Module 6 of 20

Arrays, pointers and references

Built-in arrays and std::array, pointers and nullptr, pointer arithmetic and the [begin, end) range, references and dangling, const correctness read right to left, and 2-D grids with a neighbour loop.

Every container, every string and every reference in C++ is built on the ideas in this module: a block of elements laid out one after another, an address that names one of them, and an alias that stands for an object without copying it. Learn how a built-in array decays into a pointer and why `std::array` was invented; what `&` and `*` do and what `nullptr` means; how `p + i`, `q - p` and the half-open range `[begin, end)` describe a sequence; and how a reference differs from a pointer in exactly three ways.

These are the parts of C++ that interviews probe hardest and that undefined behaviour hides in most often — an index past the end, a null dereference, a reference that outlives its object. The module teaches the discipline that keeps a program well-defined: initialise every array and pointer, check before dereferencing, form the one-past-the-end pointer but never read it, and let `const` — read right to left — say who may write what. It ends with grids, where two indices, row-major layout and a neighbour loop with a bounds check turn every board and matrix problem into the same idiom.

The exercises print values, indices and differences, never an address: reverse a `std::array` in place, pass a decayed array with its length, swap through pointers and through references, sum a `[begin, end)` range, write `strlen` by hand, edit a vector through a returned reference, write const-correct signatures, and read and walk grids. The checkpoint asks for a pointer walk over runs of equal values, a flood count on a character grid and a statistics function with reference out-parameters.

CONTENTS

1	Built-in arrays and std::array	14 min
2	Pointers — addresses, nullptr and the arrow	14 min
3	Pointer arithmetic and arrays	14 min
4	References — an alias with a rulebook	14 min
5	Const correctness — reading declarations right to left	13 min

6	Multidimensional data — grids, layout and neighbours	15 min
✓	Checkpoint — Arrays, pointers and references	25 min

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

LESSON 1 · 14 MIN

Built-in arrays and std::array

An array is a fixed number of elements of one type stored one after another in memory. C++ inherited the built-in array from C, and with it three properties that shape the whole language: the size is fixed at compile time, nothing checks that an index is in range, and the array's name quietly turns into a pointer the moment you pass it around. This lesson settles what a built-in array really is, where it bites, and why `std::array<T, N>` — the same contiguous block with value semantics and a `.size()` — is what modern code writes instead.

Declaring and initialising

```
int scores[5];                    // five ints; automatic storage, NOT initial
int primes[5] = {2, 3, 5, 7, 11}; // aggregate initialisation
int zeros[5] = {};                // every element value-initialised to 0
int partial[5] = {1, 2};          // 1, 2, 0, 0, 0 — the rest are zeroed
int deduced[] = {4, 8, 15, 16};   // size deduced from the initialiser: 4
```

The size in brackets must be a constant expression: a literal, a `constexpr int`, an enumerator. `int n; std::cin >> n; int a[n];` is a *variable-length array* — a C99 feature that GCC accepts as an extension and standard C++ does not have; Clang warns and other compilers reject it. When the size is only known at run time the answer is `std::vector` (Module 13, STL containers).

The first line above is the dangerous one: a local built-in array with no initialiser holds whatever bytes were on the stack, and reading them is undefined behaviour. `int scores[5] = {};` costs nothing; write the braces every time.

Indexing, and what the language does not check

```
int a[4] = {10, 20, 30, 40};
std::cout << a[0] << ' ' << a[3] << '\n';   // 10 40
a[4] = 50;                                    // undefined behaviour: there is
```

Indices run from `0` to `N - 1`. There is no check: `a[4]` writes four bytes past the end of `a` into whatever happens to be there — another local, the return address, nothing you can predict —

and under `-O2` the optimiser is entitled to assume it never happens and delete code around it. The array does not know its own end. You have to.

To ask an array its element count, use `std::size` from `<iterator>` (C++17). It works only while the name still denotes an array, which turns out to be a feature:

```
#include <iterator>
int a[4] = {10, 20, 30, 40};
for (std::size_t i = 0; i < std::size(a); ++i) std::cout << a[i] << ' ';
for (int x : a) std::cout << x << ' ';        // range-for knows the bounds to
```

The older `sizeof a / sizeof a[0]` gives the same number, and a silent wrong answer after decay; prefer `std::size`.

Decay: the array is not a value

A built-in array is not a first-class value. It cannot be assigned (`b = a;` is a compile error), cannot be returned from a function by value, and cannot be compared with `==`. Worse, in almost every expression the array's name **decays** into a pointer to its first element:

```
void show(int values[], int n) {          // exactly the same as void show(int
    std::cout << sizeof values << '\n';   // 8 on this platform — the size of
}

int a[4] = {1, 2, 3, 4};
show(a, 4);                               // a decays to &a[0]; the 4 has to
```

`int values[]` and `int values[4]` in a parameter list are both just `int*` — the number is ignored. That is why every C-style function that takes an array also takes a length, and why `std::size(values)` inside `show` is a compile error rather than a wrong answer: a pointer has no size to give.

Comparing two arrays with `==` compiles because both decay: it compares two addresses and is always `false` for distinct arrays (C++20 deprecates it and `-Wall` warns). If the size must survive a call, pass the array by reference — `void show(const int (&values)[4])` — or, better, stop using built-in arrays for anything you pass around.

---

std::array<T, N>: the same block, done properly

```cpp
#include <array>

std::array<int, 5> a{2, 3, 5, 7, 11};
std::array<int, 5> z{};                  // all zeros
std::cout << a.size() << ' ' << a[2] << ' ' << a.front() << ' ' << a.back() <<
a.at(2) = 6;                             // bounds-checked: at(7) throws std::
a.fill(0);                               // every element 0

std::array<int, 5> b = a;                // a real copy — value semantics
b[0] = 99;                               // a[0] is unchanged
bool same = (a == b);                    // element-wise comparison: false
```

`std::array` is a struct wrapping one built-in array: the same layout, the same zero overhead, the same fixed size — and it fixes everything above. It is copied and assigned like an `int`, compared element by element, passed by (const) reference without losing its size, and returned by value. `.at()` gives a checked access when the index came from outside; `[]` is still unchecked, exactly as fast as the built-in and exactly as undefined out of range.

```cpp
double mean(const std::array<int, 5>& values) {   // by const reference: no co
    long long sum = 0;
    for (int x : values) sum += x;
    return static_cast<double>(sum) / values.size();
}
```

The one thing `std::array` shares with the built-in is uninitialised elements when you write `std::array<int, 5> a;` with no braces. Write `a{}`.

	Built-in `T a[N]`	`std::array<T, N>`	`std::vector<T>`
Size	compile-time, fixed	compile-time, fixed	run-time, growable
Knows its size	only via `std::size`, before decay	`.size()`	`.size()`
Copy, assign, `==`	no (decays)	yes, element-wise	yes, element-wise
Bounds check	none	`.at()`	`.at()`
Storage	inline (the stack, for a local)	inline	the heap, through a handle

Two-dimensional built-in arrays, in brief

```
int grid[3][4] = {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}};
std::cout << grid[1][2] << '\n';   // 7: row 1, column 2
```

A 2-D built-in array is an array of arrays, laid out row after row in one block (row-major). Passing one to a function needs every dimension but the first — `void show(int g[][4], int rows)` — so the compiler can compute `g[r][c]`. Lesson 6 covers grids properly.

Pitfalls

Off by one. `for (int i = 0; i ⩽ N; ++i) a[i]` touches `a[N]`. Half-open ranges (`i < N`) are the habit that prevents it.

`sizeof` after decay. `sizeof values / sizeof values[0]` inside a function taking `int values[]` is `8 / 4 = 2`, always. Pass the length or use `std::array`.

Uninitialised elements. `int a[5];` and `std::array<int, 5> a;` both hold garbage. `= {}` and `{}`.

Returning a local array. `int* f() { int a[3] = {1, 2, 3}; return a; }` returns a pointer to storage that no longer exists — a dangling pointer (lesson 4 and Module 7, Memory, ownership and RAII).

Key takeaways

A built-in array is a fixed-size contiguous block with no bounds check; out-of-range access is undefined behaviour, not an exception.

Always initialise: `int a[N] = {};`.

An array's name decays to a pointer to its first element when passed; the callee loses the size, so pass it separately.

`std::array<T, N>` keeps the layout and the speed and adds value semantics, `.size()`, `.at()` and `==`.

Fixed size at compile time: `std::array`. Run-time size: `std::vector`. A raw `T a[N]` only for the smallest local scratch.

EXERCISE 1

Reverse in place

Read `n` (1 ≤ n ≤ 50) and then `n` integers into a `std::array<int, 50>`, keeping `n` as the count of elements in use. Reverse the first `n` elements **in place** — two indices walking towards each other, swapping with `std::swap` — and print them space-separated on one line. Do not build a second array.

Input: `n`, then `n` integers (whitespace-separated, possibly across several lines). **Output:** the `n` integers in reverse order on one line.

Example: input `5 1 2 3 4 5` → output `5 4 3 2 1`

INPUT 1

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

OUTPUT 1

```
5 4 3 2 1
```

INPUT 2

```
1
42
```

OUTPUT 2

```
42
```

REFERENCE SOLUTION

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

int main() {
    std::array<int, 50> values{};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::cin >> values[i];
    }
    int lo = 0;
    int hi = n - 1;
    while (lo < hi) {
        std::swap(values[lo], values[hi]);
        ++lo;
        --hi;
    }
    for (int i = 0; i < n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
    return 0;
}
```

EXERCISE 2

Above the average

Read `n` (1 ≤ n ≤ 100) integers into a built-in array `int values[100]`. Write `double average(const int* values, int n)` — the array **decays** to a pointer when you pass it, so the count travels alongside — and use it to print the mean to two decimals and how many values are strictly greater than the mean.

Input: `n`, then `n` integers. **Output:** two lines, `average=<mean with 2 decimals>` and `above=<count>`.

Example: input `4 10 20 30 41` →

```
average=25.25
above=2
```

INPUT 1

```
4
10 20 30 41
```

OUTPUT 1

```
average=25.25
above=2
```

INPUT 2

```
3
5 5 5
```

OUTPUT 2

```
average=5.00
above=0
```

REFERENCE SOLUTION

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

// The array decays to a pointer here: the function needs n to know where it ends.
double average(const int* values, int n) {
    long long sum = 0;
    for (int i = 0; i < n; ++i) {
        sum += values[i];
    }
    return static_cast<double>(sum) / n;
}

int main() {
    int values[100] = {};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::cin >> values[i];
    }
    double avg = average(values, n);
    int above = 0;
    for (int i = 0; i < n; ++i) {
        if (values[i] > avg) ++above;
    }
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "average=" << avg << '\n';
    std::cout << "above=" << above << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
int a[5] = {1, 2};
std::cout << a[1] + a[4] << '\n';
```

A. `2`

B. `3`

C. Undefined behaviour — `a[4]` is uninitialised

D. Compile error — too few initialisers

2. Which of these is **not** standard C++20?

A. `int a[3] = {1, 2, 3};`

B. `constexpr int n = 3; int a[n];`

C. `int n; std::cin >> n; int a[n];`

D. `int a[] = {1, 2, 3};`

3. Inside `void f(int values[])`, what is `sizeof values` on this platform?

A. The number of bytes in the caller's array

B. 8 — `values` is a pointer

C. 4 — the size of one `int`

D. A compile error

4. `std::array<int, 4> a; std::cout << a[0];` — what happens?

