Module 19 of 20

Performance and undefined behaviour

The undefined-behaviour catalogue and the well-defined detectors for it, the cost model of copies, allocations and calls, cache lines, padding and traversal order, what -O2 does under the as-if rule, and how to measure without being lied to.

Two questions decide whether a C++ program is trustworthy and whether it is fast, and they turn out to be the same question asked twice. The first is what the compiler is *allowed to assume*: the standard lists operations it imposes no requirements on — signed overflow, an index past the end, an uninitialised read, a dangling pointer, a data race — and in exchange the optimiser treats every one of them as unreachable and reasons backwards from that, which is why an overflow test written as `x + 1 < x` compiles to `false` and a null check after a dereference is deleted. The second is what the code you wrote *costs*: a `std::vector` passed by value is an allocation and a copy of every element, `s = s + piece` in a loop is quadratic, a column-major inner loop loads a cache line per element, and a struct declared `char, double, char` is a third larger than it needs to be. Neither the assumption nor the cost is visible in the source; both are predictable from rules this module makes explicit.

The five lessons move from the contract to the machine to the measurement. The catalogue names every common kind of undefined behaviour, shows how the optimiser turns each into a deleted check, and settles the detectors that are themselves well-defined: `__builtin_add_overflow` and its siblings, `std::numeric_limits` comparisons, an index checked as a signed value, and the sanitizers. The cost model counts what a copy, an allocation and a call are made of and applies the count to passing, returning, `reserve`, string building and views. Memory layout explains cache lines, why contiguity beats the complexity table, the padding rule and the row-major traversal rule. Compiler optimisation covers the levels, the as-if rule, inlining, constant folding, dead-code elimination and what is not worth doing by hand. Measuring takes a time with `std::chrono::steady_clock`, lists the six ways a microbenchmark lies, and orders the tools — `time`, `perf`, Valgrind, the sanitizers — into a loop.

Every exercise models performance by *counting* — operations, copies, allocations, cache-line loads, mispredictions, comparisons — and never by timing, because a duration on a shared judge is a coin toss and a count is a fact. You will write a checked-arithmetic evaluator and a bounds-checked

array that report violations instead of committing them, an instrumented type that counts copies against moves for every way of passing it, your own growing `Vec` that reports allocations and element moves for a stated factor, a struct-padding calculator checked against the platform rule, an LRU line cache that compares the two traversal orders, a constant folder with dead-store elimination, a two-bit branch predictor over random and sorted data, a linear-against-binary search comparison counter and a benchmark-sample summariser. The checkpoint adds an overflow-safe statistics report over `long long`, an array-of-structs against struct-of-arrays cache-line audit built on the padding rule, and an operation counter that runs insertion sort and selection sort on the same input.

CONTENTS

1	The undefined behaviour catalogue — what the compiler is allowed to assume	15 min
2	The cost model — counting copies, allocations and calls	14 min
3	Memory layout and the cache — contiguity, padding and traversal order	15 min
4	Compiler optimisation — what -O2 does for you and what it cannot	14 min
5	Measuring — clocks, benchmarks that lie and the tools that tell the truth	14 min
✓	Checkpoint — Performance and undefined behaviour	25 min

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

LESSON 1 · 15 MIN

The undefined behaviour catalogue — what the compiler is allowed to assume

Undefined behaviour is the contract at the centre of C++: the standard lists operations it "imposes no requirements" on, and in exchange the compiler may assume they never happen. That assumption is what makes `-O2` fast — a loop over a signed index needs no wrap-around check because signed overflow *cannot* occur — and it is what makes a program with undefined behaviour unpredictable rather than merely wrong. This lesson catalogues the operations that are undefined, shows how the optimiser turns each into a deleted check, and settles the well-defined tools for detecting them: checked arithmetic, bounds checkers, sanitizers and brace-initialisation.

Three kinds of "the standard does not say"

Category	Meaning	Example
Implementation-defined	The implementation chooses and documents	`sizeof(int)`, whether `char` is signed
Unspecified	Any of several outcomes, not necessarily consistent	the order `f(g(), h())` evaluates its arguments
Undefined	No requirements at all — anything may happen, even before the operation	signed overflow, reading past an array

The first two are portability concerns. The third is a correctness concern: a program with undefined behaviour has no meaning, and the compiler produces whatever code follows from *assuming the undefined operation is unreachable*.

The catalogue

Signed integer overflow. `INT_MAX + 1`, `INT_MIN - 1`, `INT_MIN / -1`, `INT_MIN % -1`, `100000 * 100000` in `int`. Unsigned arithmetic wraps modulo 2^n by definition; signed arithmetic has no defined result at all (Module 2, lesson 2).

Out-of-bounds access. `a[n]` on an array of `n`, `v[v.size()]`, `*end`. The one-past-the-end pointer may be formed and compared, never dereferenced.

Uninitialised reads. `int x; if (x > 0)` reads an indeterminate value; the optimiser may treat `x` as whatever is convenient — sometimes as different values in two places.

Null and dangling dereference. `*static_cast<int*>(nullptr)`; a reference to a local after its function returned; an iterator into a `std::vector` after a `push_back` reallocated (Module 13, lesson 6); a `std::string_view` of a temporary (Module 5, lesson 6).

Use after free and double free. Touching memory after `delete`, deleting twice, `new[]` paired with `delete` (Module 7).

Data races. Two threads touching one object without synchronisation, at least one writing (Module 17, lesson 2).

One-definition-rule violations. Two translation units defining the same `inline` function or class differently — undefined, *no diagnostic required*: the linker keeps one.

Strict aliasing. Reading a `float` through an `int*` (`*reinterpret_cast<int*>(&f)`); the compiler assumes an `int*` and a `float*` never alias. Use `std::memcpy` or C++20 `std::bit_cast`.

Unsequenced modification. `i++ + i++` modifies `i` twice with no ordering — undefined. C++17 tightened the rules: function arguments are *indeterminately sequenced* (so `f(i++, i++)` is unspecified in order, no longer undefined) and an assignment's right side is sequenced before its left (so `a[i] = i++` is defined). Neither is code to write.

Shifts. `x << n` with `n` negative or at least the width of `x` is undefined in every standard. Before C++20 a left shift of a negative value, or one whose result did not fit (`1 << 31` in `int`), was undefined too; C++20 defines both modulo 2^n, so `1 << 31` is now `INT_MIN`. `1 << 32` never was defined.

Integer division by zero. `/ 0` and `% 0`. Floating-point division by zero is *defined* — infinity or NaN.

Also on the list: modifying a string literal, writing through a `const_cast` to an object declared `const`, an infinite loop with no side effects, and using an object after its destructor ran.

Why the optimiser "deletes your check"

Undefined behaviour reaches *backwards*: a check written before the undefined operation can be removed because of it.

```
int first(int* p) {
    int v = *p;                  // dereferences p, so p cannot be null …
    if (p == nullptr) return -1; // … and this test is provably false: deleted
    return v;
}
```

The same reasoning removes an overflow test written in terms of the overflow:

```
bool willOverflow(int x) {
    return x + 1 < x;            // signed overflow cannot happen, so: always
}
```

GCC and Clang both compile `willOverflow` to `return false`. The intent is fine; the test is phrased in an operation the compiler is allowed to assume never overflows. The fix is not a flag (`-fwrapv` and `-ftrapv` are diagnostic tools, not portable semantics) — it is to test *before* the operation, in terms that are themselves defined.

Detecting overflow in well-defined code

GCC and Clang provide checked-arithmetic builtins that compute the exact result when it fits and report when it does not, without performing the undefined operation:

```
int a = 2147483647;
int result = 0;
if (__builtin_add_overflow(a, 1, &result)) std::cout << "overflow\n";
else std::cout << result << '\n';
```

`__builtin_sub_overflow` and `__builtin_mul_overflow` are its siblings; each returns `true` on overflow. The portable alternative tests against `std::numeric_limits` first:

```cpp
#include <limits>

bool checkedAdd(int a, int b, int& out) {
    constexpr int hi = std::numeric_limits<int>::max();
    constexpr int lo = std::numeric_limits<int>::min();
    if ((b > 0 && a > hi - b) || (b < 0 && a < lo - b)) return false;
    out = a + b;
    return true;
}
```

Every comparison there is on values that fit. For `int` operands a third route is to compute in `long long`, which holds any sum, difference or product of two `int`s, and compare with the `int` limits; it does not scale to `long long` operands, which is where the builtins earn their place. Division has one dangerous case besides zero — `min / -1` — tested by hand.

Detecting out-of-bounds access

`at()` throws `std::out_of_range` (Module 15); `operator[]` checks nothing. A wrapper that returns `std::optional` reports a violation without exceptions and, by taking the index as a *signed* type, catches the negative index that would otherwise convert to a huge `std::size_t`:

```cpp
std::optional<int> get(const std::vector<int>& v, long long i) {
    if (i < 0 || i ⩾ static_cast<long long>(v.size())) return std::nullopt;
    return v[static_cast<std::size_t>(i)];
}
```

On libstdc++, compiling with `-D_GLIBCXX_ASSERTIONS` makes `operator[]` on every container check its bounds and abort with a message — cheap enough to leave on in debug builds.

Finding what you did not detect

`-Wall -Wextra` catch the mechanical cases: a possibly uninitialised local, `size_t` compared with `int`, a constant shift that is too wide, an always-true `⩾ 0` on an unsigned.

UBSan (`-fsanitize=undefined`) prints `runtime error: signed integer overflow: 2147483647 + 1 cannot be represented in type 'int'` with the file and line, as it

happens.

ASan (`-fsanitize=address`) reports out-of-bounds, use-after-free and double free with a stack trace (Module 7, lesson 6).

Brace-initialisation rejects narrowing at compile time, and `int x{};` is never indeterminate.

The judge compiles with `-O2` and no sanitizer: a solution with undefined behaviour may print the right answer on your machine and a different one there, which is why nothing in this track relies on it.

Pitfalls

Written	What happens at `-O2`
`if (x + 1 < x)`	Folded to `false` ; the detection never runs
`*p` then `if (p == nullptr)`	The null test is deleted
`int total;` then `total += v[i]`	Indeterminate value; any result
`for (std::size_t i = n - 1; i >= 0; --i)`	Wraps at 0; reads out of range
`1 << 32`	Undefined whatever the standard
Testing for overflow *after* computing it	Too late — the assumption already shaped the code

---

Key takeaways

Undefined behaviour is an operation with no requirements; the optimiser assumes it is unreachable and reasons backwards from that.

The catalogue: signed overflow, out-of-bounds, uninitialised reads, null and dangling dereference, use after free, data races, ODR violations, strict aliasing, unsequenced modifications, bad shifts, integer division by zero.

Test *before* the operation, in defined terms: `__builtin_add_overflow` and friends, `std::numeric_limits` comparisons, a wider type, an index checked against `size()` .

Take an index from input as a signed value so a negative one is reported, not converted.

`-Wall -Wextra`, UBSan, ASan, `_GLIBCXX_ASSERTIONS` and brace-init find what your tests did not.

EXERCISE 1

A checked-arithmetic evaluator

Evaluate `int` expressions without ever performing an operation that could overflow. Read `n`, then `n` lines of the form `a op b`, where `op` is one of `+ - * / %` and both operands fit in an `int`. Compute the result **in** `int` through the checked builtins — `__builtin_add_overflow(a, b, &result)`, `__builtin_sub_overflow` and `__builtin_mul_overflow` each store the result through their third argument and return `true` when it did not fit — and test the two dangerous division cases by hand before dividing: a zero divisor, and `std::numeric_limits<int>::min()` divided by `-1` (which also overflows for `%`).

Print `a op b = result`, or `a op b = overflow`, or `a op b = division by zero`. After the last line print `flagged k of n`, where `k` counts the lines that produced no result. The scaffold reads each operand into a `long long` and casts it to `int` — the operands fit, and `-2147483648` reads cleanly that way.

Input: `n`, then `n` lines `a op b` with the three tokens separated by spaces. **Output:** `n` result lines, then the summary.

```
4
2147483647 + 1
100000 * 100000
-2147483648 - 1
7 / 2
```

prints

```
2147483647 + 1 = overflow
100000 * 100000 = overflow
-2147483648 - 1 = overflow
7 / 2 = 3
flagged 3 of 4
```

INPUT 1

```
4
2147483647 + 1
100000 * 100000
-2147483648 - 1
7 / 2
```

OUTPUT 1

```
2147483647 + 1 = overflow
100000 * 100000 = overflow
-2147483648 - 1 = overflow
7 / 2 = 3
flagged 3 of 4
```

INPUT 2

```
3
5 * 6
-7 / 2
-7 % 3
```

OUTPUT 2

```
5 * 6 = 30
-7 / 2 = -3
-7 % 3 = -1
flagged 0 of 3
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    int flagged = 0;
    for (int k = 0; k < n; ++k) {
        long long la, lb;
        char op;
        std::cin >> la >> op >> lb;
        const int a = static_cast<int>(la);
        const int b = static_cast<int>(lb);
        std::cout << a << ' ' << op << ' ' << b << " = ";
        int result = 0;
        bool bad = false;
        std::string why = "overflow";
        switch (op) {
            case '+': bad = __builtin_add_overflow(a, b, &result); break;
            case '-': bad = __builtin_sub_overflow(a, b, &result); break;
            case '*': bad = __builtin_mul_overflow(a, b, &result); break;
            case '/':
            case '%':
                // the only two divisions that are not defined: by zero, and min / -
                if (b == 0) {
                    bad = true;
                    why = "division by zero";
                } else if (a == std::numeric_limits<int>::min() && b == -1) {
                    bad = true;
                } else {
                    result = op == '/' ? a / b : a % b;
                }
                break;
        }
        if (bad) {
            std::cout << why << '\n';
            ++flagged;
        } else {
            std::cout << result << '\n';
        }
    }
    std::cout << "flagged " << flagged << " of " << n << '\n';
    return 0;
}
```

EXERCISE 2

A bounds-checked array

Write `Checked`, a wrapper around a `std::vector<int>` that reports an out-of-range index instead of committing it. `get(long long i)` returns `std::optional<int>` — empty when `i` is not in `[0, size())` — and `set(long long i, int value)` returns `false` on the same test. The index parameter must be a **signed** `long long`: an index read from input can be negative or larger than any container, and both must be reported rather than converted to a `std::size_t` that happens to be huge. Only after the check may you cast the index to `std::size_t` and subscript.

