Module 18 of 20

Streams and files

The stream state model and recovering from bad input, the reading-pattern catalogue with validation, file streams and open modes, string streams with std::quoted and std::format, binary records and std::filesystem.

Every program in this track reads standard input and writes standard output, and most of the failed submissions are not logic errors but reading errors: a count followed by a `getline` that returns an empty line, a loop on `eof()` that runs once too often, a bad token that silently sticks the stream so every later read is a no-op. C++ streams are precise about all of this — four state bits, a buffer, a character-level API — and a programmer who knows the model can read any input format, recover from malformed data and say exactly which line was wrong. The same model then extends, unchanged, to files and to strings in memory, because `std::ifstream` and `std::istringstream` are the same formatter over a different buffer.

The five lessons go from the console outward. Streams in depth opens the stream: `good` / `fail` / `eof` / `bad`, what a failed `>>` leaves in the variable and in the buffer, `clear()` and `ignore()`, and `peek()` / `get()` for a tokeniser that sees every character. Reading structured input is the catalogue — N then N tokens, N then N lines, tokens until EOF, a matrix with its dimensions checked, records with a variable field count — each with a template and its validation. File streams covers `std::ofstream` / `ifstream` / `fstream`, the open modes that truncate or append, RAII closing and why the writer must finish before the reader opens. String streams and formatting adds reuse without the state bug, `std::quoted` for text with spaces, and C++20 `std::format` / `format_to` for reports and tables. Binary I/O and the filesystem finishes with `write` / `read` of fixed-width records, padding and endianness, and `std::filesystem` for paths, sizes, directories and sorted listings.

The exercises are whole programs against the study judge, which lets a program create, write and read files in its working directory: a recovery loop that skips bad tokens and reports them, a `peek` / `ignore` tokeniser, the N-lines reader and a matrix reader that validates every row, a text file written and read back with counts, a ledger appended across two opens, an `ostringstream` report inspected before printing, a `std::quoted` round trip, a fixed-width binary record file indexed with `seekg`, and a directory created, listed in sorted order with sizes, and removed. The checkpoint adds a variable-field records reader, a `std::format` table with dynamic widths, and an inventory file written, measured, read back through a `std::map` and deleted.

CONTENTS

1	Streams in depth — state, recovery and the character API	15 min
2	Reading structured input — the pattern catalogue	15 min
3	File streams — open modes, RAII and reading back	14 min
4	String streams and formatting — building text in memory	14 min
5	Binary I/O and the filesystem	15 min
✓	Checkpoint — Streams and files	25 min

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

LESSON 1 · 15 MIN

Streams in depth — state, recovery and the character API

Module 1 taught the two reading habits — `>>` for tokens, `std::getline` for lines — and promised that every input in this track is well-formed. Real input is not. A stream that meets `abc` where it expected a number does not throw and does not skip: it sets a flag, leaves the characters where they were, and silently refuses every later read until you reset it. Programs that "stop reading half way for no reason" are almost always this. This lesson opens the stream up: the buffer, the state bits, exactly what a failed `>>` leaves behind, how to recover, and the character-level calls — `peek`, `get`, `ignore` — that let you write a tokeniser instead of hoping `>>` splits the input the way you need.

A stream is a formatter over a buffer

`std::cin` is a `std::istream`; `std::cout` is a `std::ostream`. Neither owns a file or a terminal. Each holds a pointer to a `std::streambuf`, the object that moves bytes to and from the operating system in chunks, and the stream's own job is *formatting*: turning `42` into the characters `4` `2` on the way out and back into an `int` on the way in. That split is why `std::ifstream` (lesson 3) and `std::istringstream` (lesson 4) have exactly the same interface as `std::cin` — the same formatter over a different buffer — so everything here works on all three. The buffer is also why output appears late: `std::cout << x` writes into memory, and the bytes leave when the buffer fills, on a flush (`std::flush`, `std::endl`), when the tied `std::cin` reads, or at normal exit. `std::cerr` flushes after every write, so a diagnostic there appears at once while `cout` text is still waiting.

The four state bits

Every stream carries three flag bits; a fourth name, `goodbit`, means none is set.

Bit	Set when	Query
`goodbit`	nothing has gone wrong	`good()`
`eofbit`	a read tried to go past the end of the input	`eof()`
`failbit`	a read or write could not do what was asked (a bad token, a missing file)	`fail()`
`badbit`	the underlying buffer broke (disk error, closed pipe)	`bad()`

The conversions matter more than the queries. `if (stream)` and `while (std::cin >> x)` use `operator bool`, which is `!fail()` — true as long as neither `failbit` nor `badbit` is set, and silent about `eofbit`. `good()` is true only when all three are clear.

`eof()` is the one people misuse. It becomes true when a read *bumps into* the end, not when the last item has been consumed. The input `12` with no trailing newline reads `12` successfully and sets `eofbit` in the same call, because the extraction looked for a thirteenth character to see whether the number continued. The read was fine — `operator bool` still says so. That is why `while (std::cin >> x)` is right and `while (!std::cin.eof())` runs once too often (Module 3, lesson 5).

What a failed `>>` leaves behind

```
#include <iostream>

int main() {
    int x = 42;
    std::cin >> x;                          // input: "abc 5"
    std::cout << x << ' ' << std::cin.fail() << ' ' << std::cin.eof() << '\n';
    int y = 7;
    std::cin >> y;                          // does nothing: the stream is in a j
    std::cout << y << '\n';
    return 0;
}
```

```
0 1 0
7
```

Three things happen. The stream sets `failbit`. The variable is set to `0` — required since C++11 whenever characters were examined and could not form a value; when the stream was already at the end before the read began, the conversion never runs and the variable keeps its old value, so never rely on the value after a failed read either way. And the characters `abc` are **not consumed**: they are still the next thing in the buffer. Every later `>>` sees `failbit`, gives up without touching the buffer, and leaves its variable alone — `y` stays `7`. The stream is *stuck* until something clears the state. An out-of-range number is the one variation: `99999999999` into an `int` sets `failbit`, stores the largest representable value (`2147483647`) and *does* consume the digits.

Recovering

```
std::cin.clear();                                                    // res
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');  // dis
```

`clear()` sets the state to `goodbit` (with an argument — `clear(std::ios::failbit)` — it *sets* that bit; the no-argument form is the reset). It does nothing to the buffer, so the offending characters are still waiting, and the second step decides what to do with them. `ignore(n, delim)` discards up to `n` characters or through the first `delim`, whichever comes first; with `std::numeric_limits<std::streamsize>::max()` (`<limits>`) as `n` it means "the rest of the line". `ignore()` with no arguments discards exactly one character — the form Module 1 used to drop the newline after `std::cin >> n`. The third option is to extract the bad token as a string and count it:

```
long long sum = 0;
int rejected = 0;
for (;;) {
    int x;
    if (std::cin >> x) { sum += x; continue; }
    if (std::cin.eof()) break;              // nothing left: a clean stop
    std::cin.clear();                       // a bad token: reset the state...
    std::string junk;
    std::cin >> junk;                       // ...consume the token that caused
    ++rejected;                             // ...and carry on
}
```

The order of the two tests is the whole trick. `eof()` is checked *before* clearing, because `clear()` would erase the evidence; a read that failed without reaching the end is, by

elimination, a bad token. `std::cin >> junk` cannot fail here — the bad characters are non-whitespace, so a string extraction takes them — and the loop resumes on the next token. Note what "token" means to this loop: `12abc` reads as the number `12` followed by the bad token `abc`, because `>>` stops at the first character that cannot continue the number. A reader that wants the whole word rejected reads a string first and converts it (Module 5, lesson 3), or checks `peek()` after the number.

The stricter alternative, `std::cin.exceptions(std::ios::failbit)`, makes the stream throw `std::ios_base::failure` the moment the bit is set — right when malformed input is fatal, wrong for a read-until-EOF loop, where the end of input sets `failbit` too.

The character-level API

`>>` works in tokens and skips whitespace first. When the grammar is finer than "whitespace-separated" — an expression like `x1=42+foo*(7-y)` where `*` and `(` are tokens of their own — you read one character at a time:

Call	Effect
`int c = in.peek();`	Returns the next character **without** consuming it, or `EOF` (`-1`) at the end
`int c = in.get();`	Consumes and returns the next character, or `EOF`; sets `eofbit` and `failbit` at the end
`in.unget();`	Steps back one character
`in >> std::ws;`	Skips whitespace and nothing else

`peek()` and `get()` return `int`, not `char`, precisely so that `EOF` can be told apart from every real character; compare against `EOF` before converting. Because whitespace is *not* skipped, these calls see the newlines and spaces that `>>` hides — which is what a tokeniser wants, and what makes `std::getline(std::cin >> std::ws, line)` the tidy way to skip blank lines before reading one. A tokeniser built on them has one shape: peek at the next character, decide what kind of token starts there, then consume exactly that token.

```
int c;
while ((c = std::cin.peek()) ≠ EOF) {
    if (std::isspace(c)) { std::cin.get(); continue; }                    // sk
    if (c == '#') { std::cin.ignore(std::numeric_limits<std::streamsize>::max(
    if (std::isdigit(c)) {
        std::string digits;
        while (std::isdigit(std::cin.peek())) digits += static_cast<char>(std
        std::cout << "NUMBER " << digits << '\n';
        continue;
    }
    std::cout << "OP " << static_cast<char>(std::cin.get()) << '\n';     // any
}
```

`std::isdigit(std::cin.peek())` is safe because `peek()` returns either `EOF` or a value in the `unsigned char` range — exactly the two things `<cctype>` accepts (Module 5, lesson 3). A name token is the same inner loop with `isalnum`.

Pitfalls

Mistake	What happens
Reading `x` after `std::cin >> x` failed	`0`, or the old value: never meaningful. Test the stream.
`while (!std::cin.eof())`	One extra iteration with a failed read.
Forgetting `clear()` before the next read	Every read is a silent no-op for the rest of the program.
`clear()` without discarding the bad characters	The same token fails again: an infinite loop.
`char c = std::cin.get();` then `c == EOF`	`EOF` is `-1`; keep the result in an `int`.

Key takeaways

A stream is a formatter over a `streambuf`; `cin`, an `ifstream` and an `istringstream` share one interface and one state model.

`operator bool` is `!fail()`; `eof()` means a read bumped the end, which the last successful read also does.

A failed `>>` sets `failbit`, zeroes (or leaves) the variable, consumes nothing, and sticks until `clear()`.

Recover with `clear()` and then discard the bad characters — `ignore(max, '\n')` or an extraction into a string; test `eof()` before clearing.

`peek()`, `get()` and `ignore()` see every character, whitespace included, and return `int` so `EOF` is distinguishable.

EXERCISE 1

Skip the bad tokens

Read whitespace-separated tokens from standard input until it ends. Every token is either an integer (an optional sign and digits, within `int`) or a word made only of letters — nothing like `12abc` or `3.5` appears. Read each token with `std::cin >> x` into an `int`. When the read fails and the stream is **not** at the end of the input, it met a word: `clear()` the stream state, extract the offending token into a `std::string`, and remember it. Test `eof()` before you clear, or you cannot tell the two failures apart.