A. Prints `0`

B. Reads an indeterminate value — undefined behaviour

C. Compile error: `a` is not initialised

D. Throws `std::out_of_range`

5. Which comparison compares the **elements**?

A. `int a[3] = {}; int b[3] = {}; a == b`

B. `std::array<int, 3> a{}; std::array<int, 3> b{}; a == b`

C. Both of them

D. Neither — arrays cannot be compared

6. Which parameter declaration keeps the array's size known inside the function?

A. `void f(int a[5])`

B. `void f(int* a)`

C. `void f(int (&a)[5])`

D. `void f(int a[])`

7. What does this print?

```
std::array<int, 3> a{1, 2, 3};
auto b = a;
b[0] = 9;
std::cout << a[0] << b[0] << '\n';
```

A. `19`

B. `99`

C. `11`

D. Compile error — arrays cannot be copied

ANSWER KEY · BUILT-IN ARRAYS AND STD::ARRAY

1. A Aggregate initialisation with fewer values than elements value-initialises the rest, so `a[2]`, `a[3]` and `a[4]` are 0 and `a[1] + a[4]` is `2 + 0`. Only an array with **no** initialiser at all holds indeterminate values.

2. C An array bound must be a constant expression. `int a[n]` with a run-time `n` is a variable-length array — a GCC extension that Clang warns about and other compilers reject. The answer for a run-time size is `std::vector<int> a(n)`.

3. B An array parameter is a pointer parameter in disguise: `int values[]` is `int*`, and `sizeof` a pointer is 8 on x86-64. The caller's array size is gone; it must be passed separately.

4. B `std::array` wraps a built-in array and inherits its rule: with no initialiser the elements are not initialised. Write `std::array<int, 4> a{};` to zero them. `[]` never checks bounds or initialisation; only `.at()` checks bounds.

5. B Built-in arrays decay in `a == b`, so it compares two addresses (always false for distinct arrays; C++20 deprecates it and `-Wall` warns). `std::array` defines `==` element-wise, which is one of the reasons it exists.

6. C A reference to an array of 5 keeps the size in the type and rejects an array of any other size. The other three are all exactly `int*`; the `5` in `int a[5]` is ignored by the compiler.

7. A `std::array` has value semantics: `auto b = a` makes an independent copy, so writing `b[0]` leaves `a[0]` at 1. A built-in array could not be copied with `=` at all.

LESSON 2 · 14 MIN

Pointers — addresses, nullptr and the arrow

Every object in a running C++ program sits at an address, and a pointer is a variable that holds one. That single idea is underneath half of the track: references are a disciplined pointer, iterators a generalised one, `std::vector` a pointer plus a size, and dynamic memory (Module 7, Memory, ownership and RAII) pointers with an ownership contract. This lesson settles the two operators, what a null pointer is, why you would want a pointer at all, and the one rule — check before you dereference — that keeps a program out of undefined behaviour.

Address-of and dereference

```
int x = 42;
int* p = &x;      // &x is the address of x; p now "points at" x
*p = 7;           // *p is the object p points at — this writes x
std::cout << x << ' ' << *p << '\n';   // 7 7
```

`&` applied to an object gives its address; `*` applied to a pointer gives the object at that address — an lvalue you can read or assign. The type `int*` is "pointer to `int`": it holds the address of an `int` and nothing else, so `double d; int* p = &d;` does not compile.

Read a declaration from the name outwards: `int* p` — *p is a pointer to int*. The asterisk binds to the name, not the type, which produces the oldest trap in the language:

```
int* a, b;        // a is int*, b is int — the * belongs to a only
int *c, *d;       // both pointers, C style
int* e = nullptr; // one declaration per line: no ambiguity
```

What a pointer's value is

A pointer holds an address: 8 bytes on this platform, whatever it points at. `std::cout << p` prints it in hexadecimal, and a judged program must never do so — the value changes between runs, machines and compilers. Two pointers compare equal exactly when they point at the same object, and *that* is the meaningful thing to print:

```cpp
int a = 1, b = 1;
int* p = &a;
int* q = &b;
std::cout << (*p == *q) << ' ' << (p == q) << '\n';   // 1 0 — same value, dij
q = &a;
std::cout << (p == q) << '\n';                        // 1 — now the same obje
```

The line `q = &a;` is **reseating**: a pointer variable can be assigned a new address at any time. Assigning to `q` changes where `q` points; assigning to `*q` changes the object it points at. Keeping those two apart is the whole skill.

---

nullptr

A pointer that points at nothing holds the null pointer value, spelled `nullptr` (C++11). `NULL` and `0` are the C spellings and still compile; `nullptr` has its own type and cannot be mistaken for an integer.

```cpp
int* p = nullptr;
if (p) { /* not taken */ }
if (p == nullptr) std::cout << "nothing to read\n";
```

A pointer converts to `bool` — `nullptr` is `false`, anything else is `true` — so `if (p)` is the idiom for "do I have an object?". Dereferencing a null pointer is undefined behaviour: usually a segmentation fault on this platform, but the optimiser is allowed to assume it never happens and may delete a null check that comes *after* a dereference it has already seen. The rule is mechanical: **check before you dereference, never after.**

A pointer declared without an initialiser is worse than null: it holds an indeterminate value, and even reading it — not dereferencing, just `p == nullptr` — is undefined behaviour. `int* p = nullptr;` or `int* p = &x;`. Never a bare `int* p;`.

Pointers to class objects: the arrow

```cpp
struct Point { int x; int y; };

Point pt{3, 4};
Point* pp = &pt;
(*pp).x = 10;      // dereference, then member — the parentheses are required
pp->y = 20;        // the same thing, spelled the way everyone spells it
std::cout << pt.x << ' ' << pt.y << '\n';   // 10 20
```

`a->b` is `(*a).b`. The parentheses matter because `.` binds tighter than `*`: `*pp.x` means `*(pp.x)`, and a pointer has no member `x`, so it does not compile. You will write `->` constantly with `this` (Module 8, Classes and objects) and with smart pointers (Module 7).

Pointer to pointer

A pointer is an object too, so it has an address:

```cpp
int x = 1;
int* p = &x;
int** pp = &p;     // pointer to pointer to int
**pp = 5;          // x is 5
*pp = nullptr;     // p is now null; x is untouched
```

Two stars, two hops. In practice `int**` appears as `char** argv` in `main`'s full signature (an array of C strings, lesson 3) and in C APIs that hand a pointer back through an output parameter.

Why a pointer at all?

References (lesson 4) are the default way to refer to another object, so a pointer needs a reason. There are four good ones:

The object may be absent. `void log(const Config* cfg)` can be called with `nullptr`; there is no null reference.

You need to reseat. A cursor, a "current" element, a parent link — anything that points at different objects over time.

Arrays and C APIs. A decayed array is a pointer, `argv` is a pointer, and every C library speaks pointers.

Dynamic memory. Objects created with `new` are reached only through pointers — wrapped in `std::unique_ptr` (Module 7) in modern code so that ownership is enforced.

If none of those applies, use a reference or a value.

Pointers as parameters

```
void swap_values(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int x = 1, y = 2;
swap_values(&x, &y);     // the & at the call site announces "this may be modi
```

The callee reaches back into the caller's variables through the addresses. Compare the reference version in lesson 4, where the call site reads `swap_values(x, y)` and nothing marks the mutation: some codebases prefer pointers for out-parameters for exactly that visibility, the C++ Core Guidelines prefer references and a name that says what happens.

A function may also *return* a pointer, and returning `nullptr` is the natural way to say "not found":

```
int* largest(int* values, int n) {
    if (n == 0) return nullptr;
    int* best = &values[0];
    for (int i = 1; i < n; ++i) {
        if (values[i] > *best) best = &values[i];
    }
    return best;
}

int* p = largest(scores, count);
if (p) *p = 0;              // check, then use
```

`best` is reseated each time a larger element appears; the caller gets the winner's address and may read or write through it — but must not keep `p` after `scores` is gone, because a pointer does not keep its object alive (Module 7).

Pitfalls

Mistake	What happens
`int* a, b;`	`b` is an `int`; later `*b` fails to compile with a confusing message
`int* p;` then `if (p)`	reading an indeterminate pointer — undefined behaviour
`*p` when `p` may be null	undefined behaviour; a crash if you are lucky
Testing `*p` before testing `p`	the dereference happens first; put the null check first
Returning `&local` from a function	the local is gone when the function returns — dangling

Key takeaways

- `&obj` is an address; `*ptr` is the object at that address; `ptr->m` is `(*ptr).m`.
- A pointer's *value* is an address you must never print in a judged program; compare pointers with `==` and print what they reach.
- `nullptr` means "no object"; check with `if (p)` before every dereference, and initialise every pointer.
- Assigning to `p` reseats it; assigning to `*p` writes the pointee.
- Use a pointer when the object may be absent, must be reseated, is a C array or is dynamically allocated; otherwise use a reference or a value.

EXERCISE 1

Swap, then reseat

Read three integers `a`, `b` and `k`. Write `void swap_values(int* a, int* b)` that exchanges the two integers through pointers, and call it with `&a` and `&b`. Then declare one pointer `int* target` that points at `a` when `k` is odd and at `b` when `k` is even, and add `k` to whichever integer it points at, **through the pointer**. Print the state after each step. Never print a pointer's value.

Input: one line with `a b k`. **Output:** three lines: `before: a=<a> b=<b>`, `swapped: a=<a> b=<b>`, `final: a=<a> b=<b>`.

Example: input `3 9 4` →

```
before: a=3 b=9
swapped: a=9 b=3
final: a=9 b=7
```

INPUT 1

```
3 9 4
```

OUTPUT 1

```
before: a=3 b=9
swapped: a=9 b=3
final: a=9 b=7
```

INPUT 2

```
-1 -1 1
```

OUTPUT 2

```
before: a=-1 b=-1
swapped: a=-1 b=-1
final: a=0 b=-1
```

REFERENCE SOLUTION

```cpp
#include <iostream>

void swap_values(int* a, int* b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}

int main() {
    int a = 0;
    int b = 0;
    int k = 0;
    std::cin >> a >> b >> k;
    std::cout << "before: a=" << a << " b=" << b << '\n';
    swap_values(&a, &b);
    std::cout << "swapped: a=" << a << " b=" << b << '\n';
    int* target = (k % 2 == 0) ? &a : &b;
    *target += k;
    std::cout << "final: a=" << a << " b=" << b << '\n';
    return 0;
}
```

EXERCISE 2

Largest, then zeroed

Read `n` (0 ≤ n ≤ 100) integers into `int values[100]`. Write `int* largest(int* values, int n)` returning a pointer to the **first** largest element, or `nullptr` when `n` is 0. In `main`, check the pointer before using it: print `empty` if it is null; otherwise print `largest=<value>` (reached through the pointer), set that element to 0 **through the pointer**, and print the array.