`main` reads `n` and `n` integers into the wrapper, then `q` commands: `get i` prints `a[i] = value` or `a[i] out of range`; `set i v` prints `a[i] ← v` or `a[i] out of range`. After the commands print `violations k` (the out-of-range count) and `final:` followed by the elements, one space before each.

Input: `n`, then `n` integers, then `q`, then `q` commands. **Output:** one line per command, then the two summary lines.

```
3
10 20 30
5
get 0
get 3
set 1 99
get -1
get 1
```

prints

```
a[0] = 10
a[3] out of range
a[1] ← 99
a[-1] out of range
a[1] = 99
violations 2
final: 10 99 30
```

INPUT 1

```
3
10 20 30
5
get 0
get 3
set 1 99
get -1
get 1
```

OUTPUT 1

```
a[0] = 10
a[3] out of range
a[1] ← 99
a[-1] out of range
a[1] = 99
violations 2
final: 10 99 30
```

INPUT 2

```
1
7
4
get 1
get 0
set 0 -7
get 0
```

OUTPUT 2

```
a[1] out of range
a[0] = 7
a[0] ← -7
a[0] = -7
violations 1
final: -7
```

REFERENCE SOLUTION

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

class Checked {
public:
    explicit Checked(std::vector<int> values) : data_(std::move(values)) {}

    std::optional<int> get(long long i) const {
        if (!inRange(i)) return std::nullopt;
        return data_[static_cast<std::size_t>(i)];
    }

    bool set(long long i, int value) {
        if (!inRange(i)) return false;
        data_[static_cast<std::size_t>(i)] = value;
        return true;
    }

    const std::vector<int>& data() const { return data_; }

private:
    // signed on both sides: a negative i is rejected here, never converted
    bool inRange(long long i) const {
        return i >= 0 && i < static_cast<long long>(data_.size());
    }
    std::vector<int> data_;
};

int main() {
    int n;
    std::cin >> n;
    std::vector<int> values(static_cast<std::size_t>(n));
    for (int& x : values) std::cin >> x;
    Checked a(std::move(values));

    int q;
    std::cin >> q;
    int violations = 0;
    for (int k = 0; k < q; ++k) {
        std::string cmd;
        long long i;
        std::cin >> cmd >> i;
        if (cmd == "get") {
            if (const auto x = a.get(i)) {
                std::cout << "a[" << i << "] = " << *x << '\n';
```

```
            } else {
                std::cout << "a[" << i << "] out of range\n";
                ++violations;
            }
        } else {
            int value;
            std::cin >> value;
            if (a.set(i, value)) {
                std::cout << "a[" << i << "] ← " << value << '\n';
            } else {
                std::cout << "a[" << i << "] out of range\n";
                ++violations;
            }
        }
    }
    std::cout << "violations " << violations << '\n';
    std::cout << "final:";
    for (int x : a.data()) std::cout << ' ' << x;
    std::cout << '\n';
    return 0;
}
```

QUIZ

1. What is true of this function?

```
bool willOverflow(int x) {
    return x + 1 < x;
}
```

A. It reliably returns `true` for `INT_MAX`

B. It has undefined behaviour for `INT_MAX`, and at `-O2` GCC and Clang compile it to `return false`

C. It is well-defined because the comparison happens before the addition overflows

D. It compiles only with `-fwrapv`

2. Which of these is NOT undefined behaviour?

A. `int i = INT_MAX; i + 1`

B. `unsigned int u = 0; u - 1`

C. `1 << 32` in an `int`

D. `int x; std::cout << x;`

3. `sizeof(int)` is 4 on this platform and could be something else on another. Which category does the standard put that in?

A. Implementation-defined — the implementation chooses and documents

B. Unspecified — any of several outcomes, not necessarily consistent

C. Undefined — no requirements at all

D. Ill-formed — it should not compile

4. In C++20, what is `f(i++, i++)`?

A. Undefined behaviour, as in every standard

B. Well-defined: the arguments are evaluated left to right

C. Well-defined but unspecified: the two argument evaluations are indeterminately sequenced, so which value each receives depends on the compiler

D. A compile error

5. Which test detects overflow of `a + b` for `int` operands without itself invoking undefined behaviour?

A. `if (a + b < a)`

B. `if (static_cast<long long>(a + b) > INT_MAX)`

C. `if (b > 0 && a > std::numeric_limits<int>::max() - b)`

D. `try { a + b; } catch (const std::overflow_error&)`

6. What does this do?

```
std::vector<int> v{1, 2, 3};
for (std::size_t i = v.size() - 1; i ⩾ 0; --i) std::cout << v[i];
```

A. Prints `321`

B. Prints `321`, then `i` wraps to a huge value and `v[i]` reads out of range — undefined behaviour

C. Compile error: `i ⩾ 0` is ill-formed for an unsigned type

D. Prints `123`

7. A solution has undefined behaviour but prints the right answer on your machine. What is guaranteed when the judge runs it at `-O2` on Clang?

A. The same output — undefined behaviour is deterministic for a given input

B. Nothing — the program has no meaning, and a different compiler, level or flags may print anything

C. A runtime error, because the judge runs UBSan

D. A compile error

ANSWER KEY · THE UNDEFINED BEHAVIOUR CATALOGUE — WHAT THE COMPILER IS ALLOWED TO ASSUME

1. B `x + 1` overflows for `INT_MAX`, and signed overflow is undefined; the optimiser assumes it cannot happen, so `x + 1 < x` is provably false and the function becomes `return false`. The test must be phrased in defined terms — `x == std::numeric_limits<int>::max()` or `__builtin_add_overflow` — before the operation.

2. B Unsigned arithmetic wraps modulo 2^n by definition, so `0u - 1` is `4294967295`. Signed overflow, a shift by at least the width, and reading an indeterminate value are all on the catalogue.

3. A The implementation must choose a size and document it, which is implementation-defined. Unspecified is for things like argument evaluation order; undefined is the correctness concern, not a portability one.

4. C C++17 made function arguments indeterminately sequenced with respect to each other, so the two increments no longer overlap and the program has a meaning — just not a portable one. `i++ + i++` in one expression remains unsequenced and undefined. Neither is code to write.

5. C Every comparison in the third option is on values that fit: `max - b` cannot overflow when `b > 0`. The first two compute `a + b` in `int` first, which is the overflow they meant to detect; C++ never throws on integer overflow.

6. B `i ⩾ 0` is always true for `std::size_t`, so after printing `v[0]` the decrement wraps to `SIZE_MAX` and the next subscript is out of bounds. `-Wextra` warns that the comparison is always true; a signed index or a `while (i-- > 0)` loop fixes it.

7. B Undefined behaviour is not a value the compiler picks consistently; it is the absence of requirements, and every optimiser exploits it differently. The judge compiles with `-O2` and no sanitizer, which is exactly why nothing in the track relies on it.

LESSON 2 · 14 MIN

The cost model — counting copies, allocations and calls

Performance reasoning in C++ starts long before a stopwatch: it starts by naming the operations a line of code performs and counting them. A `std::vector<int>` passed by value is an allocation plus a copy of every element; a `std::string` longer than fifteen characters is an allocation; `s = s + piece` in a loop copies the whole prefix every time. None of those costs is visible in the source, and all of them are predictable from the value semantics the earlier modules taught. This lesson builds the model — what a copy, an allocation and a call cost relative to each other — and applies it to the decisions that recur in every program: how to pass, how to return, when to `reserve`, how to build a string, and when a view beats a copy.

What a copy is made of

Copying a value means copying its bytes and, for anything that owns memory, allocating and copying what it owns:

Value	The copy is
`int`, `double`, a small `struct` of them	a register move; effectively free
`std::string` of ≤ 15 characters (libstdc++'s small-string buffer)	a copy of the handle's bytes; no allocation
`std::string` of 100 characters	one allocation plus 100 bytes copied
`std::vector<int>` of a million	one allocation plus four megabytes copied
`std::vector<std::string>`	one allocation, then *one copy per element* — each with its own rule above

A `std::vector` handle is three pointers; copying the handle is not the cost, copying the elements is. The `std::string` boundary — fifteen characters on this platform's libstdc++, twenty-two on libc++ — is why "small strings are cheap" is true and "strings are cheap" is not.

Orders of magnitude

The relative costs below are from published measurements of ordinary hardware; the absolute numbers vary by machine, the *ratios* do not:

Operation	Roughly
Integer add, compare, correctly predicted branch	under 1 ns
Read from L1 cache	~1 ns
Read from main memory (a cache miss)	~100 ns
Small allocation and free (`new` / `delete` pair)	~20–50 ns
Mispredicted branch	~5–7 ns
A system call	~200 ns and up

An allocation costs about as much as fifty arithmetic operations, so a loop that allocates per iteration is an allocation loop whatever else it does; a cache miss costs about a hundred, which lesson 3 is about.

Passing parameters

Module 4 gave the rule; the model explains it. Passing by value copies; passing by `const&` binds a name to the caller's object for the price of one pointer:

```
long long total(const std::vector<int>& v);   // no copy, cannot modify
long long total(std::vector<int> v);          // allocation + n copies per ca
```

By value is right for small trivially copyable types (`int`, `double`, a `Point` of two doubles, `std::string_view`, `std::span`), where a copy is a register move and the indirection of a reference would cost more. It is also right for a **sink** — a parameter the function will store — because the caller can then move into it:

```
void Log::add(std::string line) { lines_.push_back(std::move(line)); }

log.add(text);              // one copy (into the parameter), one move (into t
log.add(std::move(text));   // one move, one move
log.add("literal");         // constructs in place: zero copies, one move
```

The instrumented type in this lesson's exercise counts exactly these: an lvalue argument to a by-value parameter is one copy, `std::move` of it is one move, a `const&` parameter is neither, and a temporary passed by value is *nothing* — C++17 guarantees the parameter is constructed directly from the prvalue (Module 9, lesson 5).

Returning

Return by value. A prvalue return (`return std::vector<int>{…}`) is elided by rule; a named local (`return result;`) is elided by every compiler through NRVO, and moved when it cannot be. `return std::move(result);` disables NRVO and *forces* the move — a pessimisation. Output parameters (`void fill(std::vector<int>& out)`) save nothing over returning and cost the caller a declaration.

Growth and `reserve`

A `std::vector` that outgrows its capacity allocates a bigger block, moves every element across and frees the old one. With a growth factor f the total number of element moves over n pushes is bounded by $n \cdot f / (f - 1)$ — twice n for doubling — so the amortised cost per `push_back` is constant. For a doubling vector filled from empty:

Pushes	Allocations	Elements moved	Final capacity
10	5	15	16
100	8	127	128
1 000	11	1 023	1 024

`reserve(n)` first turns that into one allocation and zero moves. It is the single most common cheap win in C++ code: whenever the final size is known or bounded, reserve. The factor and the exact capacities are implementation choices — libstdc++ doubles, MSVC grows by 1.5 —

which is why the exercise models its own `Vec` with a stated factor rather than reading `capacity()`.

`emplace_back(args…)` constructs the element in place from its arguments; `push_back(x)` copies an lvalue and moves an rvalue. For a `std::vector<std::string>`, `v.push_back(s)` is an allocation and a copy of the characters, `v.push_back(std::move(s))` is a pointer swap, `v.emplace_back("text")` constructs once.

Building strings

```
std::string out;
for (const auto& piece : pieces) out += piece;          // amortised linear
for (const auto& piece : pieces) out = out + piece;     // quadratic
```

`out += piece` appends into spare capacity, reallocating geometrically like a vector, so k pieces of m characters copy about $k{\cdot}m$ bytes in total. `out = out + piece` builds a fresh temporary containing everything so far plus the piece, then moves it into `out`: the i-th iteration copies $i{\cdot}m$ bytes, and the total is $m \cdot k(k+1)/2$. For a thousand ten-character pieces that is ten kilobytes copied against five megabytes — the same output, five hundred times the work. `std::ostringstream` and `std::format` are also linear; `reserve` on the string removes the growth copies too.

Views instead of copies

`std::string::substr` allocates a new string for anything past the small buffer; `std::string_view::substr` adjusts a pointer and a length. A parser that slices its input with views does zero allocations for the slicing itself, which is the whole case for `std::string_view` (Module 5, lesson 6) — with the standing rule that the view must not outlive the string it looks into.

The method

For any line you suspect: name the operation (copy, allocation, comparison, call), count how often it runs per element, multiply by the elements, compare with the alternative. The model is coarse — it ignores caches, branch prediction and what the optimiser removes — but it ranks

alternatives correctly far more often than intuition does, and it is the form every exercise in this module takes: a counter that reports what the code did, never a clock.

Pitfalls

Written	Cost you did not see
`void f(std::vector<int> v)` for read-only `v`	An allocation and *n* copies per call
`for (auto s : names)` over strings	One copy per element (Module 3, lesson 4)
`push_back` a million times without `reserve`	~20 reallocations and ~a million element moves
`out = out + piece` in a loop	Quadratic copying
`return std::move(local);`	Forces a move where elision was free
`std::string key = line.substr(a, b)` in a hot loop	One allocation per slice; a `string_view` is none

Key takeaways

A copy of an owning type is an allocation plus a copy of what it owns; small strings and small structs are the cheap exceptions.

An allocation costs about fifty arithmetic operations; a cache miss about a hundred. Loops are judged by what they allocate and touch.

Pass by `const&` to read, by value to sink or for small types, and move into sinks; return by value and never `return std::move(local)`.

`reserve` when the size is known; geometric growth keeps `push_back` amortised constant, but the moves are real.

`+=` builds a string in linear time; `s = s + piece` is quadratic; views slice without allocating.

EXERCISE 1

Counting copies and moves

Instrument a type so that every way of passing it reports what it cost. `Payload` owns a `std::vector<int>` and keeps two `static inline long long` counters, `copies` and `moves`; write its copy constructor and its `noexcept` move constructor so each increments its counter. Three functions take it: `takeByValue(Payload)`, `takeByRef(const Payload&)`, and a `Store` whose `add(Payload p)` is a **sink** — it `push_back`s `std::move(p)` into its vector. `makeTemp()` returns `Payload{}`.

`main` reads `n` commands and applies each to one long-lived `item` (a moved-from `item` stays valid; only the counts matter). Before the loop, `reserve(n)` the store's vector so that growth never moves an element. After each command print `<command> copies=<c> moves=<m>` with the counts that command alone added, and at the end `total copies=<C> moves=<M>`.

Command	Does
`value`	`takeByValue(item)`
`ref`	`takeByRef(item)`
`temp`	`takeByValue(Payload{})`
`sink-copy`	`store.add(item)`
`sink-move`	`store.add(std::move(item))`
`sink-temp`	`store.add(Payload{})`
`make`	`Payload made = makeTemp();`

Input: `n`, then `n` command words. **Output:** `n` lines, then the total line.

