must answer `false` for equal elements. With `<=` it says both `a` before `b` and `b` before `a`, and `std::sort` may read past the end of the range — a crash on some inputs. Use `<`.

5. B Every lambda expression has its own compiler-generated type with no name, so `auto` is the exact and free way to store one. `std::function` *can* hold it, at the cost of type erasure and an indirect call; there is no `lambda` keyword.

6. A `auto` parameters make the closure's `operator()` a template, instantiated once for `(int, int)` and once for `(std::string, std::string)`. Integer `+` adds; string `+` concatenates.

7. A The predicate captures `limit` (6) by value and is true for 8 and 12 only. `std::count_if` returns how many elements satisfy it.

LESSON 6 · 14 MIN

Scope, lifetime and linkage

Three different questions hide behind "where can I use this variable?": **scope** — in which region of the source the name is visible; **lifetime** — between which two moments the object exists; and **linkage** — whether the same name in two translation units refers to one entity or two. Beginners meet them tangled: a `static` local that remembers its value, a global that two files fight over, a function that returns a reference to something already gone. This lesson pulls the three apart, because each has its own rules and its own bugs.

Block scope and shadowing

A name declared inside a block is visible from its declaration to the closing brace — and in nested blocks, unless an inner declaration **shadows** it:

```
int x = 1;
{
    int x = 2;                // a new x; the outer one is hidden here
    std::cout << x << '\n';   // 2
}
std::cout << x << '\n';       // 1
```

Loop variables and `if`-with-initialiser variables have the scope of their statement. Shadowing is legal and occasionally intended, but a parameter shadowed by a local, or an outer `count` hidden by an inner one, is a bug waiting for a reader; `-Wshadow` reports every case.

Lifetime: storage duration

Storage duration	Examples	Lives from … to
automatic	locals, parameters	its declaration … the end of its block
static	globals, namespace-scope variables, function-local `static`, string literals	before `main` (or first use) … program exit
thread	`thread_local` variables	thread start … thread end (Module 17)
dynamic	what `new` creates; a container's elements	`new` … `delete` (Module 7)

An automatic object is destroyed at its closing brace, in reverse order of declaration. Nothing about it survives — which is the rule behind the most important bug in this lesson.

Function-local `static`

```
int nextId() {
    static int counter = 0;      // initialised once, the first time control r
    return ++counter;
}

nextId();  nextId();
std::cout << nextId() << '\n';   // 3
```

The variable has **static** storage duration — it lives for the whole program — but **function** scope: nothing outside `nextId` can name it. The initialiser runs once, the first time execution reaches it, and C++11 guarantees that first run is thread-safe. It is the honest way to write a counter, an id generator or a lazily built lookup table without a global.

The cost is hidden state: the function returns different results for the same arguments, cannot be reset between tests, and is shared by every caller in the process. Use it for genuinely program-wide facts; for anything a caller might want two of, pass the state in or make it a class (Module 8).

Globals, and why to avoid them

A variable at namespace scope — outside every function — also has static storage duration:

```
int g_requests = 0;          // a global

void handle() { ++g_requests; }
```

Every function can read and write it, which is exactly the problem: reasoning about `handle()` now requires knowing every other function that touches `g_requests`, testing it requires resetting global state, and running it on two threads is a data race. Globals also initialise before `main` in an order that is unspecified *across* translation units — a global whose initialiser reads a global in another file is the "static initialisation order fiasco", and it fails only on some builds.

Constants are the exception: `constexpr double kGravity = 9.81;` at namespace scope is a global with no state. For everything else, pass parameters and return values; state that several functions share belongs in an object they are handed.