Input: `n`, then `n` integers. **Output:** `empty`, or two lines: `largest=<value>` and the `n` integers space-separated.

Example: input `5 3 8 2 8 1` →

```
largest=8
3 0 2 8 1
```

INPUT 1

```
5
3 8 2 8 1
```

OUTPUT 1

```
largest=8
3 0 2 8 1
```

INPUT 2

```
0
```

OUTPUT 2

```
empty
```

REFERENCE SOLUTION

```cpp
#include <iostream>

int* largest(int* values, int n) {
    if (n == 0) return nullptr;
    int* best = &values[0];
    for (int i = 1; i < n; ++i) {
        if (values[i] > *best) best = &values[i];
    }
    return best;
}

int main() {
    int values[100] = {};
    int n = 0;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::cin >> values[i];
    }
    int* p = largest(values, n);
    if (!p) {
        std::cout << "empty\n";
        return 0;
    }
    std::cout << "largest=" << *p << '\n';
    *p = 0;
    for (int i = 0; i < n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
int x = 3;
int* p = &x;
*p = *p + 1;
int y = *p;
std::cout << x << ' ' << y << '\n';
```

A. `3 4`

B. `4 4`

C. `4 3`

D. An address, then `4`

2. What does `int* a, b;` declare?

A. Two pointers to `int`

B. A pointer to `int` and an `int`

C. A compile error

D. An `int` and a pointer to `int`

3. Which of these is undefined behaviour?

A. `int* p = nullptr; if (p) *p = 1;`

B. `int* p = nullptr; std::cout << *p;`

C. `int x = 0; int* p = &x; *p = 5; std::cout << x;`

D. `int* p = nullptr; p = nullptr;`

4. For a `Point* pp`, `pp→x` is the same as…

A. `*pp.x`

B. `(*pp).x`

C. `&pp.x`

D. `pp.x`

5. When is a pointer parameter the right choice over a reference parameter?

A. Whenever the object is large

B. When the argument may legitimately be absent (`nullptr`) or the function must reseat it

C. Whenever the function modifies the argument

D. Never — references replaced pointers in modern C++

6. What does this print?

```
int a = 1, b = 2;
int* p = &a;
int* q = &b;
q = p;
*q = 10;
std::cout << a << ' ' << b << ' ' << (p == q) << '\n';
```

A. `1 10 1`

B. `10 2 1`

C. `10 10 1`

D. `10 2 0`

7. `int* p; std::cout << (p == nullptr);` — what happens?

A. Prints `1`

B. Prints `0`

C. Undefined behaviour — `p` holds an indeterminate value

D. Compile error

ANSWER KEY · POINTERS — ADDRESSES, NULLPTR AND THE ARROW

1. B `*p = *p + 1` writes through the pointer, so `x` becomes 4; `y` then copies that 4. Nothing here prints `p` itself, so no address appears.

2. B The `*` belongs to the declarator `a`, not to the type: `a` is `int*`, `b` is a plain `int`. Declare one pointer per line to avoid the trap.

3. B Dereferencing a null pointer is undefined behaviour. The first option checks before dereferencing, the third writes through a valid pointer, and the last merely assigns null twice.

4. B `→` dereferences and then selects the member. `*pp.x` parses as `*(pp.x)` because `.` binds tighter than `*`, and a pointer has no member `x`, so it does not compile.

5. B A reference cannot be null and cannot be reseated, so those two needs call for a pointer. Size and mutability are handled by `const T&` and `T&` respectively.

6. B `q = p` reseats `q` to point at `a`, so `*q = 10` writes `a`; `b` is untouched. Both pointers now hold the same address, so `p == q` prints `1`.

7. C A pointer declared without an initialiser is not null; it holds garbage, and even reading it is undefined behaviour. Initialise every pointer: `int* p = nullptr;`.

LESSON 3 · 14 MIN

Pointer arithmetic and arrays

A pointer into an array can be moved. `p + 1` is the next element, `q - p` is how many elements apart two pointers are, and `p[i]` is nothing more than `*(p + i)`. That is how arrays and pointers are the same thing at the machine level, how every C string is walked, and — through `begin` and `end` — how the entire standard library expresses "a range of elements". This lesson settles the arithmetic, the half-open range convention, the one pointer you may form but never dereference, and the C string.

---

p + i moves by elements

```
int a[5] = {10, 20, 30, 40, 50};
int* p = a;                          // decay: p points at a[0]
std::cout << *(p + 2) << '\n';       // 30
++p;                                 // p now points at a[1]
std::cout << *p << ' ' << p[1] << '\n';   // 20 30
```

Adding `i` to a `T*` moves it `i` *elements*, not `i` bytes: the compiler multiplies by `sizeof(T)` for you. `p + 2` on an `int*` advances 8 bytes; on a `double*` 16; on a `char*` 2. `++p`, `--p`, `p += n` and `p - n` all work the same way.

The subscript operator is defined in terms of this: `p[i]` **is** `*(p + i)`, for pointers and built-in arrays alike. So `a[3]` is `*(a + 3)` — and, because addition commutes, `3[a]` compiles too. Do not write it; do understand why it works.

The difference of two pointers

```
#include <cstddef>
int* first = a;
int* third = a + 2;
std::ptrdiff_t gap = third - first;   // 2 — elements, not bytes
```

Subtracting two pointers into the same array gives the number of elements between them as a `std::ptrdiff_t` — a signed 8-byte integer on this platform. It is negative if the left

operand comes first. The practical use is recovering an **index** from a pointer: if `p` points into `a`, `p - a` is its index — what you print when an exercise asks "where", never the address.

Subtracting pointers into two different arrays is undefined behaviour, even though the compiler will happily emit the subtraction.

The half-open range [begin, end)

```
int* begin = a;
int* end = a + 5;                  // one past the last element
for (int* p = begin; p ≠ end; ++p) std::cout << *p << ' ';
```

`a + 5` does not point at an element — there is no `a[5]` — but the language guarantees you may **form** the one-past-the-end pointer, compare with it and subtract it. You may not dereference it, and you may not go further: `a + 6` is undefined behaviour before you even read it, and so is `a - 1`.

The pair `[begin, end)` — first element included, `end` excluded — is the convention for describing a sequence in C++. Its properties are why:

- The length is `end - begin`, with no `+ 1`.
- An empty range is `begin == end`, and a loop `while (p ≠ end)` runs zero times without a special case.
- Splitting a range at `mid` gives `[begin, mid)` and `[mid, end)` with nothing lost or counted twice.

`std::begin(a)` and `std::end(a)` (from `<iterator>`) give exactly these two pointers for a built-in array, and iterators that behave the same way for a `std::vector`. Every algorithm in `<algorithm>` (Module 14, Algorithms and lambdas) takes such a pair; a function taking `(const int* begin, const int* end)` *is* an algorithm:

```
long long sum(const int* begin, const int* end) {
    long long total = 0;
    for (const int* p = begin; p != end; ++p) total += *p;
    return total;
}

const int* find_negative(const int* begin, const int* end) {
    for (const int* p = begin; p != end; ++p) {
        if (*p < 0) return p;
    }
    return end;                                    // "not found" is the end po
}
```

Returning `end` for "not found" is the library's convention too (`std::find`), and the caller tests `if (it != end)`. `const int*` because neither function modifies an element — lesson 5.

Passing an array with its length

Because an array decays, a function that receives one must be told the size. Two spellings:

```
void show(const int* values, std::size_t count);    // pointer + count
void show(const int* begin, const int* end);         // a range

int a[4] = {1, 2, 3, 4};
show(a, std::size(a));
show(a, a + std::size(a));
std::vector<int> v{1, 2, 3};
show(v.data(), v.size());                            // a vector hands out the
```

`v.data()` is a pointer to the vector's first element and `v.data() + v.size()` its end; for an empty vector `data()` may be null, and adding zero to a null pointer is defined, so the range is still empty and valid. C++20's `std::span<const int>` (Module 16, Modern C++) carries pointer and count as one parameter.

Ordering

Pointers into the same array order like their elements, so `<` works: `while (lo < hi)` is Module 3's two-pointer sweep with pointers instead of indices:

```
void reverse(int* lo, int* hi) {      // hi is one past the end
    while (lo < hi) {
        --hi;                         // step inside the range before touching
        int tmp = *lo;
        *lo = *hi;
        *hi = tmp;
        ++lo;
    }
}
```

C strings and the terminating '\0'

Before `std::string` there was the C string: an array of `char` whose end is marked by a `'\0'` character (the byte 0) rather than by a stored length. A string literal is such an array:

```
const char* s = "hello";   // points at an array of 6 chars: h e l l o \0
```

Every C string function walks until it finds the terminator. `strlen` from `<cstring>` is, in essence:

```
std::size_t my_strlen(const char* s) {
    const char* p = s;
    while (*p != '\0') ++p;
    return static_cast<std::size_t>(p - s);   // elements walked = characters
}
```

The loop condition can be written `while (*p)` because `'\0'` is the only false `char`. Reading a line into a fixed buffer safely uses `std::cin.getline(buf, sizeof buf)`, which stops before the buffer overflows and always writes the terminator:

```
char buf[256] = {};
std::cin.getline(buf, sizeof buf);
std::cout << my_strlen(buf) << '\n';
```

A `std::string` (Module 5, Strings) does all of this for you and knows its length; `s.c_str()` hands you the terminated array when a C API needs one. Write C strings only when you are talking to C.

Pitfalls

Dereferencing `end`. `*end`, `end[0]`, `for (p = begin; p ⩽ end; ++p)` — all read past the array.

Forming a pointer outside `[a, a + N]`. `a + N + 1` and `a - 1` are undefined even unread.

Subtracting unrelated pointers. Only pointers into the same array.

A missing terminator. `char name[5] = "hello";` does not fit the `'\0'` — a compile error in C++ — and `char name[5] = {'h', 'e', 'l', 'l', 'o'};` compiles with no terminator, so `strlen(name)` runs off the end.

`sizeof` on the pointer. `sizeof s` for a `const char* s` is 8, never the length.

Key takeaways

`p + i` moves by `i` elements; `p[i]` is `*(p + i)`; `q - p` counts elements and is how you turn a pointer back into an index.

`[begin, end)` is the range convention: `end` is one past the last element — form it, compare with it, never dereference it.

Empty range: `begin == end`; length: `end - begin`; "not found": return `end`.

A decayed array needs its length passed alongside; `std::vector::data()` gives the same kind of pointer.

A C string ends at `'\0'`; walking to it is `strlen`, and `std::string` exists so you rarely have to.

