Module 17 of 20

Concurrency

std::thread and std::jthread with the join rule; data races and the RAII locks; atomics and compare-exchange; condition variables and a bounded queue; futures, promises and std::async; a thread pool, split-and-reduce and the deterministic-output rule for judged programs.

A second thread is a second flow of control sharing every byte of the process, and C++ gives you the primitives with no safety net: a variable two threads write without ordering is undefined behaviour, not merely a wrong count, and a joinable thread you forget to join ends the program. The reward is real parallelism on real cores — the study judge has two — and the patterns that make it safe are few and learnable.

The module takes them in order. Threads: how one starts, how arguments travel in (`std::ref`), `join` against `detach`, C++20's self-joining `std::jthread`, and the split-and-slot shape that needs no lock at all. Mutexes and locks: the data race defined precisely, `std::mutex` behind `std::lock_guard` and `std::scoped_lock`, small critical sections, deadlock and lock ordering. Atomics: `fetch_add` as a ticket dispenser, the compare-exchange loop, when an atomic replaces a mutex and when it cannot. Condition variables: the mutex-state-predicate protocol, spurious and lost wake-ups, a bounded producer–consumer queue and its clean shutdown. Futures: `std::async`, `std::promise` and `std::packaged_task`, exceptions crossing threads, results merged in index order. Patterns: a thread pool, work splitting and reduction, immutable data and message passing, and the rule every judged concurrent program obeys — threads compute, the main thread joins them all, and only then does anything print.

Every exercise is a judged program on the Clang 18 / C++20 runtime, and every one prints the same bytes on every run because its output never depends on scheduling: a chunked parallel sum, letter counts through `std::ref`, a histogram merged under one lock per thread, transfers under `std::scoped_lock`, Collatz lengths handed out by an atomic ticket, a bounded compare-exchange claim with a running maximum, the bounded queue itself, threads taking strict turns on a condition variable, digit sums through `std::async`, searches that keep or break a promise, a thread pool counting primes, and row sums over an immutable grid. The checkpoint adds a prime census counted with an atomic, a batch of divisions through `std::async` where some throw, and a two-stage pipeline of bounded queues whose prefix sums come out in order.

CONTENTS

1	Threads — std::thread, std::jthread and the join rule	14 min
2	Mutexes and locks — making shared writes safe	15 min
3	Atomics — lock-free counters and compare-exchange	14 min
4	Condition variables — waiting without spinning	14 min
5	Futures, promises and std::async — results that arrive later	15 min
6	Concurrency patterns — pools, reductions and the deterministic-output rule	15 min
✓	Checkpoint — Concurrency	25 min

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

LESSON 1 · 14 MIN

Threads — std::thread, std::jthread and the join rule

A thread is a second flow of control inside the same process: it has its own stack and its own position in the code, and it shares everything else — the heap, globals, static locals, open files — with every other thread. Since C++11 threads are part of the standard library (`<thread>`), portable across operating systems. This lesson covers how a thread starts, how arguments travel into it, `join` versus `detach`, the C++20 `std::jthread` that joins itself, how to split a vector across threads without a lock, and the rule every judged program in this module obeys: threads compute, the main thread joins them all, and only then does anything get printed.

Starting a thread

```
#include <iostream>
#include <thread>

void work(int id, int reps, long long& out) {
    long long acc = 0;
    for (int i = 0; i < reps; ++i) acc += id;
    out = acc;
}

int main() {
    long long result = 0;
    std::thread t(work, 7, 1000, std::ref(result));   // running from this li
    t.join();                                          // wait for it to finis
    std::cout << result << '\n';                       // 7000
    return 0;
}
```

Constructing a `std::thread` with a callable and its arguments starts the thread immediately; there is no separate `start()`. `join()` blocks the calling thread until the function returns, and after it the object is no longer *joinable*. The rule the destructor enforces is harsh: destroying a `std::thread` that is still joinable calls `std::terminate`. The library refuses to guess whether you meant "wait for it" or "abandon it", and an aborted program is easier to diagnose than a thread silently running past the end of `main`. Every `std::thread` must therefore be joined (or

detached) on every path out of the scope that owns it — including the path an exception takes, which is the case `std::jthread` was added for.

How arguments travel

The constructor *copies* the callable and each argument into storage owned by the new thread, then invokes the callable with those copies as rvalues. A parameter taken by value receives the copy, as expected. A parameter declared `T&` cannot bind to an rvalue, so passing a plain variable to it is a compile error — libstdc++ says "std::thread arguments must be invocable after conversion to rvalues" — and the fix is `std::ref(x)`, a `std::reference_wrapper` the invocation unwraps; that is why `result` above is wrapped. A `const T&` parameter compiles without help but binds to the thread's private copy, so a large vector is still copied once; `std::cref(v)` passes the original. The copy rule is a safety feature: a thread that copied its arguments cannot be surprised by the caller changing them. Whenever you write `std::ref`, you take on the obligation that the referenced object outlives the thread — trivially true when the same scope joins it.

Lambdas as thread bodies

Most thread bodies are lambdas, and the same rules apply to captures. Capturing by reference is safe *because* the owning scope joins the thread before the captured objects are destroyed:

```
std::vector<int> data(1000, 3);
long long evens = 0;
std::thread t([&data, &evens] {
    for (int x : data) if (x % 2 == 0) evens += x;
});
t.join();          // data and evens are alive until here, so the references w
```

The trap is a loop variable. If four threads are started in a loop and each captures `i` by reference, every thread reads `i` whenever it happens to run — after the loop has moved on — and a thread reading a variable the loop is writing is a data race (lesson 2). Capture the index by value and the shared containers by reference: `[&partial, &data, i]`.

join, detach and joinable

`join()` waits and then makes the object non-joinable; calling it a second time throws `std::system_error`. `detach()` also makes it non-joinable, but by disowning the thread: it keeps running with nothing waiting for it. When `main` returns the process ends and every detached thread is killed wherever it happens to be, and any reference it holds to a local of its starting scope dangles the moment that scope exits. Detached threads have their uses in servers; a program that reads input, computes and prints has none, and a judged program must never detach. The habit is *one owner, one join*: the scope that creates the threads joins them, in a `std::vector<std::thread>` walked with `for (auto& w : workers) w.join();`.

std::jthread

C++20's `std::jthread` is a `std::thread` whose destructor joins: nothing terminates, an early `return` or an exception still waits for the worker, and a `std::vector<std::jthread>` joins every element when it is destroyed. It also carries a cooperative stop mechanism: if the callable's first parameter is a `std::stop_token`, the thread receives one, and `request_stop()` — called explicitly, or by the destructor before it joins — flips it.

```
#include <stop_token>
#include <thread>

std::jthread worker([](std::stop_token st, int step) {
    long long acc = 0;
    while (!st.stop_requested() && acc < 1'000'000) acc += step;
}, 5);
// the destructor calls worker.request_stop(), then join()
```

Polling `stop_requested()` is the thread's own responsibility — nothing is interrupted by force, which is the only safe design when the thread may be holding a lock or half-way through a write. Prefer `std::jthread` in new code; this module uses `std::thread` where the join is the point being made.

Splitting work without a lock

The parallel sum below is the shape of most exercises in this module. The vector is split into contiguous chunks, `[i * n / t, (i + 1) * n / t)`, a formula that spreads the remainder

without a special case and produces empty ranges when `n < t`. Each thread writes its answer into its own element of a results vector sized *before* the threads started, so no two threads touch the same memory and no lock is needed; the merge — the only place the results meet — happens on the main thread after every join.

```
#include <iostream>
#include <thread>
#include <vector>

int main() {
    std::vector<int> v(10'000, 3);
    const std::size_t t = 4;
    std::vector<long long> partial(t, 0);
    {
        std::vector<std::jthread> workers;
        for (std::size_t i = 0; i < t; ++i) {
            const std::size_t lo = i * v.size() / t;
            const std::size_t hi = (i + 1) * v.size() / t;
            workers.emplace_back([&v, &partial, i, lo, hi] {
                long long s = 0;
                for (std::size_t k = lo; k < hi; ++k) s += v[k];
                partial[i] = s;
            });
        }
    }                                   // every worker joined here
    long long total = 0;
    for (long long p : partial) total += p;
    std::cout << total << '\n';         // 30000
    return 0;
}
```

Reading `v` from four threads at once is fine — reads never race with reads. Sizing `partial` first matters: a `push_back` from inside a thread could reallocate the buffer under another thread's feet. Note the local `s`: accumulating in a register and storing once is faster and cleaner than writing `partial[i] += v[k]` on every step.

---

hardware_concurrency and what a thread costs

`std::thread::hardware_concurrency()` returns the number of threads the hardware can run at once — a hint, and 0 when unknown. On the judge it is 2, so two threads doing arithmetic finish in about half the time and eight finish no sooner, only interleaved. Starting a thread

costs tens of microseconds plus a stack (8 MB reserved on Linux), so spawn one per chunk, never one per element. And never print the value — it differs from machine to machine, and the judge compares output exactly.

The judge rule: compute, join, then print

`std::cout` is one shared object. The standard promises that concurrent `<<` calls are not a data race, but it also says characters may interleave, and which thread's line appears first depends on the scheduler — a different answer on every run. A judged program must print exactly the same bytes every time, so the rule is mechanical: threads write into their own slots; the main thread joins every worker; the main thread prints in index order. It is also good design — computation and presentation kept apart — and every exercise in this module is built on it.

Pitfalls

Mistake	What happens
A `std::thread` destroyed while joinable	`std::terminate` — the program aborts
Passing a variable to a `T&` parameter without `std::ref`	Compile error: the thread's copy is an rvalue
Capturing the loop index by reference	Every thread may read a changed `i` — a data race and the wrong slot
`detach()` in a program that then returns from `main`	The thread is killed mid-work; references to locals dangle
Printing from inside a thread	Interleaved, order-dependent output the judge rejects
One thread per element	Thousands of thread starts; slower than the serial loop

Key takeaways

Constructing a `std::thread` starts it; every joinable thread must be joined (or detached) before it is destroyed, or the program terminates.

Arguments are copied into the thread and passed as rvalues: use `std::ref` / `std::cref` for references, and keep the referenced objects alive until the join.

Capture loop indices by value; capture shared data by reference only when the owning scope joins the thread.

`std::jthread` joins in its destructor and offers a cooperative `std::stop_token`; prefer it in new code.

Split work into contiguous chunks, give each thread its own slot in a pre-sized results vector, and merge on the main thread after joining.

Threads compute; the main thread joins them all and prints aggregated results in a fixed order — never print from a worker.

EXERCISE 1

Parallel sum, chunked

Read `t` (the number of threads, 1–8) and `n`, then `n` integers. Split the indices into `t` **contiguous chunks** with the lesson's formula — chunk `i` is `[i * n / t, (i + 1) * n / t)`, which spreads the remainder and yields empty chunks when `n < t`. Start one `std::thread` per chunk; each sums its chunk into a local and stores the result in **its own slot** of a `std::vector<long long>` sized before any thread starts. Join every thread in a loop, and only then print, in index order, `thread <i>: [<lo>, <hi>) sum <s>` and finally `total <sum>`.

Nothing is printed from inside a thread, and no lock is needed: reads of the input never race, and each thread writes a different slot.

Input: `t n`, then `n` integers. **Output:** `t + 1` lines.

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

→

```
thread 0: [0, 2) sum 3
thread 1: [2, 4) sum 7
thread 2: [4, 7) sum 18
total 28
```

INPUT 1

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

OUTPUT 1

```
thread 0: [0, 2) sum 3
thread 1: [2, 4) sum 7
thread 2: [4, 7) sum 18
total 28
```

INPUT 2

```
4 2
10 -10
```

OUTPUT 2

```
thread 0: [0, 0) sum 0
thread 1: [0, 1) sum 10
thread 2: [1, 1) sum 0
thread 3: [1, 2) sum -10
total 0
```

REFERENCE SOLUTION

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

int main() {
    std::size_t t, n;
    std::cin >> t >> n;
    std::vector<long long> v(n);
    for (auto& x : v) std::cin >> x;

    std::vector<long long> partial(t, 0);   // one slot per thread, sized before any
    std::vector<std::thread> workers;
    for (std::size_t i = 0; i < t; ++i) {
        const std::size_t lo = i * n / t;
        const std::size_t hi = (i + 1) * n / t;
        workers.emplace_back([&v, &partial, i, lo, hi] {
            long long s = 0;
            for (std::size_t k = lo; k < hi; ++k) s += v[k];
            partial[i] = s;
        });
    }
    for (auto& w : workers) w.join();

    long long total = 0;
    for (std::size_t i = 0; i < t; ++i) {
        const std::size_t lo = i * n / t;
        const std::size_t hi = (i + 1) * n / t;
        std::cout << "thread " << i << ": [" << lo << ", " << hi << ") sum " << part
        total += partial[i];
    }
    std::cout << "total " << total << '\n';
    return 0;
}
```

EXERCISE 2

Letter ranges through std::ref

Write `void count_in(const std::string& text, char lo, char hi, int& out)`, which counts the characters of `text` that, lower-cased, fall in `[lo, hi]`, and stores the count in `out`. Read `t` (1–26) and a line of text. Split the alphabet into `t` contiguous ranges with the chunk formula over 26 letters — range `i` runs from `'a' + i * 26 / t` to `'a' + (i + 1) * 26 / t - 1` — and start one **function-based** thread per range: `std::thread(count_in, std::cref(text), lo, hi, std::ref(counts[i]))`. The `std::ref` is the point: a thread copies its arguments and passes them

as rvalues, so an `int&` parameter needs the wrapper, and `std::cref` spares the text a copy per thread.

After joining, print `<lo>-<hi>: <count>` per range and `letters <total>`.

Input: `t`, then one line of text. **Output:** `t + 1` lines.

