Module 20 of 20

Interview idioms

The contest template and what it costs, the STL idiom sheet, the pitfalls that fail interviews, implementing vector and a hash map by hand, clean solutions under pressure, and the C++ theory drill.

Nineteen modules taught the language. This one teaches the round: how a C++ interview or contest is actually conducted, what is actually judged, and the small set of habits that turn knowing C++ into passing.

It opens with the template — `#include <bits/stdc++.h>`, `using namespace std;`, the two fast-I/O lines, `long long` by default — presented honestly as contest idioms with a measured cost, so you can use them under a clock and explain in the same breath why production code does not. The idiom sheet names the shapes interview problems take and the two-to-six-line STL for each: sort–unique–erase, `lower_bound` for "first ≥", counting with `std::map`, a `std::priority_queue` of pairs, `std::tuple` for a multi-key sort. The pitfalls lesson is the review pass — `int` products, unsigned `size()`, `1 << 31`, `%` on negatives, `std::endl`, `map[]`, invalidated iterators — that catches most "correct but rejected" solutions. Implement-the-built-in is the question every senior C++ interview asks in some form: a `Vec<T>` with the rule of five and doubling growth, a `UniquePtr<T>`, a chaining hash map, and a string with SSO in outline. Clean solutions is about legibility; the theory drill is thirty questions with the answers an interviewer wants to hear.

The exercises are the problems interviewers set, written the way the module says to write them: statistics with the template, a character grid, distinct values with `lower_bound` queries, the k rarest words through a bounded heap, a fixer for four broken idioms, fifteen expressions to predict before running, the vector and the hash map by hand, balanced brackets with pure helpers, a leaderboard with tuple keys, and a dispatch drill. The final checkpoint — an LRU cache, a shunting-yard evaluator and a polymorphic shape store — closes the plan.

CONTENTS

1	The interview template — the contest idioms and what they cost	13 min
2	The STL idiom sheet — the two-line answers to the common shapes	14 min
3	The pitfalls that fail interviews	14 min

4	Implement the built-in — a vector, a unique_ptr, a hash map and a string by hand	15 min
5	Writing clean solutions under time pressure	12 min
6	The C++ theory drill — thirty questions, thirty answers	25 min
✓	Final checkpoint — Interview idioms	30 min

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

LESSON 1 · 13 MIN

The interview template — the contest idioms and what they cost

Every timed round starts the same way: a statement, an input format, a limit, and forty minutes. The candidates who do well have stopped thinking about the mechanics — the includes, the fast reader, which integer type — because they type a template from memory and spend the minutes on the problem. C++ has a well-known contest template, and nineteen modules of this track have told you not to write two lines of it. This lesson gives you the template, measures what each line buys and costs, and draws the boundary: these are idioms for a clock, and knowing *why* they stay out of production code is what an interviewer is listening for.

The template

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

const ll MOD = 1'000'000'007;

#ifdef LOCAL
#define dbg(x) cerr << #x << " = " << (x) << '\n'
#else
#define dbg(x)
#endif

ll solve(const vector<ll>& a) {
    ll best = 0;
    for (ll x : a) best = max(best, x);
    return best;
}

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<ll> a(n);
    for (auto& x : a) cin >> x;
    dbg(n);
    cout << solve(a) << '\n';
    return 0;
}
```

Type it until it takes ninety seconds. Every line is deliberate: `solve` is separate from I/O so the logic is readable and testable; every quantity that could pass two billion is `long long` from the start; the output is `'\n'`, never `endl`; `dbg` prints to `cerr` — which the judge ignores — and compiles to nothing unless you build with `-DLOCAL`.

`#include <bits/stdc++.h>`

The header is not in the standard. It is an internal file of libstdc++, GCC's standard library, that includes every standard header at once — and because Clang on Linux uses libstdc++ by default it works there too, including on this track's judge (Clang 18, libstdc++ 14). It does not exist on MSVC or with libc++ (Apple Clang), so a solution that uses it is not portable C++.

The other cost is compile time. Measured on the lab's g++ 14 with `-O2`: a hello-world-sized program compiled in about 0.9 s with `<iostream>`, `<vector>`, `<algorithm>` and `<string>`, and in about 3 s with `bits/stdc++.h`. Three seconds is nothing once; across a thousand translation units that each pay it, it is the difference between a build you wait for and one you do not. Under a clock, "forgetting an include is impossible" wins; in a repository, portability, build time and a file that states its own dependencies win. Know both.

`using namespace std;`

The directive makes every name in `std` visible unqualified. The cost is not style; it is that every one of those names is now a name you cannot safely declare yourself. Module 1 introduced the trade-off; here is what it looks like when it bites:

```
#include <bits/stdc++.h>
using namespace std;
int count = 0;                                  // a natural name for a count
int main() {
    vector<int> v{1, 2, 2};
    cout << count(v.begin(), v.end(), 2);       // error: reference to 'count
}
```

Unqualified lookup finds your `count` and `std::count` in the same scope and refuses to choose. A function named `rank` clashes the same way with the type trait `std::rank`; `size`, `data`, `distance`, `next`, `prev`, `left`, `right`, `apply` and `reduce` are all in the library. (Separately, `<cmath>` drags C's `y1`, `j0` and `gamma` into the global namespace, so `double y1;` fails even without the directive.) In a 60-line solution with three variables none of this happens; in a header it happens to everyone who includes it. The rule is therefore simple: the directive never goes in a header, and in an interview you can use it while saying that sentence.

The two fast-I/O lines

```
ios::sync_with_stdio(false);
cin.tie(nullptr);
```

By default every `cin` read is synchronised with C's `stdin` so that `scanf` and `cin` can interleave; switching that off lets the C++ streams keep their own buffer. Measured on the lab

reading a million integers: 400–470 ms synchronised, 155–195 ms not. The rule that comes with it: after the call, never mix `printf` / `scanf` with `cout` / `cin` — the two families no longer share a buffer, and output arrives out of order.

`cin.tie(nullptr)` unties `cin` from `cout`. A tied stream is flushed before every read so that a prompt appears before the program waits; a judge never reads prompts, so the flush is pure cost. Interactive problems — where you talk to a judge process — are the exception: there you flush explicitly after every query, and the statement says so.

`long long` by default

`int` is 32 bits on this platform: about ±2.1 × 10^9. A sum of 10^5 values up to 10^5 is 10^{10}; a product of two values near 10^5 is 10^{10}; `n * (n - 1) / 2` for `n = 10^5` is 5 × 10^9. Every one of those overflows `int`, and signed overflow is undefined behaviour — not a wrong number, but a program the optimiser may transform on the assumption that it never happens (Module 19). `long long` holds ±9.2 × 10^{18}, which covers every intermediate in a typical problem. So: `long long` for anything that accumulates or multiplies; `int` for indexes and loop variables.

The contest hack you will see is `#define int long long` with `signed main()`. It works. It also rewrites every `int` the preprocessor reaches — `sizeof`, overload choices, code pasted from elsewhere — and cannot be explained to an interviewer as anything but a trick. `using ll = long long;` costs the same keystrokes and says what you meant.

Reading patterns

Shape	Code
`n` then `n` values	`int n; cin >> n; vector<ll> a(n); for (auto& x : a) cin >> x;`
`T` test cases	`int T; cin >> T; while (T--) { … }`
Until end of input	`ll x; while (cin >> x) { … }`
A grid of characters	`vector<string> g(R); for (auto& row : g) cin >> row;`
A line with spaces	`cin >> n; cin.ignore(); getline(cin, line);`

`cin >> row` reads one whitespace-free token, which is exactly one row of `.` and `#`, and `g[r][c]` then indexes it. The mixed case is Module 1's rule — `>>` leaves the newline behind,

and a following `getline` returns an empty string unless you consume it.

Printing

`'\n'` ends a line; `endl` ends a line *and flushes*, which is a system call. Measured on the lab printing a million lines: about 500 ms with `endl`, about 40 ms with `'\n'`. Buffered output leaves when the program exits, which is all a judge needs. Decimals are printed with `fixed << setprecision(k)`, which is sticky, so set it once. For a modular answer, `const ll MOD = 1'000'000'007;` at the top (the digit separators are C++14), reduce after every multiplication, and normalise a subtraction with `((x % MOD) + MOD) % MOD` — the pitfalls lesson returns to both.

What goes wrong

Mistake	What happens
`int total = n * n;`	Overflow in `int` before any widening: undefined behaviour
`scanf` after `sync_with_stdio(false)`	Input read twice or out of order
`endl` in a loop of 10^6 lines	A flush per line — an order of magnitude slower
`while (!cin.eof())`	One extra iteration with a stale value
A global named `count`, `size` or `rank`	Ambiguous reference under `using namespace std;`
`dbg(x)` printing to `cout`	Debug text in the judged output: wrong answer

Key takeaways

The template: `bits/stdc++.h`, `using namespace std;`, the two fast-I/O lines, `using ll = long long;`, `solve` separate from `main`, `'\n'` to print, `dbg` to `cerr` behind `#ifdef LOCAL`.

`bits/stdc++.h` is a libstdc++ internal (GCC and Clang-with-libstdc++ only) that measured about three times the compile time; real headers in anything you keep.

`using namespace std;` makes `count`, `size`, `rank` and hundreds more names unusable for your own declarations — fine in a 60-line file, never in a header.

`sync_with_stdio(false)` roughly halved input time on the lab; after it, never mix C and C++ I/O. `cin.tie(nullptr)` removes a flush per read.

`long long` for anything that sums or multiplies; `int` overflow is undefined behaviour, not a wrapped number.

EXERCISE 1

Statistics with the template

Using the contest template from the lesson — `#include <bits/stdc++.h>`, `using namespace std;`, the two fast-I/O lines and `long long` for every value — read an integer `n` and then `n` integers, each between `-10^12` and `10^12`. Print one line:

`count=<n> sum=<sum> min=<min> max=<max> mean=<sum / n to 2 decimals>`

When `n` is `0` print just `count=0`.

Input: `n`, then `n` integers separated by whitespace (any line layout). **Output:** one line.

```
5
3 9 -2 7 1
```

prints

```
count=5 sum=18 min=-2 max=9 mean=3.60
```

INPUT 1

```
5
3 9 -2 7 1
```

OUTPUT 1

```
count=5 sum=18 min=-2 max=9 mean=3.60
```

INPUT 2

```
0
```

OUTPUT 2

```
count=0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    ll sum = 0, lo = LLONG_MAX, hi = LLONG_MIN;
    for (int i = 0; i < n; ++i) {
        ll x;
        cin >> x;
        sum += x;
        lo = min(lo, x);
        hi = max(hi, x);
    }
    if (n == 0) {
        cout << "count=0\n";
        return 0;
    }
    cout << "count=" << n << " sum=" << sum << " min=" << lo << " max=" << hi
         << " mean=" << fixed << setprecision(2) << static_cast<double>(sum) / n <<
    return 0;
}
```

EXERCISE 2

A grid of characters

Read `R` and `C`, then `R` rows of exactly `C` characters, each `.` or `#`. Use the template's grid pattern — a `vector<string>` with one `cin >> row` per row — and print:

- one line per row, `row <r>: <number of # in that row>` (rows numbered from 0),
- `total=<number of # in the grid>`,
- `busiest column=<index of the column holding the most #>`, the lowest index on a tie (so `0` when the grid is empty of `#`).

Input: `R C`, then `R` lines. **Output:** `R + 2` lines.

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

prints

```
row 0: 1
row 1: 3
row 2: 0
total=4
busiest column=1
```

INPUT 1

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

OUTPUT 1

```
row 0: 1
row 1: 3
row 2: 0
total=4
busiest column=1
```

INPUT 2

```
1 1
#
```

OUTPUT 2

```
row 0: 1
total=1
busiest column=0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int R, C;
    cin >> R >> C;
    vector<string> g(R);
    for (auto& row : g) cin >> row;
    int total = 0;
    vector<int> perColumn(C, 0);
    for (int r = 0; r < R; ++r) {
        int here = 0;
        for (int c = 0; c < C; ++c) {
            if (g[r][c] == '#') {
                ++here;
                ++perColumn[c];
            }
        }
        total += here;
        cout << "row " << r << ": " << here << '\n';
    }
    int busiest = static_cast<int>(max_element(perColumn.begin(), perColumn.end()) -
    cout << "total=" << total << '\n';
    cout << "busiest column=" << busiest << '\n';
    return 0;
}
```

QUIZ

1. What happens here?

```cpp
#include <bits/stdc++.h>
using namespace std;
int count = 0;
int main() {
    vector<int> v{1, 2, 2};
    cout << count(v.begin(), v.end(), 2) << '\n';
}
```

A. Prints `2`

B. Prints `0`

C. Compile error: reference to `count` is ambiguous

D. Undefined behaviour

2. What does `ios::sync_with_stdio(false);` do, and what rule comes with it?

A. Unsynchronises the C++ streams from C's `stdio`, so `cin` / `cout` keep their own buffers and run about twice as fast — after it, never mix `scanf` / `printf` with `cin` / `cout`

B. Disables output buffering so every `<<` is written immediately

C. Makes `cin >> x` read whole lines instead of tokens

D. Is required before `long long` can be read with `>>`

3. And `cin.tie(nullptr);`?

A. Stops `cin` flushing `cout` before every read — harmless on a judge, but in an interactive program a prompt may not appear until you flush

B. Closes `cout` so only `cerr` can print

C. Makes `cin` skip leading whitespace

D. Ties `cin` to `cerr` for error reporting

4. `#include <bits/stdc++.h>` is…

A. A standard header every compiler provides

B. A libstdc++ internal that includes every standard header — works on GCC and on Clang using libstdc++, not on MSVC, and measured about three times the compile time of the real headers on the lab

C. Faster to compile than the real headers because it is precompiled

D. Required before `using namespace std;` will compile

5. What does this print?