EXERCISE 1

Sum and find over [begin, end)

Read `n` (0 ≤ n ≤ 1000) integers into a `std::vector<int>` and form the range `[begin, end)` from `data()` and `data() + size()`. Write `long long sum(const int* begin, const int* end)` and `const int* find_negative(const int* begin, const int* end)` — the second returns a pointer to the first negative value, or `end` when there is none. Print the sum, then the **index** of the first negative value computed as a pointer difference, or `-1`.

Input: `n`, then `n` integers. **Output:** two lines, `sum=<total>` and `first_negative=<index or -1>`.

Example: input `4 3 -1 2 -5` →

```
sum=-1
first_negative=1
```

INPUT 1

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

OUTPUT 1

```
sum=15
first_negative=-1
```

INPUT 2

```
4
3 -1 2 -5
```

OUTPUT 2

```
sum=-1
first_negative=1
```

REFERENCE SOLUTION

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

long long sum(const int* begin, const int* end) {
    long long total = 0;
    for (const int* p = begin; p != end; ++p) {
        total += *p;
    }
    return total;
}

const int* find_negative(const int* begin, const int* end) {
    for (const int* p = begin; p != end; ++p) {
        if (*p < 0) return p;
    }
    return end;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    const int* begin = values.data();
    const int* end = values.data() + values.size();
    std::cout << "sum=" << sum(begin, end) << '\n';
    const int* neg = find_negative(begin, end);
    if (neg == end) {
        std::cout << "first_negative=-1\n";
    } else {
        std::cout << "first_negative=" << (neg - begin) << '\n';
    }
    return 0;
}
```

EXERCISE 2

strlen and reverse by hand

Read one line (at most 200 characters, possibly empty) into `char buf[256]` with `std::cin.getline(buf, sizeof buf)`. Write `std::size_t my_strlen(const char* s)` that walks a pointer to the terminating `'\0'` and returns the distance, and `void reverse_in_place(char* s)` that reverses the C string with two pointers moving towards each other. Do not use `<cstring>` or `std::string`.

Input: one line of text. **Output:** `length=<n>`, then `reversed=<the reversed text>`.

Example: input `hello` →

```
length=5
reversed=olleh
```

INPUT 1

```
hello
```

OUTPUT 1

```
length=5
reversed=olleh
```

INPUT 2

```
a b c
```

OUTPUT 2

```
length=5
reversed=c b a
```

REFERENCE SOLUTION

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

std::size_t my_strlen(const char* s) {
    const char* p = s;
    while (*p != '\0') ++p;
    return static_cast<std::size_t>(p - s);
}

void reverse_in_place(char* s) {
    std::size_t len = my_strlen(s);
    if (len < 2) return;
    char* lo = s;
    char* hi = s + len - 1;
    while (lo < hi) {
        char tmp = *lo;
        *lo = *hi;
        *hi = tmp;
        ++lo;
        --hi;
    }
}

int main() {
    char buf[256] = {};
    std::cin.getline(buf, sizeof buf);
    std::cout << "length=" << my_strlen(buf) << '\n';
    reverse_in_place(buf);
    std::cout << "reversed=" << buf << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
int a[4] = {10, 20, 30, 40};
int* p = a + 1;
std::cout << p[2] << ' ' << *(p - 1) << ' ' << (a + 3) - p << '\n';
```

A. `30 10 2`

B. `40 10 2`

C. `40 20 3`

D. `40 10 8`

2. Given `int* end = arr + n;`, which operation is well-defined?

A. `*end`

B. `p ≠ end` and `end - arr`

C. Forming `arr + n + 1`

D. `end[0]`

3. What is the type of `q - p` for two `int*` pointers into the same array?

A. `int`

B. `std::size_t`

C. `std::ptrdiff_t` — a signed integer

D. `long long`, always

4. For a `const char* p` pointing at a C string, `while (*p) ++p;` stops…

A. At the first space

B. At the terminating `'\0'`

C. Never — it is undefined behaviour

D. At the end of the line

5. `std::size_t len(const char* s) { return sizeof s; }` returns…

A. The string's length

B. 8 for every input

C. The length plus one for the terminator

D. A compile error

6. For a `double* d`, `d + 1` advances by…

A. 1 byte

B. 8 bytes — one `double`

C. 4 bytes

D. Whatever the optimiser chooses

7. Which pair of pointers may be subtracted with a well-defined result?

A. Two pointers into the same array (or one past its end)

B. Any two `int*`

C. A pointer and `nullptr`

D. Pointers to elements of two different arrays

ANSWER KEY · POINTER ARITHMETIC AND ARRAYS

1. B `p` points at `a[1]`, so `p[2]` is `a[3]` = 40 and `*(p - 1)` is `a[0]` = 10. `(a + 3) - p` is a difference in elements, 3 − 1 = 2 — never in bytes.

2. B The one-past-the-end pointer may be formed, compared and subtracted — that is what makes `[begin, end)` loops work. It may not be dereferenced (`*end`, `end[0]`), and going one further past it is undefined even without a read.

3. C Pointer subtraction yields `std::ptrdiff_t` from `<cstddef>`, signed because the result is negative when `q` comes before `p`. On this platform it is a `long`, but the name is what to write.

4. B `'\0'` is the only `char` that converts to `false`, so the loop walks exactly to the terminator. Spaces and newlines are non-zero and do not stop it.

5. B `s` is a pointer, and `sizeof` a pointer is 8 on x86-64 regardless of what it points at. Measuring a C string means walking to `'\0'`.

6. B Pointer arithmetic is in elements: the compiler multiplies by `sizeof(double)`, 8 on this platform. Moving one byte would need a `char*`.

7. A Subtraction is defined only within one array, including its one-past-the-end position. Different arrays, null pointers and unrelated objects give undefined behaviour even though the compiler emits the instruction.

LESSON 4 · 14 MIN

References — an alias with a rulebook

A reference is another name for an existing object. Write `int& r = x;` and from then on `r` *is* `x`: reading `r` reads `x`, assigning to `r` assigns to `x`, and `&r == &x`. The rest of the lesson is the rulebook that follows — must be initialised, cannot be reseated, cannot be null — the two places references matter most, parameters and return values, the failure mode every C++ programmer meets, the dangling reference, and the table that decides between a reference and a pointer.

An alias, not an object

```
int x = 5;
int& r = x;        // r is an alias for x
r = 7;             // x is now 7
++r;               // x is now 8
std::cout << x << ' ' << r << '\n';   // 8 8
int y = 100;
r = y;             // NOT a rebind: copies y's value into x — x is 100, r stil
std::cout << x << ' ' << (&r == &x) << '\n';   // 100 1
```

The line `r = y;` is the one that separates references from pointers. It does not make `r` refer to `y`; there is no syntax that does. Once bound, a reference names the same object for its whole life. The compiler may implement it as a hidden pointer, but the language treats it as the object itself: no `*`, no `→`, no arithmetic.

The rules, all consequences of "it is the object":

Must be initialised. `int& r;` is a compile error — a name for nothing is meaningless.

Cannot be reseated. Assignment goes through to the referent.

Cannot be null. A legitimately obtained reference always refers to an object; `int& r = *p;` with a null `p` is undefined behaviour, not a null reference.

No references to references, arrays of references or pointers to references. `&r` is the address of the referent, `&x`.

Reference parameters

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

int x = 1, y = 2;
swap_values(x, y);     // x is 2, y is 1 — no & at the call site
```

A `T&` parameter lets the callee modify the caller's object. Compared with the pointer version in lesson 2 there are no stars in the body and no ampersands at the call site — cleaner, and less visible, so the function's name has to say that its arguments change (`swap_values`, `sort_in_place`, `read_into`).

The read-only form is the workhorse of the language: `long long total(const std::vector<int>& values)` reads the vector without copying it and cannot modify it. Module 4 (Functions) gave the rule: by value when the type is small (`int`, `double`, a pointer) or you need your own copy; by `const T&` for anything larger that you only read; by `T&` when the function's job is to change it. `const T&` on a `std::vector` of a million elements costs eight bytes; by value costs a million copies.

---

const T& binds to temporaries

```
const int& r = 5;                    // OK: a temporary int holds 5; r refers
const std::string& s = "hello";      // OK: a temporary std::string is built
int& bad = 5;                        // error: a non-const lvalue reference c
```

A non-const `T&` must bind to an *lvalue* — a named object. A `const T&` may also bind to a temporary, and when the reference is a local the temporary lives as long as it does (lifetime extension). This is what lets `void greet(const std::string& name)` be called as `greet("Ada")`: a `std::string` is made for the call and destroyed after it. It is also why `void f(int& n)` cannot be called as `f(5)` — there is no `n` in the caller to modify. Module 9 (Copies, moves and the rule of five) introduces the third kind, `T&&`, which binds only to temporaries.

Returning a reference

```
int& at_index(std::vector<int>& v, std::size_t i) { return v[i]; }

std::vector<int> v{1, 2, 3};
at_index(v, 1) = 20;      // the call is an lvalue: assign through it — v is
```

A function that returns `T&` hands back an alias to something that already exists — an element of a container the caller passed, a member of `*this` (Module 8), a `static`. `std::vector::operator[]` and `std::map::operator[]` return references; that is why `v[i] = x` works. The one rule: **the object must outlive the call.**

Dangling references

```
int& broken() {
    int local = 3;
    return local;        // warning: reference to local variable 'local' retu
}
int& r = broken();       // r names an object that no longer exists: undefine
```

`local` is destroyed when `broken` returns; the reference outlives it and reads memory that now belongs to whichever function is called next. GCC and Clang warn — treat the warning as an error. A by-value parameter is a local too, so returning a reference to one is the same bug.

The second classic is subtler, because nothing is returned:

```
std::vector<int> v{1, 2, 3};
int& first = v[0];
v.push_back(4);          // may reallocate: the elements move to a new block,
first = 9;               // undefined behaviour if it did — first refers to fr
```

A reference (or pointer, or iterator) into a `std::vector` is valid only until the next operation that may move the elements: `push_back`, `insert`, `resize`, `reserve`. This is **invalidation**, and Module 13 lesson 6 gives the table per container. The habit: use a reference into a container before the container changes, or hold an index, which survives.

References in range-for