Linkage: internal and external

A name with **external linkage** denotes the same entity in every translation unit; the linker matches them up. Functions and non-`const` globals have it by default, so two files that each define `int helper()` collide at link time — "multiple definition of `helper()`" — even if the bodies are identical. A name with **internal linkage** is private to its translation unit. Two ways to give it:

```
// util.cpp
static int parsePositive(const std::string& s);   // C-style: static at namesp

namespace {                                        // C++ style: an anonymous n
    int callCount = 0;
    int parseDigits(const std::string& s) { return static_cast<int>(s.size());
}
```

Everything inside an anonymous namespace has internal linkage; another file may define its own `parseDigits` without conflict, and helpers that are not part of a file's interface belong there. `const` and `constexpr` variables at namespace scope are internal by default in C++ (unlike C), which is why a header of `constexpr` constants causes no multiple-definition errors.

To *use* a global defined in another file, declare it with `extern`:

```cpp
// counters.cpp
int g_requests = 0;          // the definition — exactly one across the progr

// server.cpp
extern int g_requests;       // a declaration: "defined elsewhere"
```

`extern` says "this exists; do not create it here". Without it the second line would be a second definition.

`inline` functions and variables

`inline` no longer means "please inline this call" — the optimiser decides that on its own. It means: this definition may appear in several translation units, provided they are identical, and the linker keeps one — the exemption from the one-definition rule a function defined in a header needs, since every file that includes the header gets the definition. C++17 extended it to variables:

```cpp
// config.h
inline constexpr int kMaxRetries = 3;      // one object shared by every inclu
inline int roundUp(int n, int step) { return (n + step - 1) / step * step; }
```

Module 8, lesson 6 covers the header/source split this enables.

Dangling: what lifetime means for a reference

A reference or pointer is only as alive as the object it names. When the object's lifetime ends, the reference **dangles**, and using it is undefined behaviour — often it prints the right thing in a debug build and garbage under `-O2`:

```cpp
const std::string& makeName() {
    std::string s = "temp";
    return s;                // s is destroyed on the next line; the caller re
}
```

`-Wall` reports `reference to local variable 's' returned`. The fix is to return by value — Lesson 2 showed it is free. The same rule governs a lambda that captured a local by reference and outlived it (Lesson 5) and a reference into a `std::vector` after a `push_back`

reallocated it (Module 6, lesson 4); a pointer to a `static` local is *fine*, because static lifetime ends only at exit.

The three meanings of `static`

Where	Meaning
inside a function	static storage duration — one instance, initialised once, lives to exit
at namespace scope	internal linkage — private to this translation unit
inside a class	a member shared by every object of the class (Module 8, lesson 4)

The keyword was reused rather than invented three times; read it by where it appears.

Pitfalls

Expecting a function-local `static` to reset. It never does; the second call sees what the first left.

A parameter shadowed by a local of the same name. Legal, wrong, invisible without `-Wshadow`.

The same helper defined in two `.cpp` files. Linker error; put helpers in anonymous namespaces.

Returning a reference or pointer to a local. Undefined behaviour; return by value.

`static` in a header at namespace scope. Every includer gets its own private copy — usually not what was meant; `inline` is.

Key takeaways

Scope is where a name is visible; lifetime is when the object exists; linkage is whether two files share the entity.

A function-local `static` is initialised once and lives to program exit; it is a counter or a cache, not a way to pass state.

Globals hide dependencies, break tests and race across threads; constants are the acceptable ones.

Internal linkage (anonymous namespace or `static`) keeps helpers private to a translation unit; `extern` declares a global defined elsewhere; `inline` lets a header define a function or variable once for all includers.

A reference outlives nothing: a returned reference to a local, a captured local, or an element of a reallocated vector is dangling.

EXERCISE 1

Labels from a function-local static

Write `int nextId()` that keeps its counter in a **function-local** `static int` starting at 0 and returns the incremented value (the first call returns 1, the second 2, ...). No global variable is allowed — the counter must live inside `nextId`. Then write `std::string makeLabel(const std::string& prefix)` that returns `<prefix>-<nextId()>`.

Read `n`, then `n` prefixes (single tokens; they may repeat). Print each label on its own line, then `next=<nextId()>` — one more call, showing the counter has kept counting.