```
int n = 100000;
long long total = n * n;
std::cout << total;
```

A. `10000000000`

B. Undefined behaviour — `n * n` overflows `int` before the result is widened

C. `1410065408`

D. Compile error

6. `#define int long long` with `signed main()` is…

A. Standard practice — keep it in every solution

B. A contest hack that silently rewrites every `int` the macro sees, including in `sizeof`, overloads and code copied from elsewhere; write `using ll = long long;` and choose the type deliberately

C. A compile error, because `main` must return `int`

D. The only way to get 64-bit arithmetic in C++

7. The loop that reads integers until the end of input is…

A. `while (!cin.eof()) { cin >> x; use(x); }`

B. `while (cin >> x) { use(x); }`

C. `for (;;) { cin >> x; if (x == 0) break; use(x); }`

D. `while (cin.good()) { cin >> x; use(x); }`

ANSWER KEY · THE INTERVIEW TEMPLATE — THE CONTEST IDIOMS AND WHAT THEY COST

1. C The using-directive makes `std::count` visible at global scope beside your own `count`, and unqualified lookup finds both — g++ 14 reports "reference to 'count' is ambiguous". That is the real cost of `using namespace std;`: every name in the library becomes a name you cannot safely reuse (`size`, `data`, `distance`, `next`, `left` are the usual victims).

2. A By default every `cin` read is synchronised with `stdin` so C and C++ I/O can interleave; switching that off measured 155–195 ms versus 400–470 ms for a million integers on the lab. The price is that the two I/O families no longer share a buffer, so mixing them reorders or loses data.

3. A `cin` is tied to `cout` so a prompt printed before a read is visible; untying removes a flush per read. Judges do not read prompts, so the flush is pure cost there — and the reason interactive problems tell you to flush explicitly.

4. B It is not part of the standard: it lives in GCC's library and Clang picks it up only when it uses that library, which the study judge does. A hello-world-sized program took about 3 s to compile with it and under 1 s with `<iostream>`, `<vector>`, `<algorithm>` and `<string>` — fine in a contest, a bad habit in a codebase.

5. B The multiplication happens in `int` and 10^{10} does not fit; signed overflow is undefined behaviour, and at `-O2` the optimiser may produce anything. Widen an operand first: `1LL * n * n` or `static_cast<long long>(n) * n`. Java would print `1410065408`; C++ promises nothing.

6. B It works — `signed` is `int` spelt without the macro's reach — but it doubles every counter's size, changes which overload is chosen, and cannot be explained to an interviewer as anything but a trick. A type alias costs the same keystrokes and says what you meant.

7. B `cin >> x` returns the stream, which converts to `false` once a read fails; the extraction and the test are one step. The `eof()` and `good()` forms test *before* reading and process the last value twice (or a zero) when the input ends without a trailing token; the sentinel form needs a sentinel the problem never promised.

LESSON 2 · 14 MIN

The STL idiom sheet — the two-line answers to the common shapes

Interview problems take a small number of shapes — deduplicate, find the first element not below a threshold, count things, keep the k best, sort by several keys — and for each shape the STL has a two-to-six-line answer that an experienced C++ programmer types without thinking. Knowing them is worth more than any single algorithm, because they are the vocabulary the algorithm is written in. This lesson is the sheet: each idiom, the line, the cost, and the trap beside it. Everything here was taught in Modules 13 and 14; what is new is seeing the shapes together and the names to say out loud.

Sort, unique, erase

```
std::sort(v.begin(), v.end());
v.erase(std::unique(v.begin(), v.end()), v.end());
```

`std::unique` removes *adjacent* duplicates only — hence the sort — and, like every algorithm, it cannot change the container's size: it shifts the survivors to the front and returns an iterator to the new logical end. The `erase` from that iterator to `end()` is the step people forget; without it the vector keeps its old size with unspecified values at the back. O(n log n) for the sort, O(n) for the rest. The alternative when you also need membership queries is `std::set<T> s(v.begin(), v.end());`, which is the same complexity with a larger constant.

`lower_bound`: the first element not below

```
auto it = std::lower_bound(v.begin(), v.end(), x);   // first element ≥ x
if (it ≠ v.end()) use(*it, it - v.begin());          // value and index
```

On a sorted range, `lower_bound` is "first ≥ x", `upper_bound` is "first > x", both O(log n), both returning `end()` when nothing qualifies — check before dereferencing. Together they bound the run of elements equal to `x`, so `upper - lower` counts occurrences and `lower ≠ end && *lower == x` is a membership test. On a `std::set` or `std::map` call the *member* `s.lower_bound(x)`: the free algorithm works on any range but walks a tree's iterators linearly.

Counting with `std::map`

```
std::map<std::string, int> freq;
for (const auto& w : words) ++freq[w];
for (const auto& [word, n] : freq) std::cout << word << ' ' << n << '\n';   //
```

`operator[]` inserts a value-initialised entry (zero) when the key is absent and returns a reference, so `++freq[w]` is the whole counter. The map iterates in key order, which is usually the order the answer wants. When order does not matter and n is large, `std::unordered_map` is the same code with average O(1) instead of O(log n) — but its iteration order is unspecified, so copy into a vector and sort before printing.

A `priority_queue` of pairs

```
std::priority_queue<std::pair<int, std::string>> heap;                  // lar
std::priority_queue<int, std::vector<int>, std::greater<>> minHeap;    // smal
```

The default heap is a max-heap; a min-heap names the underlying container and `std::greater<>`. Pairs compare lexicographically — first by `.first`, then `.second` — which is exactly what "by count, ties by name" needs. The bounded-heap idiom keeps the k smallest of n in O(n log k): push every element into a *max*-heap and pop the top whenever the size exceeds k, so the largest are evicted and the k smallest survive (the k largest use a min-heap, symmetrically). Remember the heap's own order is not a sorted order: only `top()` is meaningful, so move the survivors into a vector and sort them to print.

`minmax`, `gcd`, `lcm`

```
auto [lo, hi] = std::minmax({a, b, c});                  // by value: safe
auto [mn, mx] = std::minmax_element(v.begin(), v.end());  // iterators
std::gcd(12, 18);  std::lcm(4, 6);                        // <numeric>, C++17
```

The initializer-list form of `minmax` returns a `pair<T, T>` by value. The two-argument form returns a pair of *references* to its arguments, which is fine for named variables and undefined behaviour for temporaries: `auto [lo, hi] = std::minmax(f(), g());` binds references to

values destroyed at the end of the statement. `std::gcd(0, 0)` is 0, and `lcm` of large values overflows — compute it as `a / gcd(a, b) * b`, dividing first.

`accumulate` with `0LL`

```
long long total = std::accumulate(v.begin(), v.end(), 0LL);
```

The accumulator's type is the type of the initial value, not the elements'. `std::accumulate(v.begin(), v.end(), 0)` sums into an `int` whatever `v` holds, and overflows at 2.1×10^9; `0LL` makes it 64-bit. The same trap applies to `std::reduce` and to a hand-written `int sum = 0;`.

Strings: reverse, sort, build

```
std::reverse(s.begin(), s.end());                    // in place
std::string r(s.rbegin(), s.rend());                 // a reversed copy
std::sort(s.begin(), s.end());                       // "cab" → "abc": the
std::string(3, '-')  +  std::to_string(n);           // repeat, convert
```

A `std::string` is a random-access range of `char`, so every algorithm applies to it. Sorting the characters gives the canonical key for grouping anagrams; `std::string(n, c)` repeats a character; building output in a `std::string` and printing once beats a stream insertion per token when the output is large.

`iota`, `bitset` and `bit`

```
std::vector<int> idx(n);
std::iota(idx.begin(), idx.end(), 0);                    // 0, 1, …, n-1: an ind
std::sort(idx.begin(), idx.end(), [&](int a, int b) { return v[a] < v[b]; });

std::bitset<32> bits(37);                                // 00000000000000000000
bits.count();  bits.to_string();  bits[0];
std::popcount(37u);  std::bit_width(37u);               // <bit>, C++20: 3 and
```

`iota` plus a sort with a lambda is how you sort indexes by their values while leaving the values in place. `std::bitset<N>` is a fixed-width bit array with counting and printing built in; for a runtime width use an `unsigned` and `<bit>`. Note that `popcount` needs an unsigned argument.

Pairs and tuples for multi-key sorting

```
std::sort(v.begin(), v.end(), [](const Entry& a, const Entry& b) {
    return std::tie(b.score, a.time, a.name) < std::tie(a.score, b.time, b.nar
});   // score descending, then time ascending, then name ascending
```

Tuples compare lexicographically, so a comparison of two `std::tie` tuples is a multi-key comparison with no `if` chain. To reverse one key, swap `a` and `b` in that slot only — the trick above — or negate a numeric key with `std::make_tuple(-a.score, a.time, a.name)`, which is fine while negation cannot overflow. A chain such as `a.score > b.score && a.name < b.name` is *not* a strict weak ordering (two equal scores compare false both ways for any names), and `std::sort` given an inconsistent comparator is undefined behaviour. `std::pair` compares the same way, which is why a `pair<int, string>` heap orders "by count, ties by name" without a comparator.

The sheet

Shape	Idiom	Cost
Deduplicate	`sort` → `erase(unique(…), end)`	O(n log n)
First ≥ x / first > x	`lower_bound` / `upper_bound`	O(log n)
Count occurrences	`++map[key]`	O(log n) each
k smallest of n	max-heap capped at k	O(n log k)
Min and max together	`minmax({…})` , `minmax_element`	O(n)
Sum safely	`accumulate(…, 0LL)`	O(n)
Anagram key	`sort` the string	O(L log L)
Sort indexes by value	`iota` + `sort` with a lambda	O(n log n)
Multi-key sort	`tie(…) < tie(…)`	O(n log n)

Pitfalls

`unique` without `erase` leaves the vector its old size.

`lower_bound` on an unsorted range compiles and returns nonsense; on a `set` , use the member.

`map[k]` in a lookup inserts `k` — a `const` map has no `operator[]` for exactly this reason; `find` , `count` or `contains` (C++20) for reads.

A `priority_queue` iterates in heap order, not sorted order; `top()` is the only ordered access.

`minmax(x, y)` with temporaries dangles; `accumulate` with `0` sums in `int` .

A comparator that is not a strict weak ordering makes `sort` undefined behaviour.

Key takeaways

Deduplicate with sort → unique → erase; the erase is the step that shrinks.

`lower_bound` is "first ≥"; `upper_bound` is "first >"; check for `end()` ; use the member on trees.

`++freq[key]` counts; `map` iterates sorted, `unordered_map` does not.

k smallest: a max-heap capped at k; pairs and tuples compare lexicographically, so `tie` gives multi-key ordering and pairs order a heap.

`0LL` in `accumulate`, `minmax({…})` by value, `gcd` before `lcm`.

EXERCISE 1

Distinct values and first-not-below queries

Read an integer `n` and `n` integers (up to `10^12` in magnitude, so `long long`), then an integer `q` and `q` query values. Deduplicate with the **sort + unique + erase** idiom, then answer each query with `std::lower_bound` — the first distinct value that is `≥ x`.

Output:

`distinct=<number of distinct values>`

`values=<the distinct values ascending, space-separated>`, or `values=none` when there are none

for each query, `<x>: <value> at <index into the distinct list>` or `<x>: none` when every value is below `x`

```
7
5 3 9 3 1 5 9
3
4
9
10
```

prints

```
distinct=4
values=1 3 5 9
4: 5 at 2
9: 9 at 3
10: none
```

INPUT 1

```
7
5 3 9 3 1 5 9
3
4
9
10
```

OUTPUT 1

```
distinct=4
values=1 3 5 9
4: 5 at 2
9: 9 at 3
10: none
```

INPUT 2

```
0
2
1
-5
```

OUTPUT 2

```
distinct=0
values=none
1: none
-5: none
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<ll> v(n);
    for (auto& x : v) cin >> x;
    sort(v.begin(), v.end());
    v.erase(unique(v.begin(), v.end()), v.end());
    cout << "distinct=" << v.size() << '\n';
    cout << "values=";
    if (v.empty()) cout << "none";
    for (size_t i = 0; i < v.size(); ++i) cout << (i ? " " : "") << v[i];
    cout << '\n';
    int q;
    cin >> q;
    while (q--) {
        ll x;
        cin >> x;
        auto it = lower_bound(v.begin(), v.end(), x);
        if (it == v.end()) cout << x << ": none\n";
        else cout << x << ": " << *it << " at " << (it - v.begin()) << '\n';
    }
    return 0;
}
```

EXERCISE 2

Word counts and the k rarest

Read an integer `n`, then `n` words, then an integer `k` (`k ≥ 1`). Count the words with a `std::map<std::string, int>` through `++freq[word]` and print every `<word> <count>` in the map's (alphabetical) order. Then find the `k` rarest words with a `std::priority_queue` of `pair<int, string>` (a max-heap): push every `(count, word)` and, whenever the heap holds more than `k`

entries, pop the top — so the largest pairs are evicted and the `k` smallest survive. Print the survivors smallest count first, ties alphabetical, as `rarest=<word>(<count>) …`. When `k` is at least the number of distinct words, every word appears.

```
7
b a c a b a d
2
```

prints

```
a 3
b 2
c 1
d 1
rarest=c(1) d(1)
```

INPUT 1

```
7
b a c a b a d
2
```

OUTPUT 1

```
a 3
b 2
c 1
d 1
rarest=c(1) d(1)
```

INPUT 2

```
3
x x x
5
```

OUTPUT 2

```
x 3
rarest=x(3)
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    map<string, int> freq;
    for (int i = 0; i < n; ++i) {
        string w;
        cin >> w;
        ++freq[w];
    }
    int k;
    cin >> k;
    for (const auto& [word, cnt] : freq) cout << word << ' ' << cnt << '\n';

    priority_queue<pair<int, string>> heap;   // largest (count, word) on top
    for (const auto& [word, cnt] : freq) {
        heap.emplace(cnt, word);
        if (static_cast<int>(heap.size()) > k) heap.pop();   // evict the largest su
    }
    vector<pair<int, string>> rare;
    while (!heap.empty()) {
        rare.push_back(heap.top());
        heap.pop();
    }
    sort(rare.begin(), rare.end());                        // pairs compare (count,
    cout << "rarest=";
    for (size_t i = 0; i < rare.size(); ++i) {
        cout << (i ? " " : "") << rare[i].second << '(' << rare[i].first << ')';
    }
    cout << '\n';
    return 0;
}
```

QUIZ

1. After `std::sort(v.begin(), v.end()); std::unique(v.begin(), v.end());` — and nothing else — `v` is…

A. Deduplicated

B. Still the same size: `unique` moves the survivors to the front and returns the new logical end, which you must pass to `erase`

C. Sorted descending

D. Empty

2. On a sorted `v`, `std::lower_bound(v.begin(), v.end(), x)` returns an iterator to…

A. The first element `≥ x`, or `end()`

B. The first element `> x`, or `end()`

C. The last element `≤ x`

D. `true` if `x` is present

3. What does this print?

```
std::map<std::string, int> m;
if (m["a"] == 0) std::cout << m.size();
```

A. `0`

B. `1`

C. Compile error

D. Undefined behaviour

4. A min-heap of `int` is spelt…

A. `std::priority_queue<int>`

B. `std::priority_queue<int, std::vector<int>, std::greater<>>`

C. `std::priority_queue<int, std::less<>>`

D. `std::priority_queue<int, std::deque<int>>`

5. `std::accumulate(v.begin(), v.end(), 0)` over 10^5 values up to 10^9 each…

A. Is correct — `accumulate` promotes to `long long`

B. Overflows: the accumulator has the type of the initial value, `int`; pass `0LL`

C. Does not compile

D. Returns a `double`

6. What happens here?

