Module 5 of 20

Strings

std::string as a value type, the find family and slicing, characters and numeric conversions, parsing input, formatting output with iomanip and std::format, and std::string_view.

Text is the shape almost every input arrives in and almost every output leaves in, and C++ handles it with three types that look alike and behave differently: the `const char*` literal inherited from C, the owning `std::string`, and the non-owning `std::string_view`. Knowing which one you hold — and who owns the characters — decides whether a program is correct, fast, or undefined.

The module works through `std::string` as a value type; the `find` family, `npos` and the slicing calls; characters as small integers with the `<cctype>` cast that keeps them well-defined; the `stoi` family and `std::from_chars` for text to number; the parsing patterns every exercise in the track relies on; `<iomanip>` and `std::format` for exact output; and finally `std::string_view` with the lifetime rule that makes it safe.

The exercises are text programs judged on exact output: reversing and comparing, counting and replacing, splitting paths and CSV rows, validating tokens that may not be numbers, printing aligned tables, tokenising with views, and — in the checkpoint — a word-frequency count, a fixed-width report and a run-length codec.

CONTENTS

1	std::string — a value type for text	13 min
2	Searching and slicing	14 min
3	Characters and conversions	14 min
4	Parsing input	14 min
5	Formatting output	14 min
6	std::string_view — a window onto text	13 min
✓	Checkpoint — Strings	25 min

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

LESSON 1 · 13 MIN

std::string — a value type for text

C's text is an array of `char` ending in a zero byte — no length, no bounds, no ownership. C++ keeps that representation for literals and C APIs but builds `std::string` on top: an owning value type that knows its length, grows on demand, copies deeply, compares by content and frees its memory on scope exit. This lesson settles what a `std::string` is, how to build, index, grow and compare one, and where the C string boundary still shows through.

A string is a value

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

int main() {
    std::string a = "kairo";
    std::string b = a;          // a second string with its own copy of the ch
    b += "!";
    std::cout << a << ' ' << b << '\n';   // kairo kairo!
    std::cout << (a == b) << '\n';        // 0: == compares content, and it di
    return 0;
}
```

`b = a` copies the characters; changing `b` cannot touch `a`. That is **value semantics**, the rule an `int` follows and the opposite of Java or Python, where two names can share one string object. `==` compares characters, never addresses. The string owns its buffer — with this platform's libstdc++ up to 15 characters live inside the object (the *small string optimisation*), anything longer on the heap — and its destructor frees it the moment the string goes out of scope. That is RAII, which Module 7, Memory, ownership and RAII, treats in full.

Constructing one

```
std::string empty;                          // ""
std::string s1 = "hello";                   // copies the literal's characters
std::string s2("hello", 3);                 // "hel"   - the first 3 characters
std::string s3(5, '-');                     // "-----" - count, then character
std::string s4(s1, 1);                      // "ello"  - a copy of s1 from index
std::string s5(s1, 1, 3);                   // "ell"   - from index 1, 3 charact
std::string s6 = std::to_string(42);        // "42"
std::string s7 = s1 + ", " + s6;            // "hello, 42"
```

Two forms that do not do what they look like. `std::string s{5, '-'}` with braces picks the `std::initializer_list<char>` constructor and builds a two-character string (code 5, then `'-'`); use parentheses for count-and-character. And `std::string s = 'a';` does not compile — there is no conversion from a single `char` — although `s = 'a';` on an existing string does.

Size, emptiness and indexing

```
std::string word = "delta";
std::cout << word.size() << ' ' << word.length() << '\n';   // 5 5 - identical
std::cout << word.empty() << '\n';                          // 0
std::cout << word[0] << word.back() << '\n';                // da
word[0] = 'D';                                              // elements are wr
word.at(10) = 'x';                                          // throws std::out
```

`size()` returns `std::size_t`, an unsigned type (Module 2, Fundamental types), so index loops use `std::size_t i` — an `int` draws a signed/unsigned warning — and `word.size() - 1` on an empty string is not `-1` but the largest `size_t` there is, the unsigned countdown trap from Module 3, Control flow.

`operator[]` does no bounds checking; an index past the end is undefined behaviour, with one carve-out: reading `s[s.size()]` yields `'\0'` (writing there is not allowed). `at()` checks and throws `std::out_of_range` — the right call when the index came from input. `front()` and `back()` are the first and last characters, undefined on an empty string, so check `empty()` first.

Growing with + and +=

```
std::string greeting = "Hello";
greeting += ", ";                 // append a C string
greeting += name;                 // append a std::string
greeting += '!';                  // append one character
greeting.push_back('\n');         // the same, for one character
greeting.append(3, '.');          // "..."
std::string line = greeting + " again";   // + builds a new string; the operar
```

`+=` appends in amortised constant time per character — the buffer grows geometrically, like a `std::vector` — so building output character by character is fine. `s = s + c` in a loop is not: every iteration copies the whole string, and a 100 000-character build becomes billions of copied bytes. Call `s.reserve(n)` when you know the final size.

Three expressions that look like concatenation and are not:

```
std::string bad1 = "abc" + "def";     // error: two const char* cannot be adde
const char* bad2 = "abc" + 1;         // pointer arithmetic: points at "bc"
std::string s = "x"; s += 65;         // appends 'A' - += accepts a char, and
```

The first is refused; the other two compile, because a literal is a `const char*` and `+` on a pointer is arithmetic (Module 6, Arrays, pointers and references). Append a number's digits with `std::to_string`: `s += std::to_string(65)` gives `"x65"`. Once one operand is a `std::string`, `+` does the expected thing with a literal or a `char` on the other side: `std::string("a") + "b" + 'c'` is `"abc"`.

Comparing

```
std::string a = "apple", b = "Banana", c = "app";
std::cout << (b < a) << ' ' << (c < a) << ' ' << (a == "apple") << '\n';   //
std::cout << a.compare(b) << '\n';                                          //
```

All six comparison operators compare **lexicographically by character code**, treating each `char` as unsigned: digits (48–57) sort before capitals (65–90), which sort before lower-case letters (97–122), so `"Banana" < "apple"`. A prefix sorts before the longer string, so `"app" < "apple"`. This is the order `std::sort` and `std::map` use (Modules 13 and 14); for dictionary order that ignores case, normalise both sides first — lesson 2 shows how. `compare()` returns a negative, zero or positive `int` in the C tradition.

The trap: `if ("apple" < "banana")` compiles, warns, and compares two addresses. Both operands are pointers, and no operator overload is involved. Make one side a `std::string` and the content comparison you meant is what you get.

---

std::string and C strings

A **C string** is a `const char*` pointing at characters that end with a `'\0'` byte; every literal is one. `std::string` stores its own length, so it may even contain `'\0'` in the middle and `size()` still counts correctly, where C's `std::strlen` would stop early.

```
std::string path = "/tmp/out.txt";
std::FILE* f = std::fopen(path.c_str(), "w");   // C APIs want a const char*
```

`c_str()` returns a pointer into the string's own buffer, null-terminated, valid until the string is modified or destroyed — never keep it past that. Constructing a `std::string` from a literal copies every character, which is why a function taking `const std::string&` allocates when called with a literal longer than the small buffer; lesson 6's `std::string_view` is the parameter type that avoids the copy.

Iterating by character

```
for (char c : word) std::cout << static_cast<int>(c) << ' ';      // the codes
for (char& c : word) c = static_cast<char>(std::toupper(static_cast<unsigned c
std::string reversed(word.rbegin(), word.rend());                  // "ATLED"
```

`for (char c : s)` copies each element; `char&` changes it in place (Module 3, Range-for and iteration). The `unsigned char` cast around `std::toupper` is not decoration — lesson 3 explains it. The iterator-pair constructor builds a string from any range of characters, and reverse iterators make reversing a one-liner. A `std::string` is a sequence of *bytes*, not letters: `"é"` in a UTF-8 source is two `char`s and `size()` says 2; for the ASCII input this track's judge sends, byte and letter coincide.

Pitfalls

`int n = s.size() - 1;` on an empty string wraps to the largest `size_t` before the conversion; test `empty()` first.

`s[s.size()]` is a legal read of `'\0'`; `s[s.size() + 1]` and any write past the end are undefined behaviour.

`std::string s = 'a';` does not compile; `s += 65` compiles and appends `'A'`.

`"a" + "b"` is an error, `"abc" + 1` is pointer arithmetic, and `"a" < "b"` compares addresses.

Passing a `std::string` by value copies it; take `const std::string&` (Module 4, Functions) or `std::string_view` (lesson 6).

Key takeaways

`std::string` is an owning value type: copies are independent, `==` compares content, the destructor frees the buffer.

`size()` is unsigned; index with `std::size_t`; `[]` is unchecked, `at()` throws.

`+=` is cheap and amortised; `s = s + x` in a loop is quadratic.

Comparisons are lexicographic by unsigned character code — capitals before lower case, a prefix before the longer string.

A literal is a `const char*`: adding two is an error, comparing two compares addresses, and `c_str()` is valid only until the next modification.

EXERCISE 1

Mirror

Read one line and print three lines: the line reversed, then `length=<n>`, then `palindrome` if the line reads the same backwards (an exact comparison — case and spaces count) or `not palindrome` otherwise.

Build the reversed string with `+=` in a loop that walks the indices from the back, and compare the two strings with `==`. An empty line reverses to an empty line and is a palindrome.

Input: one line (it may be empty or contain spaces). **Output:** three lines.

Example: input `level` → output

```
level
length=5
palindrome
```

INPUT 1

```
level
```

OUTPUT 1

```
level
length=5
palindrome
```

INPUT 2

```
kairo
```

OUTPUT 2

```
oriak
length=5
not palindrome
```

REFERENCE SOLUTION

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

int main() {
    std::string line;
    std::getline(std::cin, line);

    std::string reversed;
    reversed.reserve(line.size());
    for (std::size_t i = line.size(); i > 0; --i) {
        reversed += line[i - 1];
    }

    std::cout << reversed << '\n';
    std::cout << "length=" << line.size() << '\n';
    std::cout << (reversed == line ? "palindrome" : "not palindrome") << '\n';
    return 0;
}
```

EXERCISE 2

Smallest and largest

Read an integer `n` (at least 1) followed by `n` words — whitespace-separated tokens, possibly spread over several lines. Using only `std::string`'s own comparison operators, print the smallest word, the largest word, and how many of the `n` words are equal to the first word read (counting the first word itself).

Input: `n`, then `n` words. **Output:** three lines: `first=<smallest>`, `last=<largest>`, `same_as_first=<count>`.

Example: input `4 pear apple fig banana` → output

```
first=apple
last=pear
same_as_first=1
```

Remember the order is by character code: `Zebra` sorts before `apple`, and `10` before `9`.

INPUT 1

```
4 pear apple fig banana
```

OUTPUT 1

```
first=apple
last=pear
same_as_first=1
```

INPUT 2

```
5
Zebra apple Apple zebra app
```

OUTPUT 2

```
first=Apple
last=zebra
same_as_first=1
```

REFERENCE SOLUTION

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

int main() {
    int n = 0;
    std::cin >> n;

    std::string first;
    std::cin >> first;
    std::string smallest = first;
    std::string largest = first;
    int same = 1;

    for (int i = 1; i < n; ++i) {
        std::string word;
        std::cin >> word;
        if (word < smallest) smallest = word;
        if (largest < word) largest = word;
        if (word == first) ++same;
    }

    std::cout << "first=" << smallest << '\n';
    std::cout << "last=" << largest << '\n';
    std::cout << "same_as_first=" << same << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```cpp
std::string a = "hi";
std::string b = a;
b += "!";
std::cout << a << b;
```

A. `hihi!`

B. `hi!hi!`

C. `hi!`

D. It does not compile

2. Which line does **not** compile?

A. `std::string s = "a" + "b";`

B. `std::string s = std::string("a") + "b";`

C. `std::string s = "a"; s += 'b';`

D. `std::string s(3, 'a');`

3. `std::string s = "abc";` — what happens for `s.at(3)` and `s[3]`?

A. Both are undefined behaviour

B. `at(3)` throws `std::out_of_range`; reading `s[3]` yields `'\0'`

C. Both return `'\0'`

D. `s[3]` throws; `at(3)` returns `'\0'`

4. Which comparison of `std::string` values is **false**?

A. `std::string("Zebra") < std::string("apple")`

B. `std::string("app") < std::string("apple")`

C. `std::string("10") < std::string("9")`

D. `std::string("apple") < std::string("Apple")`

5. What does this print?