```
5
value
ref
temp
sink-copy
sink-move
```

prints

```
value copies=1 moves=0
ref copies=0 moves=0
temp copies=0 moves=0
sink-copy copies=1 moves=1
sink-move copies=0 moves=2
total copies=2 moves=3
```

INPUT 1

```
5
value
ref
temp
sink-copy
sink-move
```

OUTPUT 1

```
value copies=1 moves=0
ref copies=0 moves=0
temp copies=0 moves=0
sink-copy copies=1 moves=1
sink-move copies=0 moves=2
total copies=2 moves=3
```

INPUT 2

```
3
sink-temp
make
value
```

OUTPUT 2

```
sink-temp copies=0 moves=1
make copies=0 moves=0
value copies=1 moves=0
total copies=1 moves=1
```

REFERENCE SOLUTION

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

struct Payload {
    std::vector<int> data;
    static inline long long copies = 0;
    static inline long long moves = 0;

    Payload() : data(16, 1) {}
    Payload(const Payload& other) : data(other.data) { ++copies; }
    Payload(Payload&& other) noexcept : data(std::move(other.data)) { ++moves; }
};

void takeByValue(Payload) {}
void takeByRef(const Payload&) {}
Payload makeTemp() { return Payload{}; }

struct Store {
    std::vector<Payload> items;
    void add(Payload p) { items.push_back(std::move(p)); }
};

int main() {
    int n;
    std::cin >> n;
    Store store;
    store.items.reserve(static_cast<std::size_t>(n));   // growth must never move ar
    Payload item;
    for (int k = 0; k < n; ++k) {
        std::string cmd;
        std::cin >> cmd;
        const long long c0 = Payload::copies;
        const long long m0 = Payload::moves;
        if (cmd == "value") takeByValue(item);
        else if (cmd == "ref") takeByRef(item);
        else if (cmd == "temp") takeByValue(Payload{});
        else if (cmd == "sink-copy") store.add(item);
        else if (cmd == "sink-move") store.add(std::move(item));
        else if (cmd == "sink-temp") store.add(Payload{});
        else if (cmd == "make") { [[maybe_unused]] Payload made = makeTemp(); }
        std::cout << cmd << " copies=" << (Payload::copies - c0)
                  << " moves=" << (Payload::moves - m0) << '\n';
    }
    std::cout << "total copies=" << Payload::copies << " moves=" << Payload::moves <
```

```
    return 0;
}
```

EXERCISE 2

A Vec that reports its growth

Model geometric growth with your own container, never by reading `std::vector::capacity()`. `Vec` holds `long long` elements in a `std::unique_ptr<long long[]>` with `size`, `capacity` and an integer growth `factor` given at construction. `reserve(want)` does nothing when `want ≤ capacity`; otherwise it allocates a block of exactly `want`, copies the `size` existing elements across (count each as one **move**), counts one **allocation**, and adopts the block. `push_back(x)` calls `reserve(capacity == 0 ? 1 : capacity * factor)` when full, then stores.

Read `k` scenarios, each `factor pushes reserve`: construct a `Vec` with the factor, call `reserve(reserve)` first (a reserve of 0 does nothing), then push `1, 2, …, pushes`. Print one line per scenario: `factor=F pushes=N reserve=R: allocations A moves M capacity C sum S`, where `S` is the sum of the stored elements read back through `operator[]`.

Input: `k`, then `k` lines `factor pushes reserve` (factor ≥ 2). **Output:** `k` lines.

```
3
2 10 0
2 100 0
2 1000 0
```

prints

```
factor=2 pushes=10 reserve=0: allocations 5 moves 15 capacity 16 sum 55
factor=2 pushes=100 reserve=0: allocations 8 moves 127 capacity 128 sum 5050
factor=2 pushes=1000 reserve=0: allocations 11 moves 1023 capacity 1024 sum 500500
```

INPUT 1

```
3
2 10 0
2 100 0
2 1000 0
```

OUTPUT 1

```
factor=2 pushes=10 reserve=0:
allocations 5 moves 15 capacity 16 sum
55
factor=2 pushes=100 reserve=0:
allocations 8 moves 127 capacity 128 sum
5050
factor=2 pushes=1000 reserve=0:
allocations 11 moves 1023 capacity 1024
sum 500500
```

INPUT 2

```
2
2 1000 1000
2 1000 600
```

OUTPUT 2

```
factor=2 pushes=1000 reserve=1000:
allocations 1 moves 0 capacity 1000 sum
500500
factor=2 pushes=1000 reserve=600:
allocations 2 moves 600 capacity 1200
sum 500500
```

REFERENCE SOLUTION

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

class Vec {
public:
    explicit Vec(std::size_t factor) : factor_(factor) {}

    void reserve(std::size_t want) {
        if (want <= capacity_) return;
        auto fresh = std::make_unique<long long[]>(want);
        for (std::size_t i = 0; i < size_; ++i) fresh[i] = data_[i];
        moves_ += static_cast<long long>(size_);
        ++allocations_;
        data_ = std::move(fresh);   // the old block is freed here
        capacity_ = want;
    }

    void push_back(long long value) {
        if (size_ == capacity_) reserve(capacity_ == 0 ? 1 : capacity_ * factor_);
        data_[size_++] = value;
    }

    std::size_t size() const { return size_; }
    std::size_t capacity() const { return capacity_; }
    long long allocations() const { return allocations_; }
    long long moves() const { return moves_; }
    long long operator[](std::size_t i) const { return data_[i]; }

private:
    std::size_t factor_;
    std::size_t size_ = 0;
    std::size_t capacity_ = 0;
    long long allocations_ = 0;
    long long moves_ = 0;
    std::unique_ptr<long long[]> data_;
};

int main() {
    int k;
    std::cin >> k;
    for (int t = 0; t < k; ++t) {
        std::size_t factor, pushes, reserve;
        std::cin >> factor >> pushes >> reserve;
        Vec v(factor);
        v.reserve(reserve);
        for (std::size_t i = 1; i <= pushes; ++i) v.push_back(static_cast<long long>
        long long sum = 0;
```

```
        for (std::size_t i = 0; i < v.size(); ++i) sum += v[i];
        std::cout << "factor=" << factor << " pushes=" << pushes << " reserve=" << r
                  << ": allocations " << v.allocations() << " moves " << v.moves()
                  << " capacity " << v.capacity() << " sum " << sum << '\n';
    }
    return 0;
}
```

QUIZ

1. A function declared `long long total(std::vector<int> v)` only reads `v`. Called with a million-element vector, what does the call cost?

A. Nothing extra — the compiler passes a reference behind the scenes

B. One allocation and a million element copies

C. Three pointer copies — the vector's handle

D. One move, because the argument is an rvalue

2. `store` has spare capacity. How many copies and moves does this perform?

```
void sink(Payload p) { store.push_back(std::move(p)); }
Payload item;
sink(std::move(item));
```

A. 0 copies, 2 moves

B. 1 copy, 1 move

C. 0 copies, 1 move

D. 1 copy, 0 moves

3. Same `sink` as above. How many copies and moves does `sink(Payload{})` perform?

A. 0 copies, 2 moves

B. 0 copies, 1 move

C. 1 copy, 1 move

D. 0 copies, 0 moves

4. A `std::vector` grows by doubling. Pushing 1 000 elements into an empty one costs how many allocations and element moves?

A. 1 000 allocations, 0 moves

B. 10 allocations, 1 000 moves

C. 11 allocations, 1 023 moves

D. 1 allocation, 0 moves

5. Which of these builds a string from `k` pieces in quadratic time?

A. `out += piece` in a loop

B. `out.append(piece)` in a loop

C. `out = out + piece` in a loop

D. `std::ostringstream` with `<<` in a loop

6. `result` is a local `std::vector<int>`. What is the difference between `return result;` and `return std::move(result);`?

A. None — both move

B. `std::move` is faster because it forces the move

C. `return result;` copies; `std::move` is required to avoid the copy

D. `return result;` is elided (NRVO) or moved as a last resort; `std::move` disables elision and forces the move — a pessimisation

7. A function reads a large `std::string` and a 16-byte `Point` of two doubles, modifying neither. Which parameter types does the cost model recommend?

A. `const std::string&` and `Point` by value

B. Both by value

C. Both by `const&`

D. `std::string` by value and `const Point&`

ANSWER KEY · THE COST MODEL — COUNTING COPIES, ALLOCATIONS AND CALLS

1. B By value means a copy, and a copy of an owning type is an allocation plus a copy of everything it owns. The handle is three pointers, but the handle is not what is copied; an lvalue argument is never moved implicitly. `const std::vector<int>&` costs one pointer.

2. A `std::move(item)` move-constructs the parameter (one), and `std::move(p)` moves it into the vector (two). Without the outer `std::move` the parameter would be a copy; with spare capacity the vector moves nothing else.

3. B C++17 guarantees the parameter is constructed directly from the prvalue — no copy, no move — and then the sink moves it into the vector once. The temporary costs nothing on the way in.

4. C Capacities 1, 2, 4, …, 1 024 are eleven allocations, and every reallocation moves the elements it holds: 1 + 2 + … + 512 = 1 023. `reserve(1000)` first is the last option — one allocation and nothing moved.

5. C `out + piece` builds a fresh temporary holding everything so far, so the i-th iteration copies i pieces and the total is quadratic. `+=`, `append` and a stream append into spare capacity and grow geometrically — amortised linear.

6. D A named local returned by value is elided by every compiler through NRVO, and moved when elision is impossible. Wrapping it in `std::move` turns the operand into an xvalue that cannot be elided, so you pay the move you were trying to avoid.

7. A The string may own heap memory, so `const&` binds a name for one pointer's worth of cost. A `Point` of two doubles is a register-sized trivially copyable value; a reference would add an indirection to something cheaper than the pointer.

LESSON 3 · 15 MIN

Memory layout and the cache — contiguity, padding and traversal order

A modern processor does not read memory a byte at a time; it reads **cache lines** of 64 bytes, keeps the lines it used recently in a small fast cache, and stalls for a hundred nanoseconds whenever it needs one that is not there. That one fact decides more C++ performance questions than any algorithm: why `std::vector` beats `std::list` at everything the complexity table says the list should win, why the order of two nested loops changes a matrix sum tenfold, why a struct's members should be sorted by size, and why an array of structs is sometimes the wrong shape. This lesson gives the model — lines, locality, padding, stride — and two counting exercises that make it concrete.

Lines and locality

An ordinary x86-64 core has three cache levels — L1 around 32–48 KB at about 4 cycles, L2 up to a couple of megabytes at about 12, a shared L3 of many megabytes at about 40 — and main memory beyond at 200 cycles and more. Every level moves whole 64-byte lines. When your code reads `v[i]`, the line holding `v[i]` arrives, and with it `v[i + 1]` to `v[i + 15]` for a four-byte element: the next fifteen reads are free. That is **spatial locality**, and the hardware **prefetcher** improves on it by recognising a sequential pattern and fetching the following lines before they are asked for.

The model for this module counts *line loads*: a cache of *K* lines, least-recently-used replacement, one load each time a line not in the cache is touched. It ignores associativity and the prefetcher, and still predicts which of two traversals is cheaper.

Why `std::vector` beats `std::list`

A `std::vector<int>` keeps a million ints in one four-megabyte block: sixteen per line, sequential, prefetchable. A `std::list<int>` keeps each in a separately allocated node of at least 24 bytes (two pointers and the value, padded), scattered wherever the allocator put them, each reachable only through the previous node's pointer. Walking the list is a chain of dependent loads — node *i + 1*'s address is unknown until node *i* has arrived — so the prefetcher cannot help, and a traversal can pay a million cache misses where the vector paid sixty-five

thousand line loads, mostly prefetched. The list's O(1) insertion in the middle is real, but finding the position is a traversal, and the vector's O(n) shuffle of contiguous bytes is a `memmove` at memory bandwidth. Default to the vector; reach for `std::list` only when iterator stability is the requirement (Module 13, lesson 3).

Struct padding

Each fundamental type has an **alignment**: on x86-64 it equals its size — `char` 1, `short` 2, `int` and `float` 4, `long long`, `double` and pointers 8. A member is placed at the next offset that is a multiple of its alignment, and the struct's size is rounded up to a multiple of its largest member alignment so that arrays of it keep every element aligned. `alignof(T)` reports the alignment; `sizeof(T)` includes the padding.