```
2
The quick brown fox jumps over the lazy dog
```

→

```
a-m: 16
n-z: 19
letters 35
```

INPUT 1

```
2
The quick brown fox jumps over the lazy
dog
```

OUTPUT 1

```
a-m: 16
n-z: 19
letters 35
```

INPUT 2

```
1
Hello, World!
```

OUTPUT 2

```
a-z: 10
letters 10
```

REFERENCE SOLUTION

```cpp
#include <cctype>
#include <functional>
#include <iostream>
#include <string>
#include <thread>
#include <vector>

void count_in(const std::string& text, char lo, char hi, int& out) {
    int n = 0;
    for (char ch : text) {
        char c = static_cast<char>(std::tolower(static_cast<unsigned char>(ch)));
        if (c >= lo && c <= hi) ++n;
    }
    out = n;
}

int main() {
    int t;
    std::cin >> t;
    std::cin.ignore();
    std::string text;
    std::getline(std::cin, text);

    std::vector<int> counts(t, 0);
    std::vector<std::thread> workers;
    for (int i = 0; i < t; ++i) {
        char lo = static_cast<char>('a' + i * 26 / t);
        char hi = static_cast<char>('a' + (i + 1) * 26 / t - 1);
        workers.emplace_back(count_in, std::cref(text), lo, hi, std::ref(counts[i]));
    }
    for (auto& w : workers) w.join();

    int total = 0;
    for (int i = 0; i < t; ++i) {
        char lo = static_cast<char>('a' + i * 26 / t);
        char hi = static_cast<char>('a' + (i + 1) * 26 / t - 1);
        std::cout << lo << '-' << hi << ": " << counts[i] << '\n';
        total += counts[i];
    }
    std::cout << "letters " << total << '\n';
    return 0;
}
```

QUIZ

1. A `std::thread` object is destroyed while still joinable. What happens?

A. The thread is joined automatically

B. The thread is detached and keeps running

C. `std::terminate` is called — the program aborts

D. Undefined behaviour

2.

```
void work(int reps, long long& out);
long long result = 0;
std::thread t(work, 1000, result);
```

Why does this fail to compile, and what is the fix?

A. `work` must be a lambda; wrap it

B. Thread arguments are copied and passed as rvalues, so `long long&` cannot bind; pass `std::ref(result)`

C. `result` must be `static`

D. The thread needs `std::launch::async`

3.

```
for (int i = 0; i < 4; ++i)
    workers.emplace_back([&] { partial[i] = compute(i); });
```

What is wrong?

A. Nothing; `[&]` is the usual capture for threads

B. `i` is captured by reference, so a thread reads whatever value the loop variable has when it runs — a data race and the wrong slot

C. Lambdas cannot be thread bodies

D. `partial` must be captured by value

4. What does the destructor of a `std::jthread` do that `std::thread`'s does not?

A. Deletes the thread's stack

B. Calls `request_stop()` on its stop source and then joins

C. Detaches the thread so `main` can return

D. Throws if the thread is still running

5. A judged program calls `detach()` on its worker and returns from `main` while the worker is still computing. What happens?

A. The worker finishes and its output is appended

B. The process ends and the worker is killed wherever it is; any reference it held to a local of `main` dangles

C. `main` waits for detached threads

D. Compile error: detached threads are not allowed

6. Why must a judged program never print from inside a worker thread?

A. `std::cout` is not thread-safe and the program will crash

B. Concurrent `<<` calls are not a data race, but characters and lines may interleave in scheduler order, so the output differs between runs

C. Only the main thread can write to `std::cout`

D. Printing blocks the other threads

7. With `n = 10` elements and `t = 4` threads, what chunks does `[i * n / t, (i + 1) * n / t)` produce?

A. `[0,3) [3,6) [6,9) [9,10)`

B. `[0,2) [2,5) [5,7) [7,10)`

C. `[0,2) [2,4) [4,6) [6,10)`

D. `[0,3) [3,5) [5,8) [8,10)`

ANSWER KEY · THREADS — STD::THREAD, STD::JTHREAD AND THE JOIN RULE

1. C The library refuses to guess between waiting and abandoning: destroying a joinable `std::thread` terminates the program. `std::jthread` is the type whose destructor joins.

2. B The constructor copies each argument into the thread's own storage and invokes the callable with rvalues; a non-const reference parameter cannot bind to those. `std::ref` wraps the original so the invocation unwraps it to a real reference.

3. B The loop keeps writing `i` while the threads read it. Capture the index by value and the shared vector by reference: `[&partial, i]`.

4. B `std::jthread` joins in its destructor, so an early return or an exception still waits for the worker, and it first requests a cooperative stop through the `std::stop_token` its callable may accept.

5. B Returning from `main` ends the process regardless of detached threads. A program that reads, computes and prints has no use for `detach`; the scope that creates threads joins them.

6. B The standard guarantees no data race on `std::cout` but nothing about ordering. The judge compares bytes, so threads write to their own slots and the main thread prints after joining.

7. B The bounds are 0, 10/4 = 2, 20/4 = 5, 30/4 = 7 and 40/4 = 10, so the remainder is spread as sizes 2, 3, 2, 3 with no special case.

LESSON 2 · 15 MIN

Mutexes and locks — making shared writes safe

The moment two threads write the same variable, the program stops being the program you wrote. This lesson defines the data race precisely — it is undefined behaviour, not merely a wrong count — and then covers the tools that remove it: `std::mutex` and the RAII lock types (`std::lock_guard`, `std::scoped_lock`, `std::unique_lock`), the discipline of small critical sections, the deadlock that two locks taken in opposite orders produce and the ordering rules that prevent it, and `std::shared_mutex` for data that is read far more often than it is written.

What a data race is

`++counter` looks like one step. To the CPU it is three — load the value, add one, store it back — and two threads can interleave them: both load 41, both store 42, and one increment is lost. Losing increments is the visible symptom, but the language rule is stronger. Two threads access the same memory location, at least one of them writes, and nothing orders the accesses: that is a **data race**, and the standard says a program containing one has undefined behaviour. The compiler assumes there is no race, so it may keep `counter` in a register for the whole loop and store it once at the end; a thread might then never see the other's writes at all. The program below is broken, and it stays broken however many times it happens to print 200000:

```
// BROKEN: a data race — undefined behaviour
#include <iostream>
#include <thread>

int counter = 0;
void bump() { for (int i = 0; i < 100000; ++i) ++counter; }

int main() {
    std::thread a(bump), b(bump);
    a.join(); b.join();
    std::cout << counter << '\n';   // often below 200000; formally, anything
    return 0;
}
```

Reads alone never race: any number of threads may read a `const` table at once. The race needs a write, and the cure is to make sure no thread can touch the location while another is

in the middle of writing it.

---

std::mutex and std::lock_guard

A mutex (*mutual exclusion*) is a lock exactly one thread can hold. `lock()` blocks until the holder calls `unlock()`; the code between the two calls is a **critical section**, and only one thread at a time is inside it. Never call the pair by hand: an early `return` or an exception between them leaves the mutex locked forever, and every other thread blocks on it until the judge kills the process. The RAII lock types (Module 7, lesson 3) unlock in their destructor on every path:

```
#include <mutex>

std::mutex m;
int counter = 0;

void bump() {
    for (int i = 0; i < 100000; ++i) {
        std::lock_guard<std::mutex> lock(m);   // locked here
        ++counter;
    }                                          // unlocked here
}
```

Now the final value is exactly 200000, on every run. The mutex also establishes a *happens-before* edge: whatever a thread wrote before unlocking is visible to the next thread that locks, so the lock protects not just the increment but every variable touched inside the section.

Give the lock a name. `std::lock_guard<std::mutex>{m};` compiles, creates a temporary, and destroys it at the semicolon — the critical section is unprotected and no warning says so. And keep the mutex next to the data it guards, in a class, with a comment saying which members it covers:

```
class Tally {
public:
    void add(long long x) { std::lock_guard<std::mutex> lock(m_); value_ += x;
    long long value() const { std::lock_guard<std::mutex> lock(m_); return val
private:
    mutable std::mutex m_;   // guards value_
    long long value_ = 0;
};
```

`mutable` lets a `const` member function lock (Module 8, lesson 3). Note that `value()` locks too: a read that skips the lock races with `add`, and the rule is symmetric — every access to the guarded data, reads included, goes through the mutex.

Keep the critical section small

A lock serialises: whatever happens inside it runs one thread at a time, and the more of the program that is inside, the less parallelism is left. The pattern is to compute on private data and lock only to publish:

```
void tally(const std::vector<int>& v, std::size_t lo, std::size_t hi,
           long long& total, std::mutex& m) {
    long long local = 0;
    for (std::size_t i = lo; i < hi; ++i) local += v[i];   // no lock: local
    std::lock_guard<std::mutex> lock(m);
    total += local;                                         // one lock per th
}
```

Same answer as locking around every `+=`, with one lock per thread instead of one per element. An uncontended lock costs about twenty nanoseconds; a contended one can cost a context switch, thousands of times more. Never do I/O, heavy allocation or a call into unknown code while holding a lock.

---

std::scoped_lock: several mutexes at once

Moving money between two accounts needs both accounts locked. Taking them one after the other invites the classic deadlock: thread 1 locks A then waits for B, thread 2 locks B then waits for A, and both wait forever. `std::scoped_lock` (C++17) takes any number of mutexes

and acquires all of them with a deadlock-avoiding algorithm, whatever order the arguments come in; it needs no template argument thanks to class template argument deduction:

```
struct Account {
    std::mutex m;              // guards balance
    long long balance = 0;
};

void transfer(Account& from, Account& to, long long amount) {
    std::scoped_lock lock(from.m, to.m);   // both, or neither — never one
    from.balance -= amount;
    to.balance += amount;
}
```

With one mutex, `std::scoped_lock` is `std::lock_guard` under a better name, and modern code uses it for both. One caveat: `from` and `to` must be different objects — locking a `std::mutex` a thread already holds is undefined behaviour, and in practice a deadlock — so a self-transfer must be rejected before the lock.

---

std::unique_lock

`std::unique_lock` is the flexible one: it can be constructed without locking (`std::defer_lock`), unlocked early and re-locked, moved to another owner, and it is the lock type `std::condition_variable` requires (lesson 4). The price is a flag saying whether it currently holds the mutex. Use it when you need one of those features; otherwise `std::scoped_lock`.

```
std::unique_lock<std::mutex> lock(m);
const auto snapshot = shared;   // copy under the lock
lock.unlock();                  // release before the slow part
process(snapshot);
```

Deadlock and lock ordering

A deadlock needs a cycle of waiting: each thread holds something another one wants. Four rules break the cycle. Hold one lock at a time whenever you can. When you must hold two, take them in one fixed global order — by account number, by address — in every thread, or let `std::scoped_lock` acquire them together. Never call code you do not control (a callback, a

virtual function) while holding a lock: it may lock something of its own. And keep sections short, so that contention stays rare. `std::recursive_mutex`, which one thread may lock repeatedly, exists for legacy designs; needing it usually means the class's locking is tangled.

---

std::shared_mutex in brief

A `std::shared_mutex` has two modes: many readers at once under `std::shared_lock`, or one writer under `std::unique_lock`. It suits a configuration table read on every request and changed once an hour. For data written often it is slower than a plain `std::mutex`, because both modes do more bookkeeping.

Pitfalls

Mistake	What happens
`std::lock_guard<std::mutex>{m};` (an unnamed temporary)	Unlocked at the semicolon; the section is unprotected
Locking on the write path but not the read path	Still a data race — reads must lock too
Two mutexes taken in different orders by two threads	Deadlock; the program hangs until the judge kills it
Locking a `std::mutex` the thread already holds	Undefined behaviour; a deadlock in practice
Holding a lock while doing the real work	Threads run one at a time; no speed-up
`lock()` / `unlock()` by hand around a `return` or a `throw`	The unlock is skipped; every other thread waits forever

---

Key takeaways

A data race — two threads, one location, at least one write, no ordering — is undefined behaviour, whatever the count happens to print.

`std::mutex` plus a named RAII lock (`std::scoped_lock`, `std::lock_guard`) makes a critical section that unlocks on every path.

Every access to guarded data, reads included, takes the lock; keep the mutex beside the data it guards.

Compute on private data and lock only to publish; small sections are what keep threads parallel.

Several mutexes: `std::scoped_lock` takes them all at once; otherwise one fixed order everywhere.

`std::unique_lock` when you need to unlock early or wait on a condition; `std::shared_mutex` for read-mostly data.

EXERCISE 1

A histogram merged under one lock per thread

Complete `class Tally`, which owns a `std::array<long long, 10>` of last-digit counts and a count of merges, both guarded by one `std::mutex` declared `mutable` so that const readers can lock too. `merge(const std::array<long long, 10>&)` adds a local histogram into the shared one and counts the merge, under a `std::lock_guard`; `snapshot()` and `merges()` are const and take the same lock — a read that skips the lock is still a data race.

Read `t` and `n`, then `n` non-negative integers. Each of `t` threads takes a contiguous chunk, builds a **local** histogram of `x % 10` with no lock at all, and calls `merge` **once** — compute privately, lock to publish. After joining, print `<d>: <count>` for digits 0–9 and `merges <t>`.

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

```
2 6
10 21 32 41 55 60
```

→

```
0: 2
1: 2
2: 1
3: 0
4: 0
5: 1
6: 0
7: 0
8: 0
9: 0
merges 2
```

INPUT 1

```
2 6
10 21 32 41 55 60
```

OUTPUT 1

```
0: 2
1: 2
2: 1
3: 0
4: 0
5: 1
6: 0
7: 0
8: 0
9: 0
merges 2
```

INPUT 2

```
3 1
7
```

OUTPUT 2

```
0: 0
1: 0
2: 0
3: 0
4: 0
5: 0
6: 0
7: 1
8: 0
9: 0
merges 3
```

REFERENCE SOLUTION