```
std::string s = "x";
s += 65;
std::cout << s;
```

A. `x65`

B. `xA`

C. It does not compile

D. `x` followed by the byte 65 as a number

6. Which statement about `c_str()` is correct?

A. It returns a copy of the characters that the caller must `delete[]`

B. It returns a null-terminated `const char*` into the string's own buffer, valid until the string is modified or destroyed

C. It converts the string to a `std::string_view`

D. It is only available on `const` strings

ANSWER KEY · STD::STRING — A VALUE TYPE FOR TEXT

1. A `b = a` copies the characters, so `b += "!"` changes only `b`: `a` is still `hi` and `b` is `hi!`. A `std::string` has value semantics — there is no shared object as there would be in Java.

2. A Two string literals are two `const char*`, and there is no `+` for pointers. Once one operand is a `std::string` the overloaded `+` applies, `+=` accepts a `char`, and `(3, 'a')` is the count-and-character constructor.

3. B `at()` bounds-checks and throws. `operator[]` does not check, but the standard allows *reading* `s[s.size()]` and defines it as `'\0'`; writing there, or any index beyond, is undefined behaviour.

4. D Comparison is by character code: `'A'` (65) is less than `'a'` (97), so `"Apple"` sorts before `"apple"` and `"Zebra"` before `"apple"`. A prefix sorts first, and `'1'` is less than `'9'`, so `"10" < "9"`.

5. B `operator+=` has a `char` overload and `65` converts to the `char` `'A'`. To append the digits use `s += std::to_string(65)`.

6. B `c_str()` is the bridge to C APIs: a pointer into the string's storage with a terminating `'\0'`. Nothing is copied and nothing must be freed — but the pointer is invalidated by the next modification.

LESSON 2 · 14 MIN

Searching and slicing

Most string work is finding a position and cutting at it: the extension after the last dot, the value after the `=`, every occurrence of a word. `std::string` gives you a family of `find` functions that all answer with an index, the sentinel `npos` for "not there", and `substr`, `erase`, `insert` and `replace` to act on the index. This lesson settles how those pieces fit, the one idiom that keeps `npos` from turning into a bug, and the two loops — count every occurrence, replace every occurrence — that every text program eventually needs.

The find family

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

int main() {
    std::string s = "one two three two";
    std::cout << s.find("two") << '\n';              // 4  - first occurrence
    std::cout << s.rfind("two") << '\n';             // 14 - last occurrence
    std::cout << s.find("two", 5) << '\n';           // 14 - first occurrence
    std::cout << s.find('t') << '\n';                // 4  - a single charac
    std::cout << s.find_first_of("aeiou") << '\n';   // 0  - first character
    std::cout << s.find_last_of("aeiou") << '\n';    // 16
    std::cout << s.find_first_not_of("one ") << '\n'; // 4  - first character
    return 0;
}
```

`find` searches for a whole substring (or one character) left to right, `rfind` right to left; both take an optional start position. The `_of` variants treat their argument as a *set* of characters and find the first (or last) character that is in the set — or, with `_not_of`, the first that is not. `find_first_not_of(" \t")` is how you skip leading whitespace; `find_last_not_of(" \t")` finds where the trailing whitespace begins. Every one of them returns a `std::size_t` index.

npos and the one idiom

When nothing matches, every `find` returns `std::string::npos`, a `static` constant equal to the largest `std::size_t` — `static_cast<std::size_t>(-1)`, which is 18 446 744 073 709 551

615 on this platform. The idiom is to compare against it, and nothing else:

```
const std::size_t at = s.find("three");
if (at ≠ std::string::npos) {
    std::cout << "found at " << at << '\n';
}
```

Two ways to get this wrong. `if (s.find("x") ⩾ 0)` is always true — `size_t` is unsigned, so *every* value is at least zero, and the compiler warns. `int at = s.find("x");` turns `npos` into `-1`, and a later `at < 0` test catches it only by accident of the conversion. Keep the result in `std::size_t` (or `auto`) and compare with `npos`.

---

substr

```
std::string s = "codekairo.cpp";
std::string ext  = s.substr(10);        // "cpp"       - from index 10 to the
std::string stem = s.substr(0, 9);      // "codekairo" - from index 0, at most
std::string tail = s.substr(9, 100);    // ".cpp"      - a count past the end
std::string boom = s.substr(20);        // throws std::out_of_range: pos > siz
```

`substr(pos, count)` copies `count` characters from `pos` into a **new string** — O(count), and it allocates. The count may run past the end and is clamped; the position may not, and `pos > size()` throws (`pos == size()` gives `""`). Combine it with `find` to cut on a delimiter:

```
std::string entry = "name=Ada Lovelace";
const std::size_t eq = entry.find('=');
if (eq ≠ std::string::npos) {
    std::string key = entry.substr(0, eq);        // "name"
    std::string value = entry.substr(eq + 1);     // "Ada Lovelace"
}
```

Lesson 6's `std::string_view::substr` does the same slice in O(1) without copying; when you only need to *look* at the slice, that is the tool.

Counting every occurrence

```
int count_of(const std::string& text, const std::string& needle) {
    int count = 0;
    for (std::size_t pos = text.find(needle); pos ≠ std::string::npos; pos =
        ++count;
    }
    return count;
}
```

The loop restarts each search *after* the match it just found — `pos + needle.size()` — so `count_of("aaaa", "aa")` is 2, the non-overlapping count; restart at `pos + 1` and overlapping matches count too: 3. Interview questions usually mean non-overlapping. The needle must not be empty: `find("")` matches at every position and the loop never advances.

---

erase, insert and replace

```
std::string s = "hello world";
s.erase(5, 1);              // "helloworld"  - erase 1 character at index 5
s.erase(5);                 // "hello"       - erase from index 5 to the end
s.insert(0, ">> ");         // ">> hello"    - insert before index 0
s.replace(3, 5, "HELLO");   // ">> HELLO"    - replace 5 characters at index 3
s.replace(3, 5, "hi");      // ">> hi"       - the replacement may be a differ
```

All three take a position (and, for `erase` and `replace`, a count), modify the string in place and return a reference to it, so calls chain. C++20 adds two free functions for whole-string cases: `std::erase(s, ' ')` removes every space and `std::erase_if(s, pred)` every character the predicate accepts.

Replacing every occurrence is the loop that most often turns into an infinite one:

```
std::string replace_all(std::string text, const std::string& from, const std::
    std::size_t pos = 0;
    while ((pos = text.find(from, pos)) ≠ std::string::npos) {
        text.replace(pos, from.size(), to);
        pos += to.size();               // continue AFTER the inserted text
    }
    return text;
}
```

Restarting the search at `pos` would find the text just inserted whenever `to` contains `from` — replacing `"a"` with `"aa"` never terminates. Advancing by `to.size()` skips the insertion and handles a shorter or empty replacement too. `text` is taken by value on purpose: the function modifies its own copy and the caller's string stays intact.

---

starts_with and ends_with

```
std::string file = "report.final.pdf";
bool pdf = file.ends_with(".pdf");        // true   (C++20)
bool tmp = file.starts_with("tmp_");      // false
bool dot = file.starts_with('.');         // a single character works too
```

Before C++20 the idiom was `file.compare(0, 4, "tmp_") == 0` or `file.rfind("tmp_", 0) == 0`, both of which you will still meet. `contains()` is C++23 and not available on this track's runtime — write `file.find("final") ≠ std::string::npos`.

Trimming

```
std::string trim(const std::string& s) {
    const std::size_t start = s.find_first_not_of(" \t");
    if (start == std::string::npos) return "";              // all whitespace
    const std::size_t end = s.find_last_not_of(" \t");
    return s.substr(start, end - start + 1);
}
```

The `npos` check must come first: on an all-whitespace line `find_first_not_of` fails, and `substr(npos, …)` would throw. Lesson 4 applies `trim` to every field it parses.

Case-insensitive comparison

There is no case-insensitive `==`; the honest approach is to normalise both sides and compare the results:

```cpp
#include <cctype>

std::string lower(std::string s) {
    for (char& c : s) c = static_cast<char>(std::tolower(static_cast<unsigned
    return s;
}

bool same_word(const std::string& a, const std::string& b) {
    return lower(a) == lower(b);
}
```

This is ASCII-only — `std::tolower` knows nothing of `'É'` — and the `unsigned char` cast is required, as lesson 3 explains. The same normalisation makes a case-insensitive search: lower-case the haystack once, lower-case the needle, then `find`.

Pitfalls

`s.find(x) ⩾ 0` is always true; compare with `std::string::npos`, and keep the result in `std::size_t`.

`substr(pos, n)` clamps `n` but throws when `pos > size()`; check the `find` result before slicing with it.

A replace-all loop must advance past the *replacement*, not the match, or `"a"` to `"aa"` never ends.

`find("")` succeeds at every position; reject an empty needle before counting or replacing.

`contains` is C++23; on C++20 write `find(...) ≠ npos`.

Key takeaways

`find` / `rfind` locate substrings; the `_of` / `_not_of` variants locate characters from a set; all return an index or `npos`.

`npos` is the largest `size_t`; the only correct test is `≠ std::string::npos`.

`substr` copies and is O(n); the count is clamped, the position is checked.

`erase`, `insert`, `replace` edit in place; C++20 `std::erase` and `std::erase_if` clear characters across the whole string.

Count occurrences by restarting the search after the match; replace all by restarting after the replacement.

`starts_with` / `ends_with` are C++20; normalise to lower case for case-insensitive comparison.

EXERCISE 1

Count and replace

Read three lines: a text, a needle (never empty) and a replacement (possibly empty). Print `count=<n>`, the number of **non-overlapping** occurrences of the needle in the text scanning left to right, and then the text with every occurrence replaced by the replacement.

Count with a `find` loop that restarts after each match; replace with a loop that restarts *after the inserted replacement* — the replacement may contain the needle.

Input: three lines. **Output:** two lines.