Input: any number of tokens, on any number of lines. **Output:** `accepted=<count> sum=<sum of the integers>`, then `rejected=<count>: <the words in order, space-separated>` — or just `rejected=0` when every token was a number. Keep the sum in a `long long`.

```
10 abc 20 x9 -5
```

prints

```
accepted=3 sum=25
rejected=2: abc x9
```

INPUT 1

```
10 abc 20 x9 -5
```

OUTPUT 1

```
accepted=3 sum=25
rejected=2: abc x9
```

INPUT 2

```
1 2 3
```

OUTPUT 2

```
accepted=3 sum=6
rejected=0
```

REFERENCE SOLUTION

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

int main() {
    long long sum = 0;
    int accepted = 0;
    std::vector<std::string> rejected;
    for (;;) {
        int x;
        if (std::cin >> x) {
            sum += x;
            ++accepted;
            continue;
        }
        if (std::cin.eof()) break;          // the input is exhausted: a clean stop
        std::cin.clear();                   // a bad token: reset failbit...
        std::string junk;
        std::cin >> junk;                   // ...and consume the token that caused
        rejected.push_back(junk);
    }
    std::cout << "accepted=" << accepted << " sum=" << sum << '\n';
    if (rejected.empty()) {
        std::cout << "rejected=0\n";
    } else {
        std::cout << "rejected=" << rejected.size() << ':';
        for (const auto& token : rejected) std::cout << ' ' << token;
        std::cout << '\n';
    }
    return 0;
}
```

EXERCISE 2

A peek-and-ignore tokeniser

Read all of standard input one character at a time with `std::cin.peek()` and `std::cin.get()` — not with `>>` — and split it into tokens:

- `NUMBER`: a maximal run of digits;
- `IDENT`: a letter or `_` followed by any letters, digits or `_`;
- `OP`: any other non-whitespace character, one character per token;
- a `#` starts a comment that runs to the end of the line — skip it with `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')`;
- whitespace (spaces, newlines) is skipped and separates nothing by itself.

Input: any text, any number of lines. **Output:** one line per token — `NUMBER 42`, `IDENT foo`, `OP +` — in input order, then `tokens=<count>`.

```
x1 = 42 + foo*(7-y) # trailing comment
```

prints

```
IDENT x1
OP =
NUMBER 42
OP +
IDENT foo
OP *
OP (
NUMBER 7
OP -
IDENT y
OP )
tokens=11
```

INPUT 1

```
x1 = 42 + foo*(7-y) # trailing comment
```

OUTPUT 1

```
IDENT x1
OP =
NUMBER 42
OP +
IDENT foo
OP *
OP (
NUMBER 7
OP -
IDENT y
OP )
tokens=11
```

INPUT 2

```
# only a comment
  total_2 ⩾ 100
```

OUTPUT 2

```
IDENT total_2
OP >
OP =
NUMBER 100
tokens=4
```

REFERENCE SOLUTION

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

int main() {
    int tokens = 0;
    int c;
    while ((c = std::cin.peek()) ≠ EOF) {
        if (std::isspace(c)) {
            std::cin.get();
            continue;
        }
        if (c == '#') {
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
            continue;
        }
        if (std::isdigit(c)) {
            std::string digits;
            while (std::isdigit(std::cin.peek())) digits += static_cast<char>(std::c
            std::cout << "NUMBER " << digits << '\n';
        } else if (std::isalpha(c) || c == '_') {
            std::string name;
            while (std::isalnum(std::cin.peek()) || std::cin.peek() == '_') name +=
            std::cout << "IDENT " << name << '\n';
        } else {
            std::cout << "OP " << static_cast<char>(std::cin.get()) << '\n';
        }
        ++tokens;
    }
    std::cout << "tokens=" << tokens << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
std::istringstream in("abc");
int x = 5;
in >> x;
std::cout << x << ' ' << in.fail() << ' ' << in.eof();
```

A. `5 1 0`

B. `0 1 0`

C. `0 1 1`

D. `5 0 1`

2. `while (std::cin >> x)` keeps looping as long as…

A. `eof()` is false

B. neither `failbit` nor `badbit` is set

C. `good()` is true

D. the last character read was not a newline

3. The whole input is `12` with no newline after it. After `std::cin >> x` succeeds, what is the stream's state?

A. `goodbit` only

B. `eofbit` set, `failbit` clear — the read succeeded

C. `failbit` set because the number was not terminated

D. `eofbit` and `failbit` both set

4. A read failed on a bad token. Which sequence gets the stream reading the *next* token?

A. `std::cin.clear();` alone

B. `std::cin.ignore();` alone

C. `std::cin.clear();` then discard the bad characters (an `ignore` or an extraction into a string)

D. `std::cin.eof();` then `std::cin.clear();`

5. Why do `std::cin.get()` and `std::cin.peek()` return `int` rather than `char`?

A. For speed: `int` is the register width

B. So that `EOF` (`-1`) can be told apart from every real character value

C. Because characters are stored as `int` inside the buffer

D. It is a historical accident with no practical effect

6. What does `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');` do?

A. Discards exactly one character

B. Discards the rest of the current line, including the newline

C. Discards everything up to the end of the input

D. Sets `failbit` if the line is longer than the maximum

7. Which standard stream flushes after every write, so its text appears immediately?

A. `std::cout`

B. `std::cerr`

C. `std::clog`

D. All three

ANSWER KEY · STREAMS IN DEPTH — STATE, RECOVERY AND THE CHARACTER API

1. B The characters were examined and could not form an `int`, so the variable is set to `0` and `failbit` is set. Nothing tried to read past the end — `abc` is still unread in the buffer — so `eofbit` stays clear.

2. B The condition uses the stream's `operator bool`, which is `!fail()` — true unless `failbit` or `badbit` is set. `eofbit` alone does not stop it, which is what lets the last token of an input with no trailing newline be processed.

3. B The extraction reads `1`, `2`, then looks for a further digit and bumps into the end, which sets `eofbit`. The value `12` was read successfully, so `failbit` stays clear and `operator bool` is still true.

4. C `clear()` resets the state bits but leaves the buffer untouched, so the same characters would fail again — an infinite loop. `ignore()` alone does nothing while `failbit` is set. Reset, then consume.

5. B A `char` can hold every byte value, leaving no spare value for "no character". Returning `int` lets `-1` mean end of input while `0`–`255` are real characters — so keep the result in an `int` when comparing with `EOF`.

6. B `ignore(n, delim)` discards up to `n` characters or through the first `delim`. With the maximum as `n`, the delimiter is what stops it: the remainder of the line goes, newline included. `ignore()` with no arguments is the one-character form.

7. B `std::cerr` is unit-buffered: every insertion is flushed. `std::cout` and `std::clog` are buffered and flush when the buffer fills, on `std::flush` / `std::endl`, when the tied `std::cin` reads, or at exit.

LESSON 2 · 15 MIN

Reading structured input — the pattern catalogue

Every judged program, log parser and configuration reader starts with the same question: what shape is the input, and which read matches that shape? Most wrong answers on this platform are reading bugs, not logic bugs — a count read with `>>` followed by a `getline` that returns an empty line, a matrix read token by token that never notices a short row, a loop on `eof()` that runs once too often. This lesson is a catalogue. For each shape there is a template you can paste, the state check that makes it robust, and the validation that turns "it crashed somewhere" into "line 7: expected 3 values, got 2".

Choosing the read

Two reads, one rule. `>>` when whitespace is only a separator and the line layout carries no meaning; `std::getline` when spaces are data (a name, a sentence) or when *which line* something is on matters. When a line has internal structure, do both: `getline` the line, then parse it with a `std::istringstream` (lesson 4), so a malformed line can only break its own parse and never the state of `std::cin`. Never mix `>>` and `getline` on the same stream without dealing with the newline `>>` leaves behind.

Shape 1: a count, then N tokens

```
int n;
if (!(std::cin >> n) || n < 0) { std::cout << "bad count\n"; return 0; }
std::vector<long long> a(n);
for (auto& x : a) std::cin >> x;
```

Layout-agnostic: `3 10 20 30` and `3` on its own line followed by three lines read identically. The count is validated before it sizes anything — a negative or missing `n` would otherwise be a huge allocation or an uninitialised loop bound. `long long` unless the prompt bounds the values.

Shape 2: a count, then N lines

```
int n;
std::cin >> n;
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');   // finis
for (int i = 0; i < n; ++i) {
    std::string line;
    std::getline(std::cin, line);
    if (!line.empty() && line.back() == '\r') line.pop_back();       // a Win
    // use line
}
```

The `ignore` is the whole difference between this and a bug: without it the first `getline` returns the empty remainder of the count's line. The `\r` check costs nothing and is the difference between a parser that works on files from every editor and one that mysteriously fails on the last field of every line.

Shape 3: tokens until the input ends

```
long long x;
while (std::cin >> x) { /* every token */ }
```

The loop ends on the first failed read — the end of the input, or a bad token; lesson 1's recovery loop tells the two apart when it matters.

Shape 4: lines until the input ends, each with structure

```
std::string line;
int lineNo = 0;
while (std::getline(std::cin, line)) {
    ++lineNo;
    if (line.empty()) continue;                 // blank lines are not records
    std::istringstream fields(line);
    std::string name;
    int qty;
    double price;
    if (!(fields >> name >> qty >> price)) {
        std::cout << "line " << lineNo << ": malformed\n";
        continue;
    }
    // use the record
}
```

This is the workhorse for records. Each line gets a fresh `istringstream`, so a failure is local: the outer `getline` loop is untouched and simply moves to the next line. The line number is counted here, at the only place that sees every line, and it is what a useful error message names.

Shape 5: a header, then records

A first line that describes the rest — `3 items`, `width=80`, a column list — is Shape 2's count with extra fields, read with `>>` into as many variables as it has, followed by an `ignore` to the end of that line before the records begin. When a record's *last* field can contain spaces (`42 3.5 Ada Lovelace`), read the fixed fields with `>>` and the rest of the line with `std::getline(fields >> std::ws, name)` — the `std::ws` swallows the separator that `>>` stopped on.

Shape 6: a matrix

```
int rows, cols;
std::cin >> rows >> cols;
std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
std::vector<std::vector<long long>> m;
for (int r = 0; r < rows; ++r) {
    std::string line;
    if (!std::getline(std::cin, line)) { std::cout << "expected " << rows << '
    std::istringstream in(line);
    std::vector<long long> row;
    long long v;
    while (in >> v) row.push_back(v);
    if (static_cast<int>(row.size()) != cols) { std::cout << "row " << r + 1 <
    m.push_back(std::move(row));
}
```

Two templates exist and they differ in what they can check. The token-wise version (`for r, for c: std::cin >> m[r][c]`) is shorter and accepts any layout, which also means it cannot notice that row 2 had four numbers and row 3 had two — it just reads twelve tokens. The line-wise version above validates every row against `cols` and reports the first violation with its row number. Use the token version when the prompt guarantees the shape; use the line version when it is the program's job to check it. A row with a non-number stops the inner `while` early, so it reports as too few values, which is the right message.

Shape 7: records with a variable number of fields

```
std::string name;
int k;
while (std::cin >> name >> k) {
    std::vector<int> values(k);
    for (auto& v : values) std::cin >> v;
    // use name, values
}
```

When each record announces its own length, the count leads and the loop follows — the same as Shape 1, repeated. When the length is not announced (`ada 3 5 9` / `bob 7`), the line is the record: Shape 4's `getline` plus an `istringstream` that reads tokens until the line runs out, and `values.size()` *is* the count. The second form is the one that can validate ("at least one value expected") and the one that survives a missing field without swallowing the next record's name as a number.