```
auto [lo, hi] = std::minmax(3, 7);
std::cout << lo << ' ' << hi;
```

A. Prints `3 7`

B. Undefined behaviour — `minmax(const T&, const T&)` returns references to the two temporaries, which are destroyed at the end of the declaration

C. Compile error

D. Prints `7 3`

7. To sort by `score` descending, then `name` ascending, the comparator is…

A. `std::tie(a.score, a.name) < std::tie(b.score, b.name)`

B. `std::tie(b.score, a.name) < std::tie(a.score, b.name)`

C. `a.score > b.score && a.name < b.name`

D. `std::tie(a.name, a.score) > std::tie(b.name, b.score)`

ANSWER KEY · THE STL IDIOM SHEET — THE TWO-LINE ANSWERS TO THE COMMON SHAPES

1. B Algorithms never change a container's size — they only see iterators. The idiom is `v.erase(std::unique(v.begin(), v.end()), v.end());` and the elements past the returned iterator are unspecified until then.

2. A `lower_bound` is "first not less than"; `upper_bound` is "first greater than"; the two together bound the run of elements equal to `x`, so `upper - lower` counts occurrences. `std::binary_search` is the yes/no form.

3. B `operator[]` inserts a value-initialised entry when the key is absent, so reading `m["a"]` created it. That is what makes `++freq[w]` a one-line counter, and what makes `[]` wrong for a lookup that must not insert — use `find`, `count` or `contains`.

4. B The default comparator is `std::less`, which puts the largest on top. The comparator is the third template argument, so the container must be named too; `std::greater<>` (transparent) flips it. The second option in the list would fail because `std::less<>` is not a container.

5. B The third parameter's deduced type is the running sum's type, whatever the elements are. `0LL` (or `long long{0}`) makes it 64-bit; the same trap hits `std::reduce`.

6. B The two-argument overload returns `std::pair<const T&, const T&>`; with prvalue arguments those references dangle immediately. With named variables it is fine; with temporaries use the initializer-list form `std::minmax({3, 7})`, which returns a `pair<T, T>` by value.

7. B Tuples compare lexicographically; swapping `a` and `b` in one slot reverses that key alone. The `&&` form is not a strict weak ordering (two entries with equal scores compare unordered both ways), and the last option sorts by name first.

LESSON 3 · 14 MIN

The pitfalls that fail interviews

Most rejected C++ solutions are not wrong algorithms. They are right algorithms with an `int` that overflowed, a `size_t` that wrapped, a `%` that went negative, a `map[]` that inserted, or an iterator that was used after the container moved. Interviewers know the list; this lesson is that list, each item with the one-line fix, organised so you can run it as a review pass over your own code before you say "done". Nothing here is new — every item was taught earlier in the track, and most of them are undefined behaviour, which in C++ means the program may do anything, including appear to work on your machine — but seeing them together is what turns knowledge into a reflex.

Arithmetic

`int` products and sums. `a[i] * a[j]` with values to 10^5 is 10^{10}; a sum of 10^5 such values is 10^{10}; `n * (n + 1) / 2` at 10^5 is 5×10^9. Signed overflow is undefined behaviour, and at `-O2` the optimiser may delete the check you wrote to detect it. Fix: `long long` for anything that accumulates or multiplies, and widen *before* the operation — `1LL * a * b`, not `(long long)(a * b)`, which casts a value that already overflowed.

`%` on negatives. `-7 % 3` is `-1`: the remainder takes the dividend's sign, and `/` truncates toward zero (`-7 / 2` is `-3`). Circular indexes, hashing and modular arithmetic need `((a % m) + m) % m`, which is always in `[0, m)`.

`1 << 31`. With a 32-bit `int` the bit lands in the sign position: undefined behaviour before C++20, and since C++20 defined to wrap to `INT_MIN` — either way not 2^{31}, and `1 << 32` is undefined in every standard. The literal's type sets the width: `1LL << k` for a 64-bit power of two, `1u << 31` for a 32-bit mask.

`pow` for integers. `std::pow` returns a `double`; the standard promises no exactness for integer arguments, some libms return 99.999… for `pow(10, 2)`, and past 2^{53} a `double` cannot hold the answer. An integer loop, or `1LL << k` for powers of two, or fast exponentiation with a modulus.

Doubles compared with `==`. `0.1 + 0.2 == 0.3` is `false`. Compare within a tolerance (`std::abs(a - b) < 1e-9`), or avoid doubles: money in `long long` cents, fractions as numerator and denominator, geometry with squared distances.

Integer division where a fraction was meant. `7 / 2` is `3`; `1 / 2 * x` is `0`. One operand must be a `double` before the division, not after.

Unsigned arithmetic

`size()` compared with an `int`. `v.size()` is `size_t`, which is unsigned. In `i < v.size() - 1` on an empty vector, `size() - 1` wraps to $2^{64} - 1$ and the loop runs into memory it does not own. Write `i + 1 < v.size()`, or guard the empty case first. `-Wall` warns on signed/unsigned comparisons; the warning is right often enough to keep.

`size() - t` when `t` may exceed the size. Same wrap. Clamp first: `std::min(t, v.size())`.

The unsigned countdown. `for (size_t i = n - 1; i ⩾ 0; --i)` never ends (Module 3). Count with `i-- > 0`, or index `v[i - 1]` with `i > 0`.

Mixed signed/unsigned comparison. `-1 < v.size()` is `false`: the `-1` converts to a huge unsigned value. Cast the size to `long long` or keep the signed value on both sides.

Containers

`map[]` in a lookup. `if (m[key] > 0)` inserts `key` with `0`. Reads use `find`, `count` or `contains` (C++20); a `const` map has no `operator[]` for exactly this reason.

Erasing while iterating. `for (auto it = v.begin(); it ≠ v.end(); ++it) if (…) v.erase(it);` increments an invalidated iterator: undefined behaviour. Write `it = v.erase(it)` in the erasing branch (and do not increment), or `std::erase_if(v, pred)` in one call.

References into a vector that then grows. `int& first = v[0]; v.push_back(…); use(first);` — the push may reallocate, and `first` refers to freed memory. Take a copy, or index after the push.

`std::vector<bool>`. A packed specialisation whose `operator[]` returns a proxy, not `bool&`: `auto& b = v[0]` fails, `&v[0]` is not a `bool*`, and the element cannot be shared between threads. `std::vector<char>` or `std::deque<bool>` when you need real elements.

Iteration order of `unordered_map`. Unspecified and different between implementations and runs; never let it reach the output. Sort the keys, or use `std::map`.

Dereferencing `end()`. `*std::max_element(v.begin(), v.end())` on an empty vector, `*m.find(k)` when the key is absent, `v.back()` on an empty vector: undefined behaviour every time. Check emptiness and the `end()` result.

Language

Uninitialised locals. `int best; for (…) best = std::max(best, x);` reads an indeterminate value — undefined behaviour that happens to print zero on some runs. Brace-initialise: `int best{};`, `long long sum = 0;`, or start from the first element.

Recursion depth. The judge's stack takes 10^5 frames comfortably (measured for this track); a DFS on a path of 10^6 nodes or a naïve recursive sum of 10^7 does not. Convert to iteration with an explicit `std::vector` stack when the depth is the input size.

`std::endl` in a loop. A flush per line: measured about 500 ms against about 40 ms for 10^6 lines on the lab. `'\n'`.

`auto` copies in range-for. `for (auto s : names)` copies every string; `for (auto& p : pairs) p.second++` modifies in place while `for (auto p : pairs)` modifies copies and changes nothing. `const auto&` to read, `auto&` to modify.

`std::string_view` into a temporary. `std::string_view sv = s + "x";` — the temporary dies at the semicolon and `sv` dangles (Module 5).

A path without `return`. A non-`void` function that falls off the end is undefined behaviour, not a default value; `-Wall` warns "control reaches end of non-void function". Every branch returns.

Comparator that is not a strict weak ordering. `<=` instead of `<`, or `a.x < b.x || a.y < b.y`: `std::sort` may crash or loop. Use `std::tie`.

The review pass

Before submitting, scan once per heading. *Arithmetic:* any `int` that multiplies or accumulates? Any `%` on a value that can be negative? A shift that should be `1LL`? *Unsigned:* any `size() - k`, any `int` compared with `size()`? *Containers:* a `map[]` in a condition? An `erase` inside a loop? A reference held across a `push_back`? *Language:* every local initialised, every function returning on every path, `'\n'` not `endl`, debug output on `cerr`? Thirty seconds, and it catches most "correct but rejected" submissions. In an interview, do it aloud — "the sum is at most 10^{14}, so `long long`; the map lookup uses `find` so it does not insert" — that visible carefulness is what the interviewer is scoring.

Interview angle

"Why is `int` overflow worse in C++ than in Java?" Java wraps; C++ calls it undefined behaviour, so the optimiser may remove the overflow check or the loop that contains it.

"What does `-7 % 3` give, and how do you get `2`?" `-1`; `((a % m) + m) % m`.

"Why did the loop over an empty vector crash?" `size() - 1` is unsigned and wrapped.

"How would you remove elements from a vector while iterating?" `it = v.erase(it)`, or the erase–remove idiom, or `std::erase_if`.

Key takeaways

Widen before you multiply (`1LL * a * b`); `long long` for sums and products; `%` on negatives needs normalising; `1 << 31` is not 2^{31}.

`size()` is unsigned: never subtract from it without a guard, never compare it with a negative.

`map[]` inserts, `erase` invalidates, `push_back` may move everything, `vector<bool>` is not a vector of `bool`.

Initialise every local, return on every path, `'\n'` not `endl`, iterate a `vector` by `const auto&`.

Run the review pass aloud; it is the cheapest thirty seconds in the interview.

EXERCISE 1

Fix the idioms

Four idioms a candidate wrote are broken. Their *intent* is given; write the program that produces the correct results.

Read `n` and `n` integers `a` (each `|a_i| ≤ 10^9`), then `m` (`1 ≤ m ≤ 10^9`), `k` (`0 ≤ k ≤ 62`) and `t` (`0 ≤ t ≤ 10^9`). Print:

`product=<the product of all a_i modulo 1000000007, in [0, MOD)>` — the candidate wrote `int p = 1; p = p * x % MOD;`, which overflows and can go negative; `1` for an empty list.

`mods=<each a_i mod m, in [0, m), space-separated>` — the candidate wrote `x % m`, which is negative for negative `x`; `mods=none` for an empty list.

`pow2=<2^k>` — the candidate wrote `1 << k`, which is wrong from `k = 31`.

`tail=<the last t elements in order>`, all of them when `t ≥ n`, `tail=none` when nothing is printed — the candidate wrote `for (int i = a.size() - t; i < a.size(); ++i)`, which wraps when `t > n`.

`pairs=<the number of adjacent pairs, n - 1, or 0 for an empty list>` — the candidate wrote `a.size() - 1`, which wraps when `n = 0`.

```
4
3 -4 5 2
3
31
2
```

prints

```
product=999999887
mods=0 2 2 2
pow2=2147483648
tail=5 2
pairs=3
```

INPUT 1

```
4
3 -4 5 2
3
31
2
```

OUTPUT 1

```
product=999999887
mods=0 2 2 2
pow2=2147483648
tail=5 2
pairs=3
```

INPUT 2

```
0
7
0
5
```

OUTPUT 2

```
product=1
mods=none
pow2=1
tail=none
pairs=0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;
const ll MOD = 1'000'000'007;

ll floorMod(ll a, ll m) { return ((a % m) + m) % m; }   // always in [0, m)

int main() {
    ios::sync_with_stdio(false);
    cin.tie(nullptr);
    int n;
    cin >> n;
    vector<ll> a(n);
    for (auto& x : a) cin >> x;
    ll m, k, t;
    cin >> m >> k >> t;

    ll product = 1;
    for (ll x : a) product = floorMod(product * floorMod(x, MOD), MOD);   // both fa
    cout << "product=" << product << '\n';

    cout << "mods=";
    if (a.empty()) cout << "none";
    for (size_t i = 0; i < a.size(); ++i) cout << (i ? " " : "") << floorMod(a[i], m
    cout << '\n';

    cout << "pow2=" << (1LL << k) << '\n';

    const size_t take = min(static_cast<size_t>(t), a.size());   // clamp before sub
    cout << "tail=";
    if (take == 0) cout << "none";
    for (size_t i = a.size() - take; i < a.size(); ++i) cout << (i > a.size() - take
    cout << '\n';

    cout << "pairs=" << (a.empty() ? 0 : static_cast<ll>(a.size()) - 1) << '\n';
    return 0;
}
```

EXERCISE 2

Predict, then verify

Read an integer `k` and `k` expression ids in `1..15`. For each id print `<id>: <value>`, where the value is what C++ computes for the expression below — write each expression literally (do not hard-code the answers), print `bool`s as `true` / `false` with `std::boolalpha`, and **predict every line before you run it**:

id	expression	id	expression
1	`7 / 2`	9	`'7' - '0'`
2	`-7 / 2`	10	`0.1 + 0.2 == 0.3`
3	`-7 % 3`	11	`5u - 10`
4	`10 % -3`	12	`std::string("abc").size() - 4`
5	`7 / 2.0`	13	`INT_MAX + 1LL`
6	`static_cast<int>(-3.99)`	14	`1LL << 40`
7	`'a' + 1`	15	`100000 * 100000LL`
8	`char('a' + 1)`		

Every expression is well-defined C++20 (none overflows a signed type). Example: `3` then `1 5 13` →

```
1: 3
5: 3.5
13: 2147483648
```

INPUT 1

```
3
1 5 13
```

OUTPUT 1

```
1: 3
5: 3.5
13: 2147483648
```

INPUT 2

```
6
2 3 4 6 7 8
```

OUTPUT 2

```
2: -3
3: -1
4: 1
6: -3
7: 98
8: b
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

string eval(int id) {
    ostringstream out;
    out << boolalpha;
    switch (id) {
        case 1: out << 7 / 2; break;
        case 2: out << -7 / 2; break;
        case 3: out << -7 % 3; break;
        case 4: out << 10 % -3; break;
        case 5: out << 7 / 2.0; break;
        case 6: out << static_cast<int>(-3.99); break;
        case 7: out << 'a' + 1; break;
        case 8: out << char('a' + 1); break;
        case 9: out << '7' - '0'; break;
        case 10: out << (0.1 + 0.2 == 0.3); break;
        case 11: out << 5u - 10; break;
        case 12: out << string("abc").size() - 4; break;
        case 13: out << INT_MAX + 1LL; break;
        case 14: out << (1LL << 40); break;
        case 15: out << 100000 * 100000LL; break;
        default: out << '?';
    }
    return out.str();
}

int main() {
    int k;
    cin >> k;
    for (int i = 0; i < k; ++i) {
        int id;
        cin >> id;
        cout << id << ": " << eval(id) << '\n';
    }
    return 0;
}
```

QUIZ

1. What happens here?

```
std::vector<int> v;
for (std::size_t i = 0; i < v.size() - 1; ++i) std::cout << i;
```

A. Prints nothing

B. Undefined behaviour — `v.size() - 1` wraps to $2^{64} - 1$ on an empty vector, so the loop runs and reads out of range

C. Compile error

D. Prints `0`

2. `-7 % 3` in C++ is…

A. `2`

B. `-1` — the remainder takes the dividend's sign; `((a % m) + m) % m` gives the non-negative remainder

C. `1`

D. Implementation-defined

3. `1 << 31` with a 32-bit `int`, compiled as C++20, is…

A. `2147483648`

B. `-2147483648` — the result wraps to `INT_MIN` (and was undefined behaviour before C++20); for 2^{31} write `1LL << 31`

C. A compile error

D. `1`

4. For an integer power such as 10^9, `std::pow(10, 9)` …

A. Is exactly `1000000000` on every platform

B. Returns a `double` that a libm may compute a hair under the true value, so `static_cast<int>` can give `999999999`; use an integer loop, or `1LL << k` for powers of two

C. Does not compile

D. Returns an `int`

5. Printing 10^6 lines with `std::endl` instead of `'\n'` …

A. Costs the same

B. Flushes on every line — measured about 500 ms against about 40 ms on the lab, a dozen times slower

C. Is faster because the buffer stays small

D. Does not compile with `sync_with_stdio(false)`

6. What happens here?