Example: input

```
the cat sat on the mat
at
AT
```

→ output

```
count=3
the cAT sAT on the mAT
```

INPUT 1

```
the cat sat on the mat
at
AT
```

OUTPUT 1

```
count=3
the cAT sAT on the mAT
```

INPUT 2

```
banana
ana
-
```

OUTPUT 2

```
count=1
b-na
```

REFERENCE SOLUTION

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

int main() {
    std::string text, needle, replacement;
    std::getline(std::cin, text);
    std::getline(std::cin, needle);
    std::getline(std::cin, replacement);

    int count = 0;
    for (std::size_t pos = text.find(needle); pos != std::string::npos; pos = text.f
        ++count;
    }

    std::size_t pos = 0;
    while ((pos = text.find(needle, pos)) != std::string::npos) {
        text.replace(pos, needle.size(), replacement);
        pos += replacement.size();
    }

    std::cout << "count=" << count << '\n';
    std::cout << text << '\n';
    return 0;
}
```

EXERCISE 2

Path parts

Read one line holding a file path such as `/home/ada/notes.final.txt` and split it with `rfind` and `substr`. Print five lines:

- `dir=` the part before the last `/`, or `(none)` if there is no slash;
- `file=` the part after the last `/` (the whole path if there is no slash);
- `stem=` and `ext=`: if the file name contains a `.` that is not its first character, `ext` is everything after the last such `.` and `stem` everything before it; otherwise `stem` is the whole file name and `ext=(none)`;
- `hidden=yes` if the file name starts with `.` (use `starts_with`), else `hidden=no`.

The path never ends with `/`.

Input: one line. **Output:** five lines.

Example: input `/home/ada/notes.final.txt` → output

```
dir=/home/ada
file=notes.final.txt
stem=notes.final
ext=txt
hidden=no
```

INPUT 1

```
/home/ada/notes.final.txt
```

OUTPUT 1

```
dir=/home/ada
file=notes.final.txt
stem=notes.final
ext=txt
hidden=no
```

INPUT 2

```
archive.tar.gz
```

OUTPUT 2

```
dir=(none)
file=archive.tar.gz
stem=archive.tar
ext=gz
hidden=no
```

REFERENCE SOLUTION

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

int main() {
    std::string path;
    std::getline(std::cin, path);

    std::string dir = "(none)";
    std::string file = path;
    const std::size_t slash = path.rfind('/');
    if (slash ≠ std::string::npos) {
        dir = path.substr(0, slash);
        file = path.substr(slash + 1);
    }

    std::string stem = file;
    std::string ext = "(none)";
    const std::size_t dot = file.rfind('.');
    if (dot ≠ std::string::npos && dot ≠ 0) {
        stem = file.substr(0, dot);
        ext = file.substr(dot + 1);
    }

    std::cout << "dir=" << dir << '\n';
    std::cout << "file=" << file << '\n';
    std::cout << "stem=" << stem << '\n';
    std::cout << "ext=" << ext << '\n';
    std::cout << "hidden=" << (file.starts_with('.') ? "yes" : "no") << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```cpp
std::string s = "banana";
std::cout << s.find("na") << ' ' << s.rfind("na") << ' ' << s.find("na", 3);
```

A. `2 4 4`

B. `2 4 2`

C. `3 5 5`

D. `2 2 4`

2. Why is `if (s.find("x") ⩾ 0)` wrong?

A. `find` returns `int`, so the comparison overflows

B. `find` returns `std::size_t`, which is unsigned, so the condition is always true — even `npos` is `⩾ 0`

C. It does not compile: `npos` cannot be compared with `0`

D. It is correct — `npos` is `-1`

3. What does this print?

```
std::string s = "hello";
s.erase(1, 2);
s.insert(0, "[");
std::cout << s;
```

A. `[hlo`

B. `[helo`

C. `[hello`

D. `[llo`

4. For `std::string s = "abcdef";`, what does `s.substr(2, 100)` do?

A. Throws `std::out_of_range` because the count exceeds the size

B. Returns `"cdef"` — a count past the end is clamped

C. Undefined behaviour

D. Returns `"cd"`

5. A replace-all loop calls `pos = text.find(from, pos)` and, after `text.replace(pos, from.size(), to)`, searches again from the same `pos`. Replacing `"a"` with `"aa"`…

A. Works, because `replace` moves `pos` forward

B. Never terminates — the search finds the `a` just inserted, replaces it again, and so on

C. Throws `std::out_of_range`

D. Replaces only the first occurrence

6. Which statement about C++20 and this track's runtime is correct?

A. `starts_with`, `ends_with` and `contains` are all available

B. `starts_with` and `ends_with` are available; `contains` is C++23 and is not

C. None of them exist — use `rfind(x, 0) == 0`

D. `contains` is available but `starts_with` is not

7. `trim` calls `s.find_first_not_of(" \t")` on a line that is entirely spaces and passes the result straight to `substr`. What happens?

A. It returns an empty string

B. It returns the original line

C. `substr` throws `std::out_of_range` because the position is `npos`

D. Undefined behaviour

ANSWER KEY · SEARCHING AND SLICING

1. A `find` returns the first match, at index 2; `rfind` the last, at 4; `find("na", 3)` starts searching at index 3 and finds the one at 4.

2. B `npos` is the largest `size_t`, not a negative number. Every `size_t` is at least zero, so the branch is always taken; compare with `std::string::npos` instead.

3. A `erase(1, 2)` removes two characters starting at index 1 (`el`), leaving `hlo`; `insert(0, "[")` puts the bracket in front.

4. B Only the *position* is checked (`pos > size()` throws); the count is silently clamped to what remains. `s.substr(20)` would throw.

5. B Each replacement inserts a new match at `pos`. Advance by `to.size()` after every replacement so the search continues after the inserted text.

6. B `starts_with` / `ends_with` were added in C++20. `std::string::contains` arrived in C++23, so on a C++20 compiler write `find(x) != std::string::npos`.

7. C `find_first_not_of` finds no such character and returns `npos`, and `substr(npos)` throws because the position exceeds the size. Test for `npos` first and return `""`.

LESSON 3 · 14 MIN

Characters and conversions

A `char` is a number that the stream prints as a letter. Hold on to that and everything here follows: why `'7' - '0'` is 7, why `c + 1` is an `int`, why `<cctype>` needs an `unsigned char` cast most tutorials omit, and how text becomes a number and back — `std::to_string`, the `std::stoi` family that throws, and C++17's `std::from_chars` that does not. Parsing a token that might not be a number is where text programs fail their hidden cases; this lesson gives two correct ways to do it.

char is a small integer

```
char c = 'A';
std::cout << c << ' ' << static_cast<int>(c) << ' ' << c + 1 << '\n';   // A
std::cout << static_cast<char>(c + 1) << '\n';                          // B
```

`'A'` is the integer 65 with type `char`. Arithmetic promotes a `char` to `int` (Module 2, Integers and bits), so `c + 1` is the `int` 66 and prints as digits; cast back to `char` to print a letter. On this platform `char` is signed, −128 to 127, so bytes above 127 — present the moment input is not ASCII — are *negative*. That fact is behind the cast in the next section.

cctype and the unsigned char cast

Function	True for
`std::isdigit(c)`	`'0'` – `'9'`
`std::isalpha(c)`	letters
`std::isalnum(c)`	letters or digits
`std::isspace(c)`	space, tab, newline, carriage return, form feed, vertical tab
`std::isupper(c)` / `std::islower(c)`	capitals / lower case
`std::ispunct(c)`	printable, not alphanumeric, not space
`std::toupper(c)` / `std::tolower(c)`	returns the converted character (as `int`)

Every function takes and returns an `int`, and the argument must be representable as `unsigned char` or be `EOF`. A negative `char` is neither, and passing one is undefined behaviour — a crash on some libraries, silently wrong on others. Cast first:

```
#include <cctype>

bool is_digit(char c) { return std::isdigit(static_cast<unsigned char>(c)) ≠
char upper(char c)    { return static_cast<char>(std::toupper(static_cast<uns:
```

Two casts for the case functions: `unsigned char` on the way in, `char` on the way out because they return `int`. Wrap them once in helpers and the rest of the program stays clean. The classification functions return a non-zero `int`, not `bool` — hence the `≠ 0`.

Digit and letter arithmetic

```
int d = '7' - '0';                          // 7   - the digits are consecutiv
char c = static_cast<char>('0' + 3);        // '3'
int index = 'k' - 'a';                      // 10  - letters are consecutive t
char shifted = static_cast<char>('a' + (index + 3) % 26);   // 'n' - a Caesar
```

Digit characters have consecutive codes, so `c - '0'` converts a digit to its value and `'0' + d` converts back. The same holds for `'a'`–`'z'` and `'A'`–`'Z'`, which is how a letter becomes an index into a 26-element count array (`count[c - 'a']`). Check with `isdigit` / `isalpha` before subtracting — `'x' - '0'` is 72 and nobody will stop you using it.

Building a number by hand is the loop every parser reduces to:

```
long long value = 0;
for (char c : text) {
    if (!is_digit(c)) break;
    value = value * 10 + (c - '0');
}
```

Use `long long` for the accumulator: ten digits overflow `int`, and signed overflow is undefined behaviour (Module 2). The hand-rolled loop also shows why the library functions matter — sign, whitespace, overflow and trailing text all need handling.

Number to text: std::to_string

```
std::string a = std::to_string(42);          // "42"
std::string b = std::to_string(-7LL);        // "-7"
std::string c = std::to_string(3.5);         // "3.500000" - always six decima
std::string d = std::to_string(1e20);        // "100000000000000000000.000000"
```

Integers convert exactly as printed. For floating point `std::to_string` behaves like `%f` — six decimals, not adjustable, enormous for large values — so use `std::format("{:.2f}", x)` (lesson 5) or a stream with `std::setprecision` whenever the format matters.

Text to number: the stoi family

```
#include <string>

int i = std::stoi("42");                              // 42
int j = std::stoi("  -17xyz");                        // -17: leading whitespace s
long l = std::stol("123456789012");
long long ll = std::stoll("-9000000000000000000");
double d = std::stod("2.5e3");                        // 2500.0
std::size_t used = 0;
int k = std::stoi("123abc", &used);                   // k = 123, used = 3 - how m
```

`std::stoi`, `stol`, `stoll`, `stoul`, `stoull`, `stof`, `stod` and `stold` wrap C's `strtol` family: they skip leading whitespace, accept a sign, parse the longest valid prefix and *ignore the rest*. The optional second argument receives the count of characters consumed — the only way to notice trailing junk. Two things go wrong loudly:

```
try {
    int v = std::stoi(token);
    std::cout << v << '\n';
} catch (const std::invalid_argument&) {    // no digits at all: "abc", "", "
    std::cout << "not a number\n";
} catch (const std::out_of_range&) {        // digits, but the value does not
    std::cout << "too big\n";
}
```

An uncaught exception ends the program with a non-zero exit code — a runtime error on the judge, not a wrong answer — so any `stoi` on input you do not control sits inside a `try`. Module 15, Errors and exceptions, covers the mechanism; here the two catch clauses are enough.

Text to number without exceptions: std::from_chars

```cpp
#include <charconv>
#include <string>
#include <system_error>

bool parse_int(const std::string& text, int& out) {
    const char* first = text.data();
    const char* last = first + text.size();
    auto [ptr, ec] = std::from_chars(first, last, out);
    return ec == std::errc() && ptr == last;      // no error, and the whole t
}
```

`std::from_chars` (C++17, `<charconv>`) parses a character range into a number and reports through a result struct: `ptr` points just past what it consumed, and `ec` is `std::errc()` on success, `std::errc::invalid_argument` when no number starts at `first`, or `std::errc::result_out_of_range` when the digits are valid but the value does not fit. It never throws, never allocates, skips no whitespace and accepts no `+` — only an optional `-` for signed targets. Checking `ptr == last` turns "starts with a number" into "is a number", the check the exercises need. It is the fastest conversion in the library, and because it takes a pointer range it works on a `std::string_view` (lesson 6) too, so a token sliced from a line parses without a copy. `std::to_chars` is the mirror image; `std::format` is usually more convenient.

	`std::stoi`	`std::from_chars`
Leading whitespace	skipped	error
`+` sign	accepted	error
Trailing text	ignored (see the `pos` argument)	reported through `ptr`
Not a number	throws `invalid_argument`	`ec == errc::invalid_argument`
Too large	throws `out_of_range`	`ec == errc::result_out_of_range`
Base	argument (default 10)	argument (default 10)

Pitfalls

`std::isdigit(c)` on a plain `char` is undefined behaviour for bytes over 127; cast to `unsigned char` first.

`std::toupper` returns `int`; assigning to a `char` without a cast is a silent narrowing, so cast explicitly.

`std::stoi("42abc")` is 42, not an error; check the consumed count or use `from_chars` with `ptr == last`.

`std::stoi("")` and `std::stoi("abc")` throw `std::invalid_argument`; `std::stoi("99999999999")` throws `std::out_of_range`; an uncaught throw is a runtime error on the judge.