Validating and reporting

Three rules make an error message worth reading. Name the position — the line number or record index the program counted. Say what was expected and what was found, with the values. And decide the policy once: *stop at the first error* (return after printing it) or *skip and continue* (print, `continue`, and keep a count of bad lines for a summary). A judged program prints its report to `std::cout` because that is what is compared; a real tool sends diagnostics to `std::cerr` so the data output stays clean for the next program in the pipeline. Never trust the count over the data: after a "count then N lines" read, a `getline` that fails before `N` lines arrived is an error to report, not a silent zero. And check the *whole* line was consumed when the format is strict — after reading the expected fields, `if (fields >> extra)` succeeding means there was trailing junk.

Pitfalls

Mistake	What happens
`std::getline` right after `std::cin >> n`	The first "line" is empty. `ignore` the rest of the count's line.
A matrix read token-wise	A short or long row is never noticed; the numbers shift into the wrong cells.
Parsing fields straight from `std::cin`	One bad field sticks the stream; every later record is lost. Parse a line copy.
Ignoring `\r`	The last field of each line carries a stray carriage return and compares unequal.
Trusting `n` over the data	A short input reads garbage into the missing rows; report the shortfall instead.

Key takeaways

- `>>` for tokens when layout is irrelevant; `std::getline` when spaces are data or the line matters; both when a line has structure.

After a count read with `>>`, `ignore(max, '\n')` before the first `getline`; strip a trailing `\r`.

Parse each line in its own `std::istringstream` so a malformed line cannot stick `std::cin`.

A line-wise matrix reader can validate row lengths; a token-wise one cannot.

Error messages name the line, the expectation and the value; decide stop-or-skip once.

EXERCISE 1

N lines with spaces

Read an integer `n`, then `n` lines of text. A line may be empty and may have any amount of whitespace between and around its words. For line `i` (1-based) print `<i>: words=<w> longest=<word>` — `w` is the number of whitespace-separated words and `longest` the first word of maximal length — or `<i>: empty` when the line has no words. After the lines print `total=<sum of all word counts>`.

Read `n` with `std::cin >>`, discard the rest of that line with `std::cin.ignore(...)`, then `std::getline` each line and split it with a `std::istringstream`. Strip a trailing `\r` if one is present.

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

```
3
the quick brown fox

  spaced   out
```

prints

```
1: words=4 longest=quick
2: empty
3: words=2 longest=spaced
total=6
```

INPUT 1

```
3
the quick brown fox

  spaced   out
```

OUTPUT 1

```
1: words=4 longest=quick
2: empty
3: words=2 longest=spaced
total=6
```

INPUT 2

```
1
hello
```

OUTPUT 2

```
1: words=1 longest=hello
total=1
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    long long total = 0;
    for (int i = 1; i <= n; ++i) {
        std::string line;
        std::getline(std::cin, line);
        if (!line.empty() && line.back() == '\r') line.pop_back();
        std::istringstream words(line);
        std::string word;
        std::string longest;
        int count = 0;
        while (words >> word) {
            ++count;
            if (word.size() > longest.size()) longest = word;   // strict >: the fir
        }
        if (count == 0) std::cout << i << ": empty\n";
        else std::cout << i << ": words=" << count << " longest=" << longest << '\n'
        total += count;
    }
    std::cout << "total=" << total << '\n';
    return 0;
}
```

EXERCISE 2

Matrix with dimension validation

Read `R C` on the first line, then the matrix with one row per line. Read the rows **line by line** (`std::getline` then a `std::istringstream`) so the shape can be checked:

- if the input ends before `R` rows were read, print `error: expected <R> rows, got <k>` and stop;

if a row does not contain exactly `C` integers, print `error: row <i> has <k> values, expected <C>` (1-based `i`, the first offending row) and stop;

otherwise print `row <i>: <sum>` for each row, then `cols: <c1> <c2> ... <cC>` with the column sums.

Use `long long` for the values and sums.

Input: `R C`, then up to `R` lines. **Output:** the report or the first error.

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

prints

```
row 1: 6
row 2: 15
cols: 5 7 9
```

and

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

prints `error: row 2 has 2 values, expected 3`.

INPUT 1

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

OUTPUT 1

```
row 1: 6
row 2: 15
cols: 5 7 9
```

INPUT 2

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

OUTPUT 2

```
error: row 2 has 2 values, expected 3
```

REFERENCE SOLUTION

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

int main() {
    int rows, cols;
    std::cin >> rows >> cols;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    std::vector<std::vector<long long>> m;
    for (int r = 0; r < rows; ++r) {
        std::string line;
        if (!std::getline(std::cin, line)) {
            std::cout << "error: expected " << rows << " rows, got " << r << '\n';
            return 0;
        }
        std::istringstream in(line);
        std::vector<long long> row;
        long long v;
        while (in >> v) row.push_back(v);
        if (static_cast<int>(row.size()) != cols) {
            std::cout << "error: row " << r + 1 << " has " << row.size() << " values
            return 0;
        }
        m.push_back(std::move(row));
    }
    std::vector<long long> colSums(cols, 0);
    for (int r = 0; r < rows; ++r) {
        long long sum = 0;
        for (int c = 0; c < cols; ++c) {
            sum += m[r][c];
            colSums[c] += m[r][c];
        }
        std::cout << "row " << r + 1 << ": " << sum << '\n';
    }
    std::cout << "cols:";
    for (long long s : colSums) std::cout << ' ' << s;
    std::cout << '\n';
    return 0;
}
```

QUIZ

1. The input is `2⏎Ada Lovelace⏎…`. The program runs `int n; std::cin >> n; std::string s; std::getline(std::cin, s);`. What is `s`?

A. `"Ada Lovelace"`

B. `"2"`

C. `""` — the empty remainder of the first line

D. `"Ada"`

2. Why parse each record line in its own `std::istringstream` instead of reading the fields straight from `std::cin`?

A. `std::istringstream` is faster than `std::cin`

B. A malformed line only fails its own stream; `std::cin` is untouched and the loop moves to the next line

C. `std::cin` cannot read more than one value per line

D. It avoids the leftover-newline problem entirely

3. A matrix is declared `3 3` but one row has four numbers. Which reader notices?

A. The token-wise reader: `for r, for c: std::cin >> m[r][c]`

B. The line-wise reader that parses each row from its own `std::istringstream` and compares the count with `C`

C. Both — `>>` sets `failbit` on the extra value

D. Neither; the extra value is silently dropped

4. A file saved on Windows is read on Linux with `std::getline`. The last field of every line compares unequal to `"done"` although it looks identical. Why?

A. Linux `getline` uses a different delimiter

B. Each line ends in `\r\n`; `getline` stops at `\n` and leaves the `\r` on the field

C. The file is UTF-16

D. `std::string` comparison is locale-sensitive

5. In `std::getline(fields >> std::ws, name)`, what does `std::ws` do?

A. Reads one whitespace-separated word into `name`

B. Skips whitespace so the line read starts at the first non-space character

C. Writes a space into the stream

D. Marks the end of the previous token

6. What is wrong with `int n; std::cin >> n; std::vector<int> v(n);` when the input is empty?

A. Nothing: `n` becomes 0 and the vector is empty

B. The read fails at the end of the input before any conversion runs, so `n` keeps its uninitialised value — the vector's size is garbage

C. `std::cin` throws `std::ios_base::failure`

D. The vector's constructor throws `std::length_error`

ANSWER KEY · READING STRUCTURED INPUT — THE PATTERN CATALOGUE

1. C `>>` stops at the newline after `2` and leaves it in the buffer. `getline` reads up to that newline — nothing — and consumes it. An `ignore` to the end of the line between the two reads fixes it.

2. B A bad field read directly from `std::cin` sets its `failbit` and every later record is lost. Reading the whole line first makes the failure local. Speed is not the reason, and the newline issue is separate (an `ignore` after a `>>` count).

3. B Token-wise reading takes nine tokens wherever they are, so the extra value shifts into the next row with no error. Only a reader that sees the row as a line can count what was on it.

4. B Windows line endings are `\r\n`. `getline` consumes the `\n` only, so `"done\r"` is what the string holds. Strip a trailing `\r` after every `getline` when input may come from another platform.

5. B `>> std::ws` consumes leading whitespace and nothing else. It is what removes the separator that `>>` stopped on before `getline` takes the rest of the line — for a last field that may contain spaces.

6. B When the stream is already at the end, the extraction fails before the numeric conversion, and only the conversion sets the variable to 0. Test the read (`if (!(std::cin >> n))`) before using `n` as a size.

LESSON 3 · 14 MIN

File streams — open modes, RAII and reading back

A file stream is `std::cin` or `std::cout` with a file behind the buffer. Everything from lessons 1 and 2 — `>>`, `getline`, the state bits, the recovery loop — works unchanged; what is new is opening, the modes that decide whether a file is truncated or appended to, the fact that a stream object *is* the open handle (so its destructor closes the file), and a handful of failure modes that a console program never meets: the file is missing, the directory does not exist, the data was written but never flushed before another stream tried to read it. The study judge lets a program create, write and read files in its working directory, so this lesson's programs write a file and read it straight back.

The three classes

`<fstream>` declares `std::ofstream` (output), `std::ifstream` (input) and `std::fstream` (both). The constructor takes a path — a `std::string`, a C string or a `std::filesystem::path` (lesson 5) — and opens it at once:

```
#include <fstream>
#include <iostream>
#include <string>

int main() {
    {
        std::ofstream out("notes.txt");
        if (!out) { std::cout << "cannot create notes.txt\n"; return 0; }
        out << "first line\n" << "second line\n";
    }                                          // out is destroyed here: flu

    std::ifstream in("notes.txt");
    std::string line;
    int count = 0;
    while (std::getline(in, line)) ++count;
    std::cout << count << " lines\n";
    return 0;
}
```

A path is resolved against the process's *working directory* — where the program was run from, not where the source file lives. `"notes.txt"` in the judge is a file beside the compiled program; the same string on your machine is a file in whatever directory your terminal was in.

Did it open?

The one check every file program needs. `is_open()` reports the handle; `operator bool` (that is, `!fail()`) reports the same thing right after construction and is the idiom, because it also catches a later failure. An `ifstream` for a file that does not exist does not throw — it sets `failbit` and every read silently fails, which is how a program prints `0 lines` for a typo in the filename. An `ofstream` fails when the directory does not exist or is not writable; it *creates* the file when only the file is missing.

Open modes

The second constructor argument is a bit mask from `std::ios`:

Mode	Meaning
`in`	open for reading (the default for `ifstream`)
`out`	open for writing; on its own, **truncates** to empty (the default for `ofstream`)
`app`	append: every write goes to the end, whatever the position
`trunc`	discard existing contents (implied by `out` alone)
`ate`	start positioned *at the end*, but later writes may seek elsewhere
`binary`	no newline translation (lesson 5)

Combine them with `|`. The three that matter in practice: `std::ofstream out("log.txt")` starts a fresh file every run; `std::ofstream out("log.txt", std::ios::app)` keeps what is there and adds to it — a second open with `app` continues after what the first open wrote, which is how a ledger accumulates across runs; and `std::fstream io("data.txt", std::ios::in | std::ios::out)` opens for both directions *without* truncating, but fails if the file does not already exist — add `std::ios::trunc` or create it first.

RAII: the object is the open file

A file stream closes itself in its destructor. That is why the writing code above sits in its own block: when `out` goes out of scope, its buffer is flushed and the descriptor released *before* `in` opens the same file. Without that block the write might still be sitting in `out`'s buffer when `in` reads, and the count comes back short. The explicit calls exist for the cases the scope cannot express — `out.close()` to finish a file and then reopen the same object with `out.open("other.txt")`, or `out.flush()` to push the bytes without closing — but the scope is the idiom, exactly as Module 7 lesson 3 described: the resource's lifetime *is* the object's lifetime, and an early return or an exception still closes the file.

Writes are not on disk until flushed. `std::endl` flushes after every line, which is the reason it is slow in loops and also the reason logs written with it survive a crash; a judged program writes with `'\n'` and lets the destructor flush once.

Reading a file

Every reading pattern from lesson 2 applies, with `in` in place of `std::cin`: `while (std::getline(in, line))` for lines, `while (in >> x)` for tokens, `in >> n; in.ignore(...)` before switching to lines. Two whole-file idioms are worth knowing:

```
std::ifstream in("notes.txt");
std::string all((std::istreambuf_iterator<char>(in)), std::istreambuf_iterator
```

The extra parentheses around the first argument stop the compiler reading the line as a function declaration (the "most vexing parse"). For a file too large to hold, stream it line by line instead — a 2 GB log does not need 2 GB of memory to be counted.

The position is queryable: `out.tellp()` is how many bytes have been written so far, `in.tellg()` where the next read starts, and `in.seekg(0)` rewinds — the basis of lesson 5's record file. A read that ran off the end leaves the stream with `eofbit` and `failbit` set, so to read a file twice you `clear()` before the `seekg`.

Deleting and checking

A program that writes a scratch file should remove it: `std::remove("notes.txt")` from `<cstdio>` returns `0` on success, or `std::filesystem::remove("notes.txt")` (lesson 5)

returns whether anything was deleted. Closing the stream first is essential — on some platforms an open file cannot be removed, and on all of them a buffered write may not have happened yet.

Error handling in a file program

The stream never throws unless asked (`exceptions()` , lesson 1). The checkable moments are: after opening (`if (!out)`), after the reading loop (`in.bad()` distinguishes a disk error from a plain end of file), and after closing (`out.close(); if (out.fail())` — the flush can fail on a full disk). A judged program that reads its own freshly written file will not hit these, but a tool that does not check its open is one typo away from computing statistics on an empty stream and reporting them with confidence.

Pitfalls

Mistake	What happens
Not checking the open	Every read fails silently; the program reports zeros.
Reading a file another stream is still writing	The writer's buffer has not been flushed: the reader sees a short file. Close or scope the writer first.
`std::ofstream out("f.txt")` to add to a file	The default truncates. Use `std::ios::app` .
`std::ios::in | std::ios::out` on a missing file	The open fails; `fstream` does not create it.
A relative path "to the source file"	Paths are relative to the working directory.
`std::endl` on every line of a big file	A flush per line; use `'\n'` and let the close flush.

Key takeaways

`std::ofstream`, `std::ifstream` and `std::fstream` are `cout` / `cin` with a file behind the buffer; every reading pattern carries over.

Check the open with `if (!stream)`; a missing input file is a silent `failbit`, not an exception.

`out` truncates, `app` appends across opens, `in | out` needs an existing file, `binary` disables newline translation.

The destructor flushes and closes: scope the writer so it finishes before the reader opens the same file.

Paths are relative to the working directory; remove scratch files after closing them.

EXERCISE 1

Write it down, read it back

Read every line of standard input until it ends and write each one, followed by a newline, to a file named `notes.txt` through a `std::ofstream`. Just before that stream closes, print `written=<bytes>` using `tellp()`. Then open `notes.txt` with a `std::ifstream` — the writer must be closed or out of scope first — and read it line by line with `std::getline`, counting lines, whitespace-separated words, and characters (the sum of the line lengths, newlines excluded), and tracking the longest line. Print `lines=<l> words=<w> chars=<c> longest=<length>`. Delete the file at the end (`std::remove` from `<cstdio>` or `std::filesystem::remove`).

Input: any number of lines. **Output:** two lines.

```
the quick brown
fox
```

prints

```
written=20
lines=2 words=4 chars=18 longest=15
```

INPUT 1

```
the quick brown
fox
```

OUTPUT 1

```
written=20
lines=2 words=4 chars=18 longest=15
```

INPUT 2

```
one

two  three
```

OUTPUT 2

```
written=16
lines=3 words=3 chars=13 longest=10
```

REFERENCE SOLUTION

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

int main() {
    const char* const path = "notes.txt";
    {
        std::ofstream out(path);
        std::string line;
        while (std::getline(std::cin, line)) out << line << '\n';
        std::cout << "written=" << out.tellp() << '\n';
    }                                                   // out is flushed and closed
    std::ifstream in(path);
    long long lines = 0, words = 0, chars = 0, longest = 0;
    std::string line;
    while (std::getline(in, line)) {
        ++lines;
        const long long length = static_cast<long long>(line.size());
        chars += length;
        if (length > longest) longest = length;
        std::istringstream fields(line);
        std::string word;
        while (fields >> word) ++words;
    }
    in.close();
    std::remove(path);
    std::cout << "lines=" << lines << " words=" << words << " chars=" << chars << "
    return 0;
}
```

EXERCISE 2

A ledger appended across two opens

The input holds two sessions of ledger entries. First an integer `n` and `n` lines, then an integer `m` and `m` lines; each entry line is `<label> <amount>` with an integer amount that may be negative. Write the first session's lines to `ledger.txt` with a fresh `std::ofstream` (the default mode, which truncates) and close it. Then open `ledger.txt` **a second time** with `std::ios::app` and write the second session's lines after the first. Finally read the whole file back and print every line as `<i>: <line>` with a 1-based number, then `entries=<count>` and `balance=<sum of the amounts>` (a `long long`). Delete the file at the end.

Use `std::getline` for the entry lines (remember the `ignore` after each count) and write each line back exactly as read.

Input: `n`, `n` lines, `m`, `m` lines. **Output:** the numbered lines, then two summary lines.

```
2
opening 100
coffee -3
1
salary 250
```

prints

```
1: opening 100
2: coffee -3
3: salary 250
entries=3
balance=347
```

INPUT 1

```
2
opening 100
coffee -3
1
salary 250
```

OUTPUT 1

```
1: opening 100
2: coffee -3
3: salary 250
entries=3
balance=347
```

INPUT 2

```
1
rent -800
0
```

OUTPUT 2

```
1: rent -800
entries=1
balance=-800
```

REFERENCE SOLUTION

```cpp
#include <filesystem>
#include <fstream>
#include <iostream>
#include <limits>
#include <sstream>
#include <string>

int main() {
    const char* const path = "ledger.txt";
    int n;
    std::cin >> n;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    {
        std::ofstream out(path);                        // truncates: a fresh ledger
        for (int i = 0; i < n; ++i) {
            std::string line;
            std::getline(std::cin, line);
            out << line << '\n';
        }
    }
    int m;
    std::cin >> m;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    {
        std::ofstream out(path, std::ios::app);         // continues after session c
        for (int i = 0; i < m; ++i) {
            std::string line;
            std::getline(std::cin, line);
            out << line << '\n';
        }
    }
    std::ifstream in(path);
    std::string line;
    int entries = 0;
    long long balance = 0;
    while (std::getline(in, line)) {
        ++entries;
        std::cout << entries << ": " << line << '\n';
        std::istringstream fields(line);
        std::string label;
        long long amount = 0;
        if (fields >> label >> amount) balance += amount;
    }
    in.close();
    std::filesystem::remove(path);
    std::cout << "entries=" << entries << '\n';
    std::cout << "balance=" << balance << '\n';
```

```
    return 0;
}
```

QUIZ

1. `data.txt` already holds ten lines. What does `std::ofstream out("data.txt");` do to it?

A. Nothing until the first write

B. Truncates it to empty immediately — `out` alone implies `trunc`

C. Opens it positioned at the end, ready to append

D. Fails, because the file exists

2. `std::ifstream in("missing.txt"); int x; in >> x;` — the file does not exist. What happens?

A. The constructor throws `std::ios_base::failure`

B. The open sets `failbit`; `in` converts to `false` and the read silently does nothing

C. `x` is set to `-1`

D. The program aborts with a runtime error

3. Which statement about closing a file stream is correct?

A. You must call `close()` or the file is leaked

B. The destructor flushes and closes the file, so leaving the scope is enough — `close()` is for reopening the same object or checking the flush

C. `close()` is needed only for `std::ofstream`

D. Files close automatically only when the program exits

4.

```
std::ofstream out("f.txt");
out << "one\ntwo\n";
std::ifstream in("f.txt");
std::string line; int n = 0;
while (std::getline(in, line)) ++n;
```

What is `n` most likely to be, and why?

A. 2 — the writes are visible at once

B. 0 — `out` has not been flushed, so the file is still empty when `in` reads it

C. 1 — only the first line is flushed

D. A compile error: two streams cannot open one file

5. `std::fstream io("cache.txt", std::ios::in | std::ios::out);` when `cache.txt` does not exist:

A. Creates it empty and opens it for both directions

B. Fails to open — `in | out` requires an existing file; add `trunc` or create it first

C. Opens for reading only

D. Throws `std::filesystem::filesystem_error`

6. What is the difference between `std::ios::app` and `std::ios::ate`?

A. None — they are synonyms

B. `app` sends every write to the end regardless of seeks; `ate` only starts at the end and later seeks may write elsewhere

C. `ate` truncates the file; `app` does not

D. `app` is for text files, `ate` for binary

ANSWER KEY · FILE STREAMS — OPEN MODES, RAII AND READING BACK

1. B The default mode for `std::ofstream` is `std::ios::out`, which discards existing contents on open. To add to the file, pass `std::ios::app`.

2. B File streams never throw unless `exceptions()` was called. A failed open leaves the stream in a failed state, so every read is a no-op — which is why a program must check `if (!in)` after opening, or it computes results from nothing.

3. B A file stream is an RAII object: its destructor releases the handle after flushing, on every path out of the scope. Explicit `close()` exists for when you want to reopen the object or inspect the state after the final flush.

4. B The text sits in `out`'s buffer until it is flushed or the stream is destroyed. Scope the writer in its own block (or call `close()`) before opening the reader.

5. B Without `trunc` (or `app`), the combined mode opens an existing file without discarding it and does not create one. `std::ofstream` alone would create the file.

6. B Append mode forces every write to the end. At-end mode positions the stream at the end once, at open, and then behaves like a normal stream, so a `seekp` can move it back.

LESSON 4 · 14 MIN

String streams and formatting — building text in memory

A string stream is the third member of the family: the same formatter as `std::cin` and `std::cout`, with a `std::string` as the buffer. Reading from one turns a line you already hold into tokens; writing to one builds a report in memory that you can measure, inspect and print in a single call. Module 5 met `std::istringstream` as a line splitter and `std::ostringstream` as a string builder; this lesson covers what that left out — reusing a stream without the state bug that bites everyone once, round-tripping strings that contain spaces with `std::quoted`, and `std::format` as the modern way to build formatted text, including formatting straight into an existing string.