Input: `n`, then `n` prefixes. **Output:** `n` labels, then the `next=` line.

Example: input `3`, `order order invoice` →

```
order-1
order-2
invoice-3
next=4
```

INPUT 1

```
3
order order invoice
```

OUTPUT 1

```
order-1
order-2
invoice-3
next=4
```

INPUT 2

```
1
x
```

OUTPUT 2

```
x-1
next=2
```

REFERENCE SOLUTION

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

int nextId();
std::string makeLabel(const std::string& prefix);

int main() {
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string prefix;
        std::cin >> prefix;
        std::cout << makeLabel(prefix) << '\n';
    }
    std::cout << "next=" << nextId() << '\n';
    return 0;
}

int nextId() {
    static int counter = 0;
    return ++counter;
}

std::string makeLabel(const std::string& prefix) {
    return prefix + "-" + std::to_string(nextId());
}
```

EXERCISE 2

Instrumented recursion in an anonymous namespace

Count how many calls the naive recursive Fibonacci makes. Inside an **anonymous namespace** (internal linkage) put a `long long callCount` and a helper `long long fibCounted(int n)` that increments `callCount` on every entry and computes `fib` the naive way (`fib(0) = 0`, `fib(1) = 1`, otherwise the sum of the two before). Outside the namespace, with ordinary external linkage, write the two functions `main` uses: `long long fib(int n)` — resets the counter to 0, then returns `fibCounted(n)` — and `long long callsForLast()` — returns the counter.

Read `q`, then `q` values of `n` (0 ≤ n ≤ 30). For each print `fib(<n>)=<value> calls=<count>`.

Input: `q`, then `q` integers. **Output:** `q` lines.

Example: input `3`, `10`, `0`, `2` →

```
fib(10)=55 calls=177
fib(0)=0 calls=1
fib(2)=1 calls=3
```

The counter is shared state, so it is hidden where no other translation unit can reach it; the two named functions are the only interface.

INPUT 1

```
3
10
0
2
```

OUTPUT 1

```
fib(10)=55 calls=177
fib(0)=0 calls=1
fib(2)=1 calls=3
```

INPUT 2

```
1
20
```

OUTPUT 2

```
fib(20)=6765 calls=21891
```

REFERENCE SOLUTION

```cpp
#include <iostream>

namespace {
    long long callCount = 0;

    long long fibCounted(int n) {
        ++callCount;
        if (n <= 1) return n;
        return fibCounted(n - 1) + fibCounted(n - 2);
    }
}

long long fib(int n);
long long callsForLast();

int main() {
    int q = 0;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        int n = 0;
        std::cin >> n;
        long long value = fib(n);
        std::cout << "fib(" << n << ")=" << value << " calls=" << callsForLast() <<
    }
    return 0;
}

long long fib(int n) {
    callCount = 0;
    return fibCounted(n);
}

long long callsForLast() {
    return callCount;
}
```

QUIZ

1. What does this print?

```
int counter() {
    static int c = 10;
    return ++c;
}
int main() {
    counter();
    counter();
    std::cout << counter() << '\n';
}
```

A. 11

B. 13

C. 10

D. Compile error

2. What does this print?

```
int x = 1;
int main() {
    int x = 2;
    {
        int x = 3;
        std::cout << x;
    }
    std::cout << x << '\n';
}
```

A. `32`

B. `31`

C. `33`

D. Compile error: `x` redeclared

3. Two `.cpp` files in the same program each define `int helper() { return 1; }` at namespace scope. What happens?

A. It compiles and links; each file uses its own

B. Linker error: multiple definition of `helper()`

C. Compile error in the second file

D. Undefined behaviour at run time

4. How do you make a helper function private to its translation unit?

A. Declare it `extern`

B. Put it in an anonymous namespace, or mark it `static` at namespace scope

C. Mark it `inline`

D. Mark it `const`

5. What does the line `extern int g_total;` mean in a source file?

A. It defines `g_total` and initialises it to 0

B. It declares that `g_total` is defined in another translation unit

C. It gives `g_total` internal linkage

D. It creates a thread-local copy of `g_total`

6. What is wrong with this function?