```
std::vector<int> v{1, 2, 3, 4};
for (auto it = v.begin(); it != v.end(); ++it)
    if (*it % 2 == 0) v.erase(it);
```

A. Removes the even numbers

B. Undefined behaviour — `erase` invalidates `it` and the loop then increments it; write `it = v.erase(it)` without the `++`, or use `std::erase_if`

C. Compile error

D. Removes only the `2`

7. `0.1 + 0.2 == 0.3` is…

A. `true`

B. `false` — compare with a tolerance such as `std::abs(a - b) < 1e-9`, or keep the values as scaled integers

C. A compile error

D. Undefined behaviour

ANSWER KEY · THE PITFALLS THAT FAIL INTERVIEWS

1. B `size()` is unsigned; subtracting from zero wraps rather than going negative. Write `i + 1 < v.size()` (addition never wraps here) or guard the empty case first. The same wrap is why `size() - t` is wrong when `t` may exceed the size.

2. B Since C++11 `/` truncates toward zero and `a % b == a - (a / b) * b`, so `-7 / 3` is `-2` and the remainder `-1`. Circular indexes and modular arithmetic need the normalised form; `std::abs` on the result is a different (wrong) function.

3. B C++20 defines signed left shift modulo 2^N, so the bit lands in the sign position; shifting by 32 or more is still undefined. The intent is almost always a 64-bit power of two, and the literal's type is what fixes it.

4. B `pow` is a floating-point function with no exactness guarantee for integral arguments in the standard; glibc happens to be exact for these, other libms are not, and past 2^{53} a `double` cannot hold the answer at all. Integer arithmetic is the idiom.

5. B `std::endl` is `'\n'` plus a flush, and a flush is a system call. Buffered output is written once when the program ends; a judge that times you never needs to see a partial line.

6. B Every iterator at or after an erased position is invalid. `erase` returns the iterator to the next element, which is the value the loop must continue from; C++20's `std::erase_if(v, pred)` does the whole thing in one call.

7. B Neither `0.1` nor `0.2` is representable in binary, and the rounding errors do not cancel. Money in `long long` cents and exact fractions as numerator/denominator sidestep it; when doubles are unavoidable, compare within an epsilon that suits the magnitude.

LESSON 4 · 15 MIN

Implement the built-in — a vector, a unique_ptr, a hash map and a string by hand

"Implement `std::vector`" is the question every senior C++ interview asks in some form, because forty lines show whether you understand ownership, can write the rule of five from memory and know why growth doubles. The interviewer is listening for four sentences — *it owns a heap buffer and frees it in the destructor*; *a copy allocates, a move steals the pointer and empties the source*; *a full push doubles, so* `push_back` *is amortised O(1)*; *reallocation invalidates every reference into the old buffer*. This lesson builds the four types they ask for: a `Vec<T>`, a `UniquePtr<T>`, a chaining hash map and, in outline, a `String` with the small-string optimisation. Module 9, Copies, moves and the rule of five, wrote the owning buffer; the new skill is producing it under a clock, narrating.

`Vec<T>`: three members and the growth rule

```cpp
template <typename T>
class Vec {
    T* data_ = nullptr;
    size_t size_ = 0;
    size_t cap_ = 0;

    void grow() {
        const size_t next = cap_ == 0 ? 1 : cap_ * 2;
        T* fresh = new T[next];
        for (size_t i = 0; i < size_; ++i) fresh[i] = std::move(data_[i]);
        delete[] data_;
        data_ = fresh;
        cap_ = next;
    }

public:
    void push_back(T v) {                    // by value: see below
        if (size_ == cap_) grow();
        data_[size_++] = std::move(v);
    }
    T& operator[](size_t i) { return data_[i]; }
    const T& operator[](size_t i) const { return data_[i]; }
    size_t size() const { return size_; }
};
```

The handle is three words, like `std::vector`'s. `grow` allocates the new buffer, moves the live elements across, frees the old one and only then updates the members, so a throwing allocation leaves the object as it was. Doubling is what makes `push_back` cheap: over `n` pushes the moves total 1 + 2 + 4 + … < 2n, a constant per push on average, where growing by a fixed amount would sum to O(n²). Name the simplification aloud: `new T[next]` default-constructs every slot; the real container allocates raw storage and constructs each element on push.

`push_back` takes its parameter by value for a reason: `v.push_back(v[0])` on a full vector must work, as it does for `std::vector`. With `const T& v` the reference points into the old buffer, `grow()` frees it, and the assignment reads freed memory; by value, the copy is made at the call, before anything is reallocated.

The rule of five, written out

```
    ~Vec() { delete[] data_; }
    Vec(const Vec& o) : data_(o.size_ ? new T[o.size_] : nullptr), size_(o.si
        std::copy(o.data_, o.data_ + o.size_, data_);
    }
    Vec(Vec&& o) noexcept : data_(o.data_), size_(o.size_), cap_(o.cap_) {
        o.data_ = nullptr;
        o.size_ = o.cap_ = 0;
    }
    Vec& operator=(const Vec& o) { Vec tmp(o); swap(tmp); return *this; }
    Vec& operator=(Vec&& o) noexcept {
        if (this != &o) {
            delete[] data_;
            data_ = o.data_; size_ = o.size_; cap_ = o.cap_;
            o.data_ = nullptr; o.size_ = o.cap_ = 0;
        }
        return *this;
    }
    void swap(Vec& o) noexcept;             // std::swap the three members
```

Each of the five has one job. The copy constructor allocates exactly `size` elements — a copy has no reason to inherit the source's slack. The move constructor steals the three members and zeroes the source, so the source's destructor deletes `nullptr`; it is `noexcept` because `std::vector<Vec>` moves its elements on reallocation only when the move constructor promises not to throw (Module 9 lesson 3). Copy assignment is copy-and-swap: build the copy first, then exchange, so a failed allocation leaves `*this` untouched (the strong guarantee) and `a = a` swaps harmlessly. Move assignment cannot use that trick without an allocation, so it guards `this != &o` — without the guard, `a = std::move(a)` frees the buffer and then adopts the freed pointer.

Declare any one of these and the compiler stops generating the moves; write none and the defaults copy the pointer, so two `Vec`s free one buffer — or apply the rule of zero: a `std::unique_ptr<T[]>` member owns the buffer and only the two copy operations need writing.

`UniquePtr<T>`

```cpp
template <typename T>
class UniquePtr {
    T* p_ = nullptr;

public:
    explicit UniquePtr(T* p = nullptr) : p_(p) {}
    ~UniquePtr() { delete p_; }
    UniquePtr(const UniquePtr&) = delete;
    UniquePtr& operator=(const UniquePtr&) = delete;
    UniquePtr(UniquePtr&& o) noexcept : p_(o.release()) {}
    UniquePtr& operator=(UniquePtr&& o) noexcept { reset(o.release()); return
    T* release() { T* p = p_; p_ = nullptr; return p; }
    void reset(T* p = nullptr) { T* old = p_; p_ = p; delete old; }
    T& operator*() const { return *p_; }
    explicit operator bool() const { return p_ != nullptr; }
};
```

Every line is an interview answer. Copying is deleted because two owners would delete twice — the mistake becomes a compile error. The moves are written through `release` and `reset`, which makes self-move-assignment safe without a guard: `release` nulls `o` first, then `reset` deletes the old pointer *after* installing the new one. `explicit operator bool` allows `if (p)` but refuses `int n = p;`. The size is one pointer, the same as a raw `T*`, which is what "zero overhead" means.

A chaining hash map

```
class HashMap {
    std::vector<std::vector<std::pair<std::string, int>>> buckets_;
    size_t size_ = 0;

    size_t bucketOf(const std::string& key) const {
        size_t h = 0;
        for (unsigned char c : key) h = (h * 31 + c) % buckets_.size();
        return h;
    }

public:
    explicit HashMap(size_t buckets) : buckets_(buckets) {}
    void put(const std::string& key, int value) {
        auto& chain = buckets_[bucketOf(key)];
        for (auto& [k, v] : chain) if (k == key) { v = value; return; }
        chain.emplace_back(key, value);
        ++size_;
    }
};
```

A hash map is a vector of buckets plus a function from key to bucket index; with chaining, each bucket is a small vector of pairs and a collision appends. `put` walks the chain first so an existing key is overwritten in place; `get` walks it the same way and returns `std::optional<int>` rather than a sentinel; `erase` finds the pair, calls `chain.erase(it)` and returns at once, so the invalidated iterator is never incremented. Reducing modulo `B` at every step keeps the intermediate below 31 × 63 + 255, and `unsigned char` stops a high-bit `char` going negative. The cost is O(1 + n/B) on average — the load factor — and O(n) when everything collides, which is why `std::unordered_map` rehashes past a load factor of 1.0; the follow-up "how would you add rehashing?" is answered with `2B` buckets and a re-`put` of every pair, amortised like the vector's growth.

`String` with SSO, in outline

```
class String {
    char* ptr_;                                   // points at local_ or a
    size_t size_ = 0;
    union { char local_[16]; size_t heapCap_; };  // 15 characters + '\0'
    bool isLocal() const { return ptr_ == local_; }
};
```

That is libstdc++'s real layout, and why `sizeof(std::string)` is 32 on this platform. A string of at most fifteen characters lives inside the object, in the bytes that would otherwise hold the heap capacity; a longer one allocates and `ptr_` points at the heap. Every member function first asks which mode is active, and the instructive case is the move: a small string has nothing on the heap to steal, so moving it copies sixteen bytes and empties the source — "moves are free" is not quite true for strings. You will not be asked to write it; you may be asked why short strings never allocate.

Pitfalls

A copy constructor that copies `cap_` allocates slack the copy never uses; a move constructor that forgets to zero the source double-frees.

Move assignment without the `this ≠ &o` guard frees its own buffer on self-move.

`push_back(const T&)` plus `grow()` dangles on `v.push_back(v[0])`.

`erase` inside a range-`for` over a chain is undefined behaviour; use the iterator form and return.

Key takeaways

`Vec<T>` is three members plus a `grow()` that doubles; 1 + 2 + 4 + … < 2n is why `push_back` is amortised O(1).

The five: the destructor frees; the copy allocates `size`; the move steals and zeroes, `noexcept`; copy assignment by copy-and-swap; move assignment guards self-assignment.

`UniquePtr` deletes copying, moves through `release` / `reset`, and is one pointer wide.

A chaining hash map is buckets of pairs: overwrite in place, append on a miss, `erase` then return; the cost is the load factor.

SSO keeps fifteen characters inside the object: small strings never allocate, and moving one is a copy.

EXERCISE 1

Vec<T> with the rule of five

Implement `template <typename T> class Vec` — a growable array with a raw `T*` buffer, `size()`, `capacity()`, `push_back`, `pop_back`, `operator[]`, and the **rule of five** written by hand (destructor, copy constructor, copy assignment, move constructor, move assignment). Growth policy: capacity starts at `0`; a push into a full vector reallocates to `1` when the capacity is `0` and to **double** otherwise. A copy allocates exactly the source's size (so its capacity equals its size); a moved-from `Vec` is left empty with capacity `0`; `pop_back` never shrinks.

Drive a `Vec<long long>` with `n` commands:

- `push x` — append; `pop` — remove the last, or print `empty` when there is none
- `get i` — print element `i`, or `out of range`
- `stats` — print `size=<s> capacity=<c>`
- `print` — the elements space-separated, or `empty`
- `copy` — copy-construct a second `Vec`, push `-1` onto the copy, then print `original: <elements> (size=<s> capacity=<c>)` and `copy: <elements> (size=<s> capacity=<c>)`
- `move` — move-construct a second `Vec` from the current one, print `moved: size=<s> capacity=<c>` and `source: size=<s> capacity=<c>`; the current `Vec` stays in its moved-from state for later commands

```
9
push 1
push 2
push 3
stats
print
get 1
get 3
pop
stats
```

prints

```
size=3 capacity=4
1 2 3
2
out of range
size=2 capacity=4
```

INPUT 1

```
9
push 1
push 2
push 3
stats
print
get 1
get 3
pop
stats
```

OUTPUT 1

```
size=3 capacity=4
1 2 3
2
out of range
size=2 capacity=4
```

INPUT 2

```
5
stats
pop
print
push 7
copy
```

OUTPUT 2

```
size=0 capacity=0
empty
empty
original: 7 (size=1 capacity=1)
copy: 7 -1 (size=2 capacity=2)
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

template <typename T>
class Vec {
    T* data_ = nullptr;
    size_t size_ = 0;
    size_t cap_ = 0;

    void grow() {
        const size_t next = cap_ == 0 ? 1 : cap_ * 2;
        T* fresh = new T[next];
        for (size_t i = 0; i < size_; ++i) fresh[i] = std::move(data_[i]);
        delete[] data_;
        data_ = fresh;
        cap_ = next;
    }

public:
    Vec() = default;
    ~Vec() { delete[] data_; }

    Vec(const Vec& o) : data_(o.size_ ? new T[o.size_] : nullptr), size_(o.size_), c
        std::copy(o.data_, o.data_ + o.size_, data_);
    }
    Vec(Vec&& o) noexcept : data_(o.data_), size_(o.size_), cap_(o.cap_) {
        o.data_ = nullptr;
        o.size_ = o.cap_ = 0;
    }
    Vec& operator=(const Vec& o) {          // copy-and-swap: strong guarantee, self
        Vec tmp(o);
        swap(tmp);
        return *this;
    }
    Vec& operator=(Vec&& o) noexcept {
        if (this != &o) {
            delete[] data_;
            data_ = o.data_;
            size_ = o.size_;
            cap_ = o.cap_;
            o.data_ = nullptr;
            o.size_ = o.cap_ = 0;
        }
        return *this;
    }
    void swap(Vec& o) noexcept {
        std::swap(data_, o.data_);
        std::swap(size_, o.size_);
        std::swap(cap_, o.cap_);
```

```
    }

    size_t size() const { return size_; }
    size_t capacity() const { return cap_; }
    bool empty() const { return size_ == 0; }
    void push_back(T v) {                   // by value: v is already a copy if it c
        if (size_ == cap_) grow();
        data_[size_++] = std::move(v);
    }
    void pop_back() { --size_; }
    T& operator[](size_t i) { return data_[i]; }
    const T& operator[](size_t i) const { return data_[i]; }
};

template <typename T>
void printElements(const Vec<T>& v) {
    if (v.empty()) cout << "empty";
    for (size_t i = 0; i < v.size(); ++i) cout << (i ? " " : "") << v[i];
}

int main() {
    int n;
    cin >> n;
    Vec<long long> v;
    for (int i = 0; i < n; ++i) {
        string cmd;
        cin >> cmd;
        if (cmd == "push") {
            long long x;
            cin >> x;
            v.push_back(x);
        } else if (cmd == "pop") {
            if (v.empty()) cout << "empty\n";
            else v.pop_back();
        } else if (cmd == "get") {
            size_t idx;
            cin >> idx;
            if (idx < v.size()) cout << v[idx] << '\n';
            else cout << "out of range\n";
        } else if (cmd == "stats") {
            cout << "size=" << v.size() << " capacity=" << v.capacity() << '\n';
        } else if (cmd == "print") {
            printElements(v);
            cout << '\n';
        } else if (cmd == "copy") {
            Vec<long long> c = v;
            c.push_back(-1);
            cout << "original: ";
            printElements(v);
```