Three classes, one header

`<sstream>` declares `std::istringstream` (read from a string), `std::ostringstream` (write into one) and `std::stringstream` (both). `str()` with no argument returns a *copy* of the buffer; `str(s)` replaces it. The copy matters: `oss.str().size()` is fine, but a loop that calls `oss.str()` on every iteration copies the whole report every time. Take it once into a variable.

```
std::ostringstream report;
report << "Total: " << std::fixed << std::setprecision(2) << 1234.5 << '\n';
const std::string text = report.str();
std::cout << text << "length=" << text.size() << '\n';
```

Every manipulator works — `std::setw`, `std::left`, `std::fixed`, `std::setprecision` from `<iomanip>` behave exactly as on `std::cout` (Module 5, lesson 5), and they are just as sticky. Building in memory is what makes it possible to know a report's line count or length *before* printing it, and it is the fast way to emit a large output: one `std::cout << text` instead of a million small writes.

Reusing a stream

The trap. A stream that has been read to its end has `eofbit` and `failbit` set; giving it a new buffer does not clear them, so the next read fails immediately:

```
std::istringstream in("10");
int v;
in >> v;              // 10; the stream is now at the end (eofbit set by the 1
in >> v;              // fails: failbit
in.str("20");         // new contents, OLD state
in >> v;              // still fails
in.clear();           // now it reads
in.str("30");
in >> v;              // 30
```

The idiom is always the pair: `in.clear(); in.str(newText);`. The same applies to an `ostringstream` emptied with `oss.str("")` — `clear()` is harmless when nothing was set, so write both every time and the bug cannot happen. For a per-line parser it is usually simpler to construct a fresh `std::istringstream` inside the loop (lesson 2, Shape 4); the object is cheap.

`std::quoted`

`>>` into a string stops at whitespace, which is fine until a field is a name with spaces in it. `std::quoted` (`<iomanip>`) is the standard library's answer: on output it wraps a string in double quotes and escapes any embedded `"` or `\` with a backslash; on input it reads a quoted string as one token, spaces included, undoing the escapes.

```
std::ostringstream out;
out << std::quoted("Ada Lovelace") << ' ' << std::quoted("say \"hi\"") << ' '
// "Ada Lovelace" "say \"hi\"" 36

std::istringstream in(out.str());
std::string name, phrase;
int age;
in >> std::quoted(name) >> std::quoted(phrase) >> age;
// name == Ada Lovelace, phrase == say "hi", age == 36
```

It is a round trip by construction: whatever `<< std::quoted(s)` wrote, `>> std::quoted(t)` reads back into an equal string, whatever `s` contained. A string without quotes in the input reads as an ordinary token, so the manipulator is forgiving of unquoted single words. Both the delimiter and the escape character can be changed (`std::quoted(s, '\'', '\\')`), which is how a single-quoted format is handled. It is the simplest correct serialisation of free text into a whitespace-separated record.

`std::format`

C++20's `<format>` builds a `std::string` from a format string and arguments, with the specification mini-language Module 5 introduced: `{}` in order, `{:>8}` right-aligned in eight, `{:<10}` left, `{:^7}` centred, `{:*^9}` with a fill character, `{:.2f}` two decimals, `{:06.2f}` zero-padded, `{:+d}` always signed, `{:#x}` hexadecimal with a prefix, and `{:>{}}` taking the width from the next argument:

```
#include <format>

const int width = 12;
std::string row = std::format("{:<{}}{:>6}{:>10.2f}\n", "widget", width, 3, 4.
// "widget      "  "     3"  "      4.50"
```

Three differences from the stream manipulators. Nothing is sticky: each `{}` is formatted from its own specification and the next call starts clean. The format string is checked at compile time when it is a literal — `std::format("{} {}", 1)` is a compile error, not a run-time surprise — and a format string only known at run time must go through `std::vformat` with `std::make_format_args`. And the result is a value: pass it to `std::cout <<`, append it to a string, put it in a container.

For building a long text, `std::format_to` writes into an output iterator instead of allocating a new string per call:

```
std::string report;
for (const auto& [name, qty] : items) {
    std::format_to(std::back_inserter(report), "{:<10}{:>5}\n", name, qty);
}
std::cout << report;
```

`std::format_to_n` caps the number of characters written, for a fixed buffer. C++23 adds `std::print` and `std::println`, which format straight to `stdout` with no `<<` at all; the study runtime is C++20, so that is reading only — `std::cout << std::format(...)` is the equivalent here.

Building a report

A table is a two-pass job whichever tool you use. The first pass reads the records and finds the widths — the longest name, the widest number — with `std::max`; the second formats every row with those widths. Compute the totals in the first pass too (`long long` for sums, `double` only for money already given as decimals), then write the header, the rows and the total line into an `ostringstream` or through `format_to`, and print once. The reason to build in memory rather than print as you go is that the header often depends on what follows: the column width, the row count, whether there was anything at all.

```
std::size_t nameWidth = 4;                                    // "Item"
for (const auto& r : rows) nameWidth = std::max(nameWidth, r.name.size());
std::string out = std::format("{:<{}}{:>6}{:>10}\n", "Item", nameWidth, "Qty",
for (const auto& r : rows) {
    std::format_to(std::back_inserter(out), "{:<{}}{:>6}{:>10.2f}\n", r.name,
}
```

Numbers to text and back

`std::to_string(3.14159)` gives `3.141590` — six decimals, no control. `std::format("{:.2f}", x)` or an `ostringstream` with `std::fixed << std::setprecision(2)` gives the precision you asked for, and both are what a judged answer needs, because the compare is exact. In the other direction, an `istringstream` reading into a `double` accepts the same forms `std::stod` does and reports failure through the state instead of an exception; `std::from_chars` (Module 5, lesson 3) is the fastest and strictest of the three.

Pitfalls

Mistake	What happens
`in.str(next)` without `in.clear()`	Reads keep failing: the old `failbit` is still set.
`oss.str()` in a loop	A full copy of the buffer every iteration.
`>>` into a string for a field with spaces	Splits at the first space; use `std::quoted` or `getline`.
Expecting `std::format` to remember a width	Nothing is sticky; every `{}` states its own spec.
`std::format(runtimeString, x)`	Compile error: a non-literal format string needs `std::vformat`.
`std::to_string` for a fixed-precision answer	Always six decimals; use `{:.2f}` or `setprecision`.

Key takeaways

String streams are the console formatter over a `std::string`; `str()` copies, `str(s)` replaces.

Reuse means `clear()` **and** `str(...)`; a fresh `istringstream` per line is the simpler habit.

`std::quoted` writes a quoted, escaped string and reads it back intact — a round trip for text with spaces.

`std::format` returns a string, checks a literal format string at compile time and is never sticky; `std::format_to` appends into an existing string.

Build reports in memory in two passes — measure, then format — and print once.

EXERCISE 1

Report builder

Read an integer `n`, then `n` lines of `<item> <qty> <price>` (a one-word item, an integer quantity, a decimal price). Build the whole report in a `std::ostringstream` and print it once at the end:

a header: `ITEM` left-aligned in a field of 10, `QTY` right-aligned in 5, `TOTAL` right-aligned in 10;

one row per item with the same widths, the total (`qty × price`) with `std::fixed` and two decimals;

a rule of 25 `-` characters;

a `TOTAL` line: `TOTAL` left in 10, an empty field of 5, then the grand total right in 10 with two decimals.

Before printing the report, print `lines=<k>` where `k` is the number of newline characters in `report.str()` — the reason to build in memory is that you can inspect the text before it goes out.

Input: `n`, then `n` lines. **Output:** `lines=<k>`, then the report.

```
2
pen 3 1.50
notebook 2 4.25
```

prints

```
lines=5
ITEM        QTY     TOTAL
pen           3      4.50
notebook      2      8.50
-------------------------
TOTAL               13.00
```

INPUT 1

```
2
pen 3 1.50
notebook 2 4.25
```

OUTPUT 1

```
lines=5
ITEM        QTY     TOTAL
pen           3      4.50
notebook      2      8.50
-------------------------
TOTAL               13.00
```

INPUT 2

```
1
widget 10 0.10
```

OUTPUT 2

```
lines=4
ITEM        QTY     TOTAL
widget       10      1.00
-------------------------
TOTAL                1.00
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::ostringstream report;
    report << std::fixed << std::setprecision(2);
    report << std::left << std::setw(10) << "ITEM" << std::right << std::setw(5) <<
    double grand = 0.0;
    for (int i = 0; i < n; ++i) {
        std::string item;
        int qty;
        double price;
        std::cin >> item >> qty >> price;
        const double total = qty * price;
        grand += total;
        report << std::left << std::setw(10) << item << std::right << std::setw(5) <
    }
    report << std::string(25, '-') << '\n';
    report << std::left << std::setw(10) << "TOTAL" << std::right << std::setw(5) <<
    const std::string text = report.str();
    std::cout << "lines=" << std::count(text.begin(), text.end(), '\n') << '\n';
    std::cout << text;
    return 0;
}
```

EXERCISE 2

std::quoted round trip

Read an integer `n`, then `n` lines, each a record `"<name>" "<city>" <age>`. The two strings are double-quoted, may contain spaces, and may contain a `"` escaped as `\"` or a backslash escaped as `\\`. Parse each line with a `std::istringstream` and `std::quoted` (`<iomanip>`); do not parse the quotes by hand.

For each record print two lines: `<name> | <city> | <age>` with the raw, unquoted values, and `→` followed by the record re-serialised with `std::quoted` (name, city, age separated by single spaces). Then print `oldest=<name of the highest age, the first on ties>` (`oldest=none` when `n` is 0). Finally take everything you serialised, parse it back with `std::quoted` and print `roundtrip=ok` if every field came back equal to the original, otherwise `roundtrip=mismatch`.

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

```
1
"Bob \"the builder\" Smith" "Rio" 51
```

prints

```
Bob "the builder" Smith | Rio | 51
  → "Bob \"the builder\" Smith" "Rio" 51
oldest=Bob "the builder" Smith
roundtrip=ok
```

INPUT 1

```
2
"Ada Lovelace" "London" 36
"Bob" "New York" 40
```

OUTPUT 1

```
Ada Lovelace | London | 36
  → "Ada Lovelace" "London" 36
Bob | New York | 40
  → "Bob" "New York" 40
oldest=Bob
roundtrip=ok
```

INPUT 2

```
1
"Bob \"the builder\" Smith" "Rio" 51
```

OUTPUT 2

```
Bob "the builder" Smith | Rio | 51
  → "Bob \"the builder\" Smith" "Rio"
51
oldest=Bob "the builder" Smith
roundtrip=ok
```

REFERENCE SOLUTION

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

struct Person {
    std::string name;
    std::string city;
    int age = 0;
};

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');
    std::vector<Person> people;
    for (int i = 0; i < n; ++i) {
        std::string line;
        std::getline(std::cin, line);
        std::istringstream in(line);
        Person p;
        in >> std::quoted(p.name) >> std::quoted(p.city) >> p.age;
        people.push_back(p);
    }
    std::ostringstream serialised;
    for (const auto& p : people) {
        std::cout << p.name << " | " << p.city << " | " << p.age << '\n';
        std::ostringstream one;
        one << std::quoted(p.name) << ' ' << std::quoted(p.city) << ' ' << p.age;
        std::cout << "  -> " << one.str() << '\n';
        serialised << one.str() << '\n';
    }
    int oldest = -1;
    for (int i = 0; i < static_cast<int>(people.size()); ++i) {
        if (oldest < 0 || people[i].age > people[oldest].age) oldest = i;
    }
    std::cout << "oldest=" << (oldest < 0 ? std::string("none") : people[oldest].nam
    std::istringstream back(serialised.str());
    bool ok = true;
    for (const auto& p : people) {
        Person q;
        if (!(back >> std::quoted(q.name) >> std::quoted(q.city) >> q.age)) { ok = f
        if (q.name != p.name || q.city != p.city || q.age != p.age) ok = false;
    }
    std::cout << (ok ? "roundtrip=ok" : "roundtrip=mismatch") << '\n';
```

```
    return 0;
}
```

QUIZ

1. What does this print?