`std::from_chars` rejects `" 42"` and `"+42"`; trim first and decide whether a `+` is allowed.

Key takeaways

`char` is an integer; arithmetic promotes to `int`, and `'0' + d` / `c - '0'` convert between digit characters and values.

Every `<cctype>` call takes `static_cast<unsigned char>(c)`; the case functions return `int`.

`std::to_string` is exact for integers and fixed at six decimals for floating point.

The `stoi` family skips whitespace, parses a prefix and throws on failure; catch `invalid_argument` and `out_of_range`.

`std::from_chars` reports through `ptr` and `ec`, never throws, and `ptr == last` means the whole token was a number.

EXERCISE 1

Classify and swap

Read one line. Using `<cctype>` with the `unsigned char` cast, count its letters (`isalpha`), digits (`isdigit`), spaces (`isspace`) and other characters, then print the line with every letter's case swapped (capitals become lower case and lower case becomes capitals); every other character is unchanged.

Input: one line (possibly empty, possibly with leading or trailing spaces). **Output:** `letters=<a> digits=<b> spaces=<c> other=<d>` on the first line, the swapped text on the second.

Example: input `Hello, World 42!` → output

```
letters=10 digits=2 spaces=2 other=2
hELLO, wORLD 42!
```

INPUT 1

```
Hello, World 42!
```

OUTPUT 1

```
letters=10 digits=2 spaces=2 other=2
hELLO, wORLD 42!
```

INPUT 2

```
C++20 is here
```

OUTPUT 2

```
letters=7 digits=2 spaces=2 other=2
c++20 IS HERE
```

REFERENCE SOLUTION

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

int main() {
    std::string line;
    std::getline(std::cin, line);

    int letters = 0, digits = 0, spaces = 0, other = 0;
    for (char& c : line) {
        const unsigned char u = static_cast<unsigned char>(c);
        if (std::isalpha(u)) {
            ++letters;
            c = static_cast<char>(std::isupper(u) ? std::tolower(u) : std::toupper(u
        } else if (std::isdigit(u)) {
            ++digits;
        } else if (std::isspace(u)) {
            ++spaces;
        } else {
            ++other;
        }
    }

    std::cout << "letters=" << letters << " digits=" << digits << " spaces=" << spac
    std::cout << line << '\n';
    return 0;
}
```

EXERCISE 2

Parse the numbers

Read lines until end of input. Each line holds one token, possibly with spaces before or after it; blank lines are skipped. Trim the line, then decide what the token is:

- a valid `int` — an optional `-` followed only by digits, and a value that fits in `int` — prints `<token> → <value>` and is added to a running `long long` sum;
- digits that do not fit in an `int` print `<token> → out of range`;
- anything else (letters, a `+` sign, a decimal point, a space inside the token, a lone `-`) prints `<token> → invalid`.

After the last line print `sum=<sum>`. Use `std::from_chars` and require that it consumed the whole token (`ptr == last`); if you prefer `std::stoi`, reject the `+` sign and trailing text yourself and catch both exceptions — an uncaught one is a runtime error.

Input: any number of lines. **Output:** one line per non-blank input line, then the sum.

Example: input

```
42
  -17
abc
2147483648
```

→ output

```
42 → 42
-17 → -17
abc → invalid
2147483648 → out of range
sum=25
```

INPUT 1

```
42
  -17
abc
2147483648
```

OUTPUT 1

```
42 → 42
-17 → -17
abc → invalid
2147483648 → out of range
sum=25
```

INPUT 2

```
1

2
3.5
```

OUTPUT 2

```
1 → 1
2 → 2
3.5 → invalid
sum=3
```

REFERENCE SOLUTION

```cpp
#include <charconv>
#include <iostream>
#include <string>
#include <system_error>

std::string trim(const std::string& s) {
    const std::size_t start = s.find_first_not_of(" \t");
    if (start == std::string::npos) return "";
    const std::size_t end = s.find_last_not_of(" \t");
    return s.substr(start, end - start + 1);
}

int main() {
    long long sum = 0;
    std::string line;
    while (std::getline(std::cin, line)) {
        const std::string token = trim(line);
        if (token.empty()) continue;

        int value = 0;
        const char* first = token.data();
        const char* last = first + token.size();
        const auto [ptr, ec] = std::from_chars(first, last, value);
        if (ec == std::errc::result_out_of_range) {
            std::cout << token << " -> out of range\n";
        } else if (ec != std::errc() || ptr != last) {
            std::cout << token << " -> invalid\n";
        } else {
            std::cout << token << " -> " << value << '\n';
            sum += value;
        }
    }
    std::cout << "sum=" << sum << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
char c = 'A';
std::cout << c + 1;
```

A. `B`

B. `66`

C. `A1`

D. It does not compile

2. Why do careful programs write `std::isalpha(static_cast<unsigned char>(c))` rather than `std::isalpha(c)`?

A. `isalpha` takes `unsigned char` and will not compile with a `char`

B. `char` is signed on this platform, so a byte above 127 is negative, and passing a negative value other than `EOF` is undefined behaviour

C. The cast makes the call faster

D. Without the cast `isalpha` returns `bool` instead of `int`

3. What does `std::stoi("42abc")` do?

A. Returns 42 — it parses the longest valid prefix and ignores the rest

B. Throws `std::invalid_argument`

C. Returns 0

D. Undefined behaviour

4. `std::from_chars` is called on the three characters `" 42"` (a leading space). What is the result?

A. 42, because leading whitespace is skipped like `stoi`

B. `ec == std::errc::invalid_argument` and `ptr` still points at the space

C. 4, because the space is treated as a digit separator

D. It does not compile with a leading space

5. What is `std::to_string(2.5)` ?

A. `"2.5"`

B. `"2.50"`

C. `"2.500000"`

D. `"2.5e+00"`

6. What is the type and value of `'7' - '0'` ?

A. `char` `'7'`

B. `int` 7

C. `int` 55

D. It does not compile — you cannot subtract characters

7. Which exception does `std::stoi("99999999999")` throw?

A. `std::invalid_argument`

B. `std::out_of_range`

C. `std::overflow_error`

D. None — the value wraps around

ANSWER KEY · CHARACTERS AND CONVERSIONS

1. B `c + 1` promotes the `char` to `int` before adding, and an `int` prints as digits: 65 + 1 = 66. `static_cast<char>(c + 1)` prints `B`.

2. B The `<cctype>` functions are defined only for values representable as `unsigned char` and for `EOF`. Non-ASCII input produces negative `char` values, and the cast maps them back into the valid range.

3. A The `stoi` family wraps `strtol`: leading whitespace is skipped, a prefix is parsed, trailing text is ignored. Pass a `std::size_t*` to learn how many characters were consumed, or use `std::from_chars` and check `ptr == last`.

4. B `from_chars` skips nothing: the pattern must begin at `first`. A space is not part of a number, so no conversion happens. Trim first, and remember it rejects a `+` sign too.

5. C For floating point, `std::to_string` formats like `printf("%f")`: always six decimals. Use `std::format("{:.1f}", x)` or a stream with `std::setprecision` to control it.

6. B Both operands promote to `int`: 55 − 48 = 7. The digit characters have consecutive codes, so subtracting `'0'` converts a digit character to its value.

7. B The digits are valid but the value does not fit an `int`, which is `std::out_of_range`. `std::invalid_argument` is for text with no number at all. Catch both, or the program ends with a runtime error.

LESSON 4 · 14 MIN

Parsing input

Every program in this track starts the same way: text arrives on standard input and must become numbers, names and records before any logic runs. The judge never sends malformed input by accident, but it does send the edges — a blank line, spaces around a token, an empty field, a count of zero — and a parser that assumes the happy path fails there. This lesson settles the reading patterns: tokens versus lines, splitting on whitespace with `std::istringstream`, splitting on a delimiter with `std::getline` or a `find` loop, trimming, "N then N lines", `key=value` pairs, and validating what you read.

Tokens, lines and the newline between them

`std::cin >> x` skips leading whitespace, reads one token and stops at the next whitespace — it never consumes the newline that ends the line. `std::getline(std::cin, line)` reads everything up to the newline, consumes the newline and stores the rest. Use one after the other and `getline` returns the empty remainder of the line that `>>` was on:

```
int n = 0;
std::cin >> n;
std::cin.ignore();                    // drop the newline that >> left behind
std::string first;
std::getline(std::cin, first);        // now the next real line
```

`std::cin.ignore()` discards exactly one character; `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n')` (from `<limits>`) discards the rest of the line however long — safer when the count could be followed by spaces. Module 1, Console I/O, introduced this; every pattern below uses one of the two.

Splitting a line on whitespace

```cpp
#include <sstream>

std::string line;
std::getline(std::cin, line);              //   ada   36 london
std::istringstream iss(line);
std::string word;
while (iss >> word) {
    std::cout << '[' << word << "]\n";     // [ada] [36] [london]
}
```

An `std::istringstream` is a stream over a string: `>>` behaves exactly as on `std::cin`, skipping any amount of whitespace, so runs of spaces, tabs and leading or trailing blanks all disappear — the right tool whenever fields are separated by "one or more spaces". Reading a number the same way also validates it:

```cpp
std::istringstream iss("42 apples");
int count = 0;
std::string what;
if (iss >> count >> what) { /* both read: 42 and "apples" */ }

std::istringstream bad("forty");
if (!(bad >> count)) { /* "forty" is not an integer: count is now 0 and the st
```

A failed `>>` sets the stream's fail bit and, since C++11, writes 0 to the target; every later `>>` on that stream fails until `clear()` is called. Module 18, Streams and files, goes deeper; for parsing, "the read failed, so the token was not a number" is enough. Two shapes of one loop cover most inputs: `while (std::cin >> x)` reads tokens to the end of input and `while (std::getline(std::cin, line))` reads lines — the condition is the stream itself, which converts to `false` once a read has failed.

Splitting on a delimiter

Comma-, colon- or pipe-separated fields need a different cut, because whitespace is now part of the data and empty fields are meaningful. `std::getline` takes a third argument, the delimiter:

```
std::istringstream iss("ada,,london");
std::string field;
while (std::getline(iss, field, ',')) {
    std::cout << '[' << field << "]\n";     // [ada] [] [london]
}
```

An empty field between two commas comes out as an empty string, which is right. But a delimiter at the very *end* is followed by nothing: the final `getline` hits end of input without extracting a character and fails, so `"a,b,"` yields two fields, not three. When a trailing delimiter must mean a trailing empty field, use the `find` loop:

```
#include <vector>

std::vector<std::string> split(const std::string& text, char delim) {
    std::vector<std::string> parts;
    std::size_t start = 0;
    while (true) {
        const std::size_t at = text.find(delim, start);
        if (at == std::string::npos) {
            parts.push_back(text.substr(start));       // the last field, poss
            return parts;
        }
        parts.push_back(text.substr(start, at - start));
        start = at + 1;
    }
}
```

`split("a,b,", ',')` gives `{"a", "b", ""}` and `split("", ',')` one empty field — a definition, so state it when it matters. Lesson 6 rewrites this with `std::string_view` so no field is copied.

Trimming every field

Fields split on a delimiter keep their spaces: `"ada , 36"` splits into `"ada "` and `" 36"`. Lesson 2's `trim` fixes each one:

```cpp
std::string trim(const std::string& s) {
    const std::size_t start = s.find_first_not_of(" \t");
    if (start == std::string::npos) return "";
    const std::size_t end = s.find_last_not_of(" \t");
    return s.substr(start, end - start + 1);
}
```

Trim *before* testing for emptiness or converting to a number: `std::from_chars` (lesson 3) rejects a leading space outright, and `" "` is not an empty field until it has been trimmed.

N then N lines

```cpp
int n = 0;
std::cin >> n;
std::cin.ignore();
for (int i = 0; i < n; ++i) {
    std::string line;
    std::getline(std::cin, line);
    // parse line
}
```

The shape behind most exercises: a count, then that many records. Trust the count, not the end of input — a second section (queries, say) may follow. When the records are single tokens rather than lines, `std::cin >> token` needs no `ignore()` at all, because `>>` skips newlines itself.

---

key=value pairs

```cpp
std::string line = " timeout = 30 ";
const std::size_t eq = line.find('=');
if (eq == std::string::npos) {
    std::cout << "invalid: no '='\n";
} else {
    std::string key = trim(line.substr(0, eq));       // "timeout"
    std::string value = trim(line.substr(eq + 1));    // "30"
    if (key.empty()) std::cout << "invalid: empty key\n";
}
```

Cut on the *first* `=` so a value may itself contain one (`url = a?b=c`). Store the pairs in a `std::map<std::string, std::string>` — Module 3 iterated one; Module 13, STL containers, covers it — and let later lines overwrite earlier ones with `settings[key] = value`. Empty lines and lines starting with `#` are skipped before any of this runs.

Validating tokens

Reading is not done until every token has been checked. Decide in advance: what is the delimiter, are runs of it one separator or several (whitespace: one; commas: several), are surrounding blanks significant, may a field be empty, what range may a number have, and what happens on a bad line — skip, report, or stop. A parser that prints `line 3: invalid` and carries on beats one that crashes on the first surprise; on the judge a crash is a runtime error, not a wrong answer.