```
std::vector<int> v{1, 2, 3};
for (int x : v) x *= 2;                    // x is a copy; v is unchanged
for (int& x : v) x *= 2;                   // x aliases each element; v is {2,
for (const auto& x : v) std::cout << x;    // read-only, no copy — the defaul
```

The same rule extends to `auto`: `auto x = v[0];` copies, `auto& x = v[0];` aliases.

References versus pointers

	Reference `T&`	Pointer `T*`
Can be null	no	yes (`nullptr`)
Must be initialised	yes	should be; the language does not force it
Can be reseated	no	yes
Syntax at use	as the object itself	`*p` , `p→m`
Arithmetic	none	`p + i` , `q - p`
Mutation visible at the call site	no (`f(x)`)	yes (`f(&x)`)
Use it for	parameters, aliases, `operator[]` returns	optional values, cursors, arrays, owning dynamic memory

Default to a reference; reach for a pointer when the object may be absent, must be reseated, or is an array or dynamic memory.

Pitfalls

- **`auto` copies.** `auto x = v[0]; x = 5;` changes a copy. Write `auto&` .
- **Returning a reference to a local**, or to a parameter passed by value (also a local).
- **Holding a reference across `push_back` .** Re-read it afterwards, or use an index.
- **`const` on the reference, not the object.** `const int& r = x; x = 9;` — `r` sees 9; the `const` only forbids writing *through* `r` .

Key takeaways

A reference is an alias: bound once at initialisation, never reseated, never null.

Parameters: `T` for small or owned, `const T&` for read-only, `T&` for in-out; name the function so the mutation is visible.

`const T&` binds to temporaries (lifetime-extended when local); `T&` binds only to lvalues.

Return a reference only to something that outlives the call — never a local, and remember a `std::vector` can move its elements.

Reference by default; pointer when absence, reseating or arrays are involved.

EXERCISE 1

Sort three by reference

Read three integers. Write `void swap_values(int& a, int& b)` and `void sort3(int& a, int& b, int& c, int& swaps)` that puts the three into ascending order using **only** `swap_values` on pairs — compare `a,b`, then `b,c`, then `a,b` again — and counts every swap performed through the `swaps` in-out reference. Print the sorted values and the count.

Input: one line with three integers. **Output:** the three integers in ascending order on one line, then `swaps=<count>`.

Example: input `3 1 2` →

```
1 2 3
swaps=2
```

INPUT 1

```
3 1 2
```

OUTPUT 1

```
1 2 3
swaps=2
```

INPUT 2

```
5 5 1
```

OUTPUT 2

```
1 5 5
swaps=2
```

REFERENCE SOLUTION

```cpp
#include <iostream>

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

void sort3(int& a, int& b, int& c, int& swaps) {
    if (a > b) {
        swap_values(a, b);
        ++swaps;
    }
    if (b > c) {
        swap_values(b, c);
        ++swaps;
    }
    if (a > b) {
        swap_values(a, b);
        ++swaps;
    }
}

int main() {
    int a = 0, b = 0, c = 0;
    std::cin >> a >> b >> c;
    int swaps = 0;
    sort3(a, b, c, swaps);
    std::cout << a << ' ' << b << ' ' << c << '\n';
    std::cout << "swaps=" << swaps << '\n';
    return 0;
}
```

EXERCISE 2

Edit through a returned reference

Read `n` (1 ≤ n ≤ 100) integers into a `std::vector<int>`, then an index `i` (0 ≤ i < n) and a value `x`. Write `int& at_index(std::vector<int>& values, std::size_t i)` returning a reference to element `i`, and assign `x` **through the call**: `at_index(values, i) = x;`. Then write `void stats(const std::vector<int>& values, long long& sum, int& largest)` that fills its two out-parameters, and print the results.

Input: `n`, then `n` integers, then `i x`. **Output:** the vector after the assignment on one line, then `sum=<sum> largest=<largest>`.

Example: input `4 1 2 3 4 2 10` →

```
1 2 10 4
sum=17 largest=10
```

INPUT 1

```
4
1 2 3 4
2 10
```

OUTPUT 1

```
1 2 10 4
sum=17 largest=10
```

INPUT 2

```
1
-5
0 -9
```

OUTPUT 2

```
-9
sum=-9 largest=-9
```

REFERENCE SOLUTION

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

int& at_index(std::vector<int>& values, std::size_t i) {
    return values[i];
}

void stats(const std::vector<int>& values, long long& sum, int& largest) {
    sum = 0;
    largest = values.front();
    for (int x : values) {
        sum += x;
        if (x > largest) largest = x;
    }
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    std::size_t i = 0;
    int x = 0;
    std::cin >> i >> x;
    at_index(values, i) = x;
    for (std::size_t k = 0; k < values.size(); ++k) {
        if (k > 0) std::cout << ' ';
        std::cout << values[k];
    }
    std::cout << '\n';
    long long sum = 0;
    int largest = 0;
    stats(values, sum, largest);
    std::cout << "sum=" << sum << " largest=" << largest << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
int x = 1, y = 2;
int& r = x;
r = y;
r = 5;
std::cout << x << ' ' << y << '\n';
```

A. `5 5`

B. `5 2`

C. `1 5`

D. `2 5`

2. Which declaration compiles?

A. `int& r;`

B. `int& r = 5;`

C. `const int& r = 5;`

D. `int& r = nullptr;`

3. `int& f() { int v = 1; return v; }` — what is wrong?

A. Nothing; it returns 1

B. Compile error: locals cannot be returned

C. It returns a dangling reference — `v` is destroyed when `f` returns, so using the result is undefined behaviour

D. It returns 0 because `v` is reset

4.

```cpp
std::vector<int> v{1, 2, 3};
int& a = v[0];
v.push_back(4);
a = 9;
```

What is true of the last line?

A. `v[0]` is now 9

B. `v[0]` is still 1 — `a` was a copy

C. It is undefined behaviour if `push_back` reallocated, because `a` refers to freed memory

D. It does not compile

5. `for (auto x : v) x *= 2;` on a `std::vector<int> v` …

A. Doubles every element

B. Doubles nothing — `x` is a copy of each element

C. Is a compile error

D. Doubles only the first element

6. `void f(const std::string& s);` is called as `f("abc");`. What happens?

A. Compile error: a literal is not a `std::string`

B. A temporary `std::string` is constructed from the literal and `s` binds to it for the call

C. `s` refers to the literal's characters directly

D. Undefined behaviour

7. Which of these does a pointer allow that a reference does not?

A. Modifying the caller's object

B. Being null, being reseated, and arithmetic

C. Referring to an element of a `std::vector`

D. Being passed to a function

ANSWER KEY · REFERENCES — AN ALIAS WITH A RULEBOOK

1. B `r = y` does not rebind `r`; it copies `y`'s value into `x`. `r = 5` then writes `x` again. `y` is never written, so it stays 2.

2. C A reference must be initialised, a non-const lvalue reference cannot bind to a temporary, and `nullptr` is not an `int`. Only `const int&` may bind to the temporary holding 5, and it extends that temporary's life.

3. C `v` lives in `f`'s stack frame, which is gone after the return. The compiler warns ("reference to local variable returned") but still compiles. Return by value, or return a reference to something that outlives the call.

4. C `push_back` may move every element to a new block and free the old one; references, pointers and iterators into the vector are then invalid. Take the reference after the container stops changing, or hold an index.

5. B `auto x` copies each element into `x`; the write goes to the copy. `for (auto& x : v)` aliases the element and modifies the vector.

6. B A `const T&` parameter accepts a temporary, so the literal is converted to a `std::string` that lives until the call returns. A non-const `std::string&` parameter would reject the call.

7. B Those three are exactly the differences: a reference is never null, is bound once, and has no `+ i`. Both can alias a caller's object, a vector element, or be passed around.

LESSON 5 · 13 MIN

Const correctness — reading declarations right to left

`const` is the promise "this code does not modify that object", and the compiler enforces it. Placed on a parameter it tells the caller their object is safe; placed on a member function it tells a `const` object which operations are allowed; placed on a pointer it says whether you may change the pointee, the pointer, or neither. This lesson settles the three pointer placements and the right-to-left reading rule, the conversions `const` allows and forbids, why the property spreads through a codebase, and why `const_cast` is a warning sign rather than a tool.

Three placements, one reading rule

```
int x = 1, y = 2;

const int* a = &x;          // pointer to const int
int const* b = &x;          // the same type, written the other way round
int* const c = &x;          // const pointer to int
const int* const d = &x;    // const pointer to const int
```

Read each declaration from the name, right to left:

Declaration	Reads as	`*p = 5`	`p = &y`
`const int* p`	p is a pointer to a const int	error	ok
`int const* p`	the same	error	ok
`int* const p`	p is a const pointer to an int	ok	error
`const int* const p`	p is a const pointer to a const int	error	error

The rule that generates the table: `const` **left** of the `*` applies to what is pointed at; `const` **right** of the `*` applies to the pointer itself. `const int*` and `int const*` differ only in style; `const T*` is what most code writes. The same reading works for references — `const int& r` is "r is a reference to a const int" — and there is no `int& const`, because a reference cannot be reseated anyway.

`int* const` is rare in practice: a pointer you never reseat is usually a reference. `const int*` is everywhere — every "look but do not touch" parameter and every `begin` / `end` pair over data you only read.

What const on the pointee actually promises

```
int x = 1;
const int* p = &x;
// *p = 2;                // error: assignment of read-only location
x = 2;                    // fine: x itself is not const
std::cout << *p << '\n'; // 2 — p sees the change
```

`const int*` restricts *your access through* `p`, not the object: anyone holding `x`, or an `int*` to it, may still write, and `p` observes the new value. `const` on a pointer or reference describes the view, not the thing.

Conversions: adding const is free, removing it is a cast

```
int x = 1;
int* p = &x;
const int* cp = p;                // fine: int* → const int* adds a restrictior
// int* q = cp;                   // error: would drop the restriction silentl
int* q = const_cast<int*>(cp);    // compiles, and is the smell discussed belo
```

`T*` converts implicitly to `const T*`, `T&` to `const T&`, and a `std::vector<int>` argument binds to a `const std::vector<int>&` parameter. Going the other way needs `const_cast`, on purpose, in writing. This asymmetry is the design: a function that takes `const T&` can be called with anything, and it cannot leak a non-const handle out.

Parameters

Module 4 gave the three choices — `T`, `T&`, `const T&` — and `const` is what makes the third one honest. A `const std::string&` parameter costs one pointer to pass, accepts a literal (a temporary binds to it, lesson 4), and guarantees the caller's string comes back unchanged:

```cpp
int count_char(const std::string& text, char c) {
    int n = 0;
    for (char ch : text) {
        if (ch == c) ++n;
    }
    return n;
}
```

A top-level `const` on a *by-value* parameter — `void f(const int n)` — protects only the function's own copy; it means nothing to callers.

---

const member functions — a preview

```cpp
struct Counter {
    int value = 0;
    int get() const { return value; }   // promises not to modify *this
    void add(int n) { value += n; }     // may modify
};

void report(const Counter& c) {
    std::cout << c.get() << '\n';       // OK: get() is const
    // c.add(1);                        // error: add() is not const
}
```

A `const` object — or a `const Counter&`, which is how most objects arrive in a function — may only call member functions marked `const`. The standard library follows this throughout: `size()`, `at()`, `front()` and `find()` all have `const` versions, so a `const std::vector<int>&` is fully readable. Module 8 lesson 3 covers `const` members in depth, including `mutable`.

Why const is contagious — and worth it

Suppose `Counter::get` were not `const`. Then `report` fails to compile, and so does every function that takes `const Counter&` and calls `get`, and every `const auto&` loop over a container of `Counter`s. The fix is to add `const` where it belongs (on `get`) — or to strip `const` from every parameter on the way up, which lies to every caller.