```
std::istringstream in("10");
int v = 0;
in >> v;
in >> v;          // fails
in.str("20");
in >> v;
std::cout << v << ' ' << in.fail();
```

A. `20 0`

B. `10 1`

C. `0 1`

D. `20 1`

2. `std::cout << std::quoted("say \"hi\"");` prints…

A. `say "hi"`

B. `"say \"hi\""`

C. `"say "hi""`

D. `'say "hi"'`

3. What string does `std::format("{:06.2f}", 3.14159)` produce?

A. `3.14`

B. `003.14`

C. `3.1416`

D. `3.14`

4. `std::format("{} {}", 1)` with a string-literal format string…

A. Prints `1 ` with an empty second field

B. Throws `std::format_error` at run time

C. Is a compile error: the literal is checked against the arguments at compile time

D. Prints `1 1`

5. Which is true of `oss.str()` on a `std::ostringstream`?

A. It returns a reference to the internal buffer

B. It returns a copy of the buffer contents, so calling it in a loop copies the whole text each time

C. It clears the stream after returning the text

D. It flushes the text to `std::cout`

6. How does `std::format_to(std::back_inserter(s), "{}", x)` differ from `s += std::format("{}", x)`?

A. It does not — they are identical

B. `format_to` writes straight into `s` through the iterator, without building a temporary string first

C. `format_to` returns the number of characters written

D. `format_to` accepts a run-time format string; `format` does not

ANSWER KEY · STRING STREAMS AND FORMATTING — BUILDING TEXT IN MEMORY

1. B The second read failed and set `failbit`; `str("20")` replaces the buffer but not the state, so the third read is a no-op and `v` stays `10`. `in.clear()` before `str(...)` is the fix.

2. B `std::quoted` wraps the string in double quotes and escapes every embedded `"` (and `\`) with a backslash, so that `>> std::quoted(s)` can read it back as one token unambiguously.

3. B `.2f` gives two decimals (`3.14`, five characters); width `6` pads to six, and the leading `0` in the spec requests zero padding after the sign, so `003.14`. A plain `{:6.2f}` would pad with spaces.

4. C C++20 requires the format string to be a compile-time constant and checks it against the argument types and count. Only a run-time format string, passed through `std::vformat`, can fail at run time with `std::format_error`.

5. B `str()` returns a `std::string` by value — a full copy. Take it once into a variable. Nothing is printed or cleared; `str("")` is the call that empties the buffer (and `clear()` resets the state).

6. B `std::format` allocates a new string and returns it; `std::format_to` formats directly into whatever the output iterator writes to. Both check a literal format string at compile time.

LESSON 5 · 15 MIN

Binary I/O and the filesystem

Text I/O turns every number into digits and back, which is slow, loses precision for floating point, and makes a record's size depend on its values. Binary I/O writes the bytes an object already occupies in memory — a `double` is eight bytes on disk exactly as it is in RAM — and reads them straight back, so a file of fixed-width records can be indexed by arithmetic: record `i` starts at byte `i * sizeof(Record)`. The price is that the bytes mean nothing without the struct that wrote them, and nothing about their order is portable across machines. This lesson covers `write` / `read`, what may safely be written that way, endianness, and `std::filesystem` — the C++17 library for paths, sizes, directories and listings that a file-handling program needs around its streams.

`write` and `read`

```
#include <cstdint>
#include <fstream>

struct Record {
    std::int32_t id;
    double score;
    char name[16];
};

int main() {
    Record r{7, 91.5, "ada"};
    {
        std::ofstream out("records.bin", std::ios::binary);
        out.write(reinterpret_cast<const char*>(&r), sizeof r);
    }
    std::ifstream in("records.bin", std::ios::binary);
    Record back{};
    in.read(reinterpret_cast<char*>(&back), sizeof back);
    // back.id == 7, back.score == 91.5, back.name is "ada"
    return 0;
}
```

`write(const char*, count)` copies `count` bytes from the address; `read(char*, count)` fills them. Both want a `char*`, and an object's address is a `Record*`, so the cast is

`reinterpret_cast` — Module 2 lesson 4's "reinterpret these bytes as that type", which is exactly the meaning here and one of its few legitimate uses. After a `read`, `in.gcount()` says how many bytes actually arrived; a short read at the end of the file sets `failbit`, so `while (in.read(...))` is the loop for a whole file of records.

What may be written this way: only *trivially copyable* types — built-in numbers, `char` arrays, and structs made of them. A `std::string` or `std::vector` is a small handle (a pointer, a size, a capacity); writing `sizeof(std::string)` bytes of it saves the pointer, not the characters, and reading it back produces an object pointing at memory that no longer exists. `static_assert(std::is_trivially_copyable_v<Record>)` next to the struct documents the requirement and stops a later `std::string` member compiling.

Fixed-width records and padding

`sizeof(Record)` above is 32 on this platform, not 28: the `double` must sit at an address that is a multiple of 8, so the compiler inserts four bytes of padding after the `int32_t`. The layout is `id` (4), padding (4), `score` (8), `name` (16). Padding is part of the object and is written and read with it, which is harmless — as long as the reader uses the same struct with the same compiler settings. Reordering the members (`double` first) removes the padding here; `#pragma pack` forces it away everywhere at the cost of misaligned access and is not portable.

Because every record has the same size, the file is an array. `in.seekg(static_cast<std::streamoff>(i) * sizeof(Record))` positions the stream at record `i`; `in.tellg()` reports the position; `std::filesystem::file_size` divided by `sizeof(Record)` is the record count. That random access is the whole reason for the format — a text file has to be read from the start to find its hundredth line.

`std::ios::binary` disables newline translation. On Linux, where the judge runs, it changes nothing; on Windows a text-mode stream rewrites `\n` as `\r\n` on the way out and silently corrupts any byte `0x0A` inside a `double`. Always specify it for `write` / `read` so the program means the same thing everywhere.

Endianness

A four-byte `int` holding `1` is stored as `01 00 00 00` on x86-64 — *little-endian*, least significant byte first — and as `00 00 00 01` on a big-endian machine. `write` emits whichever order the CPU uses, so a binary file is only readable on machines that agree.

`<bit>` reports the order (`std::endian::native == std::endian::little`), and a format meant to travel serialises each integer byte by byte with shifts (`(v >> 8) & 0xFF`, …) in a declared order, or uses `std::byteswap` — C++23, reading only here. Network protocols are big-endian by convention; most file formats written today are little-endian. For a file a program writes and reads on the same machine — a cache, a save file, this lesson's exercise — the native order is fine, and a small header with a magic number and version lets a reader refuse a file it does not understand.

`std::filesystem`

`<filesystem>` (C++17) is the portable answer to "does this file exist, how big is it, what is in this directory". The namespace is long, so the idiom is an alias: `namespace fs = std::filesystem;`.

Call	Result
`fs::path p = fs::path("data") / "q1.csv";`	joins with the platform separator
`p.filename()`, `p.stem()`, `p.extension()`, `p.parent_path()`	`q1.csv`, `q1`, `.csv`, `data`
`fs::exists(p)`, `fs::is_regular_file(p)`, `fs::is_directory(p)`	queries
`fs::file_size(p)`	bytes, as `std::uintmax_t`; throws for a directory
`fs::create_directory(p)` / `fs::create_directories(p)`	one level / the whole chain; `true` if created
`fs::remove(p)` / `fs::remove_all(p)`	one entry / a tree; returns whether removed / how many entries
`fs::rename(from, to)`, `fs::copy_file(from, to)`	as named
`for (const auto& entry : fs::directory_iterator(p))`	every entry; `entry.path()`, `entry.is_regular_file()`, `entry.file_size()`

`fs::path` is a string with path semantics: `/` joins components, `string()` converts back, and `generic_string()` always uses `/` even on Windows, which is what a path written into a file or a log should use. A file stream's constructor accepts a `path` directly.

Two properties matter for a judged program. The **order** in which `directory_iterator` yields entries is unspecified — it is whatever the operating system returns — so a listing that must be deterministic collects the names into a `std::vector` and sorts them. And `fs::current_path()` is an absolute path that differs on every machine; never print it. Timestamps (`fs::last_write_time`) are the same kind of value.

```
namespace fs = std::filesystem;
fs::create_directory("inbox");
{ std::ofstream a("inbox/a.txt"); a << "hello\n"; }
std::vector<std::string> names;
for (const auto& entry : fs::directory_iterator("inbox")) {
    names.push_back(entry.path().filename().string());
}
std::sort(names.begin(), names.end());
for (const auto& n : names) std::cout << n << ' ' << fs::file_size(fs::path("
fs::remove_all("inbox");                                    // 2: the file ar
```

Errors: two flavours

Every filesystem function comes in two overloads. The plain one throws `fs::filesystem_error` (a `std::system_error` whose `what()` names the operation and the path) on failure — `file_size` of a missing file, `create_directory` where a file of that name exists. The overload with a trailing `std::error_code&` parameter reports through it instead and never throws: `auto size = fs::file_size(p, ec); if (ec) { /* ec.message() */ }`. The second form suits a tool that inspects many files and wants to keep going; the first suits code where a missing file is a bug. Module 15 lesson 5 has `std::error_code` in general.

Pitfalls

Mistake	What happens
`out.write(reinterpret_cast<const char*>(&str), sizeof str)`	Writes the string's handle; the characters are not in the file.
Reading a record with a different struct layout	Fields shift; values are garbage with no error.
Forgetting `std::ios::binary`	Fine on Linux; corrupt on Windows.
Assuming the file's byte order matches another machine	Little- and big-endian disagree; serialise explicitly.
Printing `directory_iterator` order as it comes	Unspecified order; sort first.
`fs::file_size` on a directory or missing path	Throws `filesystem_error`; use the `error_code` overload if that is expected.

Key takeaways

- `write` / `read` move raw bytes; cast with `reinterpret_cast<const char*>` and only for trivially copyable types — never a `std::string` or `std::vector`.
- Fixed-width records make a file an array: `seekg(i * sizeof(Record))`; `sizeof` includes padding (32 for the record above, on this platform).
- Always open binary files with `std::ios::binary`; byte order is the CPU's, so serialise explicitly for portable formats.
- `std::filesystem` gives `path` arithmetic, `exists`, `file_size`, `create_directory`, `remove_all` and `directory_iterator` — whose order you sort.
- Every function has a throwing overload and an `error_code` one; never print `current_path()` or timestamps in a judged program.

EXERCISE 1

Fixed-width record file

Read an integer `n`, then `n` records `<id> <score> <name>` (an `int`, a decimal, a one-word name of at most 15 characters), then an index `k` (0-based, less than `n`). Pack every record into

```
struct Record {
    std::int32_t id;
    double score;
    char name[16];
};
```

(zero-initialise it and copy the name into the array) and write them all to `records.bin`, opened with `std::ios::binary`, using `write` and `reinterpret_cast<const char*>`. Close it and print `bytes=<std::filesystem::file_size("records.bin")>`. Reopen the file in binary mode, `seekg` straight to record `k` — its offset is `k * sizeof(Record)` — `read` it and print `record[k]=<id> <name> <score with 1 decimal>`. Then `seekg(0)` and read every record in a `while (in.read(...))` loop to print `count=<n> mean=<mean score, 2 decimals> best=<name of the highest score, first on ties>`. Delete the file at the end.

Input: `n`, `n` records, `k`. **Output:** three lines.