```
const std::string& name() {
    std::string s = "ada";
    return s;
}
```

A. Nothing — the reference keeps `s` alive

B. Compile error: a function cannot return a reference

C. Undefined behaviour: it returns a dangling reference to a destroyed local; `-Wall` warns

D. It always returns an empty string

7. What does `inline` on a function mean in modern C++?

A. The compiler must inline every call to it

B. Its definition may appear in several translation units (identically) and the linker keeps one — what a function defined in a header needs

C. It has internal linkage

D. It is evaluated at compile time

ANSWER KEY · SCOPE, LIFETIME AND LINKAGE

1. B The `static` local is initialised to 10 exactly once. Each call increments it: 11, 12, then the printed 13. A non-static `int c = 10;` would print 11.

2. A Each inner `x` shadows the outer one for the extent of its block. Inside the braces `x` is 3; after them the block's `x` is gone and `main`'s `x` (2) is visible again. The global `x` is never reached.

3. B A function has external linkage by default, so both files claim to define the one program-wide `helper()`. Each file compiles fine on its own; the linker sees two definitions and refuses.

4. B Both give internal linkage, so another file may define its own function of the same name without a clash. `extern` is the opposite (external linkage), `inline` permits *identical* definitions in many files, and `const` does not apply to functions.

5. B `extern` on a declaration without an initialiser says "this object exists elsewhere; do not create one here". Exactly one translation unit must provide the definition `int g_total = 0;`.

6. C `s` has automatic storage and is destroyed when the function returns; the caller receives a reference to nothing. GCC reports "reference to local variable 's' returned". Return `std::string` by value.

7. B Inlining calls is the optimiser's decision regardless of the keyword. `inline` relaxes the one-definition rule so a header can carry the definition; C++17 gave variables the same power with `inline constexpr`.

CHECKPOINT · 25 MIN

Checkpoint — Functions

This checkpoint covers the whole module: declarations and definitions, the three ways to pass a parameter and returning by value, overloading and default arguments, recursion and memoisation, lambdas and captures, and scope, lifetime and linkage.

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:

- What is the difference between a declaration and a definition, and which error do you get when each is missing?
- When should a parameter be `T`, `const T&`, and `T&`?
- Given `f(int)` and `f(double)`, why is `f(3L)` a compile error?
- Where may a default argument be written, and why must it be trailing?
- What two things must every recursive function have, and what happens without them?
- What is the difference between `[x]` and `[&x]` in a capture list, and what does `mutable` change?
- What does a function-local `static` do on the second call, and what does `static` mean at namespace scope?

The three programs are a recursive permutation generator, a grade report built from small const-reference functions with a default argument and an overload, and a leaderboard sorted with a lambda and numbered by a function-local `static`. Read each input format carefully; the functions named in the prompt are the contract.

EXERCISE 1

Every arrangement

Read one word of 1 to 6 **distinct** lowercase letters. Write a recursive `void permute(std::string& s, std::size_t k, std::vector<std::string>& out)` that generates every arrangement by swapping each remaining character into position `k`, recursing on `k + 1`, and swapping back. Collect the results, sort them with `std::sort`, print one per line, then print `count=<how many>`.

Input: one word. **Output:** all permutations in lexicographic order, then the count line.

Example: input `abc` →