That is the contagion, and it points in a good direction. Write every read-only parameter as `const T&`, every non-mutating member function as `const`, every read-only pointer as `const T*`, and the compiler proves, for free and forever, which code changes what: when something *is* modified unexpectedly, the suspects are exactly the non-const code. Retrofitting `const` onto a codebase that ignored it is painful, which is why it is a day-one habit rather than a cleanup.

---

const_cast is a smell

```
const int limit = 100;
const_cast<int&>(limit) = 200;     // undefined behaviour: limit was declared c
```

`const_cast` removes `const` from a pointer or reference. Writing through the result is defined only if the object was *not* declared `const` — `limit` may live in read-only memory, or the compiler may have folded `100` into every use. The one legitimate use is calling a `const`-incorrect legacy API that you know does not write; in your own code, a `const_cast` means a signature is wrong.

Reading harder declarations

Declaration	Reads as
`const char* s`	pointer to const char — a read-only C string
`const char* const* argv`	pointer to const pointer to const char
`int* const& r`	reference to a const pointer to int
`const int* p[3]`	array of 3 pointers to const int
`int (*p)[3]`	pointer to an array of 3 int

Right to left, parentheses first. `const T*`, `const T&` and `T* const` cover almost everything you write.

Pitfalls

The const on the wrong side. `int* const` when you meant `const int*` compiles and protects the wrong thing.

Believing `const int*` freezes the object. It restricts your view; the object may still change under you.

`const auto` versus `const auto&`. `for (const auto x : v)` copies each element; `const auto&` does not.

Non-const getters. One missing `const` on a member function blocks every `const` user of the class.

Key takeaways

Read declarations right to left: `const` left of `*` is the pointee, right of `*` is the pointer.

`const T*` and `const T&` describe *your* access; the object can still change through other names.

Adding `const` is implicit; removing it is `const_cast`, and writing through it to a truly const object is undefined behaviour.

`const` member functions are what make `const T&` parameters usable; mark every non-mutating one.

Const correctness spreads by construction, which is exactly why it must start on day one.

EXERCISE 1

A const-correct pipeline

Read `n` ($1 \le n \le 100$) integers, then `from` and `to`. Write three functions with exactly these signatures — `int count_matching(const std::vector<int>& values, int target)`, `void replace_all(std::vector<int>& values, int from, int to)` and `void print(const std::vector<int>& values)` — and use them: print how many elements equal `from`, replace every one of them with `to`, and print the vector. Only `replace_all` may take a non-const reference.

Input: `n`, then `n` integers, then `from to`. **Output:** `count=<k>`, then the vector space-separated on one line.

Example: input `5 1 2 1 3 1 1 9` →

```
count=3
9 2 9 3 9
```

INPUT 1

```
5
1 2 1 3 1
1 9
```

OUTPUT 1

```
count=3
9 2 9 3 9
```

INPUT 2

```
3
4 5 6
7 0
```

OUTPUT 2

```
count=0
4 5 6
```

REFERENCE SOLUTION

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

int count_matching(const std::vector<int>& values, int target) {
    int count = 0;
    for (int x : values) {
        if (x == target) ++count;
    }
    return count;
}

void replace_all(std::vector<int>& values, int from, int to) {
    for (int& x : values) {
        if (x == from) x = to;
    }
}

void print(const std::vector<int>& values) {
    for (std::size_t i = 0; i < values.size(); ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    int from = 0, to = 0;
    std::cin >> from >> to;
    std::cout << "count=" << count_matching(values, from) << '\n';
    replace_all(values, from, to);
    print(values);
    return 0;
}
```

EXERCISE 2

Which const is which

Read `n` (0 ≤ n ≤ 100) integers into a `std::vector<int>`, then a `target`. Write `int index_of(const int* begin, const int* end, int target)` — a walking pointer over elements you may **not** modify, returning the index as a pointer difference or `-1` — and `void double_all(int* const begin, int*`

`const end)` — two pointers you may **not** reseat, over elements you may modify, so the walk needs a local copy of `begin`. Print the index of `target`, double every element, and print the vector.

Input: `n`, then `n` integers, then `target`. **Output:** `index=<i or -1>`, then the doubled values space-separated (an empty line when `n` is 0).

Example: input `4 3 5 7 5 5` →

```
index=1
6 10 14 10
```

INPUT 1

```
4
3 5 7 5
5
```

OUTPUT 1

```
index=1
6 10 14 10
```

INPUT 2

```
2
1 2
9
```

OUTPUT 2

```
index=-1
2 4
```

REFERENCE SOLUTION

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

int index_of(const int* begin, const int* end, int target) {
    for (const int* p = begin; p != end; ++p) {
        if (*p == target) return static_cast<int>(p - begin);
    }
    return -1;
}

void double_all(int* const begin, int* const end) {
    for (int* p = begin; p != end; ++p) {
        *p *= 2;
    }
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    int target = 0;
    std::cin >> target;
    std::cout << "index=" << index_of(values.data(), values.data() + n, target) << '
    double_all(values.data(), values.data() + n);
    for (int i = 0; i < n; ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
    return 0;
}
```

QUIZ

1. `const int* p` means…

A. `p` cannot be reseated

B. The `int` cannot be modified through `p`, but `p` may be reseated

C. Neither `p` nor the `int` can change

D. `p` must point at a variable declared `const`

2. Given `int x = 1, y = 2; int* const p = &x;`, which statement compiles?

A. `*p = 2;`

B. `p = &y;`

C. `++p;`

D. `p = nullptr;`

3. What does this print?

```
int x = 1;
const int& r = x;
x = 2;
std::cout << r << '\n';
```

A. `1`

B. `2`

C. Compile error at `x = 2`

D. Undefined behaviour

4. Which conversion happens implicitly, with no cast?

A. `int*` to `const int*`

B. `const int*` to `int*`

C. `const int&` to `int&`

D. None — every change of constness needs a cast

5. `const int c = 5; const_cast<int&>(c) = 6;` — what is the result?

A. `c` becomes 6

B. Undefined behaviour — `c` was declared `const`

C. Compile error

D. A copy of `c` becomes 6

6.

```
struct S {
    int v = 0;
    int get() { return v; }
};
void f(const S& s) { std::cout << s.get(); }
```

What happens?

A. Prints `0`

B. Compile error: `get()` is not a `const` member function, so it cannot be called on a `const S&`

C. Undefined behaviour

D. Prints an address

7. Read right to left: `const char* const* argv` is…

A. A pointer to a const pointer to const char

B. A const pointer to a pointer to const char

C. A const pointer to a const char

D. Invalid C++

ANSWER KEY · CONST CORRECTNESS — READING DECLARATIONS RIGHT TO LEFT

1. B The `const` is left of the `*`, so it applies to the pointee: `*p = 1` is an error, `p = &other` is fine. A `const int*` may point at a non-const `int`; it only restricts access through `p`.

2. A `int* const` is a const pointer to a non-const `int`: the pointer itself is frozen, so every reseat (assignment, increment) is an error, while writing the pointee is allowed.

3. B `const` on the reference forbids writing *through* `r`; it does not freeze `x`. `r` is an alias for `x` and sees the new value.

4. A Adding `const` only restricts what the new handle may do, so it is always safe and implicit. Removing it would let `const` data be written and requires `const_cast`, on purpose.

5. B The cast compiles, but writing through it to an object that was *declared* `const` is undefined behaviour: the compiler may have placed `c` in read-only memory or folded 5 into every use. `const_cast` is only defined for objects that were never const to begin with.

6. B A `const` object may only call member functions marked `const`. Declaring `int get() const` fixes it — and this is why one missing `const` blocks every const-correct caller of the class.

7. A From `argv` outwards: `*` — pointer to; `const` — a const; `*` — pointer to; `const char` — const char. The outer pointer (`argv`) may itself be reseated; what it points at, and what those point at, may not be modified.

LESSON 6 · 15 MIN

Multidimensional data — grids, layout and neighbours

Most "2-D" problems — a board, an image, a maze, a matrix, a spreadsheet — are one-dimensional memory read through two indices. Once you see the layout, choosing a representation, reading a grid from input, visiting a cell's neighbours and staying inside the bounds all become the same small set of idioms. This lesson settles those idioms with `std::vector<std::vector<int>>`, `std::array` of `std::array` and a flat vector, and gives the neighbour loop that every grid exercise in this track — and most grid interview questions — reduces to.

Four representations

```
int fixed[3][4] = {};                                  // built-in: s
std::array<std::array<int, 4>, 3> arr{};               // same, with
std::vector<std::vector<int>> vec(3, std::vector<int>(4));   // rows × cols
std::vector<int> flat(3 * 4);                          // one block;
```

The **built-in 2-D array** and **`std::array` of `std::array`** need both sizes at compile time. They are one contiguous block, and the `std::array` form can be copied, compared and passed by reference without decay. Right for a chess board or a 3×3 kernel.

`std::vector<std::vector<int>>` is the general-purpose grid: `R` rows, each a separate `std::vector<int>` of `C` elements. The constructor `vec(R, std::vector<int>(C))` makes `R` copies of one `C`-element row; `vec(R, std::vector<int>(C, -1))` fills with `-1`. Rows may differ in length (a jagged grid) — a flexibility and a hazard.

A flat vector stores `R × C` elements in one block: cell `(r, c)` is `flat[r * C + c]`, and an `int& at(int r, int c)` helper makes it read like a grid. One allocation, cache-friendly, the layout performance-minded code prefers (Module 19, Performance and undefined behaviour).

Row-major layout

Every representation above is **row-major**: row 0's elements come first, then row 1's, and so on. For a 3×4 grid the memory order is

```
index: 0  1  2  3 | 4  5  6  7 | 8  9  10 11
cell : 00 01 02 03| 10 11 12 13| 20 21 22 23
```

which is where `r * C + c` comes from: skip `r` full rows of `C`, then `c` more. The inverse is `r = i / C`, `c = i % C`. The layout also decides which loop order is fast: iterating `r` outer and `c` inner walks memory sequentially; `c` outer and `r` inner jumps `C` elements every step. Keep rows on the outside.

Reading a grid from standard input

The conventional input is `R C` on the first line, then `R` lines of `C` values:

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

`>>` skips whitespace, so it does not matter whether the numbers arrive one row per line or all on one line. A character grid (`.` and `#`) is even simpler — a `std::string` *is* a row of `char`:

```
std::vector<std::string> board(rows);
for (std::string& row : board) std::cin >> row;    // one token per row, no sp
char cell = board[1][2];
```

Visiting the neighbours

Every flood fill, path search, Game of Life step and "count adjacent mines" is the same loop. Put the offsets in two parallel arrays so the body has one shape:

```cpp
constexpr int DR[4] = {-1, 1, 0, 0};    // up, down, left, right
constexpr int DC[4] = {0, 0, -1, 1};

for (int d = 0; d < 4; ++d) {
    int nr = r + DR[d];
    int nc = c + DC[d];
    if (nr < 0 || nr ≥ rows || nc < 0 || nc ≥ cols) continue;   // off the g
    // grid[nr][nc] is a real neighbour
}
```

For eight neighbours, either extend the arrays to eight entries or loop `dr` and `dc` over `-1..1` and skip `(0, 0)`:

```cpp
for (int dr = -1; dr ≤ 1; ++dr) {
    for (int dc = -1; dc ≤ 1; ++dc) {
        if (dr == 0 && dc == 0) continue;
        int nr = r + dr;
        int nc = c + dc;
        if (nr < 0 || nr ≥ rows || nc < 0 || nc ≥ cols) continue;
        if (grid[nr][nc] == '#') ++count;
    }
}
```

The direction-array form is what you want when the neighbour rule is part of the problem — knight moves are eight `(±1, ±2)` pairs, hex grids have six — because the rule lives in data rather than in the code.