```
int lineno = 0;
std::string line;
while (std::getline(std::cin, line)) {
    ++lineno;
    if (!line.empty() && line.back() == '\r') line.pop_back();    // Windows 
    const std::string token = trim(line);
    if (token.empty()) continue;                                  // blank li
    int value = 0;
    if (!parse_int(token, value)) {                               // lesson 3
        std::cout << "line " << lineno << ": invalid\n";
        continue;
    }
    // use value
}
```

The `'\r'` check costs one comparison and saves an afternoon when a file was edited on Windows; this judge sends `\n` only, but real input promises nothing.

Pitfalls

- `std::getline` straight after `std::cin >> n` reads the empty rest of the count's line; `ignore()` first.
- `std::getline(iss, f, ',')` drops a trailing empty field; the `find` loop keeps it — pick one and say so.

`>>` on a non-number sets the fail bit and every later read on that stream fails until `clear()`.

Fields split on a delimiter keep their surrounding spaces; trim before comparing or converting.

Cut `key=value` on the first `=`, not the last, so values may contain `=`.

Key takeaways

`>>` reads tokens and skips whitespace; `getline` reads lines; an `ignore()` sits between them.

`std::istringstream` turns a line into a stream so `>>` splits it on whitespace and validates numbers.

`getline` with a delimiter and a `find` / `substr` loop split on a character; they differ on a trailing delimiter.

Trim every field, then test for empty, then convert.

Read "N then N lines" by the count, and `while (std::getline(...))` when the input runs to the end.

Validate every token and report bad lines; a crash is a runtime error on the judge.

EXERCISE 1

CSV fields

Read lines until end of input. Each line is a comma-separated record. For every line print the number of fields and then each field with its surrounding spaces trimmed, all joined with `|`.

The rules: two commas in a row enclose an empty field, a comma at the end of the line is followed by an empty field (so `x,y,` has three fields), and a line that is empty or contains only spaces has no fields — print `0`.

Input: any number of lines. **Output:** one line per input line.

Example: input

```
ada, 36 ,london
a,,b
```

→ output

```
3|ada|36|london
3|a||b
```

A `find` / `substr` loop keeps the trailing empty field; `std::getline` with a delimiter drops it — choose accordingly.

INPUT 1

```
ada, 36 ,london
a,,b
```

OUTPUT 1

```
3|ada|36|london
3|a||b
```

INPUT 2

```
one
```

OUTPUT 2

```
1|one
```

REFERENCE SOLUTION

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

std::string trim(const std::string& s) {
    const std::size_t start = s.find_first_not_of(" \t");
    if (start == std::string::npos) return "";
    const std::size_t end = s.find_last_not_of(" \t");
    return s.substr(start, end - start + 1);
}

std::vector<std::string> split(const std::string& text, char delim) {
    std::vector<std::string> parts;
    std::size_t start = 0;
    while (true) {
        const std::size_t at = text.find(delim, start);
        if (at == std::string::npos) {
            parts.push_back(text.substr(start));
            return parts;
        }
        parts.push_back(text.substr(start, at - start));
        start = at + 1;
    }
}

int main() {
    std::string line;
    while (std::getline(std::cin, line)) {
        if (trim(line).empty()) {
            std::cout << "0\n";
            continue;
        }
        const std::vector<std::string> fields = split(line, ',');
        std::cout << fields.size();
        for (const std::string& field : fields) {
            std::cout << '|' << trim(field);
        }
        std::cout << '\n';
    }
    return 0;
}
```

EXERCISE 2

Settings file

Read an integer `N`, then `N` lines of a settings file, then an integer `M` and `M` query keys (single tokens). Each settings line is either blank, a comment starting with `#`, or `key = value`: cut on the **first** `=`, trim both sides; the value may contain spaces or further `=` signs and may be empty. A line with no `=`, or an empty key, is invalid: print `line <i>: invalid` (`i` counts all `N` lines from 1) and carry on. A key set twice keeps the last value.

Then, for each query key, print `<key>=<value>` or `<key>: not set`.

Input: `N`, `N` lines, `M`, `M` keys. **Output:** one line per invalid settings line, then one line per query.

Example: input

```
3
colour = blue
broken line
colour=green
2
colour size
```

→ output

```
line 2: invalid
colour=green
size: not set
```

INPUT 1

```
3
colour = blue
broken line
colour=green
2
colour size
```

OUTPUT 1

```
line 2: invalid
colour=green
size: not set
```

INPUT 2

```
4
name = Ada Lovelace
# comment
timeout=30

2
name timeout
```

OUTPUT 2

```
name=Ada Lovelace
timeout=30
```

REFERENCE SOLUTION

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

std::string trim(const std::string& s) {
    const std::size_t start = s.find_first_not_of(" \t");
    if (start == std::string::npos) return "";
    const std::size_t end = s.find_last_not_of(" \t");
    return s.substr(start, end - start + 1);
}

int main() {
    int n = 0;
    std::cin >> n;
    std::cin.ignore();

    std::map<std::string, std::string> settings;
    for (int i = 1; i <= n; ++i) {
        std::string line;
        std::getline(std::cin, line);
        const std::string clean = trim(line);
        if (clean.empty() || clean.front() == '#') continue;

        const std::size_t eq = clean.find('=');
        if (eq == std::string::npos) {
            std::cout << "line " << i << ": invalid\n";
            continue;
        }
        const std::string key = trim(clean.substr(0, eq));
        if (key.empty()) {
            std::cout << "line " << i << ": invalid\n";
            continue;
        }
        settings[key] = trim(clean.substr(eq + 1));
    }

    int m = 0;
    std::cin >> m;
    for (int i = 0; i < m; ++i) {
        std::string key;
        std::cin >> key;
        if (settings.contains(key)) {
            std::cout << key << '=' << settings.at(key) << '\n';
        } else {
            std::cout << key << ": not set\n";
        }
    }
```

```
    return 0;
}
```

QUIZ

1. The input is `3⏎Ada⏎`. After `int n; std::cin >> n; std::string name; std::getline(std::cin, name);`, what is `name`?

A. `"Ada"`

B. `"3"`

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

D. The read fails and `name` is unchanged

2. How many times does the loop body run?

```
std::istringstream iss("a  b   c");
std::string w;
int count = 0;
while (iss >> w) ++count;
```

A. 3

B. 1

C. 6

D. 5

3. `std::istringstream iss("a,b,"); while (std::getline(iss, f, ',')) ++k;` — what is `k` afterwards?

A. 2 — the trailing comma is followed by nothing, so the third `getline` extracts no characters and fails

B. 3 — the trailing comma yields an empty field

C. 1

D. The loop never ends

4. What is the state after this?

```
std::istringstream iss("x");
int n = 5;
iss >> n;
```

A. `n` is 5 and the stream is fine

B. `n` is 0 and the stream's fail bit is set; further reads fail until `clear()`

C. An exception is thrown

D. `n` is the character code of `'x'`

5. A `key = value` parser cuts on `line.rfind('=')`. Which input breaks it?

A. `timeout = 30`

B. `url = a?b=c` — the value's own `=` becomes the cut point

C. `name = Ada Lovelace`

D. `# comment`

6. When does `while (std::getline(std::cin, line))` stop?

A. At the first empty line

B. When a read fails — at end of input

C. When `line` equals `"EOF"`

D. It never stops without a `break`

ANSWER KEY · PARSING INPUT

1. C `>>` stops at the newline after `3` without consuming it, so `getline` reads up to that newline and returns nothing. Call `std::cin.ignore()` between them.

2. A `>>` skips any run of whitespace before each token, so the double and triple spaces count as one separator each: `a`, `b`, `c`.

3. A `getline` with a delimiter returns an empty field *between* two delimiters but not after a final one, because reaching end of input with nothing extracted sets the fail bit. A `find` / `substr` loop keeps the trailing empty field.

4. B Since C++11 a failed numeric extraction writes 0 to the target and sets `failbit`. The stream stays in that state, so `if (!(iss >> n))` is the test for "not a number".

5. B The last `=` is inside the value, so the key becomes `url = a?b` and the value `c`. Cut on the *first* `=` with `find` so values may contain the character.

6. B The condition is the stream itself, which converts to `false` once an extraction has failed. An empty line is a successful read of an empty string, so the loop continues through it.

LESSON 5 · 14 MIN

Formatting output

Output that must match a judge or line up in a report is a formatting problem before anything else: how wide is the column, which side is padded, how many decimals, zero-filled or hex. C++ has two toolkits. The stream manipulators in `<iomanip>` have been there from the start and are what most existing code uses; C++20's `std::format` says the same things in one compact specification and is the one for new code. This lesson settles both — including which stream settings are sticky and which apply once, the mistake that bites everyone — and shows how to build a table, and a string, before printing anything.

Width, alignment and fill

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

int main() {
    std::cout << '[' << std::setw(6) << 42 << "]\n";                    // [
    std::cout << '[' << std::setw(6) << std::left << 42 << "]\n";       // [
    std::cout << '[' << std::setw(6) << "ab" << "]\n";                  // [
    std::cout << std::right << std::setfill('0') << std::setw(5) << 7 << '\n';
    std::cout << std::setw(3) << 1 << std::setw(3) << 2 << '\n';        // 0
    return 0;
}
```

`std::setw(n)` sets the minimum width of the **next insertion only**; the one after it is back to width 0. `std::left`, `std::right` and `std::setfill(c)` stay in force until changed, which is why the last line above zero-fills both numbers. Numbers and strings alike default to right alignment on a stream. A value wider than the field is never truncated; the field grows.

Precision and the floating-point modes

```cpp
double x = 3.14159265;
std::cout << std::setprecision(3) << x << '\n';                    // 3.14
std::cout << std::setprecision(3) << 1234.5678 << '\n';            // 1.23e
std::cout << std::fixed << std::setprecision(3) << x << '\n';      // 3.142
std::cout << std::fixed << std::setprecision(2) << 2.0 / 3 << '\n';  // 0.67
std::cout << std::scientific << x << '\n';                         // 3.14e
std::cout << std::defaultfloat << std::setprecision(6) << x << '\n';  // 3.141
```

Without `std::fixed`, `setprecision` counts *significant* digits and the stream may switch to scientific notation for large or small values — hence `1.2e+03` for 1234.5 at `setprecision(2)`. With `std::fixed` it counts digits after the point, the mode every exercise with money or averages wants. Both are sticky; set them once at the top of `main`. Rounding is to nearest, so `2.0 / 3` at two decimals is `0.67`, while an exact tie such as `0.125` rounds to even and prints `0.12` — never build an expectation on a tie.