```cpp
#include <array>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

class Tally {
public:
    void merge(const std::array<long long, 10>& local) {
        std::lock_guard<std::mutex> lock(m_);
        for (int d = 0; d < 10; ++d) counts_[d] += local[d];
        ++merges_;
    }
    std::array<long long, 10> snapshot() const {
        std::lock_guard<std::mutex> lock(m_);
        return counts_;
    }
    int merges() const {
        std::lock_guard<std::mutex> lock(m_);
        return merges_;
    }
private:
    mutable std::mutex m_;   // guards counts_ and merges_
    std::array<long long, 10> counts_{};
    int merges_ = 0;
};

int main() {
    std::size_t t, n;
    std::cin >> t >> n;
    std::vector<long long> v(n);
    for (auto& x : v) std::cin >> x;

    Tally tally;
    std::vector<std::thread> workers;
    for (std::size_t i = 0; i < t; ++i) {
        const std::size_t lo = i * n / t, hi = (i + 1) * n / t;
        workers.emplace_back([&v, &tally, lo, hi] {
            std::array<long long, 10> local{};
            for (std::size_t k = lo; k < hi; ++k) ++local[v[k] % 10];
            tally.merge(local);
        });
    }
    for (auto& w : workers) w.join();

    const std::array<long long, 10> counts = tally.snapshot();
    for (int d = 0; d < 10; ++d) std::cout << d << ": " << counts[d] << '\n';
    std::cout << "merges " << tally.merges() << '\n';
```

```
    return 0;
}
```

EXERCISE 2

Transfers under std::scoped_lock

`struct Account { std::mutex m; long long balance = 0; }` — the mutex beside the data it guards. Write `bool transfer(Account& from, Account& to, long long amount)`: refuse (return `false`) when `from` and `to` are the **same object**, since locking a mutex a thread already holds is undefined behaviour; otherwise take **both** mutexes with one `std::scoped_lock` and move the amount. Balances may go negative — no funds check — so the final state does not depend on the order the threads happened to run in.

Read `a` accounts and `t` threads, the opening balances, then `m` transfers `from to amount`. Give each thread a contiguous chunk of the transfer list; each thread counts its own refusals in a local and the main thread sums those after joining. Print `account <i>: <balance>` for every account, `self-transfers rejected <k>` and `total <sum of balances>`. A `std::vector<Account>` cannot be resized once it exists (a mutex is neither copyable nor movable), so construct it with its final size.

Input: `a t`, a line of `a` balances, `m`, then `m` lines. **Output:** `a + 2` lines.

```
3 2
100 50 0
4
0 1 30
1 2 20
2 2 5
0 2 100
```

→

```
account 0: -30
account 1: 60
account 2: 120
self-transfers rejected 1
total 150
```

INPUT 1

```
3 2
100 50 0
4
0 1 30
1 2 20
2 2 5
0 2 100
```

OUTPUT 1

```
account 0: -30
account 1: 60
account 2: 120
self-transfers rejected 1
total 150
```

INPUT 2

```
1 3
5
2
0 0 5
0 0 1
```

OUTPUT 2

```
account 0: 5
self-transfers rejected 2
total 5
```

REFERENCE SOLUTION

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

struct Account {
    std::mutex m;              // guards balance
    long long balance = 0;
};

struct Transfer {
    int from;
    int to;
    long long amount;
};

bool transfer(Account& from, Account& to, long long amount) {
    if (&from == &to) return false;
    std::scoped_lock lock(from.m, to.m);   // both, or neither — never one
    from.balance -= amount;
    to.balance += amount;
    return true;
}

int main() {
    std::size_t a, t;
    std::cin >> a >> t;
    std::vector<Account> accounts(a);   // sized once: Account is not movable
    for (auto& acc : accounts) std::cin >> acc.balance;
    std::size_t m;
    std::cin >> m;
    std::vector<Transfer> transfers(m);
    for (auto& tr : transfers) std::cin >> tr.from >> tr.to >> tr.amount;

    std::vector<int> rejected(t, 0);
    std::vector<std::thread> workers;
    for (std::size_t i = 0; i < t; ++i) {
        const std::size_t lo = i * m / t, hi = (i + 1) * m / t;
        workers.emplace_back([&accounts, &transfers, &rejected, i, lo, hi] {
            int refused = 0;
            for (std::size_t k = lo; k < hi; ++k) {
                const Transfer& tr = transfers[k];
                if (!transfer(accounts[tr.from], accounts[tr.to], tr.amount)) ++refu
            }
            rejected[i] = refused;
        });
    }
    for (auto& w : workers) w.join();
```

```
    long long total = 0;
    for (std::size_t i = 0; i < a; ++i) {
        std::cout << "account " << i << ": " << accounts[i].balance << '\n';
        total += accounts[i].balance;
    }
    int refusedTotal = 0;
    for (int r : rejected) refusedTotal += r;
    std::cout << "self-transfers rejected " << refusedTotal << '\n';
    std::cout << "total " << total << '\n';
    return 0;
}
```

QUIZ

1. Two threads run `++counter` on a plain `int` with no synchronisation. What does the C++ standard say about the program?

A. It prints a count that may be too low

B. It has undefined behaviour — a data race — whatever it happens to print

C. It is correct on x86-64 because `++` is one instruction

D. It deadlocks

2.

```
std::lock_guard<std::mutex>{m};
++counter;
```

What is wrong?

A. Nothing; the guard protects the increment

B. The guard is an unnamed temporary destroyed at the semicolon, so the increment runs unlocked

C. `lock_guard` needs `std::defer_lock`

D. The mutex is locked twice

3. `add()` locks the mutex before writing `value_`, but `value()` reads it without locking "because it only reads". Is this correct?

A. Yes: reads never race

B. No: a read concurrent with a write is still a data race; every access to guarded data, reads included, takes the lock

C. Yes, as long as `value_` is `volatile`

D. No, but only because `value()` is `const`

4. Why is `std::scoped_lock lock(from.m, to.m);` preferred over locking `from.m` and then `to.m`?

A. It is faster

B. It acquires both with a deadlock-avoiding algorithm, so two threads transferring in opposite directions cannot each hold one and wait for the other

C. It locks only one of them

D. It makes the mutexes recursive

5. A thread that holds `std::mutex m` calls a function that locks `m` again. What happens?

A. The second lock succeeds; mutexes count nested locks

B. Undefined behaviour — in practice a deadlock, the thread waiting for itself

C. The second lock is ignored

D. An exception is thrown

6. Which version keeps the most parallelism when four threads sum a large vector into a shared total?

A. Lock around every `total += v[i]`

B. Lock once before the loop and unlock after

C. Sum into a local variable without a lock, then lock once to add the local into the total

D. Use `std::cout` to combine the results

7. Why is the mutex in `class Tally` declared `mutable std::mutex m_;`?

A. So that it can be copied

B. So that `const` member functions such as `value() const` can lock it

C. Because mutexes must be `mutable` to work across threads

D. To allow `std::scoped_lock` to take it

ANSWER KEY · MUTEXES AND LOCKS — MAKING SHARED WRITES SAFE

1. B Two unordered accesses to one location with at least one write is a data race, and a program with one has undefined behaviour. Lost increments are the visible symptom; the compiler may also keep the value in a register and never see the other thread's writes.

2. B A temporary lives until the end of its full expression. Give the guard a name — `std::lock_guard<std::mutex> lock(m);` — so it lives until the closing brace.

3. B Reads only avoid racing with other reads. A read while `add` is mid-write is undefined behaviour, so the mutex is declared `mutable` and the const reader locks it too.

4. B Taking two locks in different orders in two threads is the classic deadlock. `std::scoped_lock` (or one fixed global order) removes the cycle.

5. B `std::mutex` is not recursive; locking it twice from one thread is undefined behaviour and usually hangs. `std::recursive_mutex` exists for legacy designs; needing it signals tangled locking.

6. C Compute on private data and lock only to publish: one lock per thread instead of one per element, and no thread waits while another does the real work.

7. B Locking modifies the mutex, and a const member function may modify only `mutable` members. A reader that cannot lock would have to skip the lock, which is a data race.

LESSON 3 · 14 MIN

Atomics — lock-free counters and compare-exchange

A mutex around a single `++` is a heavy tool for a light job. `std::atomic<T>` makes the operations on one variable indivisible without a lock: a `fetch_add` is one instruction the CPU performs as a unit, and no other thread can see the value half-updated. This lesson covers what an atomic promises, the operations on `std::atomic<int>`, the compare-exchange loop that turns any read-modify-write into an atomic one, when an atomic can replace a mutex and when it cannot, the memory-ordering argument (and why the default is the one to use), `std::atomic_flag`, false sharing, and why atomics on `double` are a trap for judged programs.

What atomic means

`std::atomic<int> n{0}; n.fetch_add(1);` is a read-modify-write that no other thread can interleave with: on x86-64 it compiles to a single `lock xadd` instruction. There is no torn read (a thread seeing half of a new value) and no lost update. Every operation on an atomic is also free of data races by definition, so the counter from lesson 2 becomes well defined by changing one type:

```
#include <atomic>
#include <iostream>
#include <thread>

std::atomic<int> counter{0};
void bump() { for (int i = 0; i < 100000; ++i) counter.fetch_add(1); }

int main() {
    std::thread a(bump), b(bump);
    a.join(); b.join();
    std::cout << counter.load() << '\n';   // exactly 200000
    return 0;
}
```

`std::atomic<int>`, `<long long>`, `<bool>` and pointers are *lock-free* on every mainstream platform — `std::atomic<int>::is_always_lock_free` is `true` — meaning the hardware does the work. A `std::atomic<BigStruct>` still compiles, but the library implements it with a

hidden lock, which is rarely what anyone wanted. Atomics cannot be copied or moved: an atomic member makes the class non-copyable, and a `std::vector<std::atomic<int>>` can be sized at construction but not `push_back`ed into.

The operations

Call	Meaning
`a.load()`, `a.store(v)`	Read, write; `int x = a;` and `a = v;` are the same
`a.fetch_add(v)`, `a.fetch_sub(v)`	Add or subtract; **returns the previous value**
`++a`, `a += v`	`fetch_add` that yields the new value
`a.exchange(v)`	Store `v`, return the old value
`a.compare_exchange_weak(expected, desired)`	If `a == expected` store `desired` and return `true`; else copy `a` into `expected` and return `false`

The previous value returned by `fetch_add` is a ticket dispenser: `int mine = next.fetch_add(1);` gives every calling thread a distinct number, with no lock and no duplicates, which is how work is handed out dynamically (lesson 6).

The check-then-act trap

An atomic makes each *operation* indivisible, not a sequence of them. `if (a.load() < limit) a.store(a.load() + 1);` reads twice and writes once, and another thread can run between any two of those steps; the result is a race in disguise — not undefined behaviour, since every access is atomic, but a wrong count and a limit that is overshot. Likewise two atomics that must change together (`items` and `total`) are two independent steps, and a reader can see one updated and the other not. The rule: one variable, one operation — an atomic; a compound update, or an invariant across variables — a mutex.

---

compare_exchange: any update, atomically

When the new value depends on the old one and `fetch_add` does not express it — a maximum, a clamp, a claim that must not exceed a limit — the loop is:

```
void update_max(std::atomic<int>& best, int candidate) {
    int cur = best.load();
    while (candidate > cur && !best.compare_exchange_weak(cur, candidate)) {
        // another thread changed best; cur now holds its value — decide agai
    }
}
```

`compare_exchange_weak(expected, desired)` says: *if the atomic still holds* `expected`, *replace it with* `desired`. When it does, it returns `true` and the update is done. When some other thread got in first, it returns `false` and writes the current value into `expected`, so the next iteration decides again from fresh data. The `weak` form may also fail spuriously — return `false` even though the values matched — which is harmless inside a loop and cheaper on some processors; `compare_exchange_strong` never does, and is the one to use when there is no loop around it. The loop is *lock-free*: a thread descheduled in the middle of it holds nothing, and the others proceed.

Claiming numbered slots is the same shape with a bound:

```
int cur = next.load();
while (cur < limit && !next.compare_exchange_weak(cur, cur + 1)) { }
// cur < limit: slot cur is ours; otherwise none were left
```

Memory ordering, and why the default is fine

Every atomic operation takes an optional `std::memory_order`. The default, `memory_order_seq_cst`, gives a single global order that all threads agree on — the model you have in your head. `memory_order_relaxed` guarantees only that the operation itself is atomic, with no ordering relative to other memory; it is correct for a counter read only after every thread has been joined, because `join` itself synchronises. `memory_order_acquire` / `release` are the pair for publishing data through a flag: the writer stores the data then release-stores the flag; a reader that acquire-loads the flag as set is guaranteed to see the data. The speed difference is measurable on ARM and nearly nil on x86-64, and a wrong ordering is a bug that appears once a month on a customer's machine. Leave the default unless you can write down why a weaker order is correct.

Flags and spinlocks

`std::atomic<bool> done{false};` is the standard stop signal: a worker loops `while (!done.load())`, and the main thread stores `true`. `std::atomic_flag` is the guaranteed-lock-free primitive underneath: `test_and_set()` sets it and returns the old value, `clear()` resets it, and a spinlock is nothing more:

```
std::atomic_flag busy;                                  // clear when default-cons
void enter() { while (busy.test_and_set()) { } }    // spin until we were the
void leave() { busy.clear(); }
```

Spinning burns a core while it waits, so a spinlock is only right when the wait is nanoseconds long. C++20 added `wait()` / `notify_one()` on atomics, which park the thread instead of spinning — condition-variable behaviour without the mutex.

False sharing, in outline

CPUs move memory in 64-byte cache lines. Two atomics that sit in the same line — `std::atomic<int> a, b;` declared together, or adjacent elements of an array — are physically one unit, and two cores hammering them bounce the line back and forth even though they never touch the same variable. The symptom is a parallel program slower than the serial one. The fix is `alignas(64)` on each hot atomic (`std::hardware_destructive_interference_size` in `<new>` names the number), or, better, not to share at all: accumulate in a local and `fetch_add` the total once, so the atomic is touched once per thread rather than once per element.