```
abc
acb
bac
bca
cab
cba
count=6
```

INPUT 1

```
abc
```

OUTPUT 1

```
abc
acb
bac
bca
cab
cba
count=6
```

INPUT 2

```
z
```

OUTPUT 2

```
z
count=1
```

REFERENCE SOLUTION

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

void permute(std::string& s, std::size_t k, std::vector<std::string>& out);

int main() {
    std::string word;
    std::cin >> word;
    std::vector<std::string> out;
    permute(word, 0, out);
    std::sort(out.begin(), out.end());
    for (const auto& p : out) std::cout << p << '\n';
    std::cout << "count=" << out.size() << '\n';
    return 0;
}

void permute(std::string& s, std::size_t k, std::vector<std::string>& out) {
    if (k == s.size()) {
        out.push_back(s);
        return;
    }
    for (std::size_t i = k; i < s.size(); ++i) {
        std::swap(s[k], s[i]);
        permute(s, k + 1, out);
        std::swap(s[k], s[i]);
    }
}
```

EXERCISE 2

Grade report

Read `n`, then `n` lines `name m1 m2 m3` (marks 0–100). Write four small functions, prototypes above `main`:

- `double mean(const std::vector<int>& marks);`
- `std::string grade(double average);` — `A` for 90 and above, `B` for 80, `C` for 70, `D` for 60, otherwise `F` (judged on the unrounded average).
- `std::string grade(const std::vector<int>& marks);` — an **overload** that grades the mean of the marks.
- `std::string fmt(double value, int decimals = 1);` — the value fixed to `decimals` places, with the **default on the prototype**.

Print one line per student, `<name>: <fmt(average)> <grade>`, then `class: <fmt(class average, 2)> <grade of the class average>` where the class average is the mean of the student averages, and finally `best: <name of the highest average — the first one on ties>`.

Input: `n`, then `n` student lines. **Output:** `n + 2` lines.

Example: input `2`, `Ada 95 90 90`, `Bob 70 75 72` →

```
Ada: 91.7 A
Bob: 72.3 C
class: 82.00 B
best: Ada
```

INPUT 1

```
2
Ada 95 90 90
Bob 70 75 72
```

OUTPUT 1

```
Ada: 91.7 A
Bob: 72.3 C
class: 82.00 B
best: Ada
```

INPUT 2

```
1
Cy 59 60 61
```

OUTPUT 2

```
Cy: 60.0 D
class: 60.00 D
best: Cy
```

REFERENCE SOLUTION

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

double mean(const std::vector<int>& marks);
std::string grade(double average);
std::string grade(const std::vector<int>& marks);
std::string fmt(double value, int decimals = 1);

int main() {
    int n = 0;
    std::cin >> n;
    double classTotal = 0.0;
    std::string best;
    double bestAverage = -1.0;
    for (int i = 0; i < n; ++i) {
        std::string name;
        std::vector<int> marks(3);
        std::cin >> name >> marks[0] >> marks[1] >> marks[2];
        double average = mean(marks);
        std::cout << name << ": " << fmt(average) << ' ' << grade(marks) << '\n';
        classTotal += average;
        if (average > bestAverage) {
            bestAverage = average;
            best = name;
        }
    }
    double classAverage = classTotal / n;
    std::cout << "class: " << fmt(classAverage, 2) << ' ' << grade(classAverage) <<
    std::cout << "best: " << best << '\n';
    return 0;
}

double mean(const std::vector<int>& marks) {
    long long sum = 0;
    for (int m : marks) sum += m;
    return static_cast<double>(sum) / static_cast<double>(marks.size());
}

std::string grade(double average) {
    if (average ⩾ 90) return "A";
    if (average ⩾ 80) return "B";
    if (average ⩾ 70) return "C";
    if (average ⩾ 60) return "D";
    return "F";
}