Integers have their own modes: `std::hex`, `std::oct` and `std::dec` change the base and stay changed, `std::showbase` adds `0x`, `std::uppercase` capitalises the digits. Forgetting to switch back to `std::dec` is a classic: every integer printed afterwards comes out in hex.

Sticky or one-shot

Manipulator	Effect	Lasts
`std::setw(n)`	minimum width	next insertion only
`std::left` / `std::right` / `std::internal`	alignment within the width	until changed
`std::setfill(c)`	padding character	until changed
`std::fixed` / `std::scientific` / `std::defaultfloat`	floating-point mode	until changed
`std::setprecision(n)`	digits (meaning depends on the mode)	until changed
`std::hex` / `std::oct` / `std::dec`	integer base	until changed
`std::boolalpha`	`true` / `false` instead of `1` / `0`	until changed

std::format

```
#include <format>

std::cout << std::format("{} scored {}\n", "Ada", 97);
std::cout << std::format("[{:>6}] [{:<6}] [{:^6}]\n", 42, 42, 42);
std::cout << std::format("[{:6}] [{:6}]\n", 42, "ab");
std::cout << std::format("{:05} {:+d} {:.3f} {:8.2f}\n", 42, 7, 3.14159, 2.5);
std::cout << std::format("{:x} {:X} {:#x} {:08b} {:o}\n", 255, 255, 255, 5, 8);
std::cout << std::format("{1} before {0}\n", "b", "a");
std::cout << std::format("{:*^11}\n", "mid");
```

Each `{}` is replaced by the next argument; after a colon comes a specification in a fixed order: **fill and align** (`<` left, `>` right, `^` centre, optionally preceded by the fill character), **sign** (`+`), `#` for a base prefix, `0` for zero padding, **width**, `.precision` , and a **type** letter — `d` , `x` , `X` , `b` , `o` for integers, `f` , `e` , `g` for floating point, `s` for strings. Everything is optional; `{:8.2f}` reads "width 8, two decimals, fixed". Unlike streams, nothing is sticky: each placeholder says everything about itself, and the default alignment differs by kind — numbers right, text left, which is exactly what a table wants.

The format string must be known at compile time, because the compiler checks the placeholders against the argument types: `{:d}` with a `std::string` is a compile error, not a runtime surprise. For a string built at run time use `std::vformat(fmt, std::make_format_args(args...))`. `std::format` returns a `std::string`; `std::print` (C++23) writes straight to stdout but is not available on this runtime — write `std::cout << std::format(...)`.

Building a string first

```
#include <sstream>

std::ostringstream out;
out << std::fixed << std::setprecision(1);
for (int i = 1; i <= 3; ++i) out << i << ':' << i * 0.5 << ' ';
std::string text = out.str();       // "1:0.5 2:1.0 3:1.5 "
std::cout << text << '\n';
```

An `std::ostringstream` accepts everything `std::cout` does and hands the result back with `str()`: assemble a line before deciding whether to print it, measure a string's width before padding, or collect a large output and write it once. `std::format` covers the simple cases since it already returns a string; the stream wins when many pieces with shared settings accumulate in a loop.

A table, done right

```
struct Row { std::string name; int qty; double price; };

void print_table(const std::vector<Row>& rows) {
    std::cout << std::format("{:<10}{:>5}{:>10}\n", "item", "qty", "price");
    for (const Row& r : rows) {
        std::cout << std::format("{:<10}{:>5}{:>10.2f}\n", r.name, r.qty, r.pr
    }
}
```

```
item        qty     price
pen           3      1.20
notebook     12     45.00
```

The rules that keep a table clean and a judge happy: right-align numbers so decimal points line up; put the left-aligned text column first so no line ends in padding; give each column a width that fits its widest value and use the same widths for the header. This judge ignores trailing spaces, but a left-aligned *last* column still produces them and other judges compare bytes — end each row with a right-aligned column and the problem never arises.

Pitfalls

- `setw` is one-shot; `setprecision`, `fixed`, `left`, `setfill` and `hex` are sticky and stay on for the rest of the program.
- `setprecision(2)` without `fixed` means two *significant* digits: `1234.5` prints as `1.2e+03`.
- `std::hex` left on turns every later integer into hex; switch back with `std::dec`.
- The `std::format` string must be a compile-time constant; a `std::string` variable does not compile — use `std::vformat`.
- A left-aligned last column pads with spaces to the end of the line; put text columns first.
- Ties such as `0.125` round to even at two decimals; do not build an expectation on one.

Key takeaways

- Streams: `setw` applies once; alignment, fill, precision, floating mode and base persist.
- `std::fixed` plus `setprecision(n)` is the deterministic way to print `n` decimals; without `fixed`, `n` counts significant digits.
- `std::format("{:>8.2f}")` reads fill/align, sign, `#`, `0`, width, precision, type — nothing is sticky, numbers right-align and text left-aligns by default.
- The format string is checked at compile time; it must be a literal.
- `std::ostringstream` collects output into a string via `str()`; `std::format` returns one directly.
- Put text columns first and right-align numbers so no line ends in padding.

EXERCISE 1

Receipt

Read an integer `N` and then `N` lines, each `name qty price` — a one-word item name, an integer quantity and a decimal unit price. Print one line per item and a total line, with exact widths:

- item lines: name **left-aligned in width 12**, quantity **right-aligned in width 4**, line total (`qty * price`) **right-aligned in width 10 with two decimals** — the `std::format` spec is `{:<12}{:>4}{:>10.2f}`;
- the last line: `TOTAL` left-aligned in width 16, then the grand total right-aligned in width 10 with two decimals.

A name longer than 12 characters simply pushes its columns right. `N` may be 0, in which case only the total line is printed.

Input: `N`, then `N` lines. **Output:** `N + 1` lines.

Example: input

```
3
pen 3 1.25
notebook 2 4.5
stapler 1 12.99
```

→ output

```
pen             3      3.75
notebook        2      9.00
stapler         1     12.99
TOTAL                 25.74
```

INPUT 1

```
3
pen 3 1.25
notebook 2 4.5
stapler 1 12.99
```

OUTPUT 1

```
pen             3      3.75
notebook        2      9.00
stapler         1     12.99
TOTAL                 25.74
```

INPUT 2

```
1
coffee 10 0.99
```

OUTPUT 2

```
coffee         10      9.90
TOTAL                  9.90
```

REFERENCE SOLUTION

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

int main() {
    int n = 0;
    std::cin >> n;
    double grand = 0.0;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int qty = 0;
        double price = 0.0;
        std::cin >> name >> qty >> price;
        const double total = qty * price;
        grand += total;
        std::cout << std::format("{:<12}{:>4}{:>10.2f}\n", name, qty, total);
    }
    std::cout << std::format("{:<16}{:>10.2f}\n", "TOTAL", grand);
    return 0;
}
```

EXERCISE 2

Number bases

Read integers in the range 0 to 65535 until end of input (whitespace-separated). Print a header line and then one row per number showing it in four bases, columns separated by `|`:

- decimal right-aligned in width 5;
- hexadecimal in lower case, zero-padded to 4 digits;
- octal right-aligned in width 6;
- binary zero-padded to 16 digits.

The header uses the same widths, right-aligned: `{:>5}|{:>4}|{:>6}|{:>16}` with the words `dec`, `hex`, `oct`, `bin`; each row is `{:>5}|{:04x}|{:>6o}|{:016b}`. `std::format` makes the binary column trivial — with `<iomanip>` you would have to build it by hand. With no numbers at all, only the header is printed.

Input: zero or more integers. **Output:** the header, then one line per integer.

Example: input `255 0 65535` → output

```
  dec| hex|   oct|             bin
  255|00ff|   377|0000000011111111
    0|0000|     0|0000000000000000
65535|ffff|177777|1111111111111111
```

INPUT 1

```
255 0 65535
```

OUTPUT 1

```
  dec| hex|   oct|             bin
  255|00ff|   377|0000000011111111
    0|0000|     0|0000000000000000
65535|ffff|177777|1111111111111111
```

INPUT 2

```
10
4096
```

OUTPUT 2

```
  dec| hex|   oct|             bin
   10|000a|    12|0000000000001010
 4096|1000| 10000|0001000000000000
```

REFERENCE SOLUTION

```cpp
#include <format>
#include <iostream>

int main() {
    std::cout << std::format("{:>5}|{:>4}|{:>6}|{:>16}\n", "dec", "hex", "oct", "bir
    int value = 0;
    while (std::cin >> value) {
        std::cout << std::format("{:>5}|{:04x}|{:>6o}|{:016b}\n", value, value, valu
    }
    return 0;
}
```

QUIZ

1. What does `std::cout << std::setw(5) << 42 << 7;` print?

A. `427`

B. `42 7`

C. `427`

D. `00427`

2. What does `std::cout << std::setprecision(2) << 1234.5;` print?

A. `1234.50`

B. `1234.5`

C. `1.2e+03`

D. `1234`

3. What does `std::format("{:04}|{:>4}|{:<4}|", 7, 7, "ab")` return?

A. `0007| 7|ab |`

B. `7|0007| ab|`

C. `7000|7 |ab |`

D. `0007|7 | ab|`

4.

```
std::string fmt = "{}";
std::cout << std::format(fmt, 1);
```

What happens?

A. Prints `1`

B. A compile error — the format string must be a compile-time constant; use `std::vformat` for a runtime string

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

D. Undefined behaviour

5. After `std::cout << std::hex << 255;` the program later runs `std::cout << 16;`. What appears for the second value?

A. `16`

B. `10`

C. `0x10`

D. A compile error

6. What does `std::format("{:x} {:#x} {:X}", 255, 255, 255)` return?

A. `ff 0xff FF`

B. `FF 0xFF ff`

C. `0xff 0xff 0xFF`

D. `ff ff FF`

7. What does `std::cout << std::fixed << std::setprecision(2) << 2.0 / 3;` print?

A. `0.66`

B. `0.67`

C. `0.666667`

D. `0.7`

ANSWER KEY · FORMATTING OUTPUT

1. A `setw` applies to the next insertion only: `42` is padded to width 5, then `7` is printed with no width at all.

2. C Without `std::fixed`, `setprecision` counts *significant* digits and the default float mode switches to scientific notation when the value does not fit. `std::fixed << std::setprecision(2)` gives `1234.50`.

3. A `0` before the width zero-fills, `>` right-aligns with spaces, `<` left-aligns. Note that `{:4}` alone would already right-align a number and left-align a string.

4. B `std::format` checks the format string against the argument types at compile time, so it must be a constant expression. `std::vformat(fmt, std::make_format_args(1))` is the run-time form.

5. B `std::hex` is sticky: the stream stays in hexadecimal until `std::dec` (or `std::oct`) is inserted. 16 in hex is `10`, without a prefix unless `std::showbase` is on.

6. A `x` is lower-case hex, `#` adds the `0x` prefix, `X` gives upper-case digits (and `#X` would give `0XFF`).

7. B Fixed mode with precision 2 rounds to the nearest hundredth: 0.6667 → `0.67`. Only exact binary ties such as `0.125` round to even.

LESSON 6 · 13 MIN

std::string_view — a window onto text

Every `std::string` you pass, slice or compare costs a copy somewhere: a function taking `const std::string&` allocates when called with a literal, `substr` copies what it returns, and a parser that cuts a line into fields copies the whole line again in pieces. `std::string_view` (C++17) answers all three: a pointer and a length that *refer to* characters someone else owns. It is sixteen bytes on this platform, trivially copyable, its `substr` costs nothing — and it brings the one lifetime rule this lesson exists to make you respect.

What it is

```
#include <string_view>

std::string owner = "codekairo";
std::string_view whole = owner;              // implicit from std::string: poi
std::string_view lit = "literal";            // from a literal: strlen once, r
std::string_view part(owner.data() + 4, 5);  // pointer + length: "kairo"
std::cout << whole.size() << ' ' << part << ' ' << lit.substr(0, 3) << '\n';
```

A view holds `data()` and `size()` and nothing else — no buffer, no ownership, no null terminator of its own; copying one copies two words. It supports the whole *read-only* string interface: `size`, `empty`, `[]`, `at`, `front`, `back`, iteration, the `find` family, `substr`, `starts_with` / `ends_with`, `compare` and all six comparison operators, which compare content just as `std::string` does. Anything that would change the characters — `+=`, `push_back`, `insert`, `replace` — does not exist, because the view does not own them. Two operations exist only on views: `remove_prefix(n)` and `remove_suffix(n)` shrink the window from either end in O(1), and are how a view is trimmed.