```
struct Loose { char tag; int value; char flag; };   // 0, 4, 8 → size 12, alig
struct Tight { int value; char tag; char flag; };   // 0, 4, 5 → size 8,  alig
static_assert(sizeof(Loose) == 12 && sizeof(Tight) == 8);
```

`Loose` spends three bytes after `tag` to align `value` and three at the end to round to four; half the struct is padding. Reordering members by **descending size** removes the interior padding entirely — only tail padding can remain. For a million elements the difference between 12 and 8 bytes is four megabytes and a third fewer cache lines on every traversal. Clang's `-Wpadded` reports every padded struct; `static_assert(sizeof(Packet) == 8)` locks a layout you depend on. `#pragma pack(1)` removes the padding by force and makes every access misaligned — slower on x86, a fault on some architectures; a tool for matching a wire format, not for saving memory.

The rule is exact only for fundamental members on this platform's ABI; a library type has whatever size and alignment it has, which is why the padding exercise takes member sizes as input rather than printing `sizeof` of anything but a plain struct.

Row-major traversal

A two-dimensional array is stored **row-major**: `a[r][c]` lives at element index `r * C + c`, so a row is contiguous and consecutive rows follow each other. `int a[R][C]` and `std::array<std::array<int, C>, R>` are one block; a `std::vector<std::vector<int>>` is a block per row, contiguous within but not across (Module 6, lesson 6).

```
long long sumRows = 0, sumCols = 0;
for (int r = 0; r < R; ++r)
    for (int c = 0; c < C; ++c) sumRows += a[r][c];   // stride 4 bytes: seque

for (int c = 0; c < C; ++c)
    for (int r = 0; r < R; ++r) sumCols += a[r][c];   // stride 4·C bytes: a r
```

The two loops add the same numbers. The first touches each line once, sixteen elements per load. The second jumps `4·C` bytes per step; once `C ≥ 16` every access lands on a different line, and unless the cache can hold a line per row until the next column comes round, every access is a load — sixteen times the traffic for the same arithmetic. The exercise models this with an LRU cache of *K* lines: with *K* at least *R* the column-major loop costs the same as the row-major one; below that it collapses. The rule: **make the innermost loop walk the innermost index**. Flattening a grid into one `std::vector<int>` indexed by `r * C + c` gives the same layout with one allocation.

Arrays of structs and structs of arrays

```
struct Particle { double x, y, z; float mass; };   // 32 bytes with tail padd
std::vector<Particle> aos;                          // array of structs

struct Particles { std::vector<double> x, y, z; std::vector<float> mass; };
```

Summing every mass over the **AoS** layout reads four useful bytes out of every 32 — two particles per line, the doubles along for the ride. Over the **SoA** layout the masses are contiguous: sixteen per line, an eighth of the traffic. When a loop touches every field of each element, AoS is the natural shape and equally fast; when hot loops touch one or two fields of many elements — physics, graphics, analytics — SoA is the shape the cache wants, at the price of less convenient code. The checkpoint's second program counts the lines each layout touches for one hot field.

Branch prediction, in outline

A processor runs instructions many pipeline stages deep and cannot wait for a branch's condition before fetching what follows, so it *guesses* from the branch's history and discards the work when wrong — a **misprediction** of roughly fifteen to twenty cycles. A branch with a

pattern (a loop's back edge, `if (value < threshold)` over sorted data) is predicted almost perfectly; the same branch over random data is wrong about half the time. That is why sorting an array can make a loop over it several times faster with identical arithmetic, and why branchless forms (`total += (value < threshold) * value`, `std::min`, the conditional move the optimiser emits) win on unpredictable data. Lesson 4's exercise models a two-bit predictor over sorted and unsorted input. The related **false sharing** — two threads writing different variables on one line — is Module 17's.

Pitfalls

Layout decision	Cost
`std::list` for "lots of inserts"	A cache miss per node on every traversal
Members declared `char, double, char`	Size 24 instead of 16; a third more lines
Column-major inner loop over a wide matrix	A line load per element instead of per sixteen
`std::vector<std::vector<int>>` for a dense grid	Rows scattered; one allocation per row
AoS with one hot field of many	Most of every line is wasted

Key takeaways

- Memory moves in 64-byte lines; sequential access gets the next fifteen elements free and the prefetcher's help, scattered access pays a miss each.
- Contiguity is why `std::vector` beats node-based containers at traversal regardless of the complexity table.
- Members align to their size; order them by descending size to remove interior padding; `sizeof` includes the padding.
- Row-major storage means the innermost loop must walk the innermost index; stride is what a cache line notices.
- Choose SoA when hot loops touch few fields of many elements; sort or go branchless when a branch on data is unpredictable.

EXERCISE 1

A struct-padding calculator

Compute a struct's layout from the platform rule rather than by printing `sizeof`. Members are given as type names in declaration order, each with the size and alignment of this platform: `char` and `bool` 1, `short` 2, `int` and `float` 4, `long`, `double` and `ptr` 8. A member is placed at the next offset that is a multiple of its alignment; the struct's alignment is its largest member alignment; its size is the end of the last member rounded up to a multiple of that alignment; padding is size minus the sum of the member sizes.

Read `k` structs, each a line `m t1 … tm`. For each print `struct i: offsets o1 … om | size S align A padding P | reordered size S2 padding P2`, where the reordered figures come from laying out the same members sorted by **descending size** (a stable sort — equal sizes keep their order).

Input: `k`, then `k` lines, each a count followed by that many type names. **Output:** `k` lines.

```
2
3 char int char
3 int char char
```

prints

```
struct 1: offsets 0 4 8 | size 12 align 4 padding 6 | reordered size 8 padding 2
struct 2: offsets 0 4 5 | size 8 align 4 padding 2 | reordered size 8 padding 2
```

INPUT 1

```
2
3 char int char
3 int char char
```

OUTPUT 1

```
struct 1: offsets 0 4 8 | size 12 align 4 padding 6 | reordered size 8 padding 2
struct 2: offsets 0 4 5 | size 8 align 4 padding 2 | reordered size 8 padding 2
```

INPUT 2

```
1
3 char double char
```

OUTPUT 2

```
struct 1: offsets 0 8 16 | size 24 align 8 padding 14 | reordered size 16 padding 6
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cstddef>
#include <functional>
#include <iostream>
#include <map>
#include <string>
#include <vector>

struct Layout {
    std::vector<std::size_t> offsets;
    std::size_t size = 0;
    std::size_t align = 1;
    std::size_t padding = 0;
};

// Every member aligns to its own size (the platform rule for fundamental
// types), and the struct rounds up to its largest member alignment so that
// arrays of it keep every element aligned.
Layout layout(const std::vector<std::size_t>& sizes) {
    Layout l;
    std::size_t at = 0;
    std::size_t payload = 0;
    for (std::size_t s : sizes) {
        at = (at + s - 1) / s * s;
        l.offsets.push_back(at);
        at += s;
        payload += s;
        l.align = std::max(l.align, s);
    }
    l.size = (at + l.align - 1) / l.align * l.align;
    l.padding = l.size - payload;
    return l;
}

int main() {
    const std::map<std::string, std::size_t> width{
        {"char", 1}, {"bool", 1}, {"short", 2}, {"int", 4},
        {"float", 4}, {"long", 8}, {"double", 8}, {"ptr", 8},
    };
    int k;
    std::cin >> k;
    for (int t = 1; t ⩽ k; ++t) {
        int m;
        std::cin >> m;
        std::vector<std::size_t> sizes;
        for (int i = 0; i < m; ++i) {
            std::string type;
            std::cin >> type;
            sizes.push_back(width.at(type));
```

```
        }
        const Layout declared = layout(sizes);
        std::vector<std::size_t> sorted = sizes;
        std::stable_sort(sorted.begin(), sorted.end(), std::greater<>{});
        const Layout reordered = layout(sorted);

        std::cout << "struct " << t << ": offsets";
        for (std::size_t off : declared.offsets) std::cout << ' ' << off;
        std::cout << " | size " << declared.size << " align " << declared.align
                  << " padding " << declared.padding
                  << " | reordered size " << reordered.size
                  << " padding " << reordered.padding << '\n';
    }
    return 0;
}
```

EXERCISE 2

Row-major against column-major

Count cache-line loads for the two ways of summing an `R × C` matrix of 4-byte elements stored row-major. Model a cache of `K` lines of 64 bytes (sixteen elements each) with least-recently-used replacement: touching element `(r, c)` means touching line `(r * C + c) / 16`; a line already held costs nothing and becomes the most recently used, a line not held costs one **load** and evicts the least recently used when the cache is full. Write `LineCache` with a `touch(line)` and a `loads()` and run it over the row-major traversal (`r` outer, `c` inner) and the column-major one (`c` outer, `r` inner).

Read `t` scenarios `R C K` and print for each: `R=<R> C=<C> K=<K>: lines <total> row-major <a> column-major <b>`, where `total` is the number of lines the matrix occupies.

Input: `t`, then `t` lines `R C K` (K ≥ 1). **Output:** `t` lines.

```
3
64 16 8
64 16 64
64 16 63
```

prints

```
R=64 C=16 K=8: lines 64 row-major 64 column-major 1024
R=64 C=16 K=64: lines 64 row-major 64 column-major 64
R=64 C=16 K=63: lines 64 row-major 64 column-major 1024
```

INPUT 1

```
3
64 16 8
64 16 64
64 16 63
```

OUTPUT 1

```
R=64 C=16 K=8: lines 64 row-major 64
column-major 1024
R=64 C=16 K=64: lines 64 row-major 64
column-major 64
R=64 C=16 K=63: lines 64 row-major 64
column-major 1024
```

INPUT 2

```
2
4 20 1
4 20 4
```

OUTPUT 2

```
R=4 C=20 K=1: lines 5 row-major 5
column-major 80
R=4 C=20 K=4: lines 5 row-major 5
column-major 8
```

REFERENCE SOLUTION

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

// An LRU cache of a fixed number of cache lines: one load per line not held.
class LineCache {
public:
    explicit LineCache(long long lines) : lines_(lines) {}

    void touch(long long line) {
        const auto it = std::find(held_.begin(), held_.end(), line);
        if (it != held_.end()) {
            held_.erase(it);          // hit: becomes the most recently used
            held_.push_back(line);
            return;
        }
        ++loads_;
        if (static_cast<long long>(held_.size()) == lines_) held_.erase(held_.begin(
        held_.push_back(line);
    }

    long long loads() const { return loads_; }

private:
    long long lines_;
    std::vector<long long> held_;  // least recently used at the front
    long long loads_ = 0;
};

int main() {
    constexpr long long kPerLine = 64 / 4;

    int t;
    std::cin >> t;
    for (int k = 0; k < t; ++k) {
        long long r, c, lines;
        std::cin >> r >> c >> lines;
        const long long elements = r * c;
        const long long total = (elements + kPerLine - 1) / kPerLine;

        LineCache rowMajor(lines);
        for (long long i = 0; i < r; ++i)
            for (long long j = 0; j < c; ++j) rowMajor.touch((i * c + j) / kPerLine)

        LineCache columnMajor(lines);
        for (long long j = 0; j < c; ++j)
            for (long long i = 0; i < r; ++i) columnMajor.touch((i * c + j) / kPerLi

        std::cout << "R=" << r << " C=" << c << " K=" << lines << ": lines " << tota
```

```
                  << " row-major " << rowMajor.loads()
                  << " column-major " << columnMajor.loads() << '\n';
    }
    return 0;
}
```

QUIZ

1. On this platform, what is `sizeof(S)` for `struct S { char a; double b; char c; };`, and what does it become when the members are reordered to `double b; char a; char c;`?

A. 10, then 10

B. 24, then 16

C. 16, then 16

D. 24, then 24

2. Why does walking a `std::list<int>` of a million elements cost far more than walking a `std::vector<int>` of the same size?

A. `std::list` iterators are O(log n)

B. The vector is vectorised and the list is not — that is the whole difference

C. The list's nodes are scattered and each address depends on the previous node, so the prefetcher cannot help and most nodes are a cache miss

D. The list stores each `int` in 8 bytes

3. `int a[R][C]` is summed with `c` in the outer loop and `r` in the inner one. Once `C ⩾ 16` and the cache is small, what does each access cost in the line model?

A. One line load per element

B. One line load per sixteen elements

C. The same as the row-major loop — the compiler swaps the loops

D. Nothing — the prefetcher recognises the stride

4. `struct Particle { double x, y, z; float mass; };` is 32 bytes. Summing only `mass` over a `std::vector<Particle>`, what fraction of every loaded cache line is useful?

A. All of it — the vector is contiguous

B. Half

C. 4 of every 32 bytes — an eighth

D. 4 of every 64 bytes — a sixteenth

5. Which statement about alignment on this platform is correct?

A. Members are packed with no gaps unless you ask for padding

B. `sizeof` never includes padding

C. `alignof(double)` is 4 on x86-64

D. A member is placed at the next offset that is a multiple of its alignment, and the struct's size is rounded up to a multiple of its largest member alignment

6. A loop runs `if (v < threshold)` over an array. What is true of unsorted random data versus the same data sorted?

A. The same — the compiler removes the branch either way

B. Random: mispredicted about half the time; sorted: a handful of misses around the transition

C. Sorted is slower because the sort had to touch every element

D. Random is faster because sorted data defeats the predictor's history

7. What does `#pragma pack(1)` on a struct do?

A. Reorders the members by descending size

B. Removes the padding by force, making accesses misaligned — a tool for matching a wire format, not for saving memory

C. Fits the struct into one cache line

D. Is ignored by GCC and Clang

ANSWER KEY · MEMORY LAYOUT AND THE CACHE — CONTIGUITY, PADDING AND TRAVERSAL ORDER

1. B `b` needs an 8-aligned offset, so seven bytes of padding follow `a`; `c` sits at 16 and the size rounds up to 24. With the double first the two chars fit at 8 and 9 and only six bytes of tail padding remain: 16.

2. C A list traversal is a chain of dependent loads to separately allocated nodes; a vector is one contiguous block read sixteen ints per line with the prefetcher running ahead. The complexity table says both are O(n); the cache says one of them is a hundred times slower.

3. A The inner loop jumps `4·C` bytes per step, so with `C ⩾ 16` consecutive accesses land on different lines, and a cache that cannot hold a line per row has evicted each one before the next column returns to it. The rule: the innermost loop must walk the innermost index.

4. C Two particles fit in a 64-byte line and each contributes a 4-byte mass: 8 useful bytes of 64. A struct-of-arrays layout keeps the masses contiguous — sixteen per line — which is the case for SoA when hot loops touch one field of many elements.