```
            cout << " (size=" << v.size() << " capacity=" << v.capacity() << ")\n";
            cout << "copy: ";
            printElements(c);
            cout << " (size=" << c.size() << " capacity=" << c.capacity() << ")\n";
        } else if (cmd == "move") {
            Vec<long long> taken = std::move(v);
            cout << "moved: size=" << taken.size() << " capacity=" << taken.capacity
            cout << "source: size=" << v.size() << " capacity=" << v.capacity() << '
        }
    }
    return 0;
}
```

EXERCISE 2

A chaining hash map

Implement a hash map from `std::string` to `int` with **separate chaining**: a `vector` of `B` buckets, each a `vector<pair<string, int>>`. The hash is fixed so the buckets are deterministic: start with `h = 0` and, for each character `c` of the key taken as an `unsigned char`, set `h = (h * 31 + c) % B`. A new key is **appended** to the end of its chain; overwriting an existing key keeps its position.

Read `B` (`1 ≤ B ≤ 64`), then `n`, then `n` commands:

- `put key value` — insert or overwrite; prints nothing
- `get key` — print the value, or `missing`
- `del key` — print `deleted` or `missing`
- `dump` — one line per bucket, `<b>: k=v k=v …` in chain order or `<b>: -` when empty, then `size=<number of keys>`

```
4
7
put ab 1
put cd 2
put ab 3
get ab
get zz
dump
del cd
```

prints

```
3
missing
0: -
1: ab=3 cd=2
2: -
3: -
size=2
deleted
```

(`ab` hashes to `((97 % 4) * 31 + 98) % 4 = 1`; so does `cd`, and it joins the chain behind `ab`.)

INPUT 1

```
4
7
put ab 1
put cd 2
put ab 3
get ab
get zz
dump
del cd
```

OUTPUT 1

```
3
missing
0: -
1: ab=3 cd=2
2: -
3: -
size=2
deleted
```

INPUT 2

```
1
5
put x 1
put y 2
put z 3
del y
dump
```

OUTPUT 2

```
deleted
0: x=1 z=3
size=2
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

class HashMap {
    vector<vector<pair<string, int>>> buckets_;
    size_t size_ = 0;

    size_t bucketOf(const string& key) const {
        size_t h = 0;
        for (unsigned char c : key) h = (h * 31 + c) % buckets_.size();
        return h;
    }

public:
    explicit HashMap(size_t buckets) : buckets_(buckets) {}

    void put(const string& key, int value) {
        auto& chain = buckets_[bucketOf(key)];
        for (auto& [k, v] : chain) {
            if (k == key) {
                v = value;
                return;
            }
        }
        chain.emplace_back(key, value);
        ++size_;
    }

    optional<int> get(const string& key) const {
        const auto& chain = buckets_[bucketOf(key)];
        for (const auto& [k, v] : chain) {
            if (k == key) return v;
        }
        return nullopt;
    }

    bool erase(const string& key) {
        auto& chain = buckets_[bucketOf(key)];
        for (auto it = chain.begin(); it != chain.end(); ++it) {
            if (it->first == key) {
                chain.erase(it);
                --size_;
                return true;
            }
        }
        return false;
    }

    size_t size() const { return size_; }
```

```
    void dump() const {
        for (size_t b = 0; b < buckets_.size(); ++b) {
            cout << b << ':';
            if (buckets_[b].empty()) cout << " -";
            for (const auto& [k, v] : buckets_[b]) cout << ' ' << k << '=' << v;
            cout << '\n';
        }
        cout << "size=" << size_ << '\n';
    }
};

int main() {
    size_t B;
    int n;
    cin >> B >> n;
    HashMap table(B);
    for (int i = 0; i < n; ++i) {
        string cmd;
        cin >> cmd;
        if (cmd == "put") {
            string key;
            int value;
            cin >> key >> value;
            table.put(key, value);
        } else if (cmd == "get") {
            string key;
            cin >> key;
            if (auto v = table.get(key)) cout << *v << '\n';
            else cout << "missing\n";
        } else if (cmd == "del") {
            string key;
            cin >> key;
            cout << (table.erase(key) ? "deleted" : "missing") << '\n';
        } else if (cmd == "dump") {
            table.dump();
        }
    }
    return 0;
}
```

QUIZ

1. The rule of five says…

A. Every class must declare five constructors

B. A class that needs a user-defined destructor, copy constructor or copy assignment almost certainly needs all three, and should also define the move constructor and move assignment — or it silently loses moves

C. Five is the maximum number of members a class should have

D. Templates need five special members, other classes three

2. Growth by doubling makes `push_back` amortised O(1) because…

A. Each push copies at most one element

B. Over `n` pushes the copies total at most 1 + 2 + 4 + … < 2n, so the average per push is constant even though one push may copy everything

C. The allocator is O(1)

D. Doubling is O(log n) per push

3. A `Vec` whose `push_back(const T& v)` calls `grow()` and then writes `data_[size_++] = v;` is used as `v.push_back(v[0])` while full. What happens?

A. Works as expected

B. Undefined behaviour — `v` refers into the old buffer, which `grow()` freed before the assignment; take the parameter by value or copy it before growing

C. Compile error

D. Pushes `0`

4. Why should `Vec`'s move constructor be `noexcept`?

A. It is a language requirement for move constructors

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

C. It makes the move faster

D. It disables the copy constructor

5. Lookup in a chaining hash map with `n` keys and `B` buckets costs…

A. Always O(1)

B. O(1 + n/B) on average — constant when the load factor is bounded — and O(n) in the worst case when every key lands in one bucket

C. O(log n)

D. O(B)

6. A `UniquePtr<T>`'s copy constructor should be…

A. `= default`

B. `= delete` — ownership is exclusive, so transfer is by move only

C. A deep copy of the pointee

D. Left undeclared so the compiler generates it

7. Copy-and-swap assignment — `Vec& operator=(const Vec& o) { Vec tmp(o); swap(tmp); return *this; }` — gives you…

A. Nothing over a hand-written loop

B. Self-assignment safety and the strong exception guarantee for free: the copy is complete before anything in `*this` changes, and `tmp` destroys the old buffer on the way out

C. A compile error, because `swap` is not defined

D. A move where a copy was requested

ANSWER KEY · IMPLEMENT THE BUILT-IN — A VECTOR, A UNIQUE_PTR, A HASH MAP AND A STRING BY HAND

1. B Declaring a destructor or a copy operation suppresses the implicit moves, so a resource-owning class without hand-written moves copies where `std::vector` would have moved. The rule of zero is the alternative: let `std::vector`, `std::string` and `std::unique_ptr` members own the resources and write none of the five.

2. B The geometric series bounds the total work; growing by a constant amount instead (say +16) makes the copies sum to O(n²). `std::vector` in libstdc++ doubles; other libraries use 1.5.

3. B The reference was formed before the reallocation and dangles after it. `std::vector` guarantees `v.push_back(v[0])` works, which is exactly why a by-hand implementation must copy first; passing by value makes the copy happen at the call, before `grow()` runs.

4. B `std::move_if_noexcept` is what `vector` uses: a throwing move half-way through a reallocation would leave both buffers partly moved, so it falls back to copying. Stealing three pointers cannot throw, so say so.

5. B You hash to a bucket in O(1) and then walk its chain, whose expected length is the load factor. `std::unordered_map` rehashes to keep that near 1; with a fixed `B` and a bad hash the chains grow linearly.

6. B Two owners would `delete` the same object twice. Deleting the copy operations makes the mistake a compile error; the move constructor steals the pointer and nulls the source, and the destructor deletes whatever it still holds.

7. B If the allocation in `tmp` throws, `*this` is untouched; if `o` is `*this`, the copy is of a valid object and the swap is harmless. The cost is one extra allocation compared with reusing existing capacity, which for a reference implementation is the right trade.

LESSON 5 · 12 MIN

Writing clean solutions under time pressure

Two candidates solve the same problem correctly in the same forty minutes, and one is hired. The difference is almost always what the interviewer could *see*: a structure that made the correctness obvious, names that said what things were, edge cases handled in visible lines, `const` and `long long` in the right places, and a complexity stated without prompting. Clean code in an interview is not style points; it is how you make your thinking legible to someone deciding whether to trust you with a codebase. These are the C++ habits that produce that legibility without costing minutes.

Structure: read, solve, print

```
optional<size_t> firstFault(string_view s);          // pure: input in, answer

int main() {
    string line;
    while (getline(cin, line)) {
        if (auto at = firstFault(line)) cout << "unbalanced at " << *at << '\n
        else cout << "balanced\n";
    }
    return 0;
}
```

`main` reads and prints; the function computes and touches no stream and no global. The interviewer reads the algorithm without wading through parsing; you can check `firstFault("(]")` in your head without typing input; and when the input format changes, only `main` changes. A function that both computes and prints is twice as hard to reason about.

Take input by `const&` or `string_view` and return the answer by value: the copy is elided (Module 9 lesson 5). An output parameter — `void solve(const vector<ll>& a, ll& answer)` — is the C habit the interviewer hopes not to see. Return the right thing, too: `std::optional<T>` for "maybe an answer" rather than a `-1` sentinel that a legitimate negative result will collide with, and a `std::pair` or a two-field `struct` for two results, unpacked with a structured binding at the call.

Helpers with one job

`isOpener(c)`, `matches(open, close)`, `inBounds(r, c)`, `neighbours(node)` — a five-line helper costs fifteen seconds and removes a nested block from the main logic. It also lets you *say* the plan before typing it: "a predicate for openers, a predicate for matching pairs, then a stack walk." In C++ the helpers go above `main` as free functions, or get a one-line declaration at the top and their bodies below; a function used before it is declared is a compile error.

Names that carry the invariant

`lo` / `hi` for a binary search, `left` / `right` for two pointers, `seen`, `prefix`, `best`, `openers` — the conventional names are documentation because every reader has met them. Beyond those, a name says what is true: `firstUnsortedIndex`, `windowSum`, `remainingBudget`; never `temp`, `data`, `flag`. C++ adds a rule the other tracks do not have: under `using namespace std;` the names `count`, `size`, `rank`, `distance`, `next`, `prev`, `left` and `right` belong to the library, so a function called `rank` is an ambiguous reference — write `rankEntries`, `countPairs`. A variable that needs a comment to say what it holds needs a better name.

`const` and `auto` where they help

```
for (const auto& s : names) …            // read: no copy
for (auto& p : pairs) ++p.second;         // modify in place
for (long long x : v) sum += x * x;      // widen before the multiply
const char c = s[i];                      // will not change: say so
```

`const` on a local that never changes is a sentence the reader does not have to verify; `const T&` on a parameter is a promise not to modify the caller's object; a `const` member function is the same promise for `this`. `auto` earns its place for iterators, lambdas, structured bindings and map entries — wherever the spelled type is long and adds nothing. It does not earn its place where the type *is* the point: `auto total = 0;` is an `int`, the overflow from the pitfalls lesson in disguise; `long long total = 0;` says what you meant. The range-`for` rule: `const auto&` to read, `auto&` to modify, plain `auto` only when you want a copy; `long long x : v` over a `vector<int>` widens each element before it is used.

Make the edge cases visible

```
if (v.empty()) return 0;                    // no elements
if (v.size() == 1) return v[0];             // one element
```

Guard clauses at the top, one per line, each a sentence the interviewer can tick off. They keep the main loop free of special cases, and in C++ an early return is safe: RAII means nothing leaks on the way out, so the single-exit rule is a C-era habit you may drop. While writing the loop, state the invariant once — "at the top of each iteration `openers` holds every unmatched opener, innermost last" — which convinces the interviewer, and you, without tracing.

Comments: why, not what

`// increment i` is noise. `// v is sorted, so once v[i] > target no later pair can work` justifies a `break` that would otherwise look like a bug. Comment the non-obvious decision, the invariant, the reason a special case exists. Delete the debugging prints before saying "done": a stray `cout` is a wrong answer.

Say the complexity, then the trade-off

"O(n log n) for the sort, then O(n) for the sweep, O(n) extra for the map. A hash set would make it O(n) on average but lose the ordered output, so I would keep the tree unless n is large." That sentence, unprompted, is the highest-leverage thing you can say: it shows you know what you built and that you weighed the alternative — which is the job.

Test in your head, in order

- The example from the statement — trace it; do not glance at it.
- The smallest input: empty, one element, a single line.
- A boundary: the window exactly `k` wide, the target equal to the first element.
- An adversarial one: all equal, all negative, sorted descending, a value that overflows `int`.

Then the review pass from the pitfalls lesson — `int` products, `size()` arithmetic, `map[]`, `erase` in a loop, uninitialised locals, `endl`. Tracing a four-element input aloud takes a minute

and finds most off-by-ones.

When you are stuck

Say what you know: "I need better than O(n²); sorting gives me order, but I do not yet see how to use it." Then simplify — small n, the sorted case, one query — and grow the solution. A brute force written cleanly and *labelled* as the baseline earns partial credit and usually shows where the optimisation is: the repeated scan becomes a sort, the repeated lookup a map.

What interviewers score

Dimension	What it looks like in C++
Correctness	The example and the edge cases pass; no undefined behaviour in the hidden cases
Complexity	Stated unprompted, with the trade-off
Ownership and types	`const&` parameters, `long long` where it matters, no raw `new`, nothing dangling
Structure	read → solve → print, helpers with one job, names that carry meaning
Communication	The plan before the code, the invariant during, the review pass after

The last row is where C++ candidates most often lose: interviewers cannot score what they do not hear.

Pitfalls

- `auto total = 0;` is an `int`; write the type when the type is the point.
- `for (auto s : names)` copies every string; `for (auto p : pairs) p.second++` modifies copies.
- A helper named `rank`, `count` or `size` under the using-directive is ambiguous.
- Debug output on `cout`, or a `cerr` left inside a hot loop.

Key takeaways

Read → solve (pure, `const&` in, value out) → print; helpers with one job above `main`.

Conventional names, invariant-carrying names elsewhere; not the library's names under `using namespace std;`.

`const auto&` to read, `auto&` to modify, `long long` where it accumulates; `const` on anything that does not change.

Guard clauses for edge cases, a stated invariant for the loop, comments for *why*.

Say the complexity and the trade-off unprompted; trace the example and the smallest inputs aloud; when stuck, write the labelled brute force.

EXERCISE 1

Balanced brackets, cleanly

Read lines until the end of input. For each line decide whether its brackets `()[]{}` are balanced (every other character is ignored). Print `balanced`, or `unbalanced at <index>` where the index is the position of the first closing bracket that does not match — or, if some opening brackets are never closed, the position of the **earliest** unclosed opener.

Structure the solution the way the lesson says: small pure helpers `isOpener(char)` and `matches(char open, char close)`, and a pure `firstFault(std::string_view)` that returns `std::optional<std::size_t>` — `std::nullopt` when balanced. `main` only reads and prints.