```
3
1 91.5 ada
2 78.5 bob
3 91.5 cy
1
```

prints

```
bytes=96
record[1]=2 bob 78.5
count=3 mean=87.17 best=ada
```

INPUT 1

```
3
1 91.5 ada
2 78.5 bob
3 91.5 cy
1
```

OUTPUT 1

```
bytes=96
record[1]=2 bob 78.5
count=3 mean=87.17 best=ada
```

INPUT 2

```
1
42 100.0 zed
0
```

OUTPUT 2

```
bytes=32
record[0]=42 zed 100.0
count=1 mean=100.00 best=zed
```

REFERENCE SOLUTION

```cpp
#include <cstdint>
#include <filesystem>
#include <fstream>
#include <iomanip>
#include <iostream>
#include <string>
#include <type_traits>

struct Record {
    std::int32_t id;
    double score;
    char name[16];
};
static_assert(std::is_trivially_copyable_v<Record>);

int main() {
    const char* const path = "records.bin";
    int n;
    std::cin >> n;
    {
        std::ofstream out(path, std::ios::binary);
        for (int i = 0; i < n; ++i) {
            Record r{};                                          // zeroed, so the r
            std::string name;
            std::cin >> r.id >> r.score >> name;
            name.copy(r.name, sizeof(r.name) - 1);
            out.write(reinterpret_cast<const char*>(&r), sizeof r);
        }
    }
    int k;
    std::cin >> k;
    std::cout << "bytes=" << std::filesystem::file_size(path) << '\n';

    std::ifstream in(path, std::ios::binary);
    Record r{};
    in.seekg(static_cast<std::streamoff>(k) * static_cast<std::streamoff>(sizeof(Rec
    in.read(reinterpret_cast<char*>(&r), sizeof r);
    std::cout << std::fixed << std::setprecision(1);
    std::cout << "record[" << k << "]=" << r.id << ' ' << r.name << ' ' << r.score <

    in.clear();                                                  // in case the seek
    in.seekg(0);
    int count = 0;
    double sum = 0.0;
    double bestScore = 0.0;
    std::string best;
    while (in.read(reinterpret_cast<char*>(&r), sizeof r)) {
        ++count;
        sum += r.score;
```

```
        if (count == 1 || r.score > bestScore) {
            bestScore = r.score;
            best = r.name;
        }
    }
    in.close();
    std::filesystem::remove(path);
    std::cout << std::setprecision(2);
    std::cout << "count=" << count << " mean=" << (count > 0 ? sum / count : 0.0) <<
    return 0;
}
```

EXERCISE 2

Directory listing, sorted

Read an integer `n`, then `n` lines of `<filename> <content>` — a one-word file name (no spaces or slashes) followed by the rest of the line, which may be empty. Create a directory named `inbox` with `std::filesystem::create_directory` and write each file as `inbox/<filename>` holding the content followed by one newline. A repeated file name overwrites the earlier file.

Then list the directory with `std::filesystem::directory_iterator`, collecting each regular file's name and size, **sort the names** (the iterator's order is unspecified), and print `<name> <size>` per file, then `files=<count> total=<sum of sizes>`. Finally remove the whole directory with `std::filesystem::remove_all` and print `removed=<the number it returns>` — the files plus the directory itself.

Input: `n`, then `n` lines. **Output:** the sorted listing, then two lines.

```
3
b.txt hello world
a.txt hi
c.log
```

prints

```
a.txt 3
b.txt 12
c.log 1
files=3 total=16
removed=4
```

INPUT 1

```
3
b.txt hello world
a.txt hi
c.log
```

OUTPUT 1

```
a.txt 3
b.txt 12
c.log 1
files=3 total=16
removed=4
```

INPUT 2

```
1
notes.md line one
```

OUTPUT 2

```
notes.md 9
files=1 total=9
removed=2
```

REFERENCE SOLUTION

```cpp
#include <algorithm>
#include <cstdint>
#include <filesystem>
#include <fstream>
#include <iostream>
#include <string>
#include <utility>
#include <vector>

namespace fs = std::filesystem;

int main() {
    const fs::path dir = "inbox";
    int n;
    std::cin >> n;
    fs::create_directory(dir);
    for (int i = 0; i < n; ++i) {
        std::string name;
        std::string rest;
        std::cin >> name;
        std::getline(std::cin, rest);
        if (!rest.empty() && rest.front() == ' ') rest.erase(0, 1);
        std::ofstream out(dir / name);                          // closed at the end
        out << rest << '\n';
    }
    std::vector<std::pair<std::string, std::uintmax_t>> entries;
    for (const auto& entry : fs::directory_iterator(dir)) {
        if (entry.is_regular_file()) entries.emplace_back(entry.path().filename().st
    }
    std::sort(entries.begin(), entries.end());
    std::uintmax_t total = 0;
    for (const auto& [name, size] : entries) {
        std::cout << name << ' ' << size << '\n';
        total += size;
    }
    std::cout << "files=" << entries.size() << " total=" << total << '\n';
    std::cout << "removed=" << fs::remove_all(dir) << '\n';
    return 0;
}
```

QUIZ

1.

```
std::string s = "hello";
out.write(reinterpret_cast<const char*>(&s), sizeof s);
```

What ends up in the file?

A. The five characters `hello`

B. The `std::string` object's own bytes — a pointer, a size and a capacity — but not the characters

C. `hello` followed by a null byte

D. A compile error: `std::string` cannot be cast to `const char*`

2. `struct R { std::int32_t id; double score; };` — what is `sizeof(R)` on this platform, and why?

A. 12 — four plus eight

B. 16 — four bytes of padding after `id` so the `double` sits on an 8-byte boundary

C. 8 — the compiler packs the fields

D. It depends on the optimisation level

3. In what order does `std::filesystem::directory_iterator` yield the entries of a directory?

A. Alphabetical by file name

B. Creation order

C. Unspecified — whatever the operating system returns; sort the names yourself for a deterministic listing

D. Largest file first

4. `std::filesystem::file_size("reports")` where `reports` is a directory:

A. Returns the total size of the files inside

B. Returns 0

C. Throws `std::filesystem::filesystem_error`; the `std::error_code` overload reports the error without throwing

D. Returns `-1`

5. An `int` holding `1` is stored on x86-64 as the bytes `01 00 00 00`. This byte order is called…

A. Big-endian

B. Little-endian: the least significant byte comes first

C. Network order

D. Packed

6. What does `std::ios::binary` change when a file is opened on Linux?

A. Nothing on Linux; on Windows it disables the `\n` to `\r\n` translation that would corrupt binary data

B. It makes `>>` read raw bytes instead of formatted values

C. It disables buffering

D. It makes `read` and `write` available — they are absent in text mode

ANSWER KEY · BINARY I/O AND THE FILESYSTEM

1. B `write` copies `sizeof s` bytes starting at the object, and the object is a handle to heap memory. Only trivially copyable types can be written this way; for a string, write `s.data()` with `s.size()` bytes (and its length first).

2. B A `double` must be 8-byte aligned, so the compiler inserts padding after the 4-byte `int32_t`. The padding is part of the object and is written and read with it; the layout is fixed by the platform ABI, not by `-O2`.

3. C The standard makes no promise; the order comes from the underlying directory structure and differs between file systems. Collect the entries into a vector and sort before printing.

4. C `file_size` is defined for regular files only. Every filesystem function has a throwing overload and one taking a trailing `std::error_code&` that sets the code instead — the second is the one for code that expects failures.

5. B x86-64 is little-endian. Big-endian (network order) would store `00 00 00 01`. A binary file written with `write` uses the CPU's order, so a portable format serialises each integer byte by byte in a declared order.

6. A Binary mode only turns off newline translation, which Linux never does. `read` / `write` exist in every mode. Always specify it for binary data so the program behaves identically on Windows.

CHECKPOINT · 25 MIN

Checkpoint — Streams and files

This checkpoint covers the whole module: stream state and recovery from a bad token, the character-level API, the reading-pattern catalogue, file streams and open modes, string streams with `std::quoted` and `std::format`, binary records and `std::filesystem`.

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 does a stream's `operator bool` test, and why can `eof()` be true after a successful read?
- After `std::cin >> x` fails on `abc`, what is in `x`, where is `abc`, and which two steps get the stream reading?
- Why must `std::cin.ignore(...)` follow `std::cin >> n` before the first `std::getline`?
- What does `std::ofstream out("f.txt")` do to an existing file, and which mode adds to it instead?
- Why does `in.str(next)` on a used `std::istringstream` still fail, and what fixes it?
- Why is `sizeof` a struct of an `int32_t` and a `double` 16 rather than 12?
- Why must a `directory_iterator` listing be sorted before it is printed?

The three programs are a variable-field records reader, a `std::format` table with dynamic widths, and an inventory file written, measured, read back through a `std::map` and removed.

EXERCISE 1

Records with a variable field count

Read lines until the input ends. Each non-blank line is a record `<name> v1 v2 ... vk` with `k ⩾ 0` integer values after the name. For each record print `<name>: n=<k> sum=<s> max=<m>`, or `<name>: n=0` when there are no values. If any token after the name is not an integer, print `line <L>: malformed` instead, where `L` is the line's 1-based number counting every line, blank ones included. Skip blank lines silently. Finish with `records=<good> bad=<malformed>`.

Parse each line in its own `std::istringstream`; after the values loop stops, `eof()` on that stream tells a clean end from a bad token. Use `long long`.

Input: any number of lines. **Output:** one line per record or bad line, then the summary.

```
ada 3 5 9
bob

cy 7 x 2
dee -4 -6
```

prints

```
ada: n=3 sum=17 max=9
bob: n=0
line 4: malformed
dee: n=2 sum=-10 max=-4
records=3 bad=1
```

INPUT 1

```
ada 3 5 9
bob

cy 7 x 2
dee -4 -6
```

OUTPUT 1

```
ada: n=3 sum=17 max=9
bob: n=0
line 4: malformed
dee: n=2 sum=-10 max=-4
records=3 bad=1
```

INPUT 2

```
solo 100
```

OUTPUT 2

```
solo: n=1 sum=100 max=100
records=1 bad=0
```

REFERENCE SOLUTION

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

int main() {
    std::string line;
    int lineNo = 0;
    int good = 0;
    int bad = 0;
    while (std::getline(std::cin, line)) {
        ++lineNo;
        std::istringstream in(line);
        std::string name;
        if (!(in >> name)) continue;               // a blank line
        long long v = 0;
        long long sum = 0;
        long long max = 0;
        int count = 0;
        while (in >> v) {
            if (count == 0 || v > max) max = v;
            sum += v;
            ++count;
        }
        if (!in.eof()) {                           // the loop stopped on a token, r
            std::cout << "line " << lineNo << ": malformed\n";
            ++bad;
            continue;
        }
        ++good;
        if (count == 0) std::cout << name << ": n=0\n";
        else std::cout << name << ": n=" << count << " sum=" << sum << " max=" << ma
    }
    std::cout << "records=" << good << " bad=" << bad << '\n';
    return 0;
}
```

EXERCISE 2

A std::format table

Read an integer `n`, then `n` lines of `<name> <qty> <price>`. Print a table built with `std::format` and `std::format_to` only — no `setw`:

- the name column is left-aligned with a width of `max(4, longest name)`, passed as a dynamic width with `{:<{}}`;

`Qty` right-aligned in 5, `Price` right-aligned in 9 with two decimals, `Total` (`qty × price`) right-aligned in 10 with two decimals;

a header row `Item`, `Qty`, `Price`, `Total` in the same fields;

a rule of `=` as wide as the table (`width + 24`);

a last line with `Grand total` left-aligned in `width + 14` and the grand total right in 10 with two decimals.