Bounds checks belong before the access

`grid[r][c]` on a vector of vectors, a flat vector and a built-in array all have the same contract: an index outside `[0, size)` is undefined behaviour, and a neighbour loop is where it happens — `r = 0` going up, `c = cols - 1` going right. The check in the loop above is the whole defence; a helper keeps it in one place:

```cpp
bool inside(int r, int c, int rows, int cols) {
    return r ≥ 0 && r < rows && c ≥ 0 && c < cols;
}
```

`.at(r).at(c)` throws `std::out_of_range` instead — but an uncaught exception is still a failed run; the check is what you want. With a jagged `std::vector<std::vector<int>>`, the

row bound is `grid[r].size()`, not a shared `cols`.

Passing a grid to a function

```cpp
using Grid = std::vector<std::vector<int>>;

long long row_sum(const Grid& g, int r) {
    long long total = 0;
    for (int x : g[r]) total += x;
    return total;
}

void clear(Grid& g) {
    for (auto& row : g) {
        for (int& x : row) x = 0;
    }
}
```

By `const Grid&` to read, `Grid&` to modify — the rules from lessons 4 and 5. A `Grid` by value copies every row; it is the single most common accidental copy in grid code. A built-in 2-D array is passed as `int (&g)[3][4]` or `int g[][4]` with the row count alongside.

Transpose and rotate

Index games rather than algorithms:

```cpp
// transpose: R×C → C×R
std::vector<std::vector<int>> t(cols, std::vector<int>(rows));
for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) t[c][r] = grid[r][c];
}

// rotate 90° clockwise: (r, c) → (c, R - 1 - r)
std::vector<std::vector<int>> rot(cols, std::vector<int>(rows));
for (int r = 0; r < rows; ++r) {
    for (int c = 0; c < cols; ++c) rot[c][rows - 1 - r] = grid[r][c];
}
```

Pitfalls

`std::vector<std::vector<int>> g(R, C);` does not compile — the second argument must be a row, `std::vector<int>(C)`. The error says no matching constructor for `(int, int)`; the fix is the inner vector.

Swapped loop bounds. `for (r < cols)` and `for (c < rows)` are undefined behaviour the moment `rows ≠ cols`. Name the variables `rows` / `cols` or `R` / `C`, never `n` / `m`.

`x` / `y` confusion. `grid[y][x]` is the row-major order when you think in coordinates; `grid[x][y]` silently transposes. Use `r` / `c` and the question never comes up.

Neighbour without the bounds check. Corner cells reach index `-1`.

`std::vector<bool>` as a visited grid works but is not a real container of `bool` (Module 13); `std::vector<char>` when it matters.

Key takeaways

A grid is one block of memory read through two indices; row-major means `(r, c)` lives at `r * C + c`.

`std::vector<std::vector<int>> g(R, std::vector<int>(C))` for run-time sizes; `std::array` of `std::array` for compile-time; a flat vector for speed.

Read `R C` then `R × C` values with two nested loops; a character row is a `std::string`.

Neighbours are direction arrays plus a bounds check *before* the access.

Pass grids by `const Grid&` or `Grid&`; keep rows as the outer loop.

EXERCISE 1

Row sums, column sums, first maximum

Read `R` and `C` (1 ≤ R, C ≤ 50) and an `R × C` grid of integers into a `std::vector<std::vector<int>>` built as `grid(R, std::vector<int>(C))`. Print the sum of every row, then of every column, then the largest value and where it **first** occurs in row-major order.

Input: `R C`, then `R` rows of `C` integers. **Output:** `R` lines `row <r>: <sum>`, `C` lines `col <c>: <sum>`, then `max=<value> at r=<r> c=<c>`.

Example: input

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

→

```
row 0: 6
row 1: 15
col 0: 5
col 1: 7
col 2: 9
max=6 at r=1 c=2
```

INPUT 1

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

OUTPUT 1

```
row 0: 6
row 1: 15
col 0: 5
col 1: 7
col 2: 9
max=6 at r=1 c=2
```

INPUT 2

```
1 1
7
```

OUTPUT 2

```
row 0: 7
col 0: 7
max=7 at r=0 c=0
```

REFERENCE SOLUTION

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

int main() {
    int rows = 0, cols = 0;
    std::cin >> rows >> cols;
    std::vector<std::vector<int>> grid(rows, std::vector<int>(cols));
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            std::cin >> grid[r][c];
        }
    }
    for (int r = 0; r < rows; ++r) {
        long long total = 0;
        for (int c = 0; c < cols; ++c) total += grid[r][c];
        std::cout << "row " << r << ": " << total << '\n';
    }
    for (int c = 0; c < cols; ++c) {
        long long total = 0;
        for (int r = 0; r < rows; ++r) total += grid[r][c];
        std::cout << "col " << c << ": " << total << '\n';
    }
    int best = grid[0][0];
    int best_r = 0;
    int best_c = 0;
    for (int r = 0; r < rows; ++r) {
        for (int c = 0; c < cols; ++c) {
            if (grid[r][c] > best) {
                best = grid[r][c];
                best_r = r;
                best_c = c;
            }
        }
    }
    std::cout << "max=" << best << " at r=" << best_r << " c=" << best_c << '\n';
    return 0;
}
```

EXERCISE 2

Count the neighbours

Read `R` and `C` (1 ≤ R, C ≤ 50) and `R` rows of `C` characters, each `.` or `#`, into a `std::vector<std::string>`. For every cell, count how many of its up to eight neighbours are `#` and print the counts as a grid of digits, one row per line. Use direction arrays or a `dr` / `dc` loop over `-1..1`, and check the bounds **before** every access — a corner cell has only three neighbours.

Input: `R C`, then `R` rows. **Output:** `R` lines of `C` digits.

Example: input

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

→

```
333
343
333
```

INPUT 1

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

OUTPUT 1

```
333
343
333
```

INPUT 2

```
1 1
#
```

OUTPUT 2

```
0
```

REFERENCE SOLUTION

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

int main() {
    int rows = 0, cols = 0;
    std::cin >> rows >> cols;
    std::vector<std::string> grid(rows);
    for (std::string& row : grid) std::cin >> row;
    for (int r = 0; r < rows; ++r) {
        std::string line;
        for (int c = 0; c < cols; ++c) {
            int count = 0;
            for (int dr = -1; dr <= 1; ++dr) {
                for (int dc = -1; dc <= 1; ++dc) {
                    if (dr == 0 && dc == 0) continue;
                    int nr = r + dr;
                    int nc = c + dc;
                    if (nr < 0 || nr >= rows || nc < 0 || nc >= cols) continue;
                    if (grid[nr][nc] == '#') ++count;
                }
            }
            line += static_cast<char>('0' + count);
        }
        std::cout << line << '\n';
    }
    return 0;
}
```

QUIZ

1. `std::vector<std::vector<int>> g(3, std::vector<int>(4));` creates…

A. 3 rows of 4 zeros

B. 4 rows of 3 zeros

C. 12 empty rows

D. A compile error

2. In a flat, row-major `std::vector<int>` holding an `R × C` grid, cell `(r, c)` is at index…

A. `c * R + r`

B. `r * C + c`

C. `r * R + c`

D. `r + c`

3. Which loop order walks a row-major grid's memory sequentially?

A. `c` outer, `r` inner

B. `r` outer, `c` inner

C. Either — the cost is the same

D. Neither — vectors are not contiguous

4. With `DR = {-1, 1, 0, 0}` and `DC = {0, 0, -1, 1}`, the four directions `d = 0..3` are…

A. up, down, left, right

B. the four diagonals

C. up, left, down, right

D. right, left, up, down

5. Before reading `grid[nr][nc]` for a neighbour, the complete check is…

A. `nr < rows && nc < cols`

B. `nr ⩾ 0 && nr < rows && nc ⩾ 0 && nc < cols`

C. `nr ⩽ rows && nc ⩽ cols`

D. None — `std::vector` checks bounds for you

6. `std::vector<std::vector<int>> g(R, C);` with `int R, C` is…

A. R rows of C zeros

B. A compile error — the second argument must be a `std::vector<int>`, not an `int`

C. C rows of R zeros

D. R rows, each holding the single value C

7. `std::vector<std::string> g(R); for (auto& row : g) std::cin >> row;` reads a character grid correctly because…

A. `>>` reads one whitespace-delimited token, which is a whole row when rows contain no spaces

B. `std::string` is a `std::vector<char>`

C. `>>` reads exactly `C` characters

D. It does not — `std::getline` is required

ANSWER KEY · MULTIDIMENSIONAL DATA — GRIDS, LAYOUT AND NEIGHBOURS

1. A The outer constructor makes 3 copies of the given row, and the row is a `std::vector<int>` of 4 value-initialised (zero) elements: `g[r][c]` with `r < 3`, `c < 4`.

2. B Row-major stores each full row of `C` elements before the next: skip `r` rows (`r * C`) then `c` more. `c * R + r` is column-major, which transposes every access.

3. B With rows outside, consecutive iterations touch neighbouring elements; with columns outside, every step jumps a whole row. The results are identical; the cache behaviour is not.

4. A Pair the arrays index by index: `(-1, 0)` is one row up, `(1, 0)` one row down, `(0, -1)` one column left, `(0, 1)` one column right. Diagonals would need both offsets non-zero.

5. B Neighbours can go negative at the top and left edges and reach `rows` / `cols` at the bottom and right; all four comparisons are needed. `operator[]` never checks, and an out-of-range index is undefined behaviour.

6. B There is no `vector(int, int)` constructor for a vector of vectors: the fill constructor wants a row value, and an `int` does not convert to `std::vector<int>` implicitly. Write `g(R, std::vector<int>(C))`.

7. A A row such as `.#..#` has no whitespace, so `>>` extracts it in one go and skips the newline before the next. If rows could contain spaces you would need `std::getline` and the leftover-newline care from Module 1.

CHECKPOINT · 25 MIN

Checkpoint — Arrays, pointers and references

This checkpoint covers the whole module: built-in arrays and `std::array`, pointers and `nullptr`, pointer arithmetic and the `[begin, end)` convention, references and dangling, const correctness, and grids.

How it works. Fifteen 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 does an array's name become when you pass it to a function, and what is lost?
- What is `p[i]` in terms of `*` and `+`, and what does `q - p` give?
- Which pointer may you form but never dereference?
- Name the three things a reference cannot do that a pointer can.
- What is the difference between `const int*` and `int* const`?
- Where does cell `(r, c)` of an `R × C` grid live in a flat vector?
- Why is a reference into a `std::vector` unsafe after `push_back`?