Atomics and determinism

`std::atomic<double>` supports `fetch_add` since C++20, and it is a trap for judged programs: floating-point addition is not associative, so the sum of contributions that arrive in scheduler order differs in its last bits from run to run. Integer counters are commutative and safe; for floating point, keep each thread's partial and add them on the main thread in index order — the parallel-sum shape from lesson 1.

Pitfalls

Mistake	What happens
`a.store(a.load() + 1)`	Two operations; increments are lost — use `fetch_add`
Two atomics that must agree	A reader can see one updated and not the other; use a mutex
`compare_exchange_weak` without a loop	A spurious failure skips the update
`std::memory_order_relaxed` to publish data through a flag	The reader may see the flag before the data
`push_back` on a `std::vector<std::atomic<int>>`	Compile error: atomics are not movable
`std::atomic<double>` accumulation in a judged program	Last-bit differences between runs

Key takeaways

An atomic operation is indivisible and never a data race; `std::atomic<int>` is lock-free hardware, not a hidden mutex.

`fetch_add` returns the previous value — a ticket dispenser with no lock.

One variable, one operation: atomic. A compound update or an invariant across variables: mutex.

`compare_exchange_weak` in a loop turns any read-modify-write into an atomic one; on failure it refreshes `expected`.

Keep the default `seq_cst` ordering unless you can justify a weaker one; `relaxed` is fine for counters read after the join.

Touch shared atomics once per thread, not once per element — it avoids false sharing and is simply faster.

EXERCISE 1

Collatz lengths from a ticket dispenser

`collatz_steps(long long x)` returns how many steps `x` takes to reach 1 (halve when even, `3x + 1` when odd; 1 takes 0 steps). Read `t` and `n` and compute the step count for every `x` in `1..n` —

but hand the work out **dynamically**: a `std::atomic<int> next{0}` is the ticket dispenser, and each thread loops `int i = next.fetch_add(1); if (i ⩾ n) break;`, storing the answer for `i + 1` in `steps[i]`. Because `fetch_add` returns the previous value atomically, no index is handed out twice and none is skipped, whichever thread is faster.

After joining, on the main thread, find the number with the longest chain (the smallest such number on a tie) and the total of all step counts. Print `longest <x> steps <k>` and `total <sum>`. Which thread computed which index must not appear anywhere.

Input: `t n` (`n` at least 1). **Output:** two lines.

```
2 10
```

→

```
longest 9 steps 19
total 67
```

INPUT 1

```
2 10
```

OUTPUT 1

```
longest 9 steps 19
total 67
```

INPUT 2

```
3 1
```

OUTPUT 2

```
longest 1 steps 0
total 0
```

REFERENCE SOLUTION

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

int collatz_steps(long long x) {
    int steps = 0;
    while (x ≠ 1) {
        x = x % 2 == 0 ? x / 2 : 3 * x + 1;
        ++steps;
    }
    return steps;
}

int main() {
    int t, n;
    std::cin >> t >> n;
    std::vector<int> steps(n, 0);
    std::atomic<int> next{0};
    std::vector<std::thread> workers;
    for (int i = 0; i < t; ++i) {
        workers.emplace_back([&steps, &next, n] {
            while (true) {
                int i = next.fetch_add(1);
                if (i ⩾ n) break;
                steps[i] = collatz_steps(i + 1);
            }
        });
    }
    for (auto& w : workers) w.join();

    int longest = 0;
    long long total = 0;
    for (int i = 0; i < n; ++i) {
        if (steps[i] > steps[longest]) longest = i;
        total += steps[i];
    }
    std::cout << "longest " << longest + 1 << " steps " << steps[longest] << '\n';
    std::cout << "total " << total << '\n';
    return 0;
}
```

EXERCISE 2

Bounded claims and a running maximum

Read `t`, a limit `L`, and `n` values. The threads take contiguous chunks and, for every value in theirs, do three things. **Maximum:** fold the value into `std::atomic<long long> best` with the lesson's compare-exchange loop — load, and while the value is greater and `compare_exchange_weak` fails, try again with the refreshed `expected`. **Claim:** try to claim one of `L` slots with the bounded loop `while (cur < L && !claimed.compare_exchange_weak(cur, cur + 1)) {}` — the claim succeeds only when `cur < L` after the loop. **Big values:** count values of 100 or more in a **local** and `fetch_add` that local into `std::atomic<int> big` once per thread.

After joining print `max <best>`, `big <count>` and `claimed <count>`. Which values won a claim depends on timing and is never printed; how many did — `min(n, L)` — does not.

Input: `t L n`, then `n` integers (`n` at least 1). **Output:** three lines.

```
2 3 5
7 250 -4 100 99
```

→

```
max 250
big 2
claimed 3
```

INPUT 1

```
2 3 5
7 250 -4 100 99
```

OUTPUT 1

```
max 250
big 2
claimed 3
```

INPUT 2

```
3 10 4
-5 -1 -9 -2
```

OUTPUT 2

```
max -1
big 0
claimed 4
```

REFERENCE SOLUTION

```cpp
#include <atomic>
#include <iostream>
#include <limits>
#include <thread>
#include <vector>

void update_max(std::atomic<long long>& best, long long candidate) {
    long long cur = best.load();
    while (candidate > cur && !best.compare_exchange_weak(cur, candidate)) {
        // another thread changed best; cur now holds its value — decide again
    }
}

bool claim(std::atomic<int>& claimed, int limit) {
    int cur = claimed.load();
    while (cur < limit && !claimed.compare_exchange_weak(cur, cur + 1)) {
    }
    return cur < limit;
}

int main() {
    std::size_t t;
    int limit;
    std::size_t n;
    std::cin >> t >> limit >> n;
    std::vector<long long> v(n);
    for (auto& x : v) std::cin >> x;

    std::atomic<long long> best{std::numeric_limits<long long>::min()};
    std::atomic<int> claimed{0};
    std::atomic<int> big{0};
    std::vector<std::thread> workers;
    for (std::size_t i = 0; i < t; ++i) {
        const std::size_t lo = i * n / t, hi = (i + 1) * n / t;
        workers.emplace_back([&, lo, hi] {
            int localBig = 0;
            for (std::size_t k = lo; k < hi; ++k) {
                update_max(best, v[k]);
                claim(claimed, limit);
                if (v[k] >= 100) ++localBig;
            }
            big.fetch_add(localBig);
        });
    }
    for (auto& w : workers) w.join();
    std::cout << "max " << best.load() << '\n';
    std::cout << "big " << big.load() << '\n';
    std::cout << "claimed " << claimed.load() << '\n';
```

```
    return 0;
}
```

QUIZ

1. What does `a.fetch_add(1)` return?

A. The new value

B. The previous value

C. `true` if the add succeeded

D. Nothing; it returns `void`

2.

```
std::atomic<int> a{0};
// in each of two threads, 1000 times:
a.store(a.load() + 1);
```

What is the final value?

A. Exactly 2000

B. Between 1000 and 2000: the load and the store are two operations, and increments between them are lost

C. Undefined behaviour

D. Exactly 1000

3. `compare_exchange_weak(expected, desired)` returns `false`. What has happened to `expected`?

A. Nothing; it is unchanged

B. It now holds the atomic's current value, ready for the next iteration of the loop

C. It has been set to `desired`

D. It has been zeroed

4. Two atomics `items` and `total` must always agree (`total` is the sum of the items). Is updating them with two `fetch_add` calls enough?

A. Yes, both updates are atomic

B. No: a reader can observe one updated and the other not; an invariant across variables needs a mutex

C. Yes, if `memory_order_seq_cst` is used

D. No, atomics cannot be used in pairs

5. When is `std::memory_order_relaxed` a safe choice for `counter.fetch_add(1, ...)`?

A. Never; always use the default

B. When the counter is read only after every thread has been joined, because `join` itself synchronises

C. When the counter publishes a data structure to other threads

D. When the counter is a `double`

6. `std::vector<std::atomic<int>> counters; counters.push_back(std::atomic<int>{0});` — what happens?

A. It works

B. Compile error: atomics are neither copyable nor movable, so the vector cannot insert one; size it at construction instead

C. Undefined behaviour at run time

D. It works only with `emplace_back`

7. Why must a judged program never accumulate a floating-point sum through `std::atomic<double>::fetch_add`?

A. `fetch_add` on `double` does not exist

B. Floating-point addition is not associative, so the sum depends on the order the scheduler let the threads add, and the last bits vary between runs

C. It is a data race

D. Atomic doubles are always locked

ANSWER KEY · ATOMICS — LOCK-FREE COUNTERS AND COMPARE-EXCHANGE

1. B `fetch_add` returns the value before the addition, which is what makes it a ticket dispenser: every caller gets a distinct number. `++a` is the form that yields the new value.

2. B Each access is atomic, so there is no data race, but the sequence is not: another thread can store between the load and the store. Use `fetch_add(1)` for a single indivisible increment.

3. B On failure the current value is written into `expected`, so a loop can decide again from fresh data without a separate load. The weak form may also fail spuriously, which is why it sits in a loop.

4. B Atomicity covers one operation on one variable. A compound update, or an invariant spanning two variables, is exactly what a mutex-protected critical section is for.

5. B A relaxed operation is still atomic; it only gives up ordering relative to other memory. A counter whose final value is read after the joins needs no such ordering. Publishing data through a flag needs release/acquire.

6. B Copying an atomic would not be atomic, so the type forbids it; a vector of atomics can be constructed with a size (elements value-initialised) but never grown by insertion.

7. B C++20 provides the operation, but the order of contributions is scheduling-dependent. Keep each thread's partial and add them in index order on the main thread.

LESSON 4 · 14 MIN

Condition variables — waiting without spinning

A mutex answers "who may touch this now?". It does not answer "when will there be something to take?". A consumer that finds the queue empty could unlock, check again, unlock, check again — burning a core to discover nothing has changed — or sleep for a millisecond and add that millisecond to every hand-off. `std::condition_variable` is the third option: the thread sleeps until another thread says the state has changed. This lesson covers the protocol (a mutex, a state, a predicate), spurious and lost wake-ups, `notify_one` against `notify_all`, a bounded producer–consumer queue, and shutting such a queue down so that every thread exits.

The protocol

A condition variable is always used with three things: a `std::mutex`, some state protected by that mutex, and a predicate over the state that says whether the waiter may proceed. The waiter takes a `std::unique_lock`, then calls `wait(lock, predicate)`:

```
#include <condition_variable>
#include <mutex>

std::mutex m;
std::condition_variable cv;
bool ready = false;       // the state; guarded by m

void consumer() {
    std::unique_lock<std::mutex> lock(m);
    cv.wait(lock, [] { return ready; });   // sleeps until ready is true
    // lock is held again here, and ready is true
}

void producer() {
    {
        std::lock_guard<std::mutex> lock(m);
        ready = true;                      // change the state under the mutex
    }
    cv.notify_one();                       // then wake a waiter
}
```

`wait` atomically releases the mutex and puts the thread to sleep; when the thread is woken it re-acquires the mutex before `wait` returns. The two-argument form is exactly `while (!predicate()) wait(lock);` — the predicate is checked with the lock held, before sleeping and after every wake-up. That loop is not optional, for two reasons.

Spurious and lost wake-ups

A **spurious wake-up** is a return from `wait` that nobody asked for; operating systems permit them, and a waiter that treats "I woke up" as "the condition holds" will read an empty queue. The predicate form re-checks and goes back to sleep. A **lost wake-up** is the opposite: the producer sets `ready` and notifies *before* the consumer has started waiting, so nobody is there to receive it. That is why the state lives under the mutex and the predicate is tested with the mutex held — a consumer that arrives late sees `ready == true` and never waits at all. The notify itself carries no information; only the state does. A `notify_one` with nothing changed wakes a thread that checks the predicate and sleeps again.

`wait` requires a `std::unique_lock<std::mutex>`, not a `std::lock_guard`, because it must unlock and re-lock. `std::condition_variable_any` accepts any lock type and, in C++20, a `std::stop_token`; the plain one is faster and is what you want with a `std::mutex`.

---

notify_one and notify_all

`notify_one` wakes one waiter — right when any one of them can consume the change, such as one item pushed to a queue with several consumers. `notify_all` wakes every waiter — right when the change concerns all of them: a "shutting down" flag, a "go" signal for a group of workers, or when different waiters wait on different predicates and you cannot tell which the change satisfies. Waking everyone when one would do is a "thundering herd" — the others check, find nothing, and sleep again — but it is merely slow, never wrong. Notifying while still holding the mutex is correct; releasing first, as in `producer` above, lets the woken thread acquire the mutex without immediately blocking on it.

A bounded producer–consumer queue

The queue below is the workhorse of concurrent programs: producers `push` and block when it is full, consumers `pop` and block when it is empty, and two condition variables keep the two

kinds of waiter apart.