```
{a[b](c)}
()
((x)
```

prints

```
balanced
unbalanced at 1
unbalanced at 0
```

INPUT 1

```
{a[b](c)}
()
((x)
```

OUTPUT 1

```
balanced
unbalanced at 1
unbalanced at 0
```

INPUT 2

```
)
```

OUTPUT 2

```
balanced
unbalanced at 0
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

bool isOpener(char c) { return c == '(' || c == '[' || c == '{'; }
bool isCloser(char c) { return c == ')' || c == ']' || c == '}'; }
bool matches(char open, char close) {
    return (open == '(' && close == ')') || (open == '[' && close == ']') || (open =
}

// The index of the first offending bracket, or nullopt when balanced.
optional<size_t> firstFault(string_view s) {
    vector<size_t> openers;                        // indexes of unmatched openers, i
    for (size_t i = 0; i < s.size(); ++i) {
        const char c = s[i];
        if (isOpener(c)) {
            openers.push_back(i);
        } else if (isCloser(c)) {
            if (openers.empty() || !matches(s[openers.back()], c)) return i;
            openers.pop_back();
        }
    }
    if (openers.empty()) return nullopt;
    return openers.front();                        // the earliest unclosed opener is
}

int main() {
    string line;
    while (getline(cin, line)) {
        if (auto at = firstFault(line)) cout << "unbalanced at " << *at << '\n';
        else cout << "balanced\n";
    }
    return 0;
}
```

EXERCISE 2

A leaderboard with tuple keys

Read an integer `n` and `n` lines `name score seconds` (`score` and `seconds` are non-negative integers that may exceed `int`). Rank the entries by **score descending, then seconds ascending,**

then name ascending, using a `std::tie` / `std::tuple` comparison rather than a chain of `if`s. Print `<rank>. <name> <score> <seconds>` for each entry, then `winner=<name of rank 1>`.

Structure it: a `struct Entry`, a `readEntries(int n)` that returns a `vector<Entry>`, a `rankEntries(vector<Entry>&)` that sorts in place (the name `rank` would be ambiguous with `std::rank` under the using-directive), and a `main` that only wires them together.

```
4
ann 90 120
bob 95 300
cy 90 100
dee 95 300
```

prints

```
1. bob 95 300
2. dee 95 300
3. cy 90 100
4. ann 90 120
winner=bob
```

INPUT 1

```
4
ann 90 120
bob 95 300
cy 90 100
dee 95 300
```

OUTPUT 1

```
1. bob 95 300
2. dee 95 300
3. cy 90 100
4. ann 90 120
winner=bob
```

INPUT 2

```
1
solo 0 0
```

OUTPUT 2

```
1. solo 0 0
winner=solo
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Entry {
    string name;
    long long score;
    long long seconds;
};

vector<Entry> readEntries(int n) {
    vector<Entry> out;
    out.reserve(n);
    for (int i = 0; i < n; ++i) {
        Entry e;
        cin >> e.name >> e.score >> e.seconds;
        out.push_back(std::move(e));
    }
    return out;
}

void rankEntries(vector<Entry>& entries) {
    sort(entries.begin(), entries.end(), [](const Entry& a, const Entry& b) {
        // score descending (b before a in that slot), then seconds and name ascendi
        return tie(b.score, a.seconds, a.name) < tie(a.score, b.seconds, b.name);
    });
}

int main() {
    int n;
    cin >> n;
    vector<Entry> entries = readEntries(n);
    rankEntries(entries);
    for (size_t i = 0; i < entries.size(); ++i) {
        const Entry& e = entries[i];
        cout << i + 1 << ". " << e.name << ' ' << e.score << ' ' << e.seconds << '\r
    }
    if (!entries.empty()) cout << "winner=" << entries.front().name << '\n';
    return 0;
}
```

QUIZ

1. Keeping `solve()` separate from `main`'s reading and printing mainly buys…

A. Speed

B. Testability, readability, and an easy change when the input source changes — a pure function can be checked in your head and reused

C. Fewer includes

D. Nothing measurable

2. A guard clause such as `if (v.empty()) return 0;` at the top of a function…

A. Is poor style — one exit point only

B. Makes the edge case visible in one line and keeps the main algorithm free of special cases

C. Slows the function

D. Belongs in a `catch`

3. The highest-leverage sentence to say unprompted after coding is…

A. "I think it works"

B. The time and space complexity, and the trade-off you considered

C. "C++ is fast"

D. The line count

4. A good comment in a solution explains…

A. What each line does

B. Why a non-obvious decision is correct or which invariant holds

C. The language syntax

D. Nothing — solutions should have no comments

5. Which is the clean C++ for "sum of squares of a `vector<int>`" when the sum may exceed `int`?

A. `int s = 0; for (auto x : v) s += x * x;`

B. `long long s = 0; for (long long x : v) s += x * x;`

C. `std::accumulate(v.begin(), v.end(), 0, [](int a, int x) { return a + x * x; })`

D. `double s = 0; for (auto x : v) s += x * x;`

6. `for (const auto& s : names)` over a `std::vector<std::string>` …

A. Copies each string

B. Reads each string through a reference without copying; plain `auto` would copy every element

C. Does not compile

D. Allows `s` to be modified

7. When stuck in an interview, the right move is…

A. Silence until the optimal solution appears

B. Say what you know, write a clean, labelled brute force, then look for the redundant work in it

C. Give up on the question

D. Ask for the answer

ANSWER KEY · WRITING CLEAN SOLUTIONS UNDER TIME PRESSURE

1. B The interviewer reads the algorithm without wading through parsing, and when the question changes ("now the numbers arrive one per line") only `main` changes. A function that both computes and prints is twice as hard to reason about.

2. B One line per edge case, each a sentence the interviewer can tick off. The single-exit rule is a C-era habit that RAII made unnecessary in C++ — nothing leaks on an early return.

3. B It shows you know what you built and that you weighed the alternative — "O(n log n) for the sort; a hash set would make it O(n) but lose the ordered output". Interviewers score reasoning, not typing.

4. B `// v is sorted, so once v[i] > target no later pair can work` justifies a `break`; `// increment i` is noise. Delete the debugging prints before saying "done" — a stray `cerr` is harmless, a stray `cout` is a wrong answer.

5. B Naming the loop variable `long long` widens each element before the multiplication, so both the square and the sum are 64-bit. The first two alternatives compute `x * x` in `int` (undefined on overflow); `double` loses exactness past 2^{53}.

6. B `auto` deduces a value and so copies; `const auto&` binds to the element in place. The rule of thumb: `const auto&` to read, `auto&` to modify, `auto` only when you want a copy.

7. B A working O(n²) with a stated complexity earns partial credit and, often, shows where the O(n log n) is hiding — the repeated scan becomes a sort, the repeated lookup becomes a map.

LESSON 6 · 25 MIN

The C++ theory drill — thirty questions, thirty answers

Every C++ interview has a theory section, and its questions have barely changed since C++11: the rule of five, how a virtual call works, what `std::move` actually does, when `shared_ptr` is right. This lesson is the drill: thirty questions, each with the answer an interviewer wants to hear — the definition first, the one consequence that matters, then a stop so the follow-up can come — and a pointer to the module that teaches it in depth. Read it twice a week before interviews. A C++ answer is expected to name a mechanism: not "moves are faster" but "a move steals the pointer and leaves the source empty".

The language and the toolchain

What happens between a `.cpp` file and a running program? The preprocessor expands `#include` and macros into a translation unit; the compiler turns each unit into an object file, checking every use against a declaration; the linker joins the object files and libraries, resolving each name to exactly one definition. A compile error lives inside one unit; an "undefined reference" is a linker error — declared, never defined. *(Module 1)*

What is undefined behaviour, and how does it differ from implementation-defined and unspecified? A construct the standard places no requirements on — signed overflow, an out-of-range index, a dangling dereference, a data race — so the compiler may assume it never happens and optimise on that assumption, which is why `if (x + 1 < x)` can be deleted. Implementation-defined behaviour is documented by the compiler (`sizeof(int)`); unspecified behaviour is one of several allowed outcomes (the order function arguments are evaluated). *(1, 19)*

`const` versus `constexpr`? `const` promises the value will not change after initialisation, which may happen at run time — `const int n = read();` is legal. `constexpr` promises the value is computable at compile time, so it may be used where a constant expression is required: an array bound, a template argument, a `static_assert`. A `constexpr` function still runs at run time on run-time arguments. *(2, 16)*

What does `auto` deduce? The type of the initialiser with references and top-level `const` stripped, by the rules of template argument deduction: `auto x = ref;` is a copy, `auto& x` keeps the reference, `const auto&` binds to anything without copying. The traps are `auto total = 0;` — an `int` — and `for (auto s : names)`, which copies every string. *(2)*

References versus pointers? A reference is an alias that must be bound at initialisation, cannot be reseated and cannot be null; a pointer is a value that can be null, reassigned and used in arithmetic. A reference for a parameter that must refer to an object, a pointer for "may be absent" or "may change what it points to", a smart pointer for ownership. Both can dangle. *(6)*

By value, by reference or by `const` reference? By value for anything register-sized or that the function needs its own copy of anyway; by `const T&` for anything larger that is only read; by `T&` for an in-out parameter, which should be rare because returning a value is clearer and elided. `std::string_view` and `std::span` are the modern spellings of "read-only, any source". *(4)*

What is `std::string_view`, and what is its hazard? A non-owning pointer-plus-length over characters it does not own: cheap to copy, O(1) `substr`, the right parameter type for read-only text. It dangles the moment the string it views dies — `std::string_view sv = s + "x";` views a temporary destroyed at the semicolon — so it is a parameter or a local, never a member that outlives its source. *(5)*

What does a lambda capture, and what can go wrong? `[=]` copies each variable it uses into the closure; `[&]` stores references. A `[&]` lambda that outlives its scope — returned, stored, handed to a thread — refers to dead locals; capture by value, or `[x = std::move(x)]`, when the lambda escapes. Capturing `this` is a pointer by another name. *(4, 14)*

Objects and ownership

What is RAII? Resource Acquisition Is Initialisation: a resource's lifetime is tied to an object's lifetime — acquired in the constructor, released in the destructor, which runs on every exit from the scope, including exceptions and early returns. It is why C++ needs no `finally`: `std::vector`, `std::unique_ptr`, `std::lock_guard` and `std::ofstream` are all RAII, and a raw owning pointer with no destructor is the bug. *(7)*

The rule of three, five and zero? If you write any of the destructor, copy constructor or copy assignment, you almost certainly need all three, because the memberwise defaults copy a pointer and free it twice. Five adds the move constructor and move assignment: a user-declared destructor or copy suppresses the implicit moves, so an owning class without them copies where it could move. Zero is the alternative — let `std::vector`, `std::string` and `std::unique_ptr` members own the resources and write none. *(9)*

What does `std::move` do? Nothing at run time. It is a cast to an rvalue reference that lets overload resolution pick the move constructor or move assignment; the move happens

there, and the source is left valid but unspecified — fit for assignment and destruction, not for anything that assumes its contents. `std::move` on a `const` object silently selects the copy. *(9)*

Why return by value? Because the copy is elided: since C++17 returning a prvalue constructs the result directly in the caller, and a named local is elided by NRVO or, failing that, moved. `return std::move(local);` switches the elision off and is a pessimisation. Passing an output reference costs clarity and gains nothing. *(9)*

`std::unique_ptr` versus `std::shared_ptr`? Exclusive ownership, one pointer wide, move-only, freed in the destructor: the default. Shared ownership through a reference-counted control block, copyable, an atomic increment per copy, freed when the last owner goes: for lifetimes that are genuinely shared, which is rarer than it looks. `std::make_shared` puts object and control block in one allocation. *(7)*

Why can `shared_ptr` leak, and what is `weak_ptr`? Two objects holding `shared_ptr`s to each other never reach a count of zero, so neither is freed. A `std::weak_ptr` observes without owning: it does not count, and `lock()` yields a `shared_ptr` if the object is still alive or an empty one if not. The back edge of a parent–child relationship is its textbook place. *(7)*

Why does modern C++ almost never write `new`? Because every `new` is a `delete` someone must write on every path, including the exceptional ones; forgetting one is a leak and writing two is a double free. `std::make_unique` and the containers put the `delete` in a destructor once, and a raw pointer then means "borrowed, not owned". *(7)*

Name three ways to dangle. Return a reference to a local; keep a reference or pointer into a `std::vector` across a `push_back` that reallocates; take a `std::string_view` or `c_str()` from a temporary `std::string`. A fourth is a `[&]` lambda that escapes its scope. All are undefined behaviour, and all may "work" until the memory is reused. *(6, 7, 13)*

Polymorphism and templates

How does virtual dispatch work? A class with virtual functions carries a hidden pointer to a table of function pointers, one table per class; a call through a base pointer or reference loads the vptr, indexes the table and calls indirectly — one extra load, an indirect branch, and a pointer added to the object. The call is resolved by the dynamic type, which is why a `Dog` behind an `Animal&` barks. *(10)*

Why must a base class destructor be virtual? `delete basePtr;` on a base without a virtual destructor runs only the base destructor: the derived part is never destroyed, and the behaviour is undefined. Any class meant to be deleted through a base pointer

declares `virtual ~Base() = default;` — a `std::vector<std::unique_ptr<Shape>>` needs it too. *(10)*

What is object slicing? Copying a derived object into a base *value* — `Base b = derived;`, a by-value parameter, a `std::vector<Base>` — keeps only the base part; the derived members and the dynamic type are gone, and virtual calls on the copy run the base versions. Pass and store by reference, pointer or `std::unique_ptr<Base>`. *(10)*

Overloading, overriding and hiding? Overloading is several functions with one name and different parameters, chosen at compile time by the static types of the arguments. Overriding is a derived class replacing a base virtual function with the same signature, chosen at run time by the dynamic type; `override` makes the compiler check it. Hiding is a derived class declaring a non-virtual function with the same name: chosen by the static type, so `Animal* a` calls the base version whatever the object is. *(4, 10)*

Templates versus inheritance for polymorphism? Templates give static polymorphism: resolved at compile time, inlinable, no vtable, one instantiation per type, errors at instantiation — `std::sort` with a comparator. Virtual functions give dynamic polymorphism: one compiled function, mixed types in one container through a base pointer, an indirect call per invocation — `std::vector<std::unique_ptr<Shape>>`. Concepts give the template style a stated interface and readable errors. *(12)*

The four casts? `static_cast` for conversions the language already knows (numeric, up the hierarchy, `void*` back to its type); `dynamic_cast` for a checked downcast on a polymorphic base, returning `nullptr` on a pointer and throwing `std::bad_cast` on a reference; `const_cast` to remove `const`, a smell; `reinterpret_cast` to reinterpret bits, for serialisation and little else. A C-style cast tries all of them, which is why it is avoided. *(2, 10)*

The standard library

`std::map` versus `std::unordered_map`? A balanced tree: O(log n) everything, iteration in key order, `lower_bound`, needs only `<`. A hash table: O(1) average and O(n) worst, unspecified iteration order, needs a hash and `==`. The tree when order or range queries matter, the hash for pure lookup — and in both, `operator[]` inserts a default on a miss. *(13)*

Why is `push_back` amortised O(1), and what does it invalidate? When size reaches capacity the vector allocates a larger block — roughly double — and moves the elements across, so over n pushes the moves total less than 2n. That reallocation invalidates every

iterator, pointer and reference into the old block, which is why `int& first = v[0]; v.push_back(x);` dangles and why `reserve` up front is faster and safer. *(13)*

How do you erase while iterating? `erase` returns the iterator after the removed element, so the loop is `for (auto it = c.begin(); it ≠ c.end(); ) { if (…) it = c.erase(it); else ++it; }` — never `++it` on the iterator you erased. For a vector, `std::erase_if(v, pred)` or the erase–remove idiom does it in one pass. *(13)*

`std::vector` versus `std::list`? Contiguous storage: O(1) index, cache-friendly traversal, amortised O(1) append, O(n) insert in the middle, invalidation on reallocation. A doubly linked list: O(1) insert and erase at a known position and iterators that stay valid, but every element is an allocation and a cache miss, so `vector` wins almost everywhere. `list` earns its place for `splice` — the O(1) node move an LRU cache needs. *(13, 19)*

What is a strict weak ordering, and why does `std::sort` need it? A comparator that is irreflexive (`comp(a, a)` is false), asymmetric, transitive, and whose "neither is less" relation is also transitive. `std::sort`, `std::set` and `std::map` assume it; `≤` in the comparator, or `a.x < b.x || a.y < b.y`, breaks it, and the result is undefined behaviour — a crash or an infinite loop, not merely a wrong order. `std::tie` is correct by construction. *(14)*

Errors and concurrency

Exceptions versus error codes? Exceptions separate the error path from the happy path, cannot be ignored, and unwind the stack running destructors — the tool for a failure the immediate caller cannot handle. Return values (`bool`, `std::optional`, `std::error_code`, C++23's `std::expected`) keep the failure local and visible in the signature — the tool for an expected outcome such as "key not found". Throw at the boundary, return values inside, never use exceptions for control flow. *(15)*

What does `noexcept` mean, and why does `std::vector` care? A promise that the function does not throw; if it does, `std::terminate` runs. Move constructors and destructors should be `noexcept`: `std::vector` moves its elements during reallocation only when the move cannot throw and copies otherwise, because a throwing move half-way through would leave both buffers partly moved. The three guarantees — no-throw, strong (all or nothing), basic (valid but unspecified) — are what you promise per function. *(15)*

What is a data race, and how do you prevent it? Two threads accessing the same memory location without synchronisation, at least one writing: undefined behaviour, not "sometimes the wrong number". Prevent it with a `std::mutex` held through a `std::lock_guard` or `std::scoped_lock` around every access, or `std::atomic` for a

single counter or flag. Deadlock is two threads each holding a lock the other wants; avoid it with one lock, a fixed lock order, or `std::scoped_lock` taking both at once. *(17)*

The follow-ups they actually ask

After 2: *"So is `int` being 32 bits undefined?"* — no, implementation-defined; and `1 << 31` on it wraps since C++20 but was undefined before.

After 10: *"Show me the rule-of-five class with a `std::vector` member."* — there is none: that is the rule of zero.

After 11: *"What does `a` hold after `auto b = std::move(a);`?"* — for a `std::string` on libstdc++, nothing; the standard says only valid-but-unspecified, so never depend on it.

After 17: *"Is a virtual call inside a constructor virtual?"* — it dispatches to the class being constructed, because the derived part does not exist yet.

After 20: *"What is printed?"* — `describe(const Animal&)` and `describe(const Dog&)` called with an `Animal&` to a `Dog`: the `Animal` overload (static type), while `a.speak()` is the `Dog` override (dynamic type).

After 23: *"What if every key hashes to the same bucket?"* — one chain, O(n) per operation; rehashing fixes a high load factor, not a bad hash.

After 30: *"Is `std::atomic<int> n; n++` enough for a counter?"* — yes; `n = n + 1` is not.

How to answer theory questions

Lead with the one-sentence definition, add the consequence that matters, stop. Name the mechanism, not the adjective: "an indirect call through the vtable" beats "it is slower"; "the pointer is copied and the source nulled" beats "it is moved". Cite the standard version when you know it — "`std::erase_if`, C++20". Never guess a fact you are unsure of; "I believe it is X, but I would check cppreference" is a fine answer and a confident wrong one is not. When you do not know, connect it to what you do: "I have not used `std::pmr`, but allocators are a template parameter, so I would expect it to swap the allocation strategy."

Key takeaways

Thirty questions cover almost every C++ theory round; know the definition-plus-consequence answer to each and which module explains it.

The mechanisms interviewers listen for: RAII, the rule of five and zero, `std::move` as a cast, the vtable, reallocation invalidating references, a data race as undefined behaviour.

Expect the follow-up: the overload-versus-override snippet, the moved-from string, the single-bucket hash, the constructor's virtual call.

Name the mechanism and the standard version; never bluff; connect the unknown to the known.

EXERCISE

Static versus dynamic dispatch

Write `struct Animal` with a virtual destructor, `virtual std::string speak() const` returning `...`, and a **non-virtual** `std::string kind() const` returning `animal`. Derive `Dog` (`speak` overrides to `Woof`; `kind` is redeclared to return `dog` — hiding, not overriding) and `Cat` (`Meow`, `cat`). Add three free overloads `describe(const Animal&)`, `describe(const Dog&)`, `describe(const Cat&)` returning `describe(Animal)`, `describe(Dog)` and `describe(Cat)`.

Read `n` and `n` words `dog` / `cat`. For each, create the animal in a `std::unique_ptr<Animal>` and print one line:

`<word>: speak=<a→speak()> describe=<describe(*a)> cast=<describe(...) after a dynamic_cast to the real type> kind=<a→kind()> sliced=<speak() of an Animal copy-constructed from *a>`

Predict before running: the override is chosen at run time, the overload at compile time, the non-virtual call by the static type, and the slice keeps only the base.