```
std::string_view v = "   padded   ";
v.remove_prefix(v.find_first_not_of(' '));                  // "padded   "
v.remove_suffix(v.size() - v.find_last_not_of(' ') - 1);    // "padded"
```

substr in O(1)

```
std::string line = "2026-09-17T10:15:00";
std::string_view sv = line;
std::string_view date = sv.substr(0, 10);     // "2026-09-17": a narrower win
std::string_view time = sv.substr(11);        // "10:15:00"
```

`std::string::substr` allocates and copies; `std::string_view::substr` adjusts a pointer and a length in constant time. A tokeniser written with views allocates nothing until you decide to keep something:

```
#include <vector>

std::vector<std::string_view> words(std::string_view text) {
    std::vector<std::string_view> out;
    std::size_t i = 0;
    while (i < text.size()) {
        while (i < text.size() && text[i] == ' ') ++i;          // skip separa
        const std::size_t start = i;
        while (i < text.size() && text[i] != ' ') ++i;          // run to the e
        if (start < i) out.push_back(text.substr(start, i - start));
    }
    return out;
}
```

Every element of `out` points into `text`'s characters. That is the whole point, and the whole danger.

The right parameter type for read-only text

Parameter	Called with a literal	Called with a `std::string`	Knows its length	Null-terminated
`const char*`	fine	needs `.c_str()`	no — `strlen`	yes
`const std::string&`	constructs a temporary `std::string` (allocates past 15 chars)	fine, no copy	yes	yes
`std::string_view`	fine, no copy	fine, no copy	yes	not guaranteed

Take `std::string_view` **by value** — two words; a reference would be an indirection for nothing — whenever the function only reads text: counting, searching, parsing, printing. The caller passes a literal, a `std::string`, another view or a `(pointer, length)` pair and nothing is copied. Return a `std::string`, not a view, when the function *produces* text.

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

count_char("banana", 'a');          // 3 - no std::string was ever built
count_char(owner, 'k');             // 1 - no copy of owner
```

The lifetime rule

A view is valid exactly as long as the characters it points at. Nothing in the language checks this, and the failures are undefined behaviour that often *appears* to work:

```
std::string_view bad1 = std::string("temporary");     // the string dies at th
std::string_view bad2 = owner + "!";                  // same: + returns a tem

std::string_view first_word() {
    std::string line;
    std::getline(std::cin, line);
    return std::string_view(line).substr(0, line.find(' '));   // line is dest
}

std::vector<std::string> names = {"ada"};
std::string_view n0 = names[0];
names.push_back("grace");                             // may reallocate: n0 ma
```

The rule in one sentence: **a view must not outlive the string it looks at, and that string must not be modified while the view is in use**. In practice: take views as parameters and use them within the call; return a view only into something the caller gave you (the argument, never a local); never store a view in a struct or container unless the owner is guaranteed to live longer and stay put. `words()` above is fine because its caller still holds `text`; it becomes a bug the moment someone writes `auto ws = words(read_line());` — the temporary line is gone by the next statement.

One more: `data()` is not guaranteed to end in `'\0'`. A view of the middle of a string stops where `size()` says but the bytes continue, so `sv.data()` handed to a C function that expects a terminator reads past the window. Build a `std::string` first.

Converting back

```
std::string_view sv = "kairo";
std::string s1(sv);              // explicit construction: copies the characte
std::string s2 = sv;             // error: the conversion from a view is expli
std::string s3;
s3 = sv;                         // fine: assignment from a view is allowed
s3 += sv;                        // fine: append accepts a view
std::string s4 = std::string(sv) + "!";   // build the string, then concatenat
```

The asymmetry is deliberate: `std::string` converts to a view silently because that is free; a view becomes a `std::string` only when you write it, because that allocates. To *keep* a piece of text — a map key, a struct member, anything that outlives the input — convert and store the `std::string`.

---

constexpr views

```
constexpr std::string_view kVersion = "1.4.2";
constexpr std::string_view kMajor = kVersion.substr(0, kVersion.find('.'));
static_assert(kMajor == "1");
constexpr std::string_view kDays[] = {"mon", "tue", "wed", "thu", "fri", "sat
```

A view onto a literal is a `constexpr` value: the characters live in the program image for its whole run, so the lifetime rule holds automatically, and `find`, `substr` and `==` are `constexpr`. A `std::string` cannot be a `constexpr` variable in C++20 (its allocation must not escape compile-time evaluation), so a compile-time table of names is a table of views. Module 16, Modern C++, has more.

When not to use it

The function stores the text (a member, a container element, a key): take `std::string` by value and move it in (Module 9, Copies, moves and the rule of five).

You need a null-terminated `const char*` for a C API: `std::string` and `c_str()`.

The text will be modified: views are read-only windows.

The source is a temporary or will change while the view is alive: copy instead.

Pitfalls

`std::string_view v = s + "x";` or `= std::string(...)` dangles before the next statement.

Returning a view of a local, or of a `std::string` parameter taken by value, dangles.

A view into a `std::vector<std::string>` element dies on `push_back`; into a `std::string` on any resize.

`sv.data()` is not null-terminated; never hand it to `printf`, `fopen` or `strlen`.

`remove_prefix(n)` with `n > size()` is undefined behaviour; compute `n` from a `find` that succeeded.

Key takeaways

`std::string_view` is a pointer and a length: non-owning, sixteen bytes, trivially copyable, read-only.

`substr`, `remove_prefix` and `remove_suffix` are O(1) and never allocate; the `find` family and comparisons work as on `std::string`.

Take it by value as the parameter type for any function that only reads text; return `std::string` when producing text.

A view must not outlive its string or survive a modification of it; never view a temporary, never return a view of a local.

Converting back is explicit — `std::string(sv)` — because it copies; `data()` is not null-terminated; views of literals are `constexpr`.

EXERCISE 1

Words as views

Read one line. Write `std::vector<std::string_view> words(std::string_view text)` that splits the text on runs of spaces **without copying a single character** — every element is a view into the line,

which stays alive in `main`. Then print `words=<n>`, `longest=<word>` (the first of the longest words; `longest=(none)` if there are no words), and one line `<word> <length>` per word in order.

Input: one line (possibly empty, possibly with leading, trailing or repeated spaces). **Output:** two lines, then one line per word.

Example: input `the quick brown fox` → output

```
words=4
longest=quick
the 3
quick 5
brown 5
fox 3
```

INPUT 1

```
the quick  brown fox
```

OUTPUT 1

```
words=4
longest=quick
the 3
quick 5
brown 5
fox 3
```

INPUT 2

```
   leading and trailing
```

OUTPUT 2

```
words=3
longest=trailing
leading 7
and 3
trailing 8
```

REFERENCE SOLUTION

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

std::vector<std::string_view> words(std::string_view text) {
    std::vector<std::string_view> out;
    std::size_t i = 0;
    while (i < text.size()) {
        while (i < text.size() && text[i] == ' ') ++i;
        const std::size_t start = i;
        while (i < text.size() && text[i] != ' ') ++i;
        if (start < i) out.push_back(text.substr(start, i - start));
    }
    return out;
}

int main() {
    std::string line;
    std::getline(std::cin, line);

    const std::vector<std::string_view> ws = words(line);   // views into line, whic
    std::cout << "words=" << ws.size() << '\n';

    std::string_view longest;
    for (std::string_view w : ws) {
        if (w.size() > longest.size()) longest = w;
    }
    if (ws.empty()) {
        std::cout << "longest=(none)\n";
    } else {
        std::cout << "longest=" << longest << '\n';
    }

    for (std::string_view w : ws) {
        std::cout << w << ' ' << w.size() << '\n';
    }
    return 0;
}
```

EXERCISE 2

Prefix filter

The first line is a non-empty prefix. Every following line, until end of input, is an entry. Write `std::string_view trim(std::string_view)` that removes leading and trailing spaces and tabs

with `remove_prefix` and `remove_suffix`, and for every entry: trim it, and if the trimmed entry `starts_with` the prefix print `match: ` followed by the rest of the entry after the prefix. Finally print `matched=<k> of <total>`, where `total` counts every entry line, blank ones included.

Input: the prefix, then any number of entry lines. **Output:** one line per matching entry, then the summary.

Example: input

```
err
error: disk full
  errno 5
warning: low
err
```

→ output

```
match: or: disk full
match: no 5
match:
matched=3 of 4
```

INPUT 1

```
err
error: disk full
  errno 5
warning: low
err
```

OUTPUT 1

```
match: or: disk full
match: no 5
match:
matched=3 of 4
```

INPUT 2

```
TODO
  TODO: write tests
DONE: read

TODO
```

OUTPUT 2

```
match: : write tests
match:
matched=2 of 4
```

REFERENCE SOLUTION

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

std::string_view trim(std::string_view s) {
    const std::size_t start = s.find_first_not_of(" \t");
    if (start == std::string_view::npos) return {};
    s.remove_prefix(start);
    const std::size_t end = s.find_last_not_of(" \t");
    s.remove_suffix(s.size() - end - 1);
    return s;
}

int main() {
    std::string prefix;
    std::getline(std::cin, prefix);

    int total = 0;
    int matched = 0;
    std::string line;
    while (std::getline(std::cin, line)) {
        ++total;
        const std::string_view entry = trim(line);
        if (entry.starts_with(prefix)) {
            ++matched;
            std::cout << "match: " << entry.substr(prefix.size()) << '\n';
        }
    }
    std::cout << "matched=" << matched << " of " << total << '\n';
    return 0;
}
```

QUIZ

1.

```
std::string_view v = std::string("temp") + "x";
std::cout << v;
```

What happens?

A. Prints `tempx`

B. Undefined behaviour — `v` views a temporary `std::string` that was destroyed at the end of the first statement

C. A compile error: a view cannot be built from a temporary

D. Prints `temp`

2. Why is `std::string_view` normally passed **by value** rather than by `const&`?

A. Because it cannot be bound to a reference

B. Because it is two words (a pointer and a length) that are cheap to copy — a reference would add an indirection for nothing

C. Because passing by value copies the characters, which is safer

D. Because `const std::string_view&` does not compile

3. Given `std::string_view sv = "kairo";`, which line does **not** compile?

A. `std::string s(sv);`

B. `std::string s = sv;`

C. `std::string s; s = sv;`

D. `std::string s; s += sv;`

4. What does `sv.substr(2)` cost for a `std::string_view sv`?

A. O(n) — it copies the characters into a new string

B. O(1) — it returns a narrower view; nothing is copied or allocated

C. It throws unless `sv` is null-terminated

D. It is undefined behaviour

5. You pass `sv.data()` to `std::printf("%s")`, where `sv` is a view of the middle of a longer string. What is the risk?

A. None — `data()` is always null-terminated

B. `printf` prints past the end of the view until it finds a `'\0'`, because a view carries no terminator

C. A compile error: `data()` returns `void*`

D. `printf` prints only the first character

6. Which of these uses a `std::string_view` safely?

A. A function that returns a view of one of its local `std::string` variables

B. Storing a view of `v[0]` from a `std::vector<std::string> v` and then calling `v.push_back(...)`

C. `constexpr std::string_view kName = "codekairo";` used anywhere in the program

D. A view of `line` used after `std::getline(std::cin, line)` has replaced the contents with a longer line

ANSWER KEY · STD::STRING_VIEW — A WINDOW ONTO TEXT

1. B `+` returns a temporary `std::string`; the view copies its pointer, and the temporary dies at the semicolon. It may print correctly by luck, which is what makes the bug dangerous.