```
#include <condition_variable>
#include <mutex>
#include <optional>
#include <queue>

class BoundedQueue {
public:
    explicit BoundedQueue(std::size_t capacity) : capacity_(capacity) {}

    void push(long long v) {
        std::unique_lock<std::mutex> lock(m_);
        notFull_.wait(lock, [this] { return items_.size() < capacity_; });
        items_.push(v);
        lock.unlock();
        notEmpty_.notify_one();
    }

    std::optional<long long> pop() {
        std::unique_lock<std::mutex> lock(m_);
        notEmpty_.wait(lock, [this] { return !items_.empty() || closed_; });
        if (items_.empty()) return std::nullopt;       // closed and drained
        long long v = items_.front();
        items_.pop();
        lock.unlock();
        notFull_.notify_one();
        return v;
    }

    void close() {
        { std::lock_guard<std::mutex> lock(m_); closed_ = true; }
        notEmpty_.notify_all();
    }

private:
    std::mutex m_;                              // guards everything below
    std::condition_variable notFull_;
    std::condition_variable notEmpty_;
    std::queue<long long> items_;
    std::size_t capacity_;
    bool closed_ = false;
};
```

A `capacity` of 1 makes the producer and consumer strictly alternate; a larger one lets a fast producer run ahead by that many items. The bound is the point: an unbounded queue lets a producer outrun a slow consumer until memory runs out, and *back-pressure* — the producer sleeping in `push` — is what a bound provides.

Shutting down cleanly

The hardest part of a queue is the end. A consumer blocked in `pop` on an empty queue sleeps forever unless something wakes it, and a program with a sleeping thread never exits — the judge reports a timeout. `close()` is that something: it sets `closed_` under the mutex and calls `notify_all`, so *every* consumer wakes; the predicate `!items_.empty() || closed_` lets them through; and `pop` returns the remaining items first, `std::nullopt` only once the queue is drained. The consumer's loop is then `while (auto item = q.pop())`, and the whole program is:

```
int main() {
    BoundedQueue q(4);
    long long total = 0;
    std::thread consumer([&q, &total] {
        while (auto item = q.pop()) total += *item;    // ends on nullopt
    });
    std::thread producer([&q] {
        for (long long i = 1; i ⩽ 100; ++i) q.push(i);
        q.close();
    });
    producer.join();
    consumer.join();
    std::cout << total << '\n';                         // 5050, every run
    return 0;
}
```

The consumer's total is deterministic because the *set* of items is fixed by the producer, and integer addition does not care about the order in which the scheduler let them through. What is not deterministic — how full the queue got, how often each side waited — must never be printed.

Turn-taking with a condition variable

The same protocol imposes an order between threads when one is needed. With `int turn` under the mutex, thread `i` waits for `turn % k == i`, does its step, increments `turn`, and calls `notify_all`; the threads then run strictly round-robin, and a sequence they build together comes out identical every time. It is a demonstration, not a design — threads forced to alternate might as well be one — but it shows that a condition variable is a general "wait until the state says so", not only a queue signal.

Timeouts, and the C++20 additions

`wait_for(lock, duration, predicate)` and `wait_until` return `false` if the deadline passes with the predicate still false, which is how a thread avoids waiting forever on a producer that died. C++20 adds three simpler primitives for common special cases: `std::latch` (a countdown that opens once), `std::barrier` (a reusable rendezvous for a fixed group) and `std::counting_semaphore` (a permit count). They cover "wait for all workers to be ready" and "at most four at a time" without a mutex or a predicate; the condition variable remains the general tool.

Pitfalls

Mistake	What happens
`wait(lock)` with no predicate	A spurious wake-up proceeds on a false condition
Changing the state without the mutex	A lost wake-up; the waiter sleeps forever
Forgetting `close()` / `notify_all` at the end	Consumers never exit; the judge times out
Holding the lock while processing the item	Producers and other consumers all wait on the processing
`notify_one` for a change that concerns every waiter	Some threads never wake
`std::lock_guard` with `wait`	Compile error: `wait` needs `std::unique_lock`

Key takeaways

A condition variable is used with a mutex, a state and a predicate; `wait(lock, pred)` re-checks the predicate after every wake-up.

Change the state under the mutex, then notify; the state carries the information, the notify only says "look again".

`notify_one` when any one waiter can act; `notify_all` for shutdown, group signals and mixed predicates.

A bounded queue blocks producers when full and consumers when empty; the bound is back-pressure.

Shut down with a closed flag and `notify_all`, drain the remaining items, and return "nothing" only when empty and closed.

Totals over a fixed set of items are deterministic; queue depths and wait counts are not, so never print them.

EXERCISE 1

The bounded queue

Complete `class BoundedQueue` from the lesson: `push(v)` waits under a `std::unique_lock` until the queue is below capacity, appends, unlocks and notifies `notEmpty_`; `pop()` waits until there is an item **or the queue is closed**, returns `std::nullopt` only when it is closed and drained, otherwise takes the front, unlocks and notifies `notFull_`; `close()` sets the flag under the mutex and `notify_all`s, so every consumer wakes and exits. Every `wait` takes a predicate — a spurious wake-up must go back to sleep.

Read `capacity` and `n`, then `n` integers. A producer thread pushes them in order and then closes the queue; a consumer thread pops until `nullopt`, tracking the count, the sum and the maximum. Join both, then print `items <count> sum <sum> max <max>`. The totals are the same on every run because the set of items is fixed; how full the queue got is not, and is never printed.

Input: `capacity n` (both at least 1), then `n` integers. **Output:** one line.

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

→

```
items 5 sum 18 max 9
```

INPUT 1

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

OUTPUT 1

```
items 5 sum 18 max 9
```

INPUT 2

```
1 1
42
```

OUTPUT 2

```
items 1 sum 42 max 42
```

REFERENCE SOLUTION

```cpp
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <optional>
#include <queue>
#include <thread>
#include <vector>

class BoundedQueue {
public:
    explicit BoundedQueue(std::size_t capacity) : capacity_(capacity) {}

    void push(long long v) {
        std::unique_lock<std::mutex> lock(m_);
        notFull_.wait(lock, [this] { return items_.size() < capacity_; });
        items_.push(v);
        lock.unlock();
        notEmpty_.notify_one();
    }

    std::optional<long long> pop() {
        std::unique_lock<std::mutex> lock(m_);
        notEmpty_.wait(lock, [this] { return !items_.empty() || closed_; });
        if (items_.empty()) return std::nullopt;       // closed and drained
        long long v = items_.front();
        items_.pop();
        lock.unlock();
        notFull_.notify_one();
        return v;
    }

    void close() {
        {
            std::lock_guard<std::mutex> lock(m_);
            closed_ = true;
        }
        notEmpty_.notify_all();
    }

private:
    std::mutex m_;                              // guards everything below
    std::condition_variable notFull_;
    std::condition_variable notEmpty_;
    std::queue<long long> items_;
    std::size_t capacity_;
    bool closed_ = false;
};

int main() {
```

```
    std::size_t capacity, n;
    std::cin >> capacity >> n;
    std::vector<long long> values(n);
    for (auto& v : values) std::cin >> v;

    BoundedQueue q(capacity);
    long long count = 0, sum = 0, best = 0;
    std::thread consumer([&q, &count, &sum, &best] {
        while (auto item = q.pop()) {
            if (count == 0 || *item > best) best = *item;
            ++count;
            sum += *item;
        }
    });
    std::thread producer([&q, &values] {
        for (long long v : values) q.push(v);
        q.close();
    });
    producer.join();
    consumer.join();
    std::cout << "items " << count << " sum " << sum << " max " << best << '\n';
    return 0;
}
```

EXERCISE 2

Strict turns

Read `k` (threads, 1–8) and `r` (rounds). Start `k` threads; thread `i` (0-based) must append the letter `'A' + i` to a shared `std::string` exactly `r` times, and the threads must take **strict turns**: an `int turn` under the mutex says whose go it is, thread `i` waits with a predicate for `turn % k == i`, appends, increments `turn`, and `notify_all`s — every waiter must re-check, because only one of them can proceed and you cannot tell which is asleep. The result is the same string on every run, `ABC...ABC...`.

After joining, print `sequence <string>` and `length <k * r>`.

Input: `k r`. **Output:** two lines.

```
3 2
```

→

```
sequence ABCABC
length 6
```

INPUT 1

```
3 2
```

OUTPUT 1

```
sequence ABCABC
length 6
```

INPUT 2

```
1 4
```

OUTPUT 2

```
sequence AAAA
length 4
```

REFERENCE SOLUTION

```cpp
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <string>
#include <thread>
#include <vector>

int main() {
    int k, r;
    std::cin >> k >> r;

    std::mutex m;
    std::condition_variable cv;
    int turn = 0;              // guarded by m
    std::string sequence;      // guarded by m

    std::vector<std::thread> workers;
    for (int i = 0; i < k; ++i) {
        workers.emplace_back([&, i] {
            for (int round = 0; round < r; ++round) {
                std::unique_lock<std::mutex> lock(m);
                cv.wait(lock, [&] { return turn % k == i; });
                sequence.push_back(static_cast<char>('A' + i));
                ++turn;
                lock.unlock();
                cv.notify_all();
            }
        });
    }
    for (auto& w : workers) w.join();
    std::cout << "sequence " << sequence << '\n';
    std::cout << "length " << sequence.size() << '\n';
    return 0;
}
```

QUIZ

1. Why is `cv.wait(lock, pred)` used rather than `cv.wait(lock)` followed by the action?

A. The one-argument form does not exist

B. A wait may return spuriously; the predicate form re-checks the condition after every wake-up and sleeps again if it does not hold

C. The predicate form is faster

D. The one-argument form does not release the mutex

2. A producer sets `ready = true` without holding the mutex and calls `notify_one`. What can go wrong?

A. Nothing; the notify is what wakes the consumer

B. A lost wake-up: the consumer may check `ready`, see false, and be about to wait when the flag flips and the notify arrives with nobody waiting — it then sleeps forever

C. The notify throws

D. The consumer wakes twice

3. When is `notify_all` required rather than `notify_one`?

A. Always; `notify_one` is deprecated

B. When the change concerns every waiter — a shutdown flag, a group go-signal — or when waiters wait on different predicates and you cannot tell which the change satisfies

C. Only when more than one item was pushed

D. Never; it only wastes wake-ups

4. Why does `std::condition_variable::wait` take a `std::unique_lock<std::mutex>` and not a `std::lock_guard`?

A. `lock_guard` is slower

B. `wait` must unlock the mutex while sleeping and re-lock it before returning; `unique_lock` supports that, `lock_guard` cannot

C. `lock_guard` cannot be passed by reference

D. It is a historical accident

5. A bounded queue's `pop()` waits on `!items_.empty() || closed_`. After `close()`, what does `pop()` return while items remain?

A. `std::nullopt` immediately

B. The remaining items in order, then `std::nullopt` once the queue is empty

C. The last item only

D. It throws

6. In a producer–consumer program, which of these is safe to print in a judged case?

A. The maximum number of items the queue held at once

B. How many times the producer had to wait

C. The sum and count of all items consumed

D. Which thread consumed item 7

7. What does the capacity bound of a `BoundedQueue` provide that an unbounded queue does not?

A. Faster pushes

B. Back-pressure: a producer that outruns the consumer sleeps in `push` instead of filling memory

C. Ordering of items

D. Thread safety

ANSWER KEY · CONDITION VARIABLES — WAITING WITHOUT SPINNING

1. B Operating systems permit spurious wake-ups, and a waiter that treats waking as proof of the condition reads an empty queue. `wait(lock, pred)` is `while (!pred()) wait(lock);`.

2. B The state must change under the mutex so the consumer's predicate check and its sleep are one atomic step relative to the producer. The notify itself carries no information; only the state does.

3. B `notify_one` is right when any single waiter can consume the change. Waking everyone for a single item is merely slow; waking one for a shutdown leaves the others asleep forever.

4. B `wait` atomically releases the mutex and sleeps, then reacquires it on wake-up. Only a lock type that can be unlocked and re-locked can be handed to it.

5. B The predicate lets a consumer through when there is an item or the queue is closed; the body returns `nullopt` only when it is empty. Closing means "no more will come", not "discard what is here".

6. C The set of items is fixed by the producer, so totals over it are the same every run. Queue depth, wait counts and which thread did what depend on scheduling.

7. B Both are thread-safe and FIFO. Only the bound makes a fast producer wait, which is what keeps memory finite when consumers are slow.

LESSON 5 · 15 MIN

Futures, promises and std::async — results that arrive later

A thread that computes a value has nowhere to put it: `std::thread` returns nothing, and an exception inside its body terminates the whole program. The `<future>` header fixes both with one idea. A **future** is a one-shot channel that will hold either a value or an exception; whoever produces it is elsewhere, and `get()` waits for it and hands it over — or rethrows. This lesson covers the three producers of a future — `std::async`, `std::promise` and `std::packaged_task` — the launch policies and the trap in `std::async`'s destructor, how an exception crosses a thread boundary, and a parallel sum that collects its partial results in a fixed order.

---

std::async and std::future

```
#include <future>
#include <iostream>

long long sum_to(long long n) {
    long long s = 0;
    for (long long i = 1; i ⩽ n; ++i) s += i;
    return s;
}

int main() {
    std::future<long long> f = std::async(std::launch::async, sum_to, 1'000'00
    // … other work here runs in parallel with sum_to …
    std::cout << f.get() << '\n';   // waits if needed: 500000500000
    return 0;
}
```

`std::async` takes a callable and its arguments the way `std::thread` does — copied, `std::ref` for references — and returns a `std::future<R>` for the callable's return type. `get()` blocks until the result exists, then moves it out; a future can be `get()` exactly once, and `valid()` is false afterwards (a second `get()` throws `std::future_error`). `wait()` blocks without taking the value; `wait_for(duration)` returns a `std::future_status` — `ready`, `timeout` or `deferred`. When several threads need the same result, `f.share()` turns it into a `std::shared_future`, which may be copied and read many times.

Launch policies

The first argument decides where the work runs. `std::launch::async` starts a new thread now. `std::launch::deferred` runs the function lazily, on the thread that calls `get()`, and not at all if nobody does — a lazy value, not concurrency. The default, `std::launch::async | std::launch::deferred`, lets the implementation choose; libstdc++ picks a thread, but code that needs parallelism should say so: always write `std::launch::async` when the point is to run in parallel.

The trap is the destructor. A future obtained from `std::async` — and only that kind — **blocks in its destructor** until the task finishes. So this loop is sequential:

```
for (int i = 0; i < 4; ++i)
    std::async(std::launch::async, work, i);   // temporary future destroyed
```

Each temporary waits for its task before the next starts. Keep the futures — `std::vector<std::future<int>>` — and the tasks overlap.

Exceptions cross threads through the future

If the callable throws, the exception is caught by the library, stored in the shared state, and rethrown by `get()` on whichever thread calls it. That is the whole mechanism for error handling across threads, and it is the only one: an exception that escapes a plain `std::thread` body calls `std::terminate`.