```
2
dog cat
```

prints

```
dog: speak=Woof describe=describe(Animal) cast=describe(Dog) kind=animal sliced=...
cat: speak=Meow describe=describe(Animal) cast=describe(Cat) kind=animal sliced=...
```

INPUT 1

```
2
dog cat
```

OUTPUT 1

```
dog: speak=Woof
describe=describe(Animal)
cast=describe(Dog) kind=animal
sliced=...
cat: speak=Meow
describe=describe(Animal)
cast=describe(Cat) kind=animal
sliced=...
```

INPUT 2

```
1
cat
```

OUTPUT 2

```
cat: speak=Meow
describe=describe(Animal)
cast=describe(Cat) kind=animal
sliced=...
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Animal {
    virtual ~Animal() = default;
    virtual string speak() const { return "..."; }
    string kind() const { return "animal"; }
};
struct Dog : Animal {
    string speak() const override { return "Woof"; }
    string kind() const { return "dog"; }            // hides Animal::kind; not virt
};
struct Cat : Animal {
    string speak() const override { return "Meow"; }
    string kind() const { return "cat"; }
};

string describe(const Animal&) { return "describe(Animal)"; }
string describe(const Dog&) { return "describe(Dog)"; }
string describe(const Cat&) { return "describe(Cat)"; }

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; ++i) {
        string word;
        cin >> word;
        unique_ptr<Animal> a;
        if (word == "dog") a = make_unique<Dog>();
        else a = make_unique<Cat>();

        string cast;
        if (auto* d = dynamic_cast<Dog*>(a.get())) cast = describe(*d);
        else cast = describe(*dynamic_cast<Cat*>(a.get()));

        Animal sliced = *a;                              // copies the Animal part o
        cout << word << ": speak=" << a->speak()
             << " describe=" << describe(*a)
             << " cast=" << cast
             << " kind=" << a->kind()
             << " sliced=" << sliced.speak() << '\n';
    }
    return 0;
}
```

QUIZ

1. The rule of five, in one sentence:

A. Every class needs five constructors

B. If you write any of destructor, copy constructor or copy assignment, write all three plus the move constructor and move assignment — or write none and let members own the resources (rule of zero)

C. Five is the number of access specifiers

D. It only applies to templates

2. `std::unique_ptr` versus `std::shared_ptr`:

A. They are interchangeable

B. Exclusive ownership, zero overhead, move-only; versus shared ownership through a reference-counted control block, copyable, with atomic count updates and the cycle hazard `weak_ptr` exists to break

C. `shared_ptr` is faster

D. `unique_ptr` cannot hold arrays

3. `const` versus `constexpr`:

A. Synonyms

B. `const` means the value will not change after initialisation (possibly at run time); `constexpr` means it is computable at compile time and may be used where a constant expression is required — array bounds, template arguments, `static_assert`

C. `constexpr` is slower

D. `const` only applies to pointers

4. `std::map` versus `std::unordered_map`:

A. Same complexity, different names

B. A balanced tree: O(log n), ordered iteration, `lower_bound`, needs `<`; versus a hash table: O(1) average, O(n) worst, unspecified order, needs a hash and `==`

C. `unordered_map` is always faster

D. `map` cannot hold strings

5. References versus pointers:

A. No difference after compilation, so no difference at all

B. A reference is an alias that must be bound at initialisation, cannot be reseated and cannot be null; a pointer is a value that can be null, reseated and used for arithmetic — use references for parameters and pointers for optional or reseatable relationships

C. Pointers are safer

D. References can be null too

6. What does `std::move` do?

A. Moves the object

B. Nothing at run time: it is a cast to an rvalue reference that lets overload resolution pick the move constructor or move assignment; the source is left valid but unspecified

C. Copies the object and deletes the original

D. Zeroes the source

7. What does this print?

```
struct Base { virtual std::string f() const { return "base"; } };
struct Derived : Base { std::string f() const override { return "derived"; } };
int main() {
    Derived d;
    Base b = d;
    Base& r = d;
    std::cout << b.f() << ' ' << r.f();
}
```

A. `derived derived`

B. `base derived` — `b` is a sliced copy holding only the `Base` part; `r` refers to the whole `Derived`

C. `base base`

D. Compile error

8. Templates versus inheritance for polymorphism:

A. Always inheritance

B. Templates give static polymorphism — resolved at compile time, inlinable, no vtable, but one instantiation per type and errors at instantiation; virtual functions give dynamic polymorphism — one compiled function, a container of mixed types through a base pointer, an indirect call per invocation

C. Templates are slower

D. Inheritance cannot express interfaces

9. Undefined behaviour is…

A. A run-time error the program reports

B. A construct the standard places no requirements on — signed overflow, an out-of-range index, a dangling reference, a data race — so the compiler may assume it never happens and optimise on that assumption

C. Behaviour that differs between compilers but is documented

D. Only possible with pointers

ANSWER KEY · THE C++ THEORY DRILL — THIRTY QUESTIONS, THIRTY ANSWERS

1. B Lead with the definition, add the consequence: a user-declared destructor or copy suppresses the implicit moves, so an owning class without them copies when it could move. Then name the alternative.

2. B The default is `unique_ptr`; `shared_ptr` is for genuinely shared lifetime, and `make_shared` puts object and control block in one allocation. Two `shared_ptr`s pointing at each other never reach zero — hence `weak_ptr`.

3. B `const int n = read();` is legal and run-time; `constexpr int n = read();` is not. A `constexpr` function may still run at run time when called with run-time arguments.

4. B Choose by the operation the problem needs: range or ordered output says tree; pure lookup says hash. Say both facts and the hazards — `[]` inserts in both, and unordered iteration order must never reach the output.

5. B Both can dangle. The choice is about what the code says: a `T&` parameter promises an object; a `T*` says "may be absent" or "may change what it points to"; `std::optional` and smart pointers cover the ownership cases.

6. B The move happens in the constructor that receives the rvalue. `std::move` on a `const` object silently selects the copy, and `return std::move(local)` defeats copy elision.

7. B Dynamic dispatch needs an object of the derived type behind a base reference or pointer. Copying into a `Base` value slices the derived part away, so the virtual call has nothing to dispatch to.

8. B `std::sort` with a comparator is the template style; a `std::vector<std::unique_ptr<Shape>>` is the virtual style. Concepts (Module 12) give templates the readable interface the virtual style always had.

9. B That last clause is the interview answer: `if (x + 1 < x)` can be deleted because signed overflow "cannot" occur. Distinguish it from implementation-defined (documented, e.g. `sizeof(int)`) and unspecified (one of several, e.g. evaluation order of function arguments).

CHECKPOINT · 30 MIN

Final checkpoint — Interview idioms

The last checkpoint of the C++ plan. It covers the contest template and its costs, the STL idiom sheet, the pitfalls review pass, `Vec<T>` and the hash map by hand, clean-solution habits and the theory drill — and, being the last one, it reaches back across the whole track.

How it works. Fifteen questions and three programs; 70% on the questions and every program accepted clears the module. With every other module complete, that also completes the plan.

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

- What `1LL * a * b` fixes that `(long long)(a * b)` does not, and why `-7 % 3` is `-1`.
- Why `i < v.size() - 1` runs on an empty vector, and what `map[key]` does inside a condition.
- The rule of five in one sentence, and why a hand-written destructor silently loses the moves.
- Why `push_back` is amortised O(1), and what a reallocation invalidates.
- What `Base b = derived;` keeps, and why deleting through a base pointer needs a virtual destructor.
- Which container gives an O(1) `splice`, and what `std::endl` does that `'\n'` does not.

The three programs are the classics interviewers set: an LRU cache from `std::list` and `std::unordered_map`, a shunting-yard evaluator with two stacks, and a polymorphic shape store behind `std::unique_ptr`.

EXERCISE 1

An LRU cache

Implement a least-recently-used cache with O(1) `get` and `put` from a `std::list<std::pair<int, int>>` in recency order (front = most recent) and a `std::unordered_map<int, std::list<...>::iterator>` from key to node. A `get` on a present key moves its node to the front with `splice`; a `put` on a present key updates the value and moves it to the front; a `put` of a new key into a full cache first evicts the node at the back.

Read the capacity `C` (`C ≥ 1`), then `n`, then `n` commands `get <key>` or `put <key> <value>`. For each `get` print `hit <value>` or `miss`. After all commands print `hits=<h> misses=<m>` and `cache=`

`<keys most recent first, space-separated>` (or `cache=empty`).