5. D That is the padding rule the calculator implements: fundamental types align to their size, interior padding fills the gaps, and tail padding keeps arrays of the struct aligned. `sizeof` counts every padding byte.

6. B The predictor learns from history; a branch whose outcome is random is wrong about half the time, at fifteen to twenty cycles each, while a sorted array gives one long run of taken then one of not-taken. Sorting first, or a branchless form, is why identical arithmetic can run several times faster.

7. B Packing sets every member's alignment to 1, so the layout matches a byte-exact format but every multi-byte access is misaligned: slower on x86 and a fault on some architectures. Reordering members by size shrinks a struct without that cost.

LESSON 4 · 14 MIN

Compiler optimisation — what -O2 does for you and what it cannot

The code the processor runs is not the code you wrote. At `-O2` the compiler inlines your small functions, folds every expression it can evaluate at compile time, deletes computations whose results are never used, replaces loops with closed forms and turns some branches into conditional moves — all under one licence, the **as-if rule**: any transformation is allowed as long as the program's observable behaviour is unchanged. Knowing what the optimiser does tells you which hand-optimisations are wasted, which measurements are lies, and why `constexpr` is a guarantee where a plain constant is a hope. The exercises simulate two things — a constant folder with dead-store elimination, and the branch predictor whose misses the optimiser cannot always remove.

The levels

Flag	What it means
`-O0`	No optimisation; every variable lives in memory; the debugger shows your source. The default.
`-O1`	The cheap, safe passes: dead code, constant folding, register allocation.
`-O2`	Production: inlining, loop optimisations, scheduling, cheap vectorisation. **What the study judge uses.**
`-O3`	`-O2` plus aggressive inlining, unrolling and vectorisation; larger code, sometimes faster.
`-Os` / `-Og`	Optimise for size / without harming debugging.
`-Ofast`	`-O3` plus `-ffast-math`, which reorders floating-point operations and ignores NaN — never for money.

Measuring an `-O0` build tells you nothing about the program; every performance question in this module assumes `-O2`.

The as-if rule

The standard defines a program's meaning by its **observable behaviour**: the reads and writes of `volatile` objects, the data written to files and streams, and the interaction with the environment. Everything else — which locals exist, how many times an expression is evaluated, whether a function is called at all — the compiler may rearrange as long as the observable behaviour is what the source demands. Two corollaries: a computation whose result reaches no output may be deleted outright, and undefined behaviour, which has no defined observable behaviour, gives the optimiser a free hand (lesson 1).

Inlining

Replacing a call with the callee's body removes the call overhead — a handful of instructions — but the real win is what it *enables*: the arguments become known values inside the body, so constants propagate, branches on them fold, and the surrounding loop is optimised as one unit.

```
inline int square(int x) { return x * x; }
int area = square(12);                    // becomes: int area = 144;
```

The `inline` keyword is not the decision. It is a promise to the linker that the definition may appear in several translation units (Module 4, lesson 6); the optimiser inlines by its own heuristics — body size, call count, whether the definition is visible in this translation unit. Functions defined in headers and templates are always visible, which is one reason header-only code is fast.

Constant folding and propagation

`3 * 4` is `12` at compile time; so is `square(12)` once inlined. The compiler evaluates what it can prove and substitutes the result. `constexpr` upgrades that from *may* to *must* in one situation: an expression used where a constant is required — an array bound, a template argument, a `constexpr` variable's initialiser — is evaluated at compile time or the program does not compile. `consteval` forbids run-time evaluation altogether (Module 16, lesson 3).

Real compilers also apply algebraic identities — `x * 0` is `0` for integers, `x * 8` is `x << 3` — and replace division by a constant with a multiplication by a magic number, which is why `n /`

`10` is not slow. The exercise's folder is deliberately simpler: it folds only when both operands are constants, so `n * 0` stays a run-time value and the difference between propagation and simplification is visible.

Dead-code elimination

A value never read is a value never computed:

```
int wasted() {
    int sum = 0;
    for (int i = 0; i < 1000000; ++i) sum += i;   // sum is never used …
    return 0;                                     // … so the loop is gone
}
```

The same rule deletes an assignment overwritten before it is read (a **dead store**), a branch whose condition is a known constant, and a call whose only effect was its return value. It is why a microbenchmark that computes something and never prints it measures nothing (lesson 5), and why a loop summing `1` to `n` is emitted as `n * (n + 1) / 2`: the compiler recognises the closed form.

Loops and vectors

At `-O2` a loop over contiguous data may be **unrolled** (several iterations per branch) and **vectorised** (several elements per instruction, in SSE or AVX registers). The requirements are the ones lesson 3 argued for — contiguous memory, a simple stride, no data-dependent control flow — plus one the compiler cannot see: that the output does not alias the inputs. Given `void add(int* a, const int* b, int n)` it must assume `a` and `b` may overlap; `__restrict` promises they do not.

Branches and `volatile`

A small `if` / `else` that selects a value — `x = c ? a : b` — is often emitted as a **conditional move** with no branch at all, immune to misprediction. A branch that guards a loop body or a call stays a branch, and its cost is decided by the hardware predictor, not the compiler. C++20's `[[likely]]` and `[[unlikely]]` choose which path is laid out first; they do not touch

the predictor. The exercise models a two-bit predictor over sorted and unsorted data to show what a data-dependent branch costs and why sorting, or a branchless form, removes it.

`volatile` declares that every read and write of an object is observable: the compiler must perform each one, in order, and may not cache the value in a register. It exists for memory-mapped hardware and signal handlers, and it serves as a **sink** in a benchmark (lesson 5). It is not a synchronisation tool — a `volatile int` shared between threads is still a data race (Module 17, lesson 3).

LTO, PGO and reading assembly

Link-time optimisation (`-flto`) lets the optimiser see every translation unit at once, so a function defined in one `.cpp` can be inlined into another. **Profile-guided optimisation** (`-fprofile-generate`, run, `-fprofile-use`) feeds measured branch frequencies back into the compiler. Both are switches for after the code is right.

Compiler Explorer (godbolt.org) shows the assembly for any snippet on any compiler and level, and the signs are readable without writing assembly: a `call` that disappeared was inlined; a loop that became a formula was folded; a `ret` with a constant means the function ran at compile time; `cmov` means a branch became a move; `xmm` / `ymm` registers mean vectorisation.

What not to hand-optimise

Habit	Why it is wasted
`x << 1` for `x * 2`, `x >> 1` for `x / 2`	The compiler emits the shift itself — and for signed `x / 2` the *correct* one, which a bare shift is not for negatives
`i++` versus `++i` on an `int`	Identical code; the difference is real only for iterators (Module 3, lesson 3)
Caching `v.size()` in a local	Hoisted automatically unless the loop body might change it
Manual loop unrolling	The compiler unrolls when it pays
`register`, `inline` as a speed hint	Removed from the language, and not a hint, respectively

What the optimiser *cannot* do: change an O(n²) algorithm to O(n log n), remove an allocation you asked for, reorder your struct's members, or fix a column-major loop. Those remain yours — and they are where the time goes.

Key takeaways

- `-O2` inlines, folds, eliminates dead code and vectorises under the as-if rule: only observable behaviour is preserved.
- `inline` is about linkage; the optimiser decides inlining by heuristics, and inlining is what makes constant propagation possible.
- `constexpr` turns "may fold" into "must fold" where a constant is required; `consteval` forbids run-time evaluation.
- Dead-code elimination deletes anything whose result is unobserved — including the benchmark loop that never prints.
- Micro-tricks the compiler already does are wasted; algorithms, allocations, layout and loop order are yours.

EXERCISE 1

A constant folder with dead-store elimination

Simulate two optimiser passes over a straight-line program of integer statements. Each line is one of `input x` (`x` receives a run-time value: unknown from here on), `x = operand`, `x = operand op operand` with `op` in `+ - *`, or `print operand`; an operand is a name or an integer literal (possibly negative). The scaffold parses each line into a `Stmt` and can render one back; you write the two passes.

Fold walks forward keeping a map of names with known constant values: replace every operand whose value is known by its literal, and when *all* operands of an assignment are then literals compute the result (`long long`), rewrite the statement as `x = value` and record `x` as known. An assignment with an unknown operand, and an `input`, make the target unknown. Fold only what is fully constant — `n * 0` stays a run-time expression. A `print` of a known name becomes `print value`.

Eliminate walks backward with a set of live names: a `print` of a name makes it live; an assignment whose target is not live is a dead store and is dropped (its operands do not become live); a kept assignment makes its target dead and its name operands live; `input` is always kept and makes its name dead.

Print the surviving statements in order, then `eliminated E of N`.

Input: `N`, then `N` statement lines. **Output:** the optimised program, then the summary line.

```
6
x = 3
y = x + 4
input n
z = y * n
w = z - z
print z
```

prints

```
input n
z = 7 * n
print z
eliminated 3 of 6
```

INPUT 1

```
6
x = 3
y = x + 4
input n
z = y * n
w = z - z
print z
```

OUTPUT 1

```
input n
z = 7 * n
print z
eliminated 3 of 6
```

INPUT 2

```
4
a = 10
b = a * a
print b
print 5
```

OUTPUT 2

```
print 100
print 5
eliminated 2 of 4
```

REFERENCE SOLUTION

```cpp
#include <cctype>
#include <cstddef>
#include <iostream>
#include <map>
#include <optional>
#include <set>
#include <sstream>
#include <string>
#include <vector>

struct Stmt {
    std::string kind;              // "input", "print" or "assign"
    std::string target;            // input/assign: the variable; print: the operand
    std::vector<std::string> ops;  // assign: one operand, or two joined by op
    char op = 0;                   // '+', '-' or '*' when ops has two entries
};

bool isLiteral(const std::string& token) {
    return std::isdigit(static_cast<unsigned char>(token[0])) || token[0] == '-';
}

Stmt parse(const std::string& line) {
    std::istringstream in(line);
    std::vector<std::string> t;
    std::string tok;
    while (in >> tok) t.push_back(tok);
    Stmt s;
    if (t[0] == "input") {
        s.kind = "input";
        s.target = t[1];
    } else if (t[0] == "print") {
        s.kind = "print";
        s.target = t[1];
    } else {
        s.kind = "assign";
        s.target = t[0];
        s.ops.push_back(t[2]);
        if (t.size() == 5) {
            s.op = t[3][0];
            s.ops.push_back(t[4]);
        }
    }
    return s;
}

std::string render(const Stmt& s) {
    if (s.kind == "input") return "input " + s.target;
    if (s.kind == "print") return "print " + s.target;
    std::string out = s.target + " = " + s.ops[0];
```

```
    if (s.ops.size() == 2) out += std::string(" ") + s.op + " " + s.ops[1];
    return out;
}

// Pass 1, forward: substitute every operand whose value is known and fold an
// operation whose operands are all constants; an input line makes a name unknown.
void fold(std::vector<Stmt>& program) {
    std::map<std::string, long long> known;
    const auto value = [&](const std::string& token) -> std::optional<long long> {
        if (isLiteral(token)) return std::stoll(token);
        const auto it = known.find(token);
        if (it == known.end()) return std::nullopt;
        return it->second;
    };
    for (Stmt& s : program) {
        if (s.kind == "input") {
            known.erase(s.target);
            continue;
        }
        if (s.kind == "print") {
            if (const auto v = value(s.target)) s.target = std::to_string(*v);
            continue;
        }
        std::vector<std::optional<long long>> vals;
        for (std::string& o : s.ops) {
            const auto v = value(o);
            vals.push_back(v);
            if (v) o = std::to_string(*v);
        }
        if (s.ops.size() == 1) {
            if (vals[0]) known[s.target] = *vals[0];
            else known.erase(s.target);
        } else if (vals[0] && vals[1]) {
            const long long a = *vals[0];
            const long long b = *vals[1];
            const long long r = s.op == '+' ? a + b : s.op == '-' ? a - b : a * b;
            s.ops = {std::to_string(r)};
            s.op = 0;
            known[s.target] = r;
        } else {
            known.erase(s.target);   // n * 0 stays a run-time value: no algebra her
        }
    }
}

// Pass 2, backward: an assignment nobody reads before it is overwritten is dead.
std::vector<Stmt> eliminate(const std::vector<Stmt>& program) {
    std::set<std::string> live;
    std::vector<bool> keep(program.size(), true);
```

```
    for (std::size_t i = program.size(); i-- > 0;) {
        const Stmt& s = program[i];
        if (s.kind == "print") {
            if (!isLiteral(s.target)) live.insert(s.target);
            continue;
        }
        if (s.kind == "input") {
            live.erase(s.target);
            continue;
        }
        if (!live.count(s.target)) {
            keep[i] = false;
            continue;
        }
        live.erase(s.target);
        for (const std::string& o : s.ops)
            if (!isLiteral(o)) live.insert(o);
    }
    std::vector<Stmt> out;
    for (std::size_t i = 0; i < program.size(); ++i)
        if (keep[i]) out.push_back(program[i]);
    return out;
}

int main() {
    int k;
    std::cin >> k;
    std::cin.ignore();
    std::vector<Stmt> program;
    for (int i = 0; i < k; ++i) {
        std::string line;
        std::getline(std::cin, line);
        program.push_back(parse(line));
    }
    fold(program);
    const std::vector<Stmt> kept = eliminate(program);
    for (const Stmt& s : kept) std::cout << render(s) << '\n';
    std::cout << "eliminated " << (program.size() - kept.size()) << " of " << progra
    return 0;
}
```

EXERCISE 2

A two-bit branch predictor

Model the cost of a data-dependent branch. For each scenario `n threshold seed` generate `n` values with the linear congruential generator `x = (x * 1103515245 + 12345) mod 2^31` in unsigned

64-bit arithmetic, starting from `x = seed`, taking `value = (x >> 16) % 256` after each step. The branch is `value < threshold`, and the predictor is a **two-bit saturating counter**: state 0 or 1 predicts "not taken", 2 or 3 predicts "taken"; a taken branch moves the state up by one (saturating at 3), a not-taken one down by one (saturating at 0); the state starts at 0. Count a **miss** whenever the prediction disagrees with the outcome.