The three programs are a pointer walk over runs of equal values, a flood count on a character grid, and a statistics function with const-correct reference parameters. Print values, indices and differences — never an address.

EXERCISE 1

Runs through a pointer walk

Read `n` (1 ≤ n ≤ 1000) integers into a `std::vector<int>`. Walk the range `[begin, end)` with `const int*` pointers and print each maximal run of equal consecutive values as `<value>x<length>`, space-separated on one line. Write `const int* run_end(const int* p, const int* end)` that returns the pointer one past the run starting at `p`; the length of a run is a pointer difference.

Input: `n`, then `n` integers. **Output:** one line of runs.

Example: input `6 1 1 2 2 2 3` → output `1x2 2x3 3x1`

INPUT 1

```
6
1 1 2 2 2 3
```

OUTPUT 1

```
1x2 2x3 3x1
```

INPUT 2

```
1
7
```

OUTPUT 2

```
7x1
```

REFERENCE SOLUTION

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

const int* run_end(const int* p, const int* end) {
    const int* q = p;
    while (q != end && *q == *p) ++q;
    return q;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    const int* begin = values.data();
    const int* end = begin + values.size();
    bool first = true;
    for (const int* p = begin; p != end;) {
        const int* q = run_end(p, end);
        if (!first) std::cout << ' ';
        std::cout << *p << 'x' << (q - p);
        first = false;
        p = q;
    }
    std::cout << '\n';
    return 0;
}
```

EXERCISE 2

Region size

Read `R` and `C` (1 ≤ R, C ≤ 50), `R` rows of `.` (open) and `#` (wall), then a start cell `r c`. Print the number of open cells reachable from the start by moving up, down, left or right through open cells,

counting the start itself; if the start is a wall, print `region=0`. Use a `std::vector<std::pair<int, int>>` as an explicit stack, a `seen` grid, direction arrays and a bounds check before every access.

Input: `R C`, `R` rows, then `r c`. **Output:** `region=<count>`.

Example: input

```
3 4
..#.
.#..
..#.
0 0
```

→ `region=5`

INPUT 1

```
3 4
..#.
.#..
..#.
0 0
```

OUTPUT 1

```
region=5
```

INPUT 2

```
3 4
..#.
.#..
..#.
1 3
```

OUTPUT 2

```
region=4
```

REFERENCE SOLUTION

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

int main() {
    int rows = 0, cols = 0;
    std::cin >> rows >> cols;
    std::vector<std::string> grid(rows);
    for (std::string& row : grid) std::cin >> row;
    int sr = 0, sc = 0;
    std::cin >> sr >> sc;
    if (grid[sr][sc] == '#') {
        std::cout << "region=0\n";
        return 0;
    }
    std::vector<std::vector<char>> seen(rows, std::vector<char>(cols, 0));
    const int DR[4] = {-1, 1, 0, 0};
    const int DC[4] = {0, 0, -1, 1};
    std::vector<std::pair<int, int>> stack;
    stack.push_back({sr, sc});
    seen[sr][sc] = 1;
    int size = 0;
    while (!stack.empty()) {
        auto [r, c] = stack.back();
        stack.pop_back();
        ++size;
        for (int d = 0; d < 4; ++d) {
            int nr = r + DR[d];
            int nc = c + DC[d];
            if (nr < 0 || nr >= rows || nc < 0 || nc >= cols) continue;
            if (grid[nr][nc] == '#' || seen[nr][nc]) continue;
            seen[nr][nc] = 1;
            stack.push_back({nr, nc});
        }
    }
    std::cout << "region=" << size << '\n';
    return 0;
}
```

EXERCISE 3

Analyse and normalise

Read `n` (1 ≤ n ≤ 100) integers into a `std::vector<int>`. Write `void analyse(const std::vector<int>& values, int& lo, int& hi, double& mean)` — a const reference in, three

references out — and `void normalise(std::vector<int>& values, int shift)` that subtracts `shift` from every element. Print the minimum, maximum and mean (two decimals), then the values after subtracting the minimum so that the smallest becomes 0.

Input: `n`, then `n` integers. **Output:** `min=<lo> max=<hi> mean=<mean>`, then the normalised values on one line.

Example: input `4 3 8 1 4` →

```
min=1 max=8 mean=4.00
2 7 0 3
```

INPUT 1

```
4
3 8 1 4
```

OUTPUT 1

```
min=1 max=8 mean=4.00
2 7 0 3
```

INPUT 2

```
1
-5
```

OUTPUT 2

```
min=-5 max=-5 mean=-5.00
0
```

REFERENCE SOLUTION

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

void analyse(const std::vector<int>& values, int& lo, int& hi, double& mean) {
    lo = values.front();
    hi = values.front();
    long long sum = 0;
    for (int x : values) {
        if (x < lo) lo = x;
        if (x > hi) hi = x;
        sum += x;
    }
    mean = static_cast<double>(sum) / static_cast<double>(values.size());
}

void normalise(std::vector<int>& values, int shift) {
    for (int& x : values) x -= shift;
}

int main() {
    int n = 0;
    std::cin >> n;
    std::vector<int> values(n);
    for (int& x : values) std::cin >> x;
    int lo = 0, hi = 0;
    double mean = 0.0;
    analyse(values, lo, hi, mean);
    std::cout << std::fixed << std::setprecision(2);
    std::cout << "min=" << lo << " max=" << hi << " mean=" << mean << '\n';
    normalise(values, lo);
    for (std::size_t i = 0; i < values.size(); ++i) {
        if (i > 0) std::cout << ' ';
        std::cout << values[i];
    }
    std::cout << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What does this print?

```cpp
int a[3] = {1, 2, 3};
std::cout << a[3] << '\n';
```

A. `0`

B. `3`

C. Undefined behaviour — `a[3]` is past the end

D. Compile error

2. Which type gives a fixed-size array value semantics — copyable, comparable with `==`, with a `.size()`?

A. `int[N]`

B. `std::array<T, N>`

C. `int*`

D. `std::size`

3. What does this print?

```cpp
int x = 1;
int* p = &x;
int** pp = &p;
**pp = 4;
std::cout << x << '\n';
```

A. `1`

B. `4`

C. An address

D. Compile error

4. `p[i]` is defined as...

A. `*(p + i)`

B. `*p + i`

C. `p + i`

D. `&p[i]`

5. For `int* begin = a; int* end = a + n;` with `n == 0`, `end - begin` is...

A. Undefined behaviour

B. `0`

C. `1`

D. `-1`

6. What does this print?

```
int a = 1, b = 2;
int& r = a;
r = b;
b = 3;
std::cout << a << r << b << '\n';
```

A. `123`

B. `223`

C. `233`

D. `333`

7. Given `int x = 0; int* const p = &x;`, which statement is allowed?

A. `p = &x;`

B. `*p = 5;`

C. `++p;`

D. None of them

8. In a flat row-major vector holding an `R × C` grid, cell `(r, c)` is at index…

A. `r * C + c`

B. `c * R + r`

C. `r * R + c`

D. `r + c * C`

9. Which function returns a dangling reference?

A. `int& f(std::vector<int>& v) { return v[0]; }`

B. `int& f() { static int n = 0; return n; }`

C. `int& f() { int n = 0; return n; }`

D. `int& f(int& n) { return n; }`

10. `std::vector<int> v{1, 2, 3}; auto x = v[0]; x = 9;` — afterwards `v[0]` is…

A. `9`

B. `1` — `x` is a copy; `auto&` would alias the element

C. Undefined behaviour

D. A compile error

11. Inside `void f(int values[])`, the expression `std::size(values)` …

A. Returns the caller's array length

B. Returns 8

C. Does not compile — `values` is a pointer and has no size

D. Returns 0

12. To read the upward neighbour `(r - 1, c)` safely you must first check…

A. `r > 0`

B. `r ⩾ 0`

C. `r < rows`

D. Nothing — `std::vector` bounds-checks `[]`

13. `std::array<int, 3> a{}; a.at(5)` …

A. Returns 0

B. Is undefined behaviour

C. Throws `std::out_of_range`

D. Does not compile

14. `void total(std::vector<int> v)` versus `void total(const std::vector<int>& v)` for summing a million elements — the difference is…

A. None once the optimiser has run

B. The first copies a million `int`s on every call; the second passes one pointer-sized reference

C. The second is slower because of the extra indirection

D. The first is required — a `const` vector cannot be iterated

15. `int x = 1; const int* p = &x; x = 5;` — `*p` now reads…

A. `1`, the value when `p` was bound

B. `5` — `const` restricts writes through `p`, not changes to `x`

C. Undefined behaviour

D. A compile error at `x = 5`

ANSWER KEY · CHECKPOINT — ARRAYS, POINTERS AND REFERENCES

1. C Valid indices are 0, 1 and 2. Nothing checks the subscript; reading `a[3]` is undefined behaviour, whatever it happens to print today.

2. B `std::array` wraps the built-in block in a struct, so it copies, compares and reports its size like any other value. A built-in array decays and cannot be assigned; `std::size` is a function, not a container.

3. B `*pp` is `p`, and `**pp` is `*p`, which is `x`. Two dereferences reach the `int`, and the write lands in `x`.

4. A Subscripting a pointer (or a decayed array) is pointer arithmetic followed by a dereference. `*p + i` adds to the first element's value; `p + i` is the address of the element, not the element.

5. B An empty range has `begin == end`, so the difference is 0 and a `p != end` loop runs zero times. Forming `a + 0` is always allowed.

6. B `r = b` copies 2 into `a` (`r` still names `a`); `b = 3` changes only `b`. So `a` is 2, `r` reads `a` as 2, and `b` is 3.

7. B The `const` is right of the `*`, so the pointer is frozen and the pointee is not: writing `*p` is fine, reseating or moving `p` is an error.

8. A Skip `r` complete rows of `C` elements, then `c` more. The other formulas either transpose the grid or use the wrong stride.

9. C The local `n` is destroyed when `f` returns. A vector element, a `static` and a reference parameter all refer to objects that outlive the call.

10. B `auto` deduces `int`, dropping the reference `operator[]` returned, so `x` is an independent copy. `auto& x = v[0]` makes `x` an alias and the write would reach the vector.

11. C The parameter is `int*`; `std::size` has no overload for a pointer, so the call is a hard error rather than a silently wrong number. That is the safe failure — pass the length explicitly.

12. A `r - 1` must be at least 0, which means `r > 0`. `r ⩾ 0` still allows `r == 0` and an index of −1; `operator[]` never checks.

13. C `.at()` is the checked access and throws on a bad index. `a[5]` would be undefined behaviour — the same unchecked access as a built-in array.

14. B Pass by value copies the whole vector; `const&` costs eight bytes and forbids modification, which is all a sum needs. Const containers are fully iterable through their `const` overloads.

15. B `const int*` is a read-only view of an object that may still be written through other names. `p` sees whatever `x` currently holds.