```
2
7
put 1 1
put 2 2
get 1
put 3 3
get 2
get 3
get 1
```

prints

```
hit 1
miss
hit 3
hit 1
hits=3 misses=1
cache=1 3
```

INPUT 1

```
2
7
put 1 1
put 2 2
get 1
put 3 3
get 2
get 3
get 1
```

OUTPUT 1

```
hit 1
miss
hit 3
hit 1
hits=3 misses=1
cache=1 3
```

INPUT 2

```
1
5
put 5 50
get 5
put 6 60
get 5
get 6
```

OUTPUT 2

```
hit 50
miss
hit 60
hits=2 misses=1
cache=6
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

class LruCache {
    size_t cap_;
    list<pair<int, int>> order_;                                  // front = most rec
    unordered_map<int, list<pair<int, int>>::iterator> where_;

public:
    explicit LruCache(size_t cap) : cap_(cap) {}

    optional<int> get(int key) {
        auto it = where_.find(key);
        if (it == where_.end()) return nullopt;
        order_.splice(order_.begin(), order_, it->second);     // relink in place: t
        return it->second->second;
    }

    void put(int key, int value) {
        if (auto it = where_.find(key); it != where_.end()) {
            it->second->second = value;
            order_.splice(order_.begin(), order_, it->second);
            return;
        }
        if (order_.size() == cap_) {
            where_.erase(order_.back().first);
            order_.pop_back();
        }
        order_.emplace_front(key, value);
        where_[key] = order_.begin();
    }

    const list<pair<int, int>>& order() const { return order_; }
};

int main() {
    size_t C;
    int n;
    cin >> C >> n;
    LruCache cache(C);
    int hits = 0, misses = 0;
    for (int i = 0; i < n; ++i) {
        string cmd;
        cin >> cmd;
        if (cmd == "get") {
            int key;
            cin >> key;
            if (auto v = cache.get(key)) {
                ++hits;
```

```
                cout << "hit " << *v << '\n';
            } else {
                ++misses;
                cout << "miss\n";
            }
        } else {
            int key, value;
            cin >> key >> value;
            cache.put(key, value);
        }
    }
    cout << "hits=" << hits << " misses=" << misses << '\n';
    cout << "cache=";
    if (cache.order().empty()) cout << "empty";
    bool first = true;
    for (const auto& [key, value] : cache.order()) {
        cout << (first ? "" : " ") << key;
        first = false;
    }
    cout << '\n';
    return 0;
}
```

EXERCISE 2

A shunting-yard evaluator

Read lines until the end of input; each is an arithmetic expression over non-negative integer literals with `+ - * / %`, parentheses and optional spaces (the input is always well-formed; there is no unary minus). Evaluate it in `long long` with the **shunting-yard** two-stack method — a value stack and an operator stack; `* / %` bind tighter than `+ -`, every operator is left-associative, and an operator is applied whenever the one on top of the stack has precedence at least as high as the incoming one. Division truncates toward zero. Print the value, or `error` if a division or modulo by zero occurs anywhere in the expression.

```
1 + 2 * 3
(1 + 2) * 3
10 / 3
```

prints

```
7
9
3
```

INPUT 1

```
1 + 2 * 3
(1 + 2) * 3
10 / 3
```

OUTPUT 1

```
7
9
3
```

INPUT 2

```
7 - 2 - 1
8 / (3 - 3)
```

OUTPUT 2

```
4
error
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;
using ll = long long;

int precedence(char op) { return (op == '+' || op == '-') ? 1 : 2; }

// Pops two values, applies op, pushes the result; false on division by zero.
bool applyOperator(vector<ll>& values, char op) {
    const ll b = values.back(); values.pop_back();
    const ll a = values.back(); values.pop_back();
    ll r = 0;
    switch (op) {
        case '+': r = a + b; break;
        case '-': r = a - b; break;
        case '*': r = a * b; break;
        case '/': if (b == 0) return false; r = a / b; break;
        default:  if (b == 0) return false; r = a % b; break;
    }
    values.push_back(r);
    return true;
}

optional<ll> evaluate(const string& s) {
    vector<ll> values;
    vector<char> ops;
    auto applyTop = [&]() {
        const char op = ops.back();
        ops.pop_back();
        return applyOperator(values, op);
    };
    for (size_t i = 0; i < s.size(); ++i) {
        const char c = s[i];
        if (isspace(static_cast<unsigned char>(c))) continue;
        if (isdigit(static_cast<unsigned char>(c))) {
            ll v = 0;
            while (i < s.size() && isdigit(static_cast<unsigned char>(s[i]))) v = v
            --i;                                            // the for loop steps
            values.push_back(v);
        } else if (c == '(') {
            ops.push_back(c);
        } else if (c == ')') {
            while (ops.back() != '(') if (!applyTop()) return nullopt;
            ops.pop_back();                                 // drop the '('
        } else {
            while (!ops.empty() && ops.back() != '(' && precedence(ops.back()) >= pr
                if (!applyTop()) return nullopt;
            }
            ops.push_back(c);
        }
```

```
    }
    while (!ops.empty()) if (!applyTop()) return nullopt;
    return values.back();
}

int main() {
    string line;
    while (getline(cin, line)) {
        if (auto v = evaluate(line)) cout << *v << '\n';
        else cout << "error\n";
    }
    return 0;
}
```

EXERCISE 3

Shapes behind unique_ptr

Write an abstract `Shape` with a virtual destructor and pure virtual `name()`, `area()` and `perimeter()`; derive `Circle(r)`, `Rect(w, h)` and `Square(s)` (mark the leaves `final` and every override `override`). Keep them in a `std::vector<std::unique_ptr<Shape>>` and run `n` commands:

- `circle r`, `rect w h`, `square s` — add a shape (dimensions are decimals)
- `list` — one line per shape, `<index>: <name> area=<a> perimeter=<p>`
- `total` — `total area=<sum of areas>`
- `largest` — `largest=<name> <area>` for the largest area (earliest on a tie), or `largest=none`
- `remove i` — erase the shape at index `i`, or print `no such shape`

Use `std::numbers::pi` and print every number with exactly two decimals.

```
5
circle 1
rect 2 3
square 2
list
total
```

prints

```
0: circle area=3.14 perimeter=6.28
1: rect area=6.00 perimeter=10.00
2: square area=4.00 perimeter=8.00
total area=13.14
```

INPUT 1

```
5
circle 1
rect 2 3
square 2
list
total
```

OUTPUT 1

```
0: circle area=3.14 perimeter=6.28
1: rect area=6.00 perimeter=10.00
2: square area=4.00 perimeter=8.00
total area=13.14
```

INPUT 2

```
3
largest
square 3
largest
```

OUTPUT 2

```
largest=none
largest=square 9.00
```

REFERENCE SOLUTION

```cpp
#include <bits/stdc++.h>
using namespace std;

struct Shape {
    virtual ~Shape() = default;
    virtual string name() const = 0;
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
};

class Circle final : public Shape {
    double r_;
public:
    explicit Circle(double r) : r_(r) {}
    string name() const override { return "circle"; }
    double area() const override { return numbers::pi * r_ * r_; }
    double perimeter() const override { return 2 * numbers::pi * r_; }
};

class Rect final : public Shape {
    double w_, h_;
public:
    Rect(double w, double h) : w_(w), h_(h) {}
    string name() const override { return "rect"; }
    double area() const override { return w_ * h_; }
    double perimeter() const override { return 2 * (w_ + h_); }
};

class Square final : public Shape {
    double s_;
public:
    explicit Square(double s) : s_(s) {}
    string name() const override { return "square"; }
    double area() const override { return s_ * s_; }
    double perimeter() const override { return 4 * s_; }
};

int main() {
    cout << fixed << setprecision(2);
    int n;
    cin >> n;
    vector<unique_ptr<Shape>> shapes;
    for (int i = 0; i < n; ++i) {
        string cmd;
        cin >> cmd;
        if (cmd == "circle") {
            double r;
            cin >> r;
            shapes.push_back(make_unique<Circle>(r));
```

```
        } else if (cmd == "rect") {
            double w, h;
            cin >> w >> h;
            shapes.push_back(make_unique<Rect>(w, h));
        } else if (cmd == "square") {
            double s;
            cin >> s;
            shapes.push_back(make_unique<Square>(s));
        } else if (cmd == "list") {
            for (size_t k = 0; k < shapes.size(); ++k) {
                cout << k << ": " << shapes[k]->name() << " area=" << shapes[k]->are
                     << " perimeter=" << shapes[k]->perimeter() << '\n';
            }
        } else if (cmd == "total") {
            double total = 0;
            for (const auto& s : shapes) total += s->area();
            cout << "total area=" << total << '\n';
        } else if (cmd == "largest") {
            if (shapes.empty()) {
                cout << "largest=none\n";
            } else {
                auto it = max_element(shapes.begin(), shapes.end(),
                                      [](const auto& a, const auto& b) { return a->a
                cout << "largest=" << (*it)->name() << ' ' << (*it)->area() << '\n';
            }
        } else if (cmd == "remove") {
            size_t idx;
            cin >> idx;
            if (idx < shapes.size()) shapes.erase(shapes.begin() + static_cast<ptrdi
            else cout << "no such shape\n";
        }
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. With `using namespace std;` in force, declaring a global `int count;` and then calling `count(v.begin(), v.end(), 3)` …

A. Works — the global wins

B. Is a compile error: the reference to `count` is ambiguous between your variable and `std::count`

C. Calls `std::count`

D. Is undefined behaviour

2. `int n = 100000; long long sq = n * n;` is fixed by…

A. `long long sq = (long long)(n * n);`

B. `long long sq = 1LL * n * n;` — widen an operand *before* the multiplication

C. `unsigned sq = n * n;`

D. Nothing; it already works

3. To read integers until the input ends you write…

A. `while (!cin.eof()) { cin >> x; … }`

B. `while (cin >> x) { … }`

C. `while (cin.peek() ≠ EOF) …`

D. `do { cin >> x; } while (x);`

4. The complete deduplication idiom for a `std::vector<int> v` is…

A. `std::unique(v.begin(), v.end());`

B. `std::sort(v.begin(), v.end()); v.erase(std::unique(v.begin(), v.end()), v.end());`

C. `std::sort(v.begin(), v.end()); std::unique(v.begin(), v.end());`

D. `v.erase(std::unique(v.begin(), v.end()));`

5. `std::lower_bound` versus `std::upper_bound` on a sorted range:

A. Same thing

B. First element `≥ x` versus first element `> x`; their difference is the count of `x`

C. Last element `≤ x` versus last element `< x`

D. `lower_bound` works on unsorted ranges

6. `if (m[key] > 0)` on a `std::map<std::string, int> m` …

A. Only reads

B. Inserts `key` with `0` when absent — use `find`, `count` or `contains` for a read that must not insert

C. Throws when the key is absent

D. Does not compile on a `const` map only

7. `-7 / 2` and `-7 % 3` are…

A. `-4` and `2`

B. `-3` and `-1` — division truncates toward zero and the remainder takes the dividend's sign

C. `-3` and `2`

D. `-4` and `-1`

8. `for (int i = 0; i < v.size() - 1; ++i)` on an empty vector…

A. Runs zero times

B. Runs until it reads out of range: `v.size() - 1` is `size_t` and wraps to $2^{64} - 1$, and the signed `i` converts to unsigned for the comparison

C. Is a compile error

D. Runs once

9. `std::endl` differs from `'\n'` in that it…

A. Prints a platform-specific line ending

B. Also flushes the stream — a system call per line, measured about a dozen times slower over 10^6 lines

C. Is faster

D. Works only with `std::cout`

10. A class with a raw owning pointer and only a hand-written destructor…

A. Is complete

B. Double-deletes on copy: the implicit copy constructor copies the pointer; write the copy operations (or delete them) and the moves — the rule of five — or hold the resource in a `unique_ptr` (the rule of zero)

C. Cannot be instantiated

D. Is fine as long as it is never returned

11. Doubling capacity on every reallocation gives `push_back` …

A. O(1) always

B. Amortised O(1): total copies over `n` pushes stay below `2n`, although the push that triggers a reallocation is O(n)

C. O(log n)

D. O(n) always

12. An O(1) LRU cache is built from…

A. A `std::vector` scanned on every access

B. A `std::list` in recency order plus an `unordered_map` from key to list iterator, with `splice` moving a node to the front without invalidating anything

C. A `std::map` ordered by timestamp

D. Two `std::stack`s

13. In the shunting-yard method, on reading a binary operator you first…

A. Push it unconditionally

B. Apply operators from the stack while the top is not `(` and has precedence at least as high as the new one (for left-associativity), then push

C. Apply every operator on the stack

D. Discard the value stack

14. A `std::vector<std::unique_ptr<Shape>>` requires `Shape` to have…

A. A default constructor

B. A virtual destructor — `unique_ptr<Shape>` deletes through the base pointer, and without it destroying a `Circle` that way is undefined behaviour

C. A copy constructor

D. No pure virtual functions

15. Before saying "done" in a C++ interview you should…

A. Add more features

B. Run the pitfalls pass — `int` products, unsigned `size()` arithmetic, `%` on negatives, `map[]`, invalidated iterators, `endl` — remove debug output, and state the complexity

C. Replace every loop with an algorithm

D. Switch to `#define int long long`

ANSWER KEY · FINAL CHECKPOINT — INTERVIEW IDIOMS

1. B A using-directive puts every `std` name into the lookup set of the enclosing scope; a clash with your own name at the same scope is ambiguous. This is the concrete cost of the contest idiom, and why it stays out of headers and production code.

2. B The cast after the multiplication is too late: the overflow, and the undefined behaviour, already happened in `int`. Multiplying by `1LL` first converts `n` to `long long` and the rest follows the usual arithmetic conversions.

3. B The extraction's result *is* the loop test. The `eof()` form runs one extra iteration with a stale `x` when the input ends after whitespace.

4. B `unique` removes only *adjacent* duplicates and only returns the new end; the sort brings equal values together and the erase shrinks the container.

5. B Both are O(log n) on a random-access range and both return `end()` when nothing qualifies — check before dereferencing. On a `std::set` or `std::map` use the *member* `lower_bound`, which is O(log n); the free algorithm on a set walks its iterators linearly.

6. B `operator[]` returns a reference to the mapped value, creating it if necessary; on a `const` map it does not exist at all, which is the compiler telling you the same thing.

7. B Fixed since C++11. Floor division needs `(a - ((a % b) + b) % b) / b` or `std::floor` on doubles; a non-negative remainder needs `((a % m) + m) % m`.

8. B `-Wall` warns about the signed/unsigned comparison, which is the hint. `i + 1 < v.size()` cannot wrap; so does guarding the empty case first.

9. B The line ending is the same character. Buffered output leaves once, when the program exits or the buffer fills; flush explicitly only for interactive protocols.

10. B Returning it, passing it by value or storing it in a `std::vector` copies it; two objects then own one buffer. The compiler generates the copy silently, which is why the rule exists.

11. B Growth by a constant increment gives O(n²) total; geometric growth is what makes `std::vector` the default container. `reserve` when you know the final size removes even the amortised copies.

12. B The map finds the node in O(1); the list moves and evicts in O(1); list iterators stay valid across `splice`, which a vector's would not. It is the standard "which container and why" question.

13. B The `≥` is what makes `7 - 2 - 1` evaluate left to right; `>` would make it right-associative. A `)` reduces down to its `(`, and the end of input reduces everything.

14. B `= default` is enough; it makes the delete dispatch to the most-derived destructor. The same rule applies to any polymorphic base, whether or not smart pointers are involved.

15. B Thirty seconds that catches most "correct but rejected" submissions and shows visible carefulness. Doing it aloud — "the sum is at most 10^{14}, so `long long`" — is what gets people hired.