```
int parse_positive(const std::string& s) {
    const int v = std::stoi(s);                    // may throw std::invali
    if (v <= 0) throw std::out_of_range("not positive: " + s);
    return v;
}

auto f = std::async(std::launch::async, parse_positive, std::string("-3"));
try {
    std::cout << f.get() << '\n';
} catch (const std::exception& e) {
    std::cout << "error: " << e.what() << '\n';      // error: not positive:
}
```

Catch by `const&` at the `get()`, as in Module 15; the dynamic type is preserved, so a `std::out_of_range` handler matches only that.

---

std::promise: setting the value by hand

`std::async` produces the value by returning it. A `std::promise<T>` lets any code, on any thread, put a value into a future explicitly — once:

```
#include <future>
#include <thread>

void find_first(const std::vector<int>& v, int target, std::promise<int> resul
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == target) { result.set_value(static_cast<int>(i)); return; }
    }
    result.set_exception(std::make_exception_ptr(std::runtime_error("not found
}

std::promise<int> p;
std::future<int> f = p.get_future();                // get_future() once, befor
std::thread worker(find_first, std::cref(v), 42, std::move(p));
const int index = f.get();                          // or throws runtime_error
worker.join();
```

`set_value` or `set_exception` is called exactly once; a second call throws `std::future_error` (`promise_already_satisfied`). A promise destroyed without either call stores a `std::future_error` with `broken_promise`, so the waiter does not hang. `std::promise<void>` carries no value and is a one-shot signal — "the worker has initialised". Inside a `catch` block, `std::current_exception()` captures whatever was thrown, ready for `set_exception`. Promises are move-only, so they travel into the thread with `std::move`.

---

std::packaged_task: a callable with a future attached

A `std::packaged_task<R(Args...)>` wraps a callable; calling the task runs it and stores the return value or exception in a future obtained beforehand:

```
std::packaged_task<long long(long long)> task(sum_to);
std::future<long long> f = task.get_future();
std::thread t(std::move(task), 1000);   // run it on another thread …
t.join();
std::cout << f.get() << '\n';           // 500500
```

The task can be called on any thread, later, by anyone who holds it — which is exactly what a thread pool needs: the pool stores tasks in a queue, a worker calls them, and the submitter already holds the future (lesson 6). Like promises, packaged tasks are move-only.

A parallel sum, joined deterministically

```
std::vector<std::future<long long>> parts;
for (std::size_t i = 0; i < t; ++i) {
    const std::size_t lo = i * v.size() / t, hi = (i + 1) * v.size() / t;
    parts.push_back(std::async(std::launch::async, [&v, lo, hi] {
        long long s = 0;
        for (std::size_t k = lo; k < hi; ++k) s += v[k];
        return s;
    }));
}
long long total = 0;
for (auto& p : parts) total += p.get();   // index order, whatever finished fi
```

The threads finish in whatever order the scheduler decides; the `get()` loop consumes them in index order, so the merge is the same on every run — and `parts[i].get()` is the per-thread partial, printable as "thread i". A future per chunk (not per element) keeps thread creation cheap, and the vector of futures does the joining, since each `get()` waits.

Which one when

Tool	Who produces the value	Typical use
`std::async`	The function's return	Run a function in parallel and collect its result
`std::packaged_task`	The wrapped callable, when someone calls the task	Queued work: thread pools
`std::promise`	Explicit `set_value` / `set_exception`	Hand-built protocols, a signal, a result produced mid-function
`std::thread` + `std::ref`	You write into a slot	Fire-and-join workers with no error path

Pitfalls

Mistake	What happens
Discarding the future from `std::async`	Its destructor waits; the loop runs sequentially
Omitting the launch policy and expecting a thread	Allowed to be deferred; runs at `get()`
`get()` twice	`std::future_error`; use `std::shared_future` for several readers
A promise destroyed without a value	The waiter gets `broken_promise`
Calling `get_future()` after moving the promise	The moved-from promise has no shared state
Throwing inside a `std::thread` body	No future to carry it — `std::terminate`

Key takeaways

A future is a one-shot channel for a value *or* an exception; `get()` waits, moves the value out, and rethrows what the producer threw.

`std::async(std::launch::async, f, args…)` runs `f` on a new thread; name the policy, and keep the future — an `std::async` future's destructor blocks.

`std::promise` sets a result by hand (`set_value` / `set_exception`, exactly once); `std::packaged_task` attaches a future to any callable.

Exceptions cross threads only through futures; an exception escaping a `std::thread` terminates the program.

Collect futures in a vector and `get()` them in index order: deterministic merging whatever finished first.

EXERCISE 1

Digit sums through std::async

Read `t` and `n`, then `n` non-negative integers. Split the indices into `t` contiguous chunks and launch each with `std::async(std::launch::async, ...)` — name the policy, so the work really runs in parallel — computing the sum of the **digit sums** of the values in the chunk (`305` contributes `3 + 0 + 5 = 8`). Keep every `std::future<long long>` in a vector; a future from `std::async` blocks in its destructor, so discarding it would make the loop sequential. Then `get()` them **in index order**, whatever order they finished, printing `chunk <i>: <sum>` and finally `total <sum>`.

Input: `t n`, then `n` integers. **Output:** `t + 1` lines.

```
2 5
12 305 7 99 1000
```

→

```
chunk 0: 11
chunk 1: 26
total 37
```

INPUT 1

```
2 5
12 305 7 99 1000
```

OUTPUT 1

```
chunk 0: 11
chunk 1: 26
total 37
```

INPUT 2

```
3 2
0 0
```

OUTPUT 2

```
chunk 0: 0
chunk 1: 0
chunk 2: 0
total 0
```

REFERENCE SOLUTION

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

int digit_sum(long long x) {
    int s = 0;
    while (x > 0) {
        s += static_cast<int>(x % 10);
        x /= 10;
    }
    return s;
}

int main() {
    std::size_t t, n;
    std::cin >> t >> n;
    std::vector<long long> v(n);
    for (auto& x : v) std::cin >> x;

    std::vector<std::future<long long>> parts;
    for (std::size_t i = 0; i < t; ++i) {
        const std::size_t lo = i * n / t, hi = (i + 1) * n / t;
        parts.push_back(std::async(std::launch::async, [&v, lo, hi] {
            long long s = 0;
            for (std::size_t k = lo; k < hi; ++k) s += digit_sum(v[k]);
            return s;
        }));
    }
    long long total = 0;
    for (std::size_t i = 0; i < t; ++i) {
        long long s = parts[i].get();
        std::cout << "chunk " << i << ": " << s << '\n';
        total += s;
    }
    std::cout << "total " << total << '\n';
    return 0;
}
```

EXERCISE 2

A promise kept or broken

Write `void find_first(const std::vector<int>& v, int target, std::promise<int> result)`, which searches `v` for `target` and either `set_value`s the first index or `set_exception`s a `std::runtime_error("not found")` — exactly one of the two, once. Read `n` values, then `q` targets. For each target create a `std::promise<int>`, take its future **before** moving the promise into

a `std::thread`, and keep the futures and the threads in vectors. Then, in query order, `get()` each future inside a `try`: print `<target> found at <index>` or, on `std::runtime_error`, `<target> not found`. Join every thread before returning.

Input: `n`, a line of `n` integers, `q`, a line of `q` targets. **Output:** `q` lines.

```
5
4 8 15 16 23
3
15 4 42
```

→

```
15 found at 2
4 found at 0
42 not found
```

INPUT 1

```
5
4 8 15 16 23
3
15 4 42
```

OUTPUT 1

```
15 found at 2
4 found at 0
42 not found
```

INPUT 2

```
1
7
2
7 8
```

OUTPUT 2

```
7 found at 0
8 not found
```

REFERENCE SOLUTION

```cpp
#include <exception>
#include <functional>
#include <future>
#include <iostream>
#include <stdexcept>
#include <thread>
#include <vector>

void find_first(const std::vector<int>& v, int target, std::promise<int> result) {
    for (std::size_t i = 0; i < v.size(); ++i) {
        if (v[i] == target) {
            result.set_value(static_cast<int>(i));
            return;
        }
    }
    result.set_exception(std::make_exception_ptr(std::runtime_error("not found")));
}

int main() {
    std::size_t n;
    std::cin >> n;
    std::vector<int> v(n);
    for (auto& x : v) std::cin >> x;
    std::size_t q;
    std::cin >> q;
    std::vector<int> targets(q);
    for (auto& x : targets) std::cin >> x;

    std::vector<std::future<int>> results;
    std::vector<std::thread> workers;
    for (int target : targets) {
        std::promise<int> p;
        results.push_back(p.get_future());
        workers.emplace_back(find_first, std::cref(v), target, std::move(p));
    }
    for (std::size_t i = 0; i < q; ++i) {
        try {
            int index = results[i].get();
            std::cout << targets[i] << " found at " << index << '\n';
        } catch (const std::runtime_error&) {
            std::cout << targets[i] << " not found\n";
        }
    }
    for (auto& w : workers) w.join();
    return 0;
}
```

QUIZ

1.

```
for (int i = 0; i < 4; ++i)
    std::async(std::launch::async, work, i);
```

How do the four tasks run?

A. In parallel, four threads at once

B. One after another: each temporary future is destroyed at the semicolon and its destructor waits for the task

C. Not at all; the futures are discarded

D. In parallel, but their results are lost

2. `std::async(work, 5)` with no launch policy — is the work guaranteed to run on another thread?

A. Yes, always

B. No: the default policy allows deferred execution on the thread that calls `get()`; write `std::launch::async` when parallelism is the point

C. No: it always runs deferred

D. Yes, unless `work` throws

3. What happens on the second call to `get()` on the same `std::future`?

A. It returns the same value again

B. It throws `std::future_error`; for several readers use `share()` to obtain a `std::shared_future`

C. It blocks forever

D. It returns a default value

4. A function run through `std::async` throws `std::out_of_range`. Where does the exception surface?

A. In the worker thread, calling `std::terminate`

B. From `future.get()` on whichever thread calls it, with its dynamic type preserved

C. It is silently dropped

D. From the future's destructor

5. A `std::promise<int>` is destroyed without `set_value` or `set_exception` having been called. What does the waiting `get()` do?

A. Blocks forever

B. Throws `std::future_error` with `broken_promise`

C. Returns 0

D. Returns immediately with an indeterminate value

6. An exception escapes the body of a `std::thread`. What happens?

A. It is caught by the thread that called `join()`

B. `std::terminate` is called — there is no future to carry it

C. It is stored and rethrown by `join()`

D. The thread is detached

7. What does `std::packaged_task<long long(long long)>` add to a plain callable?

A. A new thread

B. A future that will hold the return value or exception when someone calls the task — which is what a thread pool needs to queue work

C. Automatic retries

D. A mutex around the call

ANSWER KEY · FUTURES, PROMISES AND STD::ASYNC — RESULTS THAT ARRIVE LATER

1. B A future obtained from `std::async` blocks in its destructor until the task completes. Keep the futures in a vector and the tasks overlap.

2. B The default is `async | deferred` and the implementation chooses. libstdc++ starts a thread, but code that needs concurrency should say so explicitly.

3. B `get()` moves the result out and leaves the future invalid. A `std::shared_future` may be copied and read many times.

4. B The library catches the exception, stores it in the shared state, and `get()` rethrows it. That is the only mechanism for carrying exceptions across threads; a plain `std::thread` body that throws terminates.

5. B The promise's destructor stores a `broken_promise` error so that no waiter hangs. `set_value` twice throws `promise_already_satisfied`.

6. B Only futures transport exceptions between threads. A worker started with `std::thread` must catch its own exceptions or be wrapped in a `std::packaged_task`.

7. B The task can be moved into a queue and called later by any worker; the submitter already holds the future from `get_future()`.

LESSON 6 · 15 MIN

Concurrency patterns — pools, reductions and the deterministic-output rule

The earlier lessons gave the primitives; this one gives the shapes they are combined into, because production concurrency is a handful of patterns applied carefully rather than clever synchronisation. A thread pool turns "a thread per task" into "a queue and a fixed number of workers". Work splitting and reduction is the shape of every parallel loop. Immutable data and message passing remove most locks by removing the sharing. A short list of rules avoids deadlock in practice. The lesson ends with the interview answer to "threads or async?" and the checklist that keeps a judged concurrent program deterministic.

A minimal thread pool

Starting a thread per task is expensive and unbounded. A pool starts `w` workers once; each loops taking the next task from a queue guarded by a mutex and a condition variable; `submit` wraps the caller's function in a `std::packaged_task`, keeps the future for the caller, and pushes the task; the destructor flips a stop flag, wakes everyone and joins. The version below runs tasks that return `long long`; a generic `submit` template is the same code with `std::invoke_result_t` for the return type.

```cpp
#include <condition_variable>
#include <deque>
#include <functional>
#include <future>
#include <mutex>
#include <thread>
#include <vector>

class ThreadPool {
public:
    explicit ThreadPool(int workers) {
        for (int i = 0; i < workers; ++i) threads_.emplace_back([this] { run()
    }
    ~ThreadPool() {
        { std::lock_guard<std::mutex> lock(m_); stopping_ = true; }
        cv_.notify_all();
        for (auto& t : threads_) t.join();
    }
    std::future<long long> submit(std::function<long long()> f) {
        std::packaged_task<long long()> task(std::move(f));
        std::future<long long> result = task.get_future();
        { std::lock_guard<std::mutex> lock(m_); tasks_.push_back(std::move(ta
        cv_.notify_one();
        return result;
    }
private:
    void run() {
        while (true) {
            std::packaged_task<long long()> task;
            {
                std::unique_lock<std::mutex> lock(m_);
                cv_.wait(lock, [this] { return stopping_ || !tasks_.empty(); }
                if (tasks_.empty()) return;            // stopping and draine
                task = std::move(tasks_.front());
                tasks_.pop_front();
            }
            task();                                    // outside the lock
        }
    }
    std::vector<std::thread> threads_;
    std::deque<std::packaged_task<long long()>> tasks_;   // guarded by m_
    std::mutex m_;
    std::condition_variable cv_;
```

```
    bool stopping_ = false;
};
```

Three details carry the design. The task runs *outside* the lock, so workers overlap. The destructor drains the queue before the workers exit — `return` only when `tasks_` is empty — because a task thrown away would leave its future with a `broken_promise`. And the futures come back in submission order, so `for (auto& f : results) f.get()` yields results in the order the tasks were submitted, whatever order the workers finished them. Real pools add a generic `submit`, priorities and per-thread queues; the skeleton is what interviews ask for.