Run the predictor over the values in generated order, then over the same values sorted ascending, and print one line per scenario: `n=<n> threshold=<t> seed=<s>: taken <count> random misses <a> sorted misses <b>`, where `taken` is how many values are below the threshold.

Input: `t`, then `t` lines `n threshold seed`. **Output:** `t` lines.

```
3
1000 128 42
1000 128 7
1000 200 42
```

prints

```
n=1000 threshold=128 seed=42: taken 527 random misses 479 sorted misses 4
n=1000 threshold=128 seed=7: taken 506 random misses 483 sorted misses 4
n=1000 threshold=200 seed=42: taken 815 random misses 219 sorted misses 4
```

INPUT 1

```
3
1000 128 42
1000 128 7
1000 200 42
```

OUTPUT 1

```
n=1000 threshold=128 seed=42: taken 527
random misses 479 sorted misses 4
n=1000 threshold=128 seed=7: taken 506
random misses 483 sorted misses 4
n=1000 threshold=200 seed=42: taken 815
random misses 219 sorted misses 4
```

INPUT 2

```
3
0 128 1
50 0 3
50 256 3
```

OUTPUT 2

```
n=0 threshold=128 seed=1: taken 0 random
misses 0 sorted misses 0
n=50 threshold=0 seed=3: taken 0 random
misses 0 sorted misses 0
n=50 threshold=256 seed=3: taken 50
random misses 2 sorted misses 2
```

REFERENCE SOLUTION

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

// A two-bit saturating predictor: states 0 and 1 predict "not taken",
// 2 and 3 predict "taken"; every outcome nudges the state one step.
long long mispredictions(const std::vector<int>& values, int threshold) {
    int state = 0;
    long long wrong = 0;
    for (int v : values) {
        const bool taken = v < threshold;
        const bool predicted = state >= 2;
        if (taken != predicted) ++wrong;
        state = taken ? std::min(3, state + 1) : std::max(0, state - 1);
    }
    return wrong;
}

int main() {
    int t;
    std::cin >> t;
    for (int k = 0; k < t; ++k) {
        int n, threshold;
        std::uint64_t seed;
        std::cin >> n >> threshold >> seed;

        std::vector<int> values;
        std::uint64_t x = seed;
        for (int i = 0; i < n; ++i) {
            x = (x * 1103515245ULL + 12345ULL) % 2147483648ULL;
            values.push_back(static_cast<int>((x >> 16) % 256));
        }
        const long long taken = std::count_if(values.begin(), values.end(),
                                              [&](int v) { return v < threshold; });
        const long long random = mispredictions(values, threshold);
        std::vector<int> sorted = values;
        std::sort(sorted.begin(), sorted.end());
        const long long ordered = mispredictions(sorted, threshold);

        std::cout << "n=" << n << " threshold=" << threshold << " seed=" << seed
                  << ": taken " << taken << " random misses " << random
                  << " sorted misses " << ordered << '\n';
    }
    return 0;
}
```

QUIZ

1. At `-O2`, what remains of the loop?

```
int wasted() {
    int sum = 0;
    for (int i = 0; i < 1000000; ++i) sum += i;
    return 0;
}
```

A. The loop, because loops always execute

B. Nothing — `sum` reaches no output, so the loop is dead code and is deleted

C. The loop, vectorised eight elements at a time

D. A closed form `n * (n + 1) / 2` stored into `sum`

2. What does the `inline` keyword actually do?

A. Forces the compiler to inline the function at every call

B. Is required before the optimiser may inline a function

C. Is a linkage promise — the definition may appear in several translation units — while the optimiser inlines by its own heuristics

D. Makes the function `constexpr`

3. Which level does the study judge compile with, and which does `g++` use when you pass no `-O` flag?

A. `-O2`; `-O0`

B. `-O3`; `-O2`

C. `-O2`; `-O1`

D. `-Ofast`; `-O0`

4. What does the as-if rule permit the compiler to do?

A. Only transformations that keep every variable and every function call

B. Any transformation that preserves observable behaviour — `volatile` accesses, data written to files and streams, interaction with the environment

C. Reorder floating-point operations freely

D. Remove any loop it considers too slow

5. `constexpr int twice(int x) { return x * 2; }` and later `int y = twice(n);` where `n` was read from `std::cin`. What happens?

A. Compile error — a `constexpr` function needs constant arguments

B. Undefined behaviour

C. It compiles and runs at run time: `constexpr` permits compile-time evaluation and requires it only where a constant is required

D. `y` is computed at compile time as 0

6. Two threads share a `volatile int counter` and both increment it without a mutex or atomic. What is true?

A. It is safe — `volatile` makes every access happen

B. It is still a data race, which is undefined behaviour; `volatile` is not a synchronisation tool

C. It is atomic on x86

D. It is a compile error

7. Which of these is worth doing by hand at `-O2`?

A. Replacing `x * 2` with `x << 1`

B. Caching `v.size()` in a local before a read-only loop

C. Fixing a column-major inner loop, or removing an allocation from a loop

D. Writing `++i` instead of `i++` on an `int` loop counter

ANSWER KEY · COMPILER OPTIMISATION — WHAT -O2 DOES FOR YOU AND WHAT IT CANNOT

1. B Under the as-if rule a value never observed need never be computed; the whole loop goes. The closed form appears only when `sum` is returned or printed — and that is also why a benchmark that never prints its result measures nothing.

2. C `inline` relaxes the one-definition rule so a header can carry a definition. Inlining is decided from body size, call count and visibility; a function without the keyword is inlined just as readily when its definition is visible.

3. A The judge builds at `-O2`, the production level. The compiler's default is `-O0` — no optimisation, every variable in memory — which is why timing an unflagged build tells you nothing about the shipped program.

4. B The program's meaning is its observable behaviour; which locals exist, how often an expression is evaluated, whether a function is called at all are all fair game. Floating-point reordering needs `-ffast-math`, which changes results and is not as-if.

5. C `constexpr` marks a function *usable* in constant expressions. With a run-time argument it is an ordinary call; with a constant argument in a context that demands a constant — an array bound, a template argument — it must fold or the program is ill-formed. `consteval` is the keyword that forbids run-time calls.

6. B `volatile` forbids caching the value in a register and reordering its accesses — the contract for memory-mapped hardware and signal handlers — but says nothing about other threads. `std::atomic<int>` or a mutex is the fix.

7. C The optimiser already emits the shift, hoists an unchanging `size()` and generates identical code for either increment on an `int`. What it cannot do is change your algorithm, remove an allocation you asked for, reorder your struct or swap your loops — those are the changes that move the time.

LESSON 5 · 14 MIN

Measuring — clocks, benchmarks that lie and the tools that tell the truth

Everything in this module so far has been a model: count the copies, count the lines, count the branch misses. Models rank alternatives; only a measurement says how much time an alternative costs, and measurements of small pieces of code are wrong more often than right — the optimiser deleted the work, the input was a compile-time constant, the first run paid for page faults the others did not, the laptop changed clock speed halfway through. This lesson settles how to take a time that means something with `std::chrono::steady_clock`, why the judge never asserts on one, the pitfalls that make a microbenchmark lie, which statistic to report, and the tools that find where the time goes: `perf`, Valgrind, the sanitizers and `-fanalyzer`.

Taking a time

```
#include <chrono>
#include <iostream>

int main() {
    const auto start = std::chrono::steady_clock::now();
    long long total = 0;
    for (int i = 0; i < 10'000'000; ++i) total += i % 7;
    const auto end = std::chrono::steady_clock::now();
    const auto us = std::chrono::duration_cast<std::chrono::microseconds>(end
    std::cout << "total=" << total << '\n';                    // the sink: t
    std::cerr << "elapsed " << us.count() << " us\n";          // the timing.
    return 0;
}
```

`steady_clock` is monotonic — it never goes backwards or jumps when the system time is adjusted — which is the property a duration needs. `system_clock` is wall-clock time for timestamps, not for intervals; `high_resolution_clock` is an alias for one of the other two and is best avoided by name. Subtracting two `time_point`s gives a `duration`; `duration_cast` converts to the unit you want and `.count()` gives the number (Module 16, lesson 4).

Two things about that snippet matter for the judge. The result goes to `stdout` and the time to `stderr`: only `stdout` is compared, and no expected output in this track contains a

duration. The judge runs on a shared machine with a 5 s CPU ceiling, and a program that took 40 ms on one run may take 90 ms on the next with no change; a case that asserted "under 50 ms" would be a coin toss. Time limits are a ceiling against runaway loops, never a measurement.

Six ways a microbenchmark lies

The work was deleted. The result of the loop is never used, so dead-code elimination removes the loop (lesson 4) and the benchmark reports the cost of two clock reads. *Fix:* make the result observable — accumulate into a value that is printed, or write it to a `volatile` sink, or use a library's `DoNotOptimize`.

The input was a constant. The compiler saw `fib(30)` with a literal argument and computed it at compile time. *Fix:* read the input from `stdin` or a `volatile`, so its value is unknown until run time.

No warm-up. The first iterations pay for page faults on freshly allocated memory, cold caches and the CPU raising its frequency; a single run measures those, not the code. *Fix:* run several times and discard the first.

The timer's resolution. A microsecond clock cannot time a ten-nanosecond operation. *Fix:* time a loop of millions of iterations and divide.

The machine was busy. Another process, a background update, thermal throttling. *Fix:* repeat, and report the **minimum** or the **median** rather than one run or the mean.

Measuring the wrong build. `-O0` measures a program the compiler never optimised; `-O2` with the sanitizers on measures the instrumentation. *Fix:* benchmark the flags you ship.

A benchmark that survives all six is a library's job — Google Benchmark handles warm-up, repetition, statistics and the sink — but a hand-rolled harness with a sink, a warm-up and a median is enough to rank two implementations.

Which number to report

The runs of a benchmark are not normally distributed: there is a floor set by the code and a long tail of interruptions above it. The **minimum** is the best estimate of what the code itself costs; the **median** is what a typical run costs; the **mean** is dragged upward by every interruption and is the least informative of the three. Report the number of samples and the spread — `(max - min) / min` — so a reader can see whether the measurement is stable at

all, and never compare two implementations measured on different machines. The exercise's summariser applies exactly these rules to samples that arrive as input.

Complexity against constant factors

Big-O ranks algorithms as n grows; at the sizes real loops see, the constant factors decide. Binary search over eight sorted elements is slower than a linear scan: the scan touches one cache line and its branch is perfectly predicted, while the search's three probes each mispredict about half the time. `std::sort` knows this — libstdc++'s introsort switches to insertion sort below sixteen elements. A `std::map` with fifty entries loses to a sorted `std::vector` with `std::lower_bound`; a `std::unordered_map` with ten entries loses to a linear scan of a vector of pairs. The operation counter in this lesson's first exercise makes the trade visible: linear search costs n comparisons in the worst case and binary search about $\log_2 n$, and the ratio between them at $n = 8$ is not the ratio at $n = 8\,000\,000$.

The tools

Tool	What it tells you
`time ./a.out`	Real, user and system seconds for the whole run — the first measurement, before anything else
`perf stat -e cycles,instructions,cache-misses,branch-misses ./a.out`	Hardware counters: instructions per cycle, misses per instruction
`perf record ./a.out` then `perf report`	Which functions the time went to, sampled
`valgrind --tool=callgrind`	Exact instruction counts per function, slowly; `kcachegrind` draws the call graph
`valgrind --tool=cachegrind`	A simulated cache: misses per line of source
`-fsanitize=address,undefined`	Memory errors and undefined behaviour at run time, ~2× slower
`-fsanitize=thread`	Data races, ~5–15× slower (Module 17)
`-fanalyzer` (GCC)	Static analysis at compile time: leaks, null dereferences, double frees along paths
`clang-tidy`	Style and correctness checks, including performance ones (`performance-*`)

The order matters. `time` says whether there is a problem. A profiler says *where* — and it is almost never where intuition pointed, which is the whole reason to profile before optimising. A sanitizer says whether the fast version is still correct. Amdahl's rule bounds the reward: a function that takes 10 % of the run cannot be optimised into more than a 10 % improvement.

The loop

Measure the whole program. Profile to find the hottest function. Form a hypothesis from the models in this module — an allocation in a loop, a copy where a reference would do, a column-major traversal, a branch on unsorted data, an O(n²) where a sort would give O(n log n). Change one thing. Measure again with the same harness, and keep the change only if the median moved by more than the spread. Run the sanitizers on the result. Then profile again, because the hottest function has moved.

Pitfalls

Written	What was measured
A loop whose result is never printed	Nothing — the loop was eliminated
`benchmark(fib(30))` with a literal	A compile-time constant
One run, reported as the mean	The page faults and the frequency ramp
`system_clock` for a duration	A clock that can jump backwards
Timing at `-O0`	A program nobody ships
An expected output with a time in it	A coin toss on the judge

Key takeaways

- `std::chrono::steady_clock::now()` twice, subtract, `duration_cast`: the only correct way to take an interval, and it goes to `stderr`, never into a judged output.
- A microbenchmark must sink its result, take an unknown input, warm up, repeat, and measure the flags you ship.
- Report the minimum or median with the sample count and spread; the mean hides the tail.
- Constant factors beat complexity at small *n*: a scan over one cache line beats a binary search that mispredicts.
- `time`, then `perf`, then change one thing, then the sanitizers — and the hot spot is never where you guessed.

EXERCISE 1

Linear against binary search, in comparisons

Make the constant-factor trade visible by counting. Read `n` sorted distinct integers and `q` queries. For each query run both searches and count the work: **linear** search compares `a[i] == x` for `i = 0, 1, …` and stops at the first match (an absent value costs `n` comparisons); **binary** search counts one comparison per probe, exactly this loop:

```
int lo = 0, hi = n - 1;
while (lo ⩽ hi) {
    int mid = lo + (hi - lo) / 2;   // one probe
    if (a[mid] == x) return found;
    if (a[mid] < x) lo = mid + 1; else hi = mid - 1;
}
```

Print `x: linear <c1> binary <c2> found|absent` per query, then `total linear <L> binary <B> ratio <L/B>` with the ratio fixed to two decimals — or `ratio n/a` when `B` is zero.

Input: `n`, then `n` sorted integers, then `q`, then `q` integers. **Output:** `q` lines, then the total line.