std::string grade(const std::vector<int>& marks) {
```

```
    return grade(mean(marks));
}

std::string fmt(double value, int decimals) {
    std::ostringstream out;
    out.precision(decimals);
    out << std::fixed << value;
    return out.str();
}
```

EXERCISE 3

Leaderboard with a lambda and a static rank

Read `k passMark`, then `n`, then `n` lines `name score`. Sort the entries with **a lambda comparator**: higher score first, and equal scores by name ascending. Write `int nextRank()` whose counter is a **function-local `static`** (first call returns 1) and `std::string row(const std::string& name, int score)` that returns `<nextRank()>. <name> <score>`. Print the first `k` rows (or all of them when there are fewer than `k`), then `passed=<count>` — the number of entries with `score ⩾ passMark`, from **`std::count_if` with a lambda that captures `passMark`**.

Input: `k passMark`, `n`, then `n` entries. **Output:** up to `k` rows, then the `passed=` line.

Example: input `3 50`, `5`, `zoe 70`, `al 90`, `bea 70`, `cid 40`, `dan 90` →

```
1. al 90
2. dan 90
3. bea 70
passed=4
```

INPUT 1

```
3 50
5
zoe 70
al 90
bea 70
cid 40
dan 90
```

OUTPUT 1

```
1. al 90
2. dan 90
3. bea 70
passed=4
```

INPUT 2

```
2 100
2
x 10
y 5
```

OUTPUT 2

```
1. x 10
2. y 5
passed=0
```

REFERENCE SOLUTION

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

struct Entry {
    std::string name;
    int score;
};

int nextRank();
std::string row(const std::string& name, int score);

int main() {
    int k = 0, passMark = 0;
    std::size_t n = 0;
    std::cin >> k >> passMark >> n;
    std::vector<Entry> entries(n);
    for (std::size_t i = 0; i < n; ++i) std::cin >> entries[i].name >> entries[i].sc

    std::sort(entries.begin(), entries.end(), [](const Entry& a, const Entry& b) {
        if (a.score != b.score) return a.score > b.score;
        return a.name < b.name;
    });

    std::size_t shown = std::min(static_cast<std::size_t>(k), entries.size());
    for (std::size_t i = 0; i < shown; ++i) {
        std::cout << row(entries[i].name, entries[i].score) << '\n';
    }

    long long passed = std::count_if(entries.begin(), entries.end(), [passMark](cons
        return e.score >= passMark;
    });
    std::cout << "passed=" << passed << '\n';
    return 0;
}

int nextRank() {
    static int rank = 0;
    return ++rank;
}

std::string row(const std::string& name, int score) {
    return std::to_string(nextRank()) + ". " + name + " " + std::to_string(score);
}
```

CHECKPOINT QUESTIONS

1. The message `undefined reference to 'area(int)'` is reported by...

A. The preprocessor

B. The compiler

C. The linker

D. The runtime, on the first call

2. Which of these is part of a function's signature?

A. The return type

B. The parameter names

C. The parameter types

D. The default arguments

3. What does this print?

```
void f(int v)  { v = 0; }
void g(int& v) { v = 0; }
int main() {
    int a = 7, b = 7;
    f(a);
    g(b);
    std::cout << a << b << '\n';
}
```

A. `77`

B. `07`

C. `70`

D. `00`

4. A function receives a `std::vector<int>` of a million elements and only reads it. The parameter should be…

A. `std::vector<int> v`

B. `const std::vector<int>& v`

C. `std::vector<int>& v`

D. `const std::vector<int> v`

5. `void f(long); void f(double);` — what does `f(1)` do?

A. Calls `f(long)`

B. Calls `f(double)`

C. Compile error: ambiguous

D. Calls `f(long)` because it is declared first

6. A function has a prototype above `main` and a definition below it. Where does its default argument go?

A. On both, with the same value

B. On the definition only

C. On the prototype only

D. Anywhere — the compiler merges them

7. What does this print?

```
long long p(long long b, int e) {
    return e == 0 ? 1 : b * p(b, e - 1);
}
int main() { std::cout << p(2, 10) << '\n'; }
```

A. 1024

B. 20

C. 2048

D. Stack overflow

8. You memoise `f(x, y)` where `x` and `y` range over $-10^6..10^6$ but only a few thousand pairs are ever asked for. The best memo is…