Build the whole text into one `std::string` with `std::format_to(std::back_inserter(out), ...)` and print it once.

Input: `n`, then `n` lines. **Output:** `n + 3` lines.

```
2
pen 3 1.50
notebook 12 4.25
```

prints

```
Item      Qty    Price     Total
pen         3     1.50      4.50
notebook   12     4.25     51.00
================================
Grand total               55.50
```

INPUT 1

```
2
pen 3 1.50
notebook 12 4.25
```

OUTPUT 1

```
Item      Qty    Price     Total
pen         3     1.50      4.50
notebook   12     4.25     51.00
================================
Grand total               55.50
```

INPUT 2

```
1
ab 1 0.99
```

OUTPUT 2

```
Item  Qty    Price     Total
ab      1     0.99      0.99
============================
Grand total              0.99
```

REFERENCE SOLUTION

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

struct Row {
    std::string name;
    int qty = 0;
    double price = 0.0;
};

int main() {
    int n;
    std::cin >> n;
    std::vector<Row> rows;
    std::size_t width = 4;                                          // "Item"
    for (int i = 0; i < n; ++i) {
        Row r;
        std::cin >> r.name >> r.qty >> r.price;
        width = std::max(width, r.name.size());
        rows.push_back(r);
    }
    std::string out = std::format("{:<{}}{:>5}{:>9}{:>10}\n", "Item", width, "Qty",
    double grand = 0.0;
    for (const auto& r : rows) {
        const double total = r.qty * r.price;
        grand += total;
        std::format_to(std::back_inserter(out), "{:<{}}{:>5}{:>9.2f}{:>10.2f}\n", r.
    }
    std::format_to(std::back_inserter(out), "{}\n", std::string(width + 24, '='));
    std::format_to(std::back_inserter(out), "{:<{}}{:>10.2f}\n", "Grand total", widt
    std::cout << out;
    return 0;
}
```

EXERCISE 3

Inventory file round trip

Read lines of `<sku> <qty>` (a one-word SKU and an integer quantity, possibly negative) until the input ends. Write each pair to `inventory.txt` as `<sku> <qty>` on its own line through a `std::ofstream`; close it and print `bytes=<std::filesystem::file_size("inventory.txt")>`. Then reopen the file with a `std::ifstream`, read the pairs back with `>>`, and add each quantity into a `std::map<std::string, long long>` keyed by SKU. Print `<sku> <total>` for every SKU in map

order, then `skus=<count> units=<sum of all totals>`. Finally delete the file with `std::filesystem::remove` and print `exists=yes` or `exists=no` from `std::filesystem::exists`.

Input: any number of `sku qty` lines. **Output:** `bytes=`, the SKU lines, the summary, `exists=`.

```
bolt 10
nut 5
bolt 7
```

prints

```
bytes=21
bolt 17
nut 5
skus=2 units=22
exists=no
```

INPUT 1

```
bolt 10
nut 5
bolt 7
```

OUTPUT 1

```
bytes=21
bolt 17
nut 5
skus=2 units=22
exists=no
```

INPUT 2

```
washer 1
```

OUTPUT 2

```
bytes=9
washer 1
skus=1 units=1
exists=no
```

REFERENCE SOLUTION

```cpp
#include <filesystem>
#include <fstream>
#include <iostream>
#include <map>
#include <string>

namespace fs = std::filesystem;

int main() {
    const fs::path path = "inventory.txt";
    {
        std::ofstream out(path);
        std::string sku;
        long long qty;
        while (std::cin >> sku >> qty) out << sku << ' ' << qty << '\n';
    }
    std::cout << "bytes=" << fs::file_size(path) << '\n';
    std::map<std::string, long long> stock;
    {
        std::ifstream in(path);
        std::string sku;
        long long qty;
        while (in >> sku >> qty) stock[sku] += qty;
    }
    long long units = 0;
    for (const auto& [sku, total] : stock) {
        std::cout << sku << ' ' << total << '\n';
        units += total;
    }
    std::cout << "skus=" << stock.size() << " units=" << units << '\n';
    fs::remove(path);
    std::cout << "exists=" << (fs::exists(path) ? "yes" : "no") << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. `if (std::cin >> x)` is true when…

A. `eof()` is false

B. neither `failbit` nor `badbit` is set after the read

C. `good()` is true

D. `x` is non-zero

2.

```
int x = 9;
std::istringstream in("seven");
in >> x;
std::cout << x;
```

prints…

A. `9`

B. `0`

C. `7`

D. Nothing — the program terminates

3. Why does `while (!std::cin.eof()) { std::cin >> x; use(x); }` process one value too many?

A. `eof()` is set only after a read has tried to go past the end, so the loop enters once more, the read fails, and `use(x)` runs on a failed value

B. `eof()` is never set on `std::cin`

C. `>>` reads two tokens per call

D. The newline at the end of the input counts as a value

4. After `int n; std::cin >> n;`, what must happen before `std::getline(std::cin, line)` reads the first real line?

A. Nothing — `getline` skips leading newlines

B. `std::cin.clear()`

C. The rest of the count's line, including its newline, must be discarded — `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')`

D. `std::cin.sync()`

5. Which reading strategy can report "row 3 has 2 values, expected 4" for a matrix?

A. `std::cin >> m[r][c]` in two nested loops

B. Reading each row as a line with `std::getline` and counting the tokens in a `std::istringstream`

C. `std::cin.exceptions(std::ios::failbit)`

D. Reading the whole input as one string and splitting on spaces

6. Which open mode adds to an existing file rather than replacing it?

A. `std::ofstream out("log.txt");`

B. `std::ofstream out("log.txt", std::ios::trunc);`

C. `std::ofstream out("log.txt", std::ios::app);`

D. `std::ofstream out("log.txt", std::ios::in);`

7. A program writes `data.txt` through `std::ofstream out` and, in the same scope, opens `std::ifstream in("data.txt")` to read it back. The reader sees an empty file. Why?

A. Two streams cannot open the same file

B. The writer's buffer has not been flushed yet; scope or `close()` the writer before opening the reader

C. `std::ifstream` reads from the start of the file only after `seekg(0)`

D. The file is locked until the program exits

8. What is the correct way to reuse a `std::istringstream in` for a new line of text?

A. `in.str(next);`

B. `in.clear(); in.str(next);`

C. `in.str(next); in.seekg(0);`

D. `in.reset(next);`

9. `std::istringstream in("\"New York\" 8"); std::string city; int n; in >> std::quoted(city) >> n;` leaves `city` as…

A. `"New`

B. `New York` — the quotes are removed and the space kept

C. `"New York"` with the quotes

D. The read fails: `>>` stops at the space

10. What does `std::format("[{:>6.1f}]", 3.14159)` produce?

A. `[3.1]`

B. `[ 3.1]`

C. `[3.1 ]`

D. `[003.14]`

11.

```
std::vector<int> v = {1, 2, 3};
out.write(reinterpret_cast<const char*>(&v), sizeof v);
```

What is written?

A. The three integers, 12 bytes

B. The vector's handle — three pointers, 24 bytes on this platform — and none of the elements

C. The three integers followed by a length

D. Nothing: a compile error

12. `struct P { char tag; double value; };` — `sizeof(P)` on this platform is…

A. 9

B. 16 — seven bytes of padding after `tag` align the `double` to 8

C. 12

D. 8

13. A judged program lists a directory with `std::filesystem::directory_iterator` and prints the names as they come. Why might it fail on the judge and pass locally?

A. The judge has no filesystem

B. The iteration order is unspecified and differs between file systems; the names must be sorted before printing

C. `directory_iterator` skips files created by the same program

D. The judge runs with a different locale

14. `std::filesystem::file_size(p, ec)` with a `std::error_code ec` on a missing file…

A. Throws `filesystem_error`

B. Sets `ec` and returns `static_cast<std::uintmax_t>(-1)` without throwing

C. Returns 0 and leaves `ec` clear

D. Creates the file

ANSWER KEY · CHECKPOINT — STREAMS AND FILES

1. B The stream's `operator bool` is `!fail()`. `eofbit` on its own does not make it false, so the last token of an input without a trailing newline still counts as read.

2. B Characters were examined and could not form an `int`, so the C++11 rule sets `x` to `0` and `failbit`. (Only when the stream is already at the end before the read does the variable keep its old value.)

3. A After the last value is read, `eofbit` is still clear (unless there was no trailing newline). The loop runs again, the read fails, and the body sees `0`. `while (std::cin >> x)` tests the read itself.

4. C `>>` leaves the newline after `n` in the buffer, and `getline` returns the empty remainder of that line. An `ignore` through the newline is the fix; `clear()` touches the state bits only.

5. B Only a reader that sees where a line ends can count what was on it. Token-wise reading takes `R × C` values wherever they fall and never notices a short row.

6. C `app` sends every write to the end of the existing contents. The default (`out` alone) and `trunc` both empty the file on open; `in` on an `ofstream` is not a sensible mode.

7. B Output sits in the `ofstream`'s buffer until a flush; the destructor performs one. Put the writer in its own block so it is destroyed before the reader is constructed.

8. B A stream that was read to its end has `eofbit` and `failbit` set; `str(...)` swaps the buffer but leaves the state, so the next read fails. `clear()` resets the state. (`reset` does not exist.)

9. B `std::quoted` on input reads a double-quoted string as a single token, strips the quotes and undoes any `\"` escapes. Plain `>> city` would have stopped at the space and left `"New`.

10. B `.1f` gives one decimal (`3.1`), and `>6` right-aligns it in a field of six, so three spaces of padding precede it. Left alignment would need `<`; zero padding would need a `0` before the width.

11. B A `std::vector` object is a small handle to heap storage. To write the elements use `v.data()` and `v.size() * sizeof(int)`; only trivially copyable types may be written as their own bytes.

12. B The `double` needs 8-byte alignment, so the compiler pads after the 1-byte `char`. The padding travels with the record in `write` / `read`; putting the `double` first would shrink nothing here (the struct's size is still rounded to its alignment) but it does matter for structs with several small members.

13. B Nothing about the order is guaranteed. Collect the entries and sort them — the same rule as for `std::unordered_map` iteration in Module 13.

14. B The overload with a trailing `std::error_code&` never throws: it reports the failure through `ec` and returns the sentinel value. The overload without it throws `std::filesystem::filesystem_error` instead.