```
8
1 2 3 4 5 6 7 8
3
8
1
9
```

prints

```
8: linear 8 binary 4 found
1: linear 1 binary 3 found
9: linear 8 binary 4 absent
total linear 17 binary 11 ratio 1.55
```

INPUT 1

```
8
1 2 3 4 5 6 7 8
3
8
1
9
```

OUTPUT 1

```
8: linear 8 binary 4 found
1: linear 1 binary 3 found
9: linear 8 binary 4 absent
total linear 17 binary 11 ratio 1.55
```

INPUT 2

```
5
-10 -3 0 7 42
4
0
-10
42
5
```

OUTPUT 2

```
0: linear 3 binary 1 found
-10: linear 1 binary 2 found
42: linear 5 binary 3 found
5: linear 5 binary 2 absent
total linear 14 binary 8 ratio 1.75
```

REFERENCE SOLUTION

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

struct Probe {
    bool found;
    long long comparisons;
};

Probe linearSearch(const std::vector<int>& a, int x) {
    long long comparisons = 0;
    for (int v : a) {
        ++comparisons;
        if (v == x) return {true, comparisons};
    }
    return {false, comparisons};
}

Probe binarySearch(const std::vector<int>& a, int x) {
    long long probes = 0;
    int lo = 0;
    int hi = static_cast<int>(a.size()) - 1;
    while (lo <= hi) {
        const int mid = lo + (hi - lo) / 2;   // never lo + hi: that sum can overflo
        ++probes;
        if (a[mid] == x) return {true, probes};
        if (a[mid] < x) lo = mid + 1;
        else hi = mid - 1;
    }
    return {false, probes};
}

int main() {
    int n;
    std::cin >> n;
    std::vector<int> a(static_cast<std::size_t>(n));
    for (int& v : a) std::cin >> v;
    int q;
    std::cin >> q;
    long long linearTotal = 0;
    long long binaryTotal = 0;
    for (int k = 0; k < q; ++k) {
        int x;
        std::cin >> x;
        const Probe lin = linearSearch(a, x);
        const Probe bin = binarySearch(a, x);
        linearTotal += lin.comparisons;
        binaryTotal += bin.comparisons;
        std::cout << x << ": linear " << lin.comparisons << " binary " << bin.compar
```

```
                  << (bin.found ? " found" : " absent") << '\n';
    }
    std::cout << "total linear " << linearTotal << " binary " << binaryTotal << " ra
    if (binaryTotal == 0) std::cout << "n/a\n";
    else std::cout << std::fixed << std::setprecision(2)
                   << static_cast<double>(linearTotal) / static_cast<double>(binaryT
    return 0;
}
```

EXERCISE 2

Summarising benchmark runs

Apply the lesson's reporting rules to timings that arrive as input, so the judge compares statistics and never a clock. Two implementations, `A` and `B`, were each run several times; for each, read `n w` and then `n` integer durations in microseconds, the first `w` of which are warm-up runs to discard (`w < n`, every duration at least 1). For the remaining samples report the count, the **minimum**, the **median** (the mean of the two middle values for an even count, printed with one decimal), the **mean** (two decimals) and the **spread** `(max - min) / min` (two decimals). Then print `speedup` as `median(A) / median(B)` to two decimals — above 1 means `B` is faster.

Input: `n w`, `n` integers, then the same again for `B`. **Output:** two summary lines and the speedup line.

```
6 1
90 41 40 44 43 42
6 1
70 33 31 30 31 35
```

prints

```
A: samples 5 min 40 median 42.0 mean 42.00 spread 0.10
B: samples 5 min 30 median 31.0 mean 32.00 spread 0.17
speedup 1.35
```

INPUT 1

```
6 1
90 41 40 44 43 42
6 1
70 33 31 30 31 35
```

OUTPUT 1

```
A: samples 5 min 40 median 42.0 mean
42.00 spread 0.10
B: samples 5 min 30 median 31.0 mean
32.00 spread 0.17
speedup 1.35
```

INPUT 2

```
4 0
12 10 11 13
5 2
99 98 20 24 22
```

OUTPUT 2

```
A: samples 4 min 10 median 11.5 mean
11.50 spread 0.30
B: samples 3 min 20 median 22.0 mean
22.00 spread 0.20
speedup 0.52
```

REFERENCE SOLUTION

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

struct Summary {
    std::size_t samples;
    long long min;
    double median;
    double mean;
    double spread;
};

// Drop the warm-up runs, sort, and take the statistics the lesson recommends:
// the minimum is what the code costs, the median what a typical run costs,
// the mean is dragged up by every interruption and is reported last.
Summary summarise(std::vector<long long> runs, std::size_t warmup) {
    runs.erase(runs.begin(), runs.begin() + static_cast<std::ptrdiff_t>(warmup));
    std::sort(runs.begin(), runs.end());
    const std::size_t n = runs.size();
    long long total = 0;
    for (long long r : runs) total += r;
    const double median = n % 2 == 1
        ? static_cast<double>(runs[n / 2])
        : (static_cast<double>(runs[n / 2 - 1]) + static_cast<double>(runs[n / 2]))
    const double spread = static_cast<double>(runs.back() - runs.front()) / static_c
    return {n, runs.front(), median, static_cast<double>(total) / static_cast<double
}

Summary readSet() {
    std::size_t n, warmup;
    std::cin >> n >> warmup;
    std::vector<long long> runs(n);
    for (long long& r : runs) std::cin >> r;
    return summarise(runs, warmup);
}

void print(const char* name, const Summary& s) {
    std::cout << name << ": samples " << s.samples << " min " << s.min
              << " median " << std::fixed << std::setprecision(1) << s.median
              << " mean " << std::setprecision(2) << s.mean
              << " spread " << s.spread << '\n';
}

int main() {
    const Summary a = readSet();
    const Summary b = readSet();
    print("A", a);
```

```
    print("B", b);
    std::cout << "speedup " << std::fixed << std::setprecision(2) << a.median / b.me
    return 0;
}
```

QUIZ

1. Which clock should measure how long a piece of code took?

A. `std::chrono::system_clock` — it has the highest resolution

B. `std::chrono::steady_clock` — monotonic, never adjusted, so a difference of two readings is a duration

C. `std::time(nullptr)`

D. `std::chrono::high_resolution_clock`, always

2. A benchmark loop computes a sum a million times and reports 0 ns. The most likely reason?

A. The timer's resolution is too coarse for a million iterations

B. The CPU was throttled

C. Dead-code elimination removed the loop because its result is never observed — print it or write it to a `volatile` sink

D. The loop was vectorised down to a single instruction

3. You ran a benchmark twenty times. Which number should you report, and why not the mean?

A. The minimum or the median — the mean is dragged upward by every interruption in the long tail

B. The mean — it uses all the data

C. The maximum — the worst case is what matters

D. The first run — later runs are warmed up and unrepresentative

4. Why does no expected output in this track contain a duration?

A. `std::chrono` is not available on the judge

B. A time on the shared judge varies from run to run with no change to the code; asserting on it would be a coin toss

C. The standard requires durations to go to `stderr`

D. The judge strips numbers from output before comparing

5. Searching eight sorted `int`s: which is typically faster, a linear scan or a binary search, and why?

A. Binary search — O(log n) always beats O(n)

B. They are identical at every size

C. The scan — it touches one cache line with a perfectly predicted branch, while the search's probes each mispredict about half the time

D. Binary search, because it touches fewer cache lines

6. Which tool tells you *where* the run time went, by sampling?

A. `time ./a.out`

B. `perf record ./a.out` followed by `perf report`

C. `-fsanitize=address`

D. `-fanalyzer`

7. A function accounts for 10 % of a program's run time. What is the most you can gain by making it infinitely fast?

A. 10 % of the total

B. Unbounded — it depends on the function

C. 50 % of the total

D. 90 % of the total

ANSWER KEY · MEASURING — CLOCKS, BENCHMARKS THAT LIE AND THE TOOLS THAT TELL THE TRUTH

1. B `steady_clock` cannot go backwards or jump when the system time is corrected, which is the property an interval needs. `system_clock` is wall-clock time for timestamps; `high_resolution_clock` is an alias for one of the others and is best avoided by name.

2. C Under the as-if rule an unobserved computation need not happen, so the benchmark timed two clock reads. Making the result observable is the first of the six fixes; a resolution problem would give a small nonzero number, not zero.

3. A Run times have a floor set by the code and a long tail of interruptions above it; the minimum estimates the code's own cost and the median a typical run. The first run is the one to discard — it pays the page faults and the frequency ramp.

4. B A program that took 40 ms may take 90 ms on the next run of a shared machine. Time limits are a ceiling against runaway loops, never a measurement; the exercises count operations instead, and a count is a fact.

5. C Big-O ranks algorithms as n grows; at eight elements the constant factors decide, and the scan's predictable loop over one line wins. libstdc++'s `std::sort` switches to insertion sort below sixteen elements for the same reason.

6. B `time` says whether there is a problem; `perf record` samples the running program and `perf report` attributes the samples to functions. ASan finds memory errors at run time and `-fanalyzer` is a compile-time path analysis — neither is a profiler.

7. A Amdahl's rule: removing a part removes at most that part's share. Which is why the loop is measure, profile, change the hottest thing, and profile again — the hot spot moves.

CHECKPOINT · 25 MIN

Checkpoint — Performance and undefined behaviour

This checkpoint covers the whole module: the undefined behaviour catalogue and the well-defined detectors for it, the cost model of copies, allocations and calls, cache lines, struct padding and traversal order, what the optimiser does under the as-if rule, and how to measure without being lied to.

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

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

- Why does `if (x + 1 < x)` never detect an overflow at `-O2`, and what do you write instead?
- What does passing a `std::vector<int>` by value cost, and by `const&`?
- How many elements are moved over a thousand `push_back`s into a doubling vector, and what does `reserve` change?
- What is the size of `struct { char a; double b; char c; }` on this platform, and what does reordering do?
- Why does a column-major inner loop over a wide matrix load a cache line per element?
- What does the as-if rule permit, and why does it make an unprinted benchmark result worthless?
- Which statistic of a set of benchmark runs should you report, and why not the mean?

The three programs are an overflow-safe statistics report over `long long` values, an array-of-structs against struct-of-arrays cache-line audit built on the padding rule, and an operation counter that runs insertion sort and selection sort on the same input and reports the work each did. Every one of them counts; none of them times.

EXERCISE 1

Overflow-safe statistics

Report statistics over `long long` values without ever computing a sum, difference or product that does not fit. Read `n` and `n` values (each fits in a `long long`). Print `count n`; if `n` is 0 print `empty` and stop. Otherwise print, one per line: `min`, `max`, `range` (max − min), `sum`, `mean` (the integer quotient `sum / n`, truncated toward zero) and `squares` (the sum of every value squared). Any figure that would overflow prints the word `overflow` in its place — the mean too, when the sum did. Use `__builtin_add_overflow`, `__builtin_sub_overflow` and `__builtin_mul_overflow` on `long long` operands; once an accumulation has overflowed it stays overflowed.

Input: `n`, then `n` integers. **Output:** `count`, then either `empty` or the six statistic lines.

```
4
-3 9 2 6
```

prints

```
count 4
min -3
max 9
range 12
sum 14
mean 3
squares 130
```

INPUT 1

```
4
-3 9 2 6
```

OUTPUT 1

```
count 4
min -3
max 9
range 12
sum 14
mean 3
squares 130
```

INPUT 2

```
2
9223372036854775807
1
```

OUTPUT 2

```
count 2
min 1
max 9223372036854775807
range 9223372036854775806
sum overflow
mean overflow
squares overflow
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::vector<long long> values(static_cast<std::size_t>(n));
    for (long long& v : values) std::cin >> v;

    std::cout << "count " << n << '\n';
    if (n == 0) {
        std::cout << "empty\n";
        return 0;
    }

    long long lo = values[0];
    long long hi = values[0];
    long long sum = 0;
    long long squares = 0;
    bool sumOverflow = false;
    bool squaresOverflow = false;
    for (long long v : values) {
        if (v < lo) lo = v;
        if (v > hi) hi = v;
        if (!sumOverflow && __builtin_add_overflow(sum, v, &sum)) sumOverflow = true
        if (!squaresOverflow) {
            long long square = 0;
            if (__builtin_mul_overflow(v, v, &square) || __builtin_add_overflow(squa
                squaresOverflow = true;
        }
    }
    long long range = 0;
    const bool rangeOverflow = __builtin_sub_overflow(hi, lo, &range);

    std::cout << "min " << lo << '\n';
    std::cout << "max " << hi << '\n';
    std::cout << "range ";
    if (rangeOverflow) std::cout << "overflow\n";
    else std::cout << range << '\n';
    std::cout << "sum ";
    if (sumOverflow) std::cout << "overflow\n";
    else std::cout << sum << '\n';
    std::cout << "mean ";
    if (sumOverflow) std::cout << "overflow\n";
    else std::cout << sum / n << '\n';
    std::cout << "squares ";
    if (squaresOverflow) std::cout << "overflow\n";
    else std::cout << squares << '\n';
```

```
    return 0;
}
```

EXERCISE 2

AoS against SoA: a cache-line audit

Audit the cache lines two layouts touch, built on the padding rule from lesson 3. Read a struct as `m` type names (`char` / `bool` 1, `short` 2, `int` / `float` 4, `long` / `double` / `ptr` 8; each aligns to its size, the struct rounds up to its largest alignment), then `n` (≥ 1) elements and the 0-based index `hot` of the member a hot loop reads. Lines are 64 bytes and both layouts start at offset 0.

> **Array of structs:** element `i`'s hot member starts at byte `i * size + offset`; count the distinct lines those bytes fall in (a member never straddles a line here, since every width divides 64 and every offset is a multiple of its width).
>
> **Struct of arrays:** the hot member alone is a contiguous array of `n × width` bytes: `ceil(bytes / 64)` lines.
>
> **All fields:** AoS is `ceil(n × size / 64)`; SoA is the sum of `ceil(n × width / 64)` over every member.

Print `struct size S hot offset O width W`, `hot field: aos A soa B`, `all fields: aos C soa D`, and `hot ratio A/B` fixed to two decimals.

Input: `m`, `m` type names, `n`, `hot`. **Output:** four lines.