A. A `std::vector<long long>` indexed by `x`

B. A `std::map<std::pair<int, int>, long long>`

C. A function-local `static long long`

D. No memo is possible for two arguments

9. What does this print?

```
int base = 10;
auto f = [base](int x) mutable { base += x; return base; };
f(1);
std::cout << f(1) << ' ' << base << '\n';
```

A. `12 10`

B. `12 12`

C. `11 10`

D. Compile error

10. Which comparator lambda sorts a `std::vector<int>` in **descending** order correctly?

A. `[](int a, int b) { return a > b; }`

B. `[](int a, int b) { return a ⩾ b; }`

C. `[](int a, int b) { return a < b; }`

D. `[](int a, int b) { return b ⩽ a; }`

11. When is a function-local `static` variable initialised?

A. On every call to the function

B. Once, the first time control reaches its declaration

C. When the program starts, before `main`, always

D. When the function returns for the first time

12. Which declaration gives `n` internal linkage?

A. `int n = 0;` at namespace scope

B. `namespace { int n = 0; }`

C. `extern int n;`

D. `inline int n = 0;`

13. A lambda captures a local `int total` by reference, is returned from the function that declared `total`, and is called by the caller. What happens?

A. It works: the capture keeps `total` alive

B. Undefined behaviour: `total` was destroyed when its function returned, so the capture dangles

C. Compile error: a lambda cannot be returned

D. It returns 0

14. What does `[[nodiscard]]` on a function do?

A. Prevents the function from being inlined

B. Makes calling it as a bare statement — ignoring the result — a compiler warning

C. Forces the result to be stored in a `const` variable

D. Deletes the return value after use

ANSWER KEY · CHECKPOINT — FUNCTIONS

1. C The compiler accepted the call because it saw a declaration; the linker then failed to find a definition with that exact signature. A missing *declaration* would be a compiler error instead.

2. C The signature is the name plus the parameter types, which is why overloads must differ there and why neither the return type nor names nor defaults can distinguish two functions.

3. C `f` receives a copy of `a` and zeroes the copy; `a` stays 7. `g` receives a reference to `b` and zeroes `b` itself. Output `70`.

4. B `const&` avoids the million-element copy and promises not to modify. A non-const reference would work but lies about intent and rejects temporaries; either by-value form copies.

5. C `int` to `long` and `int` to `double` are both standard conversions of equal rank; there is no `int` overload to be an exact match, so the compiler refuses to choose. Declaration order is irrelevant.

6. C A default is specified once, on the first declaration the calls see. Repeating it on the definition is an error, and putting it only on a definition that follows the calls means they never see it.

7. A Ten frames deep, each multiplying by 2 on the way back up: 2^10 = 1024. The base case `e == 0` is reached because `e` decreases by one per call.

8. B The key is a pair and the space is sparse and includes negatives, so a dense vector is impossible and a map keyed by the pair stores exactly the results computed. A single `static` holds one value, not a table.

9. A The lambda owns a `mutable` copy of `base` that persists between calls: 11 after the first call, 12 after the second. `main`'s `base` is never touched and is still 10.

10. A `a > b` is a strict weak ordering that puts larger values first. `a >= b` and `b <= a` return `true` for equal elements and are undefined behaviour with `std::sort`; `a < b` sorts ascending.

11. B The initialiser runs exactly once, on first reach, and C++11 guarantees that first run is thread-safe. Afterwards the variable keeps whatever the last call left in it.

12. B An anonymous namespace makes every member private to the translation unit. A plain non-`const` global and an `inline` variable have external linkage; `extern` declares an external one defined elsewhere.

13. B `[&total]` stores a reference, not the object. `total` has automatic storage and dies with its frame; the returned closure then refers to a dead object. Capture by value to make the lambda self-contained.

14. B It flags functions whose return value is the point of the call. `std::vector::empty()` carries it since C++20 so that `v.empty();` is caught as a probable `clear()`.