Work splitting and reduction

A parallel loop has two halves. **Split**: give each thread a contiguous chunk — `[i * n / t, (i + 1) * n / t)` — rather than every `t`-th element, because a contiguous chunk walks memory in order and stays in one cache line at a time. Static chunks are right when every element costs the same; when costs vary (primality tests, path searches) a dynamic split hands out the next index or the next block through `next.fetch_add(1)`, so a fast thread takes more. **Reduce**: each thread accumulates into a local; after the join, the main thread combines the locals in index order. For integers, addition, counting, minimum and maximum are associative and commutative, so the result is the same whichever thread got which chunk. Floating-point addition is not, which is why a parallel `double` sum is reduced in a fixed order on the main thread and never through a shared atomic.

Immutable data and message passing

Every lock in this module guards *mutation*. Data that no thread writes needs no lock: a `const std::vector<int>&` read by eight threads is safe, and so is a lookup table built before the first thread starts. The design move is to make as much as possible immutable while threads run — build the input, freeze it, hand out slices — and to give each thread exclusive ownership of its output. When threads must talk, prefer passing messages through a queue to sharing a structure: a producer moves an item into a `BoundedQueue`, the consumer moves it out, and at no moment do two threads hold the same object. That is the discipline Go's slogan describes — share memory by communicating — and it is available in C++ with `std::move` and a queue.

Deadlock avoidance rules

Prefer no shared mutable state; then no lock is needed.

Hold one lock at a time; if two, `std::scoped_lock` or one fixed global order.

Never call unknown code — a callback, a virtual function, a user-supplied comparator — while holding a lock.

Keep critical sections to the update itself; never block inside one (no I/O, no `join`, no waiting on another thread).

Every waiter has a way out: a stop flag with `notify_all`, a closed queue, a timeout.

Join every thread in a known place; a thread nobody joins is a thread that can still be running when its data is destroyed.

Threads or async? The interview answer

`std::thread` (or `std::jthread`) is a *resource*: you own it, join it, and communicate through memory you arrange yourself. `std::async` with a future is a *result*: fire a function, collect its value or its exception, no join to write. Use threads for long-lived workers with a lifecycle — a pool, a consumer loop, a server — and `std::async` for a fixed set of independent computations whose answers you want back. In both cases the safe shape is the same: split, compute privately, reduce on one thread. The follow-up questions are about primitives, and the one-line answers are: a data race is two unordered accesses with a write and is undefined behaviour; a mutex protects a compound update, an atomic a single one; a condition variable waits for a state change without spinning; `join` synchronises, so after it the main thread sees everything the worker wrote.

The deterministic-output rule for judged programs

The judge runs your program once per case and compares stdout byte for byte, so anything that varies with scheduling fails the case. The checklist:

Threads write into their own slots or return through futures; nothing prints until every thread has been joined.

Print in index order, or in submission order for futures — never in completion order.

Print aggregates that do not depend on interleaving: sums, counts, maxima, sorted merges.

Never print `hardware_concurrency()`, `std::this_thread::get_id()`, a duration, a queue's high-water mark or "which thread did it".

Reduce floating point on the main thread in a fixed order.

Every worker must exit: close queues, set stop flags, `notify_all`, join.

Pitfalls

Mistake	What happens
Pool destructor that stops without draining	Pending futures throw `broken_promise`
Running a task while holding the queue's lock	One worker at a time — a serial program with extra steps
Interleaved split (`i, i + t, i + 2t, …`) for a numeric loop	Cache lines shared between threads; slower than contiguous chunks
Reducing `double`s through a shared atomic	Different last bits on different runs
Printing results in completion order	Output depends on the scheduler
A pool with more CPU-bound workers than cores	No speed-up, more context switches

Key takeaways

A thread pool is a queue, a condition variable, `w` workers and `std::packaged_task`; futures return in submission order.

Split into contiguous chunks (or hand out indices with `fetch_add`), compute into locals, reduce on the main thread in a fixed order.

Immutable input needs no lock; give each thread its own output; pass messages instead of sharing structures.

Deadlock is avoided by discipline: one lock at a time, fixed order, no unknown code under a lock, every waiter has an exit.

Threads for long-lived workers with a lifecycle; `std::async` for independent results with an error path.

Judged output: join everything, print in index order, aggregates only, nothing that depends on scheduling.

EXERCISE 1

A thread pool

Complete the lesson's `ThreadPool`. `submit(std::function<long long()>)` wraps the function in a `std::packaged_task<long long()>`, takes its future, pushes the task onto the queue under the mutex, notifies one worker and returns the future. `run()` — the body of every worker — loops: wait under a `std::unique_lock` until stopping or a task is available; return only when stopping **and** the queue is drained (a task thrown away would break its future's promise); otherwise move the front task out, **unlock**, and run it outside the lock so workers overlap. The constructor and destructor are given.

Read `w` (workers, 1–4) and `n`, then `n` integers `k`. Submit one task per `k` computing how many primes are `≤ k` (`count_primes` is given), keep the futures in a vector, and print `task <i>: <count>` by calling `get()` in **submission order** — deterministic whatever order the workers finished. End with `done <n>`.

Input: `w n`, then `n` integers (each at most 100000). **Output:** `n + 1` lines.

```
2 4
10 100 1 30
```

→

```
task 0: 4
task 1: 25
task 2: 0
task 3: 10
done 4
```

INPUT 1

```
2 4
10 100 1 30
```

OUTPUT 1

```
task 0: 4
task 1: 25
task 2: 0
task 3: 10
done 4
```

INPUT 2

```
1 2
2 1000
```

OUTPUT 2

```
task 0: 1
task 1: 168
done 2
```

REFERENCE SOLUTION

```cpp
#include <condition_variable>
#include <deque>
#include <functional>
#include <future>
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

class ThreadPool {
public:
    explicit ThreadPool(int workers) {
        for (int i = 0; i < workers; ++i) threads_.emplace_back([this] { run(); });
    }
    ~ThreadPool() {
        {
            std::lock_guard<std::mutex> lock(m_);
            stopping_ = true;
        }
        cv_.notify_all();
        for (auto& t : threads_) t.join();
    }
    std::future<long long> submit(std::function<long long()> f) {
        std::packaged_task<long long()> task(std::move(f));
        std::future<long long> result = task.get_future();
        {
            std::lock_guard<std::mutex> lock(m_);
            tasks_.push_back(std::move(task));
        }
        cv_.notify_one();
        return result;
    }
private:
    void run() {
        while (true) {
            std::packaged_task<long long()> task;
            {
                std::unique_lock<std::mutex> lock(m_);
                cv_.wait(lock, [this] { return stopping_ || !tasks_.empty(); });
                if (tasks_.empty()) return;             // stopping and drained
                task = std::move(tasks_.front());
                tasks_.pop_front();
            }
            task();                                     // outside the lock
        }
    }
    std::vector<std::thread> threads_;
    std::deque<std::packaged_task<long long()>> tasks_;   // guarded by m_
    std::mutex m_;
```

```
    std::condition_variable cv_;
    bool stopping_ = false;
};

long long count_primes(int k) {
    long long count = 0;
    for (int x = 2; x <= k; ++x) {
        bool prime = true;
        for (int d = 2; d * d <= x; ++d) {
            if (x % d == 0) { prime = false; break; }
        }
        if (prime) ++count;
    }
    return count;
}

int main() {
    int w, n;
    std::cin >> w >> n;
    std::vector<std::future<long long>> results;
    {
        ThreadPool pool(w);
        for (int i = 0; i < n; ++i) {
            int k;
            std::cin >> k;
            results.push_back(pool.submit([k] { return count_primes(k); }));
        }
        for (int i = 0; i < n; ++i) std::cout << "task " << i << ": " << results[i].
    }   // the pool stops and joins here
    std::cout << "done " << n << '\n';
    return 0;
}
```

EXERCISE 2

Row sums over an immutable grid

Read `t`, `R` and `C`, then an `R × C` grid of integers into a `std::vector<std::vector<long long>>` that is **never written once the threads start** — pass it to them as `const`. Split the rows into `t` contiguous chunks. Each thread writes the sum of each of its rows into its own slots of a pre-sized `rowSums` vector, and adds its chunk's total into a `std::atomic<long long> grand` with **one** `fetch_add` after its loop. After joining, print `row <r>: <sum>` for every row, `total <grand>`, and `widest row <r>` for the row with the largest sum (the first on a tie), found on the main thread.

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

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

→

```
row 0: 6
row 1: 15
row 2: -6
total 15
widest row 1
```

INPUT 1

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

OUTPUT 1

```
row 0: 6
row 1: 15
row 2: -6
total 15
widest row 1
```

INPUT 2

```
4 1 2
5 5
```

OUTPUT 2

```
row 0: 10
total 10
widest row 0
```

REFERENCE SOLUTION

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

int main() {
    std::size_t t, R, C;
    std::cin >> t >> R >> C;
    std::vector<std::vector<long long>> grid(R, std::vector<long long>(C));
    for (auto& row : grid) for (auto& x : row) std::cin >> x;
    const std::vector<std::vector<long long>>& input = grid;   // frozen from here o

    std::vector<long long> rowSums(R, 0);
    std::atomic<long long> grand{0};
    std::vector<std::thread> workers;
    for (std::size_t i = 0; i < t; ++i) {
        const std::size_t lo = i * R / t, hi = (i + 1) * R / t;
        workers.emplace_back([&input, &rowSums, &grand, lo, hi] {
            long long chunkTotal = 0;
            for (std::size_t r = lo; r < hi; ++r) {
                long long s = 0;
                for (long long x : input[r]) s += x;
                rowSums[r] = s;
                chunkTotal += s;
            }
            grand.fetch_add(chunkTotal);
        });
    }
    for (auto& w : workers) w.join();

    std::size_t widest = 0;
    for (std::size_t r = 0; r < R; ++r) {
        std::cout << "row " << r << ": " << rowSums[r] << '\n';
        if (rowSums[r] > rowSums[widest]) widest = r;
    }
    std::cout << "total " << grand.load() << '\n';
    std::cout << "widest row " << widest << '\n';
    return 0;
}
```

QUIZ

1. A thread pool's destructor sets `stopping_`, notifies, and its workers return as soon as they see the flag, even with tasks still queued. What do the callers holding those tasks' futures see?

A. The results, computed by the destructor

B. `std::future_error` with `broken_promise` from `get()`, because the packaged tasks were destroyed uncalled

C. Default values

D. `get()` blocks forever

2. A pool's worker runs each task while still holding the queue's mutex. What is the effect?

A. Tasks run faster because there is no contention

B. Only one task runs at a time — a serial program with extra threads

C. Tasks may run twice

D. The queue becomes corrupted

3. Why is the output of `for (auto& f : results) std::cout << f.get()` deterministic even though the pool's workers finish tasks in any order?

A. Because the pool has only one worker

B. Because the futures are stored in submission order and each `get()` waits for its own task, so the merge order is fixed by the loop, not the scheduler

C. Because `std::cout` sorts its output

D. It is not deterministic

4. Why are contiguous chunks (`[i * n / t, (i + 1) * n / t)`) preferred over an interleaved split (`i, i + t, i + 2t, ...`) for a numeric loop?

A. Interleaving does not compile

B. A contiguous chunk walks memory in order and keeps each thread in its own cache lines; interleaving makes threads share lines and slows them down

C. Contiguous chunks are always equal in size

D. Interleaved splits produce wrong sums

5. Four threads each compute a partial sum of `double`s. How should the total be formed for a judged program?

A. Each thread `fetch_add`s its partial into a `std::atomic<double>`

B. Each thread stores its partial in its own slot; the main thread adds the slots in index order after joining

C. Each thread prints its partial

D. Use `float` to avoid the problem

6. Interview question: when would you choose `std::async` over `std::thread`?

A. Never; `std::async` is slower

B. For a fixed set of independent computations whose values (or exceptions) you want back — no join to write, results through futures

C. For a long-lived worker with a lifecycle, such as a consumer loop

D. Only when there is exactly one core

7. Which rule prevents the deadlock where a callback invoked under your lock tries to lock something of its own?

A. Use `std::recursive_mutex` everywhere

B. Never call code you do not control — a callback, a virtual function, a user comparator — while holding a lock

C. Lock in alphabetical order

D. Use `notify_all` before calling back

ANSWER KEY · CONCURRENCY PATTERNS — POOLS, REDUCTIONS AND THE DETERMINISTIC-OUTPUT RULE

1. B A destroyed `packaged_task` that was never invoked breaks its promise. The worker loop must return only when stopping *and* the queue is empty.

2. B The lock serialises everything inside it. Move the task out under the lock, release it, then call the task so other workers can take the next one.

3. B Completion order is scheduling-dependent; consumption order is the program's. Printing in index or submission order is the deterministic-output rule in practice.

4. B Both give correct results; the difference is cache behaviour. Adjacent elements sit in the same 64-byte line, and two cores writing the same line bounce it between them.

5. B Floating-point addition is not associative, so a shared accumulator's result depends on arrival order. A fixed-order reduction on one thread gives identical bits on every run.

6. B `std::async` is a *result*; a thread is a *resource*. Pools, consumer loops and servers own threads; a batch of computations is collected from futures.

7. B Unknown code may lock in any order and create the waiting cycle you cannot see. Copy what you need under the lock, release it, then call out.

CHECKPOINT · 25 MIN

Checkpoint — Concurrency

This checkpoint covers the whole module: starting and joining threads, data races and the RAII locks, atomics and compare-exchange, condition variables and the bounded queue, futures, promises and `std::async`, and the patterns — a thread pool, split-and-reduce, message passing — that keep concurrent programs correct and their output deterministic.

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

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

- What happens when a joinable `std::thread` is destroyed, and how does `std::jthread` differ?
- Why is an unsynchronised `++counter` from two threads undefined behaviour rather than merely inaccurate?
- When does an atomic replace a mutex, and when does it not?
- What does `compare_exchange_weak` do to `expected` when it fails, and why must it sit in a loop?
- Why must `cv.wait` be given a predicate, and what is a lost wake-up?
- What does `std::future::get()` do with an exception thrown in the task?
- Why does a judged program print only after every thread has been joined?

The three programs are a prime census that splits a range across threads and counts with an atomic, a batch of divisions run through `std::async` where some throw, and a two-stage pipeline of bounded queues whose prefix sums must come out in order. Join everything before you print.

EXERCISE 1

A prime census

Read `t`, `lo` and `hi` and count the primes in the inclusive range `[lo, hi]` with `t` threads. Split the range into contiguous chunks with the chunk formula over `n = hi - lo + 1` numbers (chunk `i` covers `[lo + i * n / t, lo + (i + 1) * n / t)`). Each thread counts the primes in its chunk in a **local**, stores that partial in its own slot, and adds it into a `std::atomic<long long> total` with **one**

`fetch_add`. After joining, print `chunk <i>: [<a>, <b>) primes <p>` for every chunk in index order, then `total <count>` read from the atomic. `is_prime` is given; `lo` is at least 1.

Input: `t lo hi`. **Output:** `t + 1` lines.