```
4
double double double float
32
3
```

prints

```
struct size 32 hot offset 24 width 4
hot field: aos 16 soa 2
all fields: aos 16 soa 14
hot ratio 8.00
```

INPUT 1

```
4
double double double float
32
3
```

OUTPUT 1

```
struct size 32 hot offset 24 width 4
hot field: aos 16 soa 2
all fields: aos 16 soa 14
hot ratio 8.00
```

INPUT 2

```
3
char int char
100
1
```

OUTPUT 2

```
struct size 12 hot offset 4 width 4
hot field: aos 19 soa 7
all fields: aos 19 soa 11
hot ratio 2.71
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cstddef>
#include <iomanip>
#include <iostream>
#include <map>
#include <string>
#include <vector>

struct Layout {
    std::vector<long long> offsets;
    long long size = 0;
    long long align = 1;
};

Layout layout(const std::vector<long long>& sizes) {
    Layout l;
    long long at = 0;
    for (long long s : sizes) {
        at = (at + s - 1) / s * s;
        l.offsets.push_back(at);
        at += s;
        l.align = std::max(l.align, s);
    }
    l.size = (at + l.align - 1) / l.align * l.align;
    return l;
}

constexpr long long kLine = 64;

long long linesFor(long long bytes) { return (bytes + kLine - 1) / kLine; }

// Distinct cache lines holding the member at byte offset (width bytes wide) across
// contiguous structs of stride bytes each. Members never straddle a line here:
// every width divides 64 and every offset is a multiple of the width, so the
// line of the first byte is the line of the whole member.
long long aosLines(long long n, long long stride, long long offset) {
    long long lines = 0;
    long long last = -1;
    for (long long i = 0; i < n; ++i) {
        const long long line = (i * stride + offset) / kLine;
        if (line != last) {
            ++lines;
            last = line;
        }
    }
    return lines;
}

int main() {
```

```
    const std::map<std::string, long long> width{
        {"char", 1}, {"bool", 1}, {"short", 2}, {"int", 4},
        {"float", 4}, {"long", 8}, {"double", 8}, {"ptr", 8},
    };
    int m;
    std::cin >> m;
    std::vector<long long> sizes;
    for (int i = 0; i < m; ++i) {
        std::string type;
        std::cin >> type;
        sizes.push_back(width.at(type));
    }
    long long n;
    int hot;
    std::cin >> n >> hot;

    const Layout l = layout(sizes);
    const long long hotOffset = l.offsets[static_cast<std::size_t>(hot)];
    const long long hotWidth = sizes[static_cast<std::size_t>(hot)];

    const long long aosHot = aosLines(n, l.size, hotOffset);
    const long long soaHot = linesFor(n * hotWidth);
    const long long aosAll = linesFor(n * l.size);
    long long soaAll = 0;
    for (long long s : sizes) soaAll += linesFor(n * s);

    std::cout << "struct size " << l.size << " hot offset " << hotOffset << " width
    std::cout << "hot field: aos " << aosHot << " soa " << soaHot << '\n';
    std::cout << "all fields: aos " << aosAll << " soa " << soaAll << '\n';
    std::cout << "hot ratio " << std::fixed << std::setprecision(2)
              << static_cast<double>(aosHot) / static_cast<double>(soaHot) << '\n';
    return 0;
}
```

EXERCISE 3

Insertion against selection: the work each did

Run two sorts on the same input and report the work each did, counted, never timed. Read `n` integers. **Insertion sort**: for `i` from 1, take `key = a[i]` and walk `j` down from `i`; each evaluation of `a[j - 1] > key` is one **comparison**; each `a[j] = a[j - 1]` is one **shift**; stop when the comparison fails or `j` reaches 0, then store the key. **Selection sort**: for each `i` from 0 to `n - 2`, scan `j` from `i + 1` to the end, one **comparison** per `a[j] < a[best]`; if the best index differs from `i`, `std::swap` and count one **swap**.

Print `sorted:` followed by the sorted values (one space before each), then `insertion comparisons C shifts S`, `selection comparisons C swaps S`, and `cheaper insertion`, `cheaper selection` or `cheaper tie` comparing comparisons + shifts against comparisons + swaps.

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

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

prints

```
sorted: 1 2 3 4 5 6
insertion comparisons 12 shifts 9
selection comparisons 15 swaps 3
cheaper selection
```

INPUT 1

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

OUTPUT 1

```
sorted: 1 2 3 4 5 6
insertion comparisons 12 shifts 9
selection comparisons 15 swaps 3
cheaper selection
```

INPUT 2

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

OUTPUT 2

```
sorted: 1 2 3 4 5
insertion comparisons 4 shifts 0
selection comparisons 10 swaps 0
cheaper insertion
```

REFERENCE SOLUTION

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

struct Work {
    long long comparisons = 0;
    long long moves = 0;   // insertion: shifts; selection: swaps
};

Work insertionSort(std::vector<int>& a) {
    Work w;
    for (std::size_t i = 1; i < a.size(); ++i) {
        const int key = a[i];
        std::size_t j = i;
        // invariant: a[0..i) is sorted; key sinks left past every larger element
        while (j > 0) {
            ++w.comparisons;
            if (a[j - 1] <= key) break;
            a[j] = a[j - 1];
            ++w.moves;
            --j;
        }
        a[j] = key;
    }
    return w;
}

Work selectionSort(std::vector<int>& a) {
    Work w;
    const std::size_t n = a.size();
    for (std::size_t i = 0; i + 1 < n; ++i) {
        std::size_t best = i;
        for (std::size_t j = i + 1; j < n; ++j) {
            ++w.comparisons;
            if (a[j] < a[best]) best = j;
        }
        if (best != i) {
            std::swap(a[i], a[best]);
            ++w.moves;
        }
    }
    return w;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<int> input(static_cast<std::size_t>(n));
```

```
    for (int& v : input) std::cin >> v;

    std::vector<int> a = input;
    const Work ins = insertionSort(a);
    std::vector<int> b = input;
    const Work sel = selectionSort(b);

    std::cout << "sorted:";
    for (int v : a) std::cout << ' ' << v;
    std::cout << '\n';
    std::cout << "insertion comparisons " << ins.comparisons << " shifts " << ins.mo
    std::cout << "selection comparisons " << sel.comparisons << " swaps " << sel.mov
    const long long insWork = ins.comparisons + ins.moves;
    const long long selWork = sel.comparisons + sel.moves;
    if (insWork < selWork) std::cout << "cheaper insertion\n";
    else if (selWork < insWork) std::cout << "cheaper selection\n";
    else std::cout << "cheaper tie\n";
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What is true of this snippet?

```
int total;
for (int v : values) total += v;
std::cout << total;
```

A. Prints the sum — `total` starts at 0

B. Undefined behaviour: `total` is read before it is initialised, and the optimiser may produce any result

C. Compile error: `total` is used uninitialised

D. Prints the sum at `-O0` and garbage at `-O2`, both guaranteed

2. Besides a zero divisor, which `int` division is undefined?

A. `INT_MIN / -1`

B. `INT_MAX / -1`

C. `-7 / 2`

D. `INT_MIN / 2`

3. Reading a `float`'s bits through `*reinterpret_cast<int*>(&f)` is undefined. What is the well-defined way?

A. `static_cast<int>(f)`

B. `std::memcpy` into an `int`, or C++20 `std::bit_cast<int>(f)`

C. A union of `int` and `float`

D. `(int&)f` — a C-style cast is allowed

4. `names` is a `std::vector<std::string>` of long strings. What does `for (auto s : names) total += s.size();` cost that `for (const auto& s : names)` does not?

A. Nothing — the compiler removes the copy

B. One allocation and one copy of every string

C. One move per element

D. One virtual call per element

5. A doubling vector receives 100 `push_back`s from empty. Allocations and element moves?

A. 100 and 0

B. 7 and 100

C. 8 and 127

D. 1 and 0

6. Which line removes every reallocation from a loop that will `push_back` exactly `n` elements into an empty `std::vector<int> v`?

A. `v.resize(n);` before the loop

B. `v.reserve(n);` before the loop

C. `v.shrink_to_fit();` after the loop

D. `v.emplace_back` instead of `push_back`

7. `struct Loose { char tag; int value; char flag; };` — what are `sizeof(Loose)` and the size after ordering the members by descending size?

A. 6, then 6

B. 12, then 8

C. 8, then 8

D. 12, then 12

8. What is the rule for nested loops over a row-major matrix, and why?

A. Make the innermost loop walk the innermost index, so consecutive accesses are consecutive bytes and each cache line serves sixteen elements

B. Make the innermost loop walk the outermost index, so rows are finished quickly

C. Order does not matter — the compiler swaps the loops

D. Use recursion to avoid nested loops

9. When is a struct-of-arrays layout the better shape?

A. Always — it removes padding

B. When hot loops touch one or two fields of many elements, so the loaded lines hold only the bytes those loops read

C. When every loop touches every field of each element

D. Never — AoS is what the language provides

10. Which of these can the optimiser NOT do for you?

A. Fold `3 * 4` into `12` and inline a small function

B. Delete a loop whose result is never observed

C. Replace an O(n²) algorithm with O(n log n), remove an allocation you asked for, or reorder your struct's members

D. Emit a shift for `x * 8`

11. A colleague benchmarks two implementations by compiling once with no `-O` flag and reporting the mean of one run each. Which of the six lies does that survive?

A. All of them — the comparison is fair

B. None: it measures an `-O0` build nobody ships, one run pays warm-up, and a single sample has no spread

C. Only the warm-up lie

D. Only the dead-code lie

12. Why is the mean the least informative statistic of a set of benchmark runs?

A. Because it needs floating-point arithmetic

B. Because run times sit on a floor set by the code with a long tail of interruptions above it, and every interruption drags the mean upward

C. Because it is always larger than the maximum

D. Because the median is faster to compute

13. Which clock and which stream for a timing inside a judged program?

A. `system_clock`, printed to `stdout` so it is checked

B. `steady_clock`, printed to `stderr` — `stdout` is compared and must never contain a duration

C. `high_resolution_clock`, printed to `stdout`

D. Any clock; the judge ignores numbers

14. A loop over unsorted data has a branch on each element's value and runs several times slower than the same loop over the sorted data. What explains it, and which fixes apply?

A. The sorted loop is vectorised; nothing to do for the unsorted one

B. The branch mispredicts about half the time on random data; sort first, or write a branchless form such as `total += (v < t) * v`

C. The unsorted data has more cache misses; add `reserve`

D. The optimiser inlines only the sorted version; mark the function `inline`

ANSWER KEY · CHECKPOINT — PERFORMANCE AND UNDEFINED BEHAVIOUR

1. B An automatic `int` without an initialiser holds an indeterminate value, and reading it is on the catalogue. `-Wall` warns, `int total{};` is never indeterminate, and nothing about the outcome is guaranteed at any level.

2. A `INT_MIN / -1` would be `INT_MAX + 1`, which does not fit: signed overflow. `-7 / 2` is `-3` by truncation toward zero, and the other two fit comfortably. The checked evaluator tests exactly this pair before dividing.

3. B Strict aliasing lets the compiler assume an `int*` and a `float*` never refer to the same object; `memcpy` and `bit_cast` copy the bytes without that assumption. `static_cast` converts the value, not the bits; type punning through a union is undefined in C++.

4. B `auto s` declares a new `std::string` initialised from each element — a copy, and for a string past the small buffer an allocation. `const auto&` binds a name to the element for the price of one pointer.

5. C Capacities 1, 2, 4, 8, 16, 32, 64, 128 are eight allocations, and each reallocation moves what it holds: 1 + 2 + … + 64 = 127. `reserve(100)` first would be the last option.

6. B `reserve(n)` makes one allocation of capacity `n` and moves nothing. `resize(n)` also creates `n` zeros, so the pushes would land after them; `shrink_to_fit` runs after the damage; `emplace_back` changes how the element is constructed, not whether the vector grows.

7. B Three bytes of padding after `tag` align `value` to 4, and three more round the end up to 12; `int, char, char` needs only two tail bytes: 8. `static_assert(sizeof(Tight) == 8)` locks it.

8. A `a[r][c]` lives at `r * C + c`, so the `c` loop is sequential and the `r` loop strides `4·C` bytes. The compiler does not generally interchange loops for you; the loop order is yours to get right.

9. B Summing one `float` out of a 32-byte struct reads an eighth of each line usefully; the same field in its own array fills every line. When a loop needs every field, AoS is the natural shape and about as fast.

10. C The as-if rule lets the compiler rewrite anything unobservable, but your algorithm, your allocations, your layout and your loop order are observable choices it preserves. Those are where the time goes and where hand work pays.

11. B `-O0` tells you about a program the compiler never optimised; a single run includes page faults and the frequency ramp; a single sample cannot show whether the measurement is stable. Benchmark the flags you ship, repeat, and report the minimum or median with the spread.

12. B The distribution is not symmetric: the minimum is the best estimate of what the code itself costs and the median of a typical run. Report one of those with the sample count and the spread so a reader can see whether the measurement is stable.

13. B `steady_clock` is monotonic; `stderr` is not judged. A duration in `stdout` would make the case a coin toss on a shared machine, so the track's programs count operations and keep any timing on the diagnostic stream.

14. B Same arithmetic, same memory, different predictability: a random branch costs a misprediction of fifteen to twenty cycles about every other element. Sorting turns it into two long runs; a branchless form removes the branch altogether.