```
2 1 20
```

→

```
chunk 0: [1, 11) primes 4
chunk 1: [11, 21) primes 4
total 8
```

INPUT 1

```
2 1 20
```

OUTPUT 1

```
chunk 0: [1, 11) primes 4
chunk 1: [11, 21) primes 4
total 8
```

INPUT 2

```
3 90 100
```

OUTPUT 2

```
chunk 0: [90, 93) primes 0
chunk 1: [93, 97) primes 0
chunk 2: [97, 101) primes 1
total 1
```

REFERENCE SOLUTION

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

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

int main() {
    long long t, lo, hi;
    std::cin >> t >> lo >> hi;
    const long long n = hi - lo + 1;
    std::vector<long long> partial(t, 0);
    std::atomic<long long> total{0};
    std::vector<std::thread> workers;
    for (long long i = 0; i < t; ++i) {
        const long long a = lo + i * n / t, b = lo + (i + 1) * n / t;
        workers.emplace_back([&partial, &total, i, a, b] {
            long long count = 0;
            for (long long x = a; x < b; ++x) {
                if (is_prime(x)) ++count;
            }
            partial[i] = count;
            total.fetch_add(count);
        });
    }
    for (auto& w : workers) w.join();
    for (long long i = 0; i < t; ++i) {
        const long long a = lo + i * n / t, b = lo + (i + 1) * n / t;
        std::cout << "chunk " << i << ": [" << a << ", " << b << ") primes " << part
    }
    std::cout << "total " << total.load() << '\n';
    return 0;
}
```

EXERCISE 2

Divisions through std::async

Read `n` pairs `a b`. Launch every division as its own `std::async(std::launch::async, ...)` task computing `a / b` (integer division) and throwing `std::domain_error("division by zero")` when

`b` is 0. Keep the futures in a vector, then `get()` them in order inside a `try`: print `task <i>: <quotient>` or `task <i>: error division by zero` — the exception crossed the thread boundary through the future and is caught by `const std::domain_error&`. Finish with `ok <k> failed <m>`.

Input: `n`, then `n` lines `a b`. **Output:** `n + 1` lines.

```
4
10 2
7 0
-9 4
0 5
```

→

```
task 0: 5
task 1: error division by zero
task 2: -2
task 3: 0
ok 3 failed 1
```

INPUT 1

```
4
10 2
7 0
-9 4
0 5
```

OUTPUT 1

```
task 0: 5
task 1: error division by zero
task 2: -2
task 3: 0
ok 3 failed 1
```

INPUT 2

```
1
1 0
```

OUTPUT 2

```
task 0: error division by zero
ok 0 failed 1
```

REFERENCE SOLUTION

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

long long divide(long long a, long long b) {
    if (b == 0) throw std::domain_error("division by zero");
    return a / b;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<std::future<long long>> tasks;
    for (int i = 0; i < n; ++i) {
        long long a, b;
        std::cin >> a >> b;
        tasks.push_back(std::async(std::launch::async, divide, a, b));
    }
    int ok = 0, failed = 0;
    for (int i = 0; i < n; ++i) {
        try {
            long long q = tasks[i].get();
            std::cout << "task " << i << ": " << q << '\n';
            ++ok;
        } catch (const std::domain_error& e) {
            std::cout << "task " << i << ": error " << e.what() << '\n';
            ++failed;
        }
    }
    std::cout << "ok " << ok << " failed " << failed << '\n';
    return 0;
}
```

EXERCISE 3

A two-stage pipeline

`BoundedQueue` is given, complete, from lesson 4. Build a pipeline of three stages over two queues of capacity 2. **Stage 1** (a thread) pushes the `n` input values into queue `A` in order, then closes `A`. **Stage 2** (a thread) pops from `A` until `nullopt`, keeps a running sum, pushes each prefix sum into queue `B`, and closes `B` when `A` is drained — closing must propagate, or stage 3 waits forever. **Stage 3** (the main thread) pops from `B` into a vector until `nullopt`. Join both threads, then print `prefix <the sums, space-separated>` and `count <n>`. Every queue is FIFO with one consumer, so the prefix sums come out in input order on every run.

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

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

→

```
prefix 3 4 8 9 14
count 5
```

INPUT 1

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

OUTPUT 1

```
prefix 3 4 8 9 14
count 5
```

INPUT 2

```
1
-7
```

OUTPUT 2

```
prefix -7
count 1
```

REFERENCE SOLUTION

```cpp
#include <condition_variable>
#include <iostream>
#include <mutex>
#include <optional>
#include <queue>
#include <thread>
#include <vector>

class BoundedQueue {
public:
    explicit BoundedQueue(std::size_t capacity) : capacity_(capacity) {}
    void push(long long v) {
        std::unique_lock<std::mutex> lock(m_);
        notFull_.wait(lock, [this] { return items_.size() < capacity_; });
        items_.push(v);
        lock.unlock();
        notEmpty_.notify_one();
    }
    std::optional<long long> pop() {
        std::unique_lock<std::mutex> lock(m_);
        notEmpty_.wait(lock, [this] { return !items_.empty() || closed_; });
        if (items_.empty()) return std::nullopt;
        long long v = items_.front();
        items_.pop();
        lock.unlock();
        notFull_.notify_one();
        return v;
    }
    void close() {
        {
            std::lock_guard<std::mutex> lock(m_);
            closed_ = true;
        }
        notEmpty_.notify_all();
    }
private:
    std::mutex m_;
    std::condition_variable notFull_;
    std::condition_variable notEmpty_;
    std::queue<long long> items_;
    std::size_t capacity_;
    bool closed_ = false;
};

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

```
    BoundedQueue a(2), b(2);
    std::thread stage1([&a, &values] {
        for (long long v : values) a.push(v);
        a.close();
    });
    std::thread stage2([&a, &b] {
        long long running = 0;
        while (auto item = a.pop()) {
            running += *item;
            b.push(running);
        }
        b.close();
    });
    std::vector<long long> prefix;
    while (auto item = b.pop()) prefix.push_back(*item);
    stage1.join();
    stage2.join();

    std::cout << "prefix";
    for (long long p : prefix) std::cout << ' ' << p;
    std::cout << '\n';
    std::cout << "count " << prefix.size() << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What happens when a joinable `std::thread` is destroyed, and how does `std::jthread` differ?

A. Both join silently

B. `std::thread` calls `std::terminate`; `std::jthread` requests a stop and joins in its destructor

C. `std::thread` detaches; `std::jthread` terminates

D. Both throw

2. Why is an unsynchronised `++counter` from two threads undefined behaviour rather than merely inaccurate?

A. Because `++` is not atomic on any hardware

B. Because the standard defines two unordered accesses with a write as a data race, and a program with one has no defined meaning — the compiler may assume it cannot happen

C. Because `int` is too small

D. It is not undefined; it is only inaccurate

3. When does an atomic replace a mutex, and when does it not?

A. Always; atomics are faster

B. One variable, one operation: atomic. A compound update or an invariant across variables: mutex

C. Never; atomics are only for flags

D. Atomics replace mutexes only on x86-64

4. `compare_exchange_weak(expected, desired)` fails. What happened to `expected`, and why must the call sit in a loop?

A. `expected` is unchanged; the loop retries the same value

B. `expected` now holds the atomic's current value; the loop decides again from it, and also absorbs the spurious failures the weak form permits

C. `expected` was set to `desired`

D. No loop is needed with the weak form

5. Why must `cv.wait` be given a predicate, and what is a lost wake-up?

A. The predicate is optional; a lost wake-up is a notify that arrived twice

B. Spurious wake-ups make an unpredicated wait proceed on a false condition; a lost wake-up is a notify sent before the waiter waits, which is why the state changes under the mutex

C. The predicate speeds up the wait; a lost wake-up is a crash

D. The predicate replaces the mutex

6. What does `std::future::get()` do with an exception thrown in the task?

A. Swallows it and returns a default value

B. Rethrows it on the calling thread, with its dynamic type preserved

C. Calls `std::terminate`

D. Returns it as an error code

7. Why does a judged program print only after every thread has been joined?

A. Because `std::cout` cannot be used while threads exist

B. Because output written before or during the workers' run would depend on scheduling, and the join is also what makes the workers' writes visible to the main thread

C. Because the judge kills threads that print

D. Because printing is slow

8.

```
for (int i = 0; i < 4; ++i)
    std::async(std::launch::async, work, i);
```

What is the problem?

A. `work` runs four times in parallel with no way to collect results

B. Each temporary future's destructor blocks until its task finishes, so the four tasks run one after another

C. The loop starts too many threads

D. Nothing

9. A `std::vector<std::thread>` of eight workers is created, and a `std::vector<long long> partial` is `push_back`ed to from inside each worker. What is wrong?

A. Nothing; `push_back` is thread-safe

B. Concurrent `push_back`s race on the vector's size and may reallocate its buffer under another thread — undefined behaviour; size the vector before the threads start and write to `partial[i]`

C. Eight workers is too many

D. `long long` cannot be shared

10. `std::lock_guard<std::mutex>{m};` appears at the top of a critical section. What does it protect?

A. The whole block

B. Nothing — the unnamed temporary is destroyed at the semicolon and the section runs unlocked

C. Only the next statement

D. The mutex itself

11. A thread pool's `run()` loop returns as soon as `stopping_` is true, without checking the queue. What is the consequence?

A. Nothing; the destructor runs the remaining tasks

B. Queued tasks are destroyed uncalled and their futures throw `broken_promise`

C. Workers keep running after the destructor

D. The mutex is never unlocked

12. Which of these outputs may a judged concurrent program print?

A. `std::this_thread::get_id()` of the thread that finished first

B. `std::thread::hardware_concurrency()`

C. The sum of per-thread partials added in index order on the main thread

D. The elapsed time of the parallel section

13. Thread 1 locks `a.m` then `b.m`; thread 2 locks `b.m` then `a.m`. What can happen, and what is the fix?

A. Nothing can go wrong; mutexes are reentrant

B. Each may hold one and wait for the other forever — deadlock; take both with `std::scoped_lock`, or lock in one fixed global order everywhere

C. A data race; use atomics

D. One thread is terminated

14. In a bounded producer–consumer program, why is `close()` implemented as "set the flag under the mutex, then `notify_all`" rather than `notify_one`?

A. `notify_one` is slower

B. Shutdown concerns every consumer; with `notify_one` only one wakes and the others sleep forever, and the program never exits

C. `notify_all` also empties the queue

D. `notify_one` cannot be called after `close()`

ANSWER KEY · CHECKPOINT — CONCURRENCY

1. B The plain type refuses to guess and aborts the program; `std::jthread` was added so that scope exit, early return and exceptions all wait for the worker.

2. B Losing increments is the visible symptom; the rule is stronger. The optimiser may keep the counter in a register for the whole loop, so a thread never sees the other's writes at all.

3. B An atomic makes a single operation indivisible. A check-then-act, or two variables that must agree, needs a critical section.

4. B On failure the current value is written into `expected`, and the weak form may fail even when the values matched, so a loop is the only correct shape; `compare_exchange_strong` never fails spuriously.

5. B The predicate form re-checks the state after every wake-up and, tested under the mutex, cannot miss a change that happened before the wait began.

6. B The library stores the exception in the shared state; `get()` rethrows it, so a `catch (const std::domain_error&)` at the `get()` matches exactly that type.

7. B Byte-for-byte comparison tolerates no interleaving. Threads compute into slots; the main thread joins them, then prints in a fixed order.

8. B Futures from `std::async` block in their destructors. Keep them in a vector to let the tasks overlap.

9. B A container is not thread-safe for concurrent modification. Pre-sized slots give each thread its own memory and remove the need for a lock.

10. B The guard must be a named local so its lifetime spans the block: `std::lock_guard<std::mutex> lock(m);`.

11. B The worker must return only when stopping *and* the queue is empty; a packaged task destroyed without being invoked breaks its promise.

12. C Ids, core counts and durations vary by machine and run; a fixed-order reduction of integer partials does not.

13. B A cycle of waiting is a deadlock, and the judge reports it as a timeout. `std::scoped_lock(a.m, b.m)` acquires both without the cycle.

14. B A change that every waiter must see needs `notify_all`. Each woken consumer re-checks the predicate, drains what remains, and returns `nullopt` when the queue is empty and closed.