2. B A view is sixteen bytes on this platform and trivially copyable; passing it by value is the idiom, and it never copies the characters it refers to.

3. B Constructing a `std::string` from a view is `explicit` because it allocates and copies, so copy-initialisation with `=` is refused. Direct construction, assignment and `+=` all accept a view.

4. B A view's `substr` adjusts a pointer and a length. `std::string::substr`, by contrast, allocates and copies. Only `pos > size()` throws.

5. B A view knows its length but the bytes continue after it. C APIs need a terminator, so build a `std::string` first and pass `c_str()`.

6. C A literal lives for the whole program, so a view of it can never dangle. The other three view something that is destroyed, may be reallocated, or has been resized under the view.

CHECKPOINT · 25 MIN

Checkpoint — Strings

This checkpoint covers the whole module: `std::string` as a value type, the `find` family and `npos`, slicing and replacing, characters as small integers and the `<cctype>` cast, `stoi` and `std::from_chars`, splitting and validating input, manipulators and `std::format`, and `std::string_view` with its lifetime rule.

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 `std::string::find` return when nothing matches, and how do you test for it?
- Why must a `char` be cast to `unsigned char` before calling `std::isalpha`?
- What is the difference between `std::stoi("42abc")` and `std::from_chars` on the same text?
- Which stream manipulators are sticky, and which one applies to the next insertion only?
- What does `std::setprecision(2)` print for `1234.5` without `std::fixed`?
- When does a `std::string_view` dangle, and what is the one rule that prevents it?
- Why does a replace-all loop advance by the replacement's length rather than the match's?

The three programs are a word-frequency count, a fixed-width report and a run-length codec; get the reading idiom right before the logic.

EXERCISE 1

Word frequency

Read all of standard input. A word is a maximal run of letters and digits (`std::isalnum` with the `unsigned char` cast), converted to lower case; every other character separates words. Print each distinct word with its count as `<word> <count>`, one per line, in ascending `std::string` order, then `total=<n> distinct=<m>`.

Input: any number of lines. **Output:** one line per distinct word, then the summary.

Example: input

```
Hello, hello! HELLO world.
World 42 42
```

→ output

```
42 2
hello 3
world 2
total=7 distinct=3
```

A `std::map<std::string, int>` keeps the words in order for you.

INPUT 1

```
Hello, hello! HELLO world.
World 42 42
```

OUTPUT 1

```
42 2
hello 3
world 2
total=7 distinct=3
```

INPUT 2

```
don't stop
```

OUTPUT 2

```
don 1
stop 1
t 1
total=3 distinct=3
```

REFERENCE SOLUTION

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

int main() {
    std::map<std::string, int> freq;
    long long total = 0;
    std::string word;

    auto flush = [&]() {
        if (!word.empty()) {
            ++freq[word];
            ++total;
            word.clear();
        }
    };

    std::string line;
    while (std::getline(std::cin, line)) {
        for (char c : line) {
            const unsigned char u = static_cast<unsigned char>(c);
            if (std::isalnum(u)) {
                word += static_cast<char>(std::tolower(u));
            } else {
                flush();
            }
        }
        flush();
    }

    for (const auto& [w, count] : freq) {
        std::cout << w << ' ' << count << '\n';
    }
    std::cout << "total=" << total << " distinct=" << freq.size() << '\n';
    return 0;
}
```

EXERCISE 2

Temperature report

Read an integer `N` (at least 1) and then `N` lines. Each line is a city name — one or more words separated by single spaces — followed by two decimals, the maximum and minimum temperature. Parse from the right: the last two tokens are the numbers and everything before them is the name.

Print a header and one row per city with `std::format`: header `{:<14}{:>7}{:>7}{:>7}` with `City`, `Max`, `Min`, `Range`; rows `{:<14}{:>7.1f}{:>7.1f}{:>7.1f}` with the name, max, min and range (max − min). Then print `Widest range: <city> (<range with one decimal>)` — the first city on a tie.

Input: `N`, then `N` lines. **Output:** `N + 2` lines.

Example: input

```
3
New York 28.5 19.0
Oslo 12.3 -3.5
Lima 21 15
```

→ output

```
City              Max    Min  Range
New York         28.5   19.0    9.5
Oslo             12.3   -3.5   15.8
Lima             21.0   15.0    6.0
Widest range: Oslo (15.8)
```

INPUT 1

```
3
New York 28.5 19.0
Oslo 12.3 -3.5
Lima 21 15
```

OUTPUT 1

```
City              Max    Min  Range
New York         28.5   19.0    9.5
Oslo             12.3   -3.5   15.8
Lima             21.0   15.0    6.0
Widest range: Oslo (15.8)
```

INPUT 2

```
1
San Jose de Costa Rica 30 20
```

OUTPUT 2

```
City              Max    Min  Range
San Jose de Costa Rica   30.0   20.0
10.0
Widest range: San Jose de Costa Rica
(10.0)
```

REFERENCE SOLUTION

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

int main() {
    int n = 0;
    std::cin >> n;
    std::cin.ignore();

    std::cout << std::format("{:<14}{:>7}{:>7}{:>7}\n", "City", "Max", "Min", "Range

    std::string widest_city;
    double widest = 0.0;
    bool have_widest = false;

    for (int i = 0; i < n; ++i) {
        std::string line;
        std::getline(std::cin, line);

        std::istringstream iss(line);
        std::vector<std::string> tokens;
        std::string token;
        while (iss >> token) tokens.push_back(token);

        const double mn = std::stod(tokens[tokens.size() - 1]);
        const double mx = std::stod(tokens[tokens.size() - 2]);
        std::string city;
        for (std::size_t k = 0; k + 2 < tokens.size(); ++k) {
            if (!city.empty()) city += ' ';
            city += tokens[k];
        }

        const double range = mx - mn;
        std::cout << std::format("{:<14}{:>7.1f}{:>7.1f}{:>7.1f}\n", city, mx, mn, r
        if (!have_widest || range > widest) {
            have_widest = true;
            widest = range;
            widest_city = city;
        }
    }

    std::cout << std::format("Widest range: {} ({:.1f})\n", widest_city, widest);
    return 0;
}
```

EXERCISE 3

Run-length codec

Read lines until end of input. Each line is `encode <text>` or `decode <text>`; the text contains no spaces.

> `encode`: the text is letters only. Replace every maximal run of one character with `<count><char>`: `aaabcc` → `3a1b2c`.
>
> `decode`: the text should be a sequence of `<count><char>` groups where the count is one or more digits and the char is a letter; expand it: `3a1b2c` → `aaabcc`. Counts may have several digits (`12x`). If the text is not well-formed — a count with no letter after it, a letter with no count before it, a count of zero, or an empty text — print `invalid`.

Use `c - '0'` to accumulate counts, `std::to_string` to emit them, and the `<cctype>` functions with the `unsigned char` cast.

Input: any number of command lines. **Output:** one line per command.

Example: input

```
encode aaabcc
decode 3a1b2c
decode 3
```

→ output

```
3a1b2c
aaabcc
invalid
```

INPUT 1

```
encode aaabcc
decode 3a1b2c
decode 3
```

OUTPUT 1

```
3a1b2c
aaabcc
invalid
```

INPUT 2

```
encode z
decode 12x
```

OUTPUT 2

```
1z
xxxxxxxxxxxx
```

REFERENCE SOLUTION

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

std::string encode(const std::string& text) {
    std::string out;
    std::size_t i = 0;
    while (i < text.size()) {
        std::size_t j = i;
        while (j < text.size() && text[j] == text[i]) ++j;
        out += std::to_string(j - i);
        out += text[i];
        i = j;
    }
    return out;
}

bool decode(const std::string& text, std::string& out) {
    out.clear();
    std::size_t i = 0;
    while (i < text.size()) {
        long long count = 0;
        std::size_t digits = 0;
        while (i < text.size() && std::isdigit(static_cast<unsigned char>(text[i]))) {
            count = count * 10 + (text[i] - '0');
            if (count > 1000000) return false;
            ++i;
            ++digits;
        }
        if (digits == 0 || count == 0) return false;
        if (i >= text.size() || !std::isalpha(static_cast<unsigned char>(text[i])))
        out.append(static_cast<std::size_t>(count), text[i]);
        ++i;
    }
    return !text.empty();
}

int main() {
    std::string line;
    while (std::getline(std::cin, line)) {
        std::istringstream iss(line);
        std::string command, text;
        iss >> command >> text;
        if (command == "encode") {
            std::cout << encode(text) << '\n';
        } else if (command == "decode") {
            std::string result;
            std::cout << (decode(text, result) ? result : "invalid") << '\n';
```

```
        }
    }
    return 0;
}
```

CHECKPOINT QUESTIONS

1. What does this print?

```
std::string s(3, 'x');
s += std::to_string(7);
std::cout << s;
```

A. `xxx7`

B. `3x7`

C. `xxx` followed by the character with code 7

D. It does not compile

2. `if ("abc" < "abd")` compiles with a warning. What does it compare?

A. The two strings' contents, lexicographically

B. The addresses of the two literals — an unspecified result

C. Their lengths

D. It always evaluates to `true`

3. What is `std::string::npos`?

A. `-1` as an `int`

B. The largest possible `std::size_t`, returned when a search finds nothing

C. `0`

D. The size of the string

4. What does this print?

```
std::string s = "a.b.c";
std::cout << s.substr(s.rfind('.') + 1);
```

A. `c`

B. `b.c`

C. `.c`

D. `a`

5. A counting loop restarts its search at `pos + needle.size()` after each match. How many matches of `"aa"` does it count in `"aaaa"`?

A. 2

B. 3

C. 4

D. 1

6. What is `static_cast<char>('a' + 2)`?

A. `'c'`

B. `'a2'`

C. `99` printed as a number

D. Undefined behaviour

7. After `auto [ptr, ec] = std::from_chars(first, last, value);`, which condition means "the whole token was a valid number"?

A. `ec == std::errc()`

B. `ptr != first`

C. `ec == std::errc() && ptr == last`

D. `value != 0`

8. What does `std::stoi("")` do?

A. Returns 0

B. Throws `std::invalid_argument`

C. Throws `std::out_of_range`

D. Undefined behaviour

9. `std::istringstream iss("a,,b"); while (std::getline(iss, f, ',')) ...` — what fields are produced?

A. `a`, `b`

B. `a`, `` (empty), `b`

C. `a,`, `,b`

D. One field, `a,,b`

10. `iss >> n` fails because the next token is `"x"`. Which statement is true?

A. `n` keeps its old value and the stream is unaffected

B. `n` becomes 0, the fail bit is set, and every later `>>` on `iss` fails until `iss.clear()`

C. An exception is thrown

D. The token `x` is skipped and the next one is read

11. Which manipulator applies to the **next insertion only**?

A. `std::setprecision`

B. `std::fixed`

C. `std::setw`

D. `std::setfill`

12. What does `std::format("{:>8.3f}", 3.14159)` return?

A. `3.142`

B. `3.142`

C. `3.14159`

D. `3.141`

13. Which is the rule that keeps a `std::string_view` safe?

A. Always convert it to a `std::string` before use

B. It must not outlive the string it views, and that string must not be modified while the view is in use

C. Only views of literals are allowed

D. It must be passed by `const&`

14. `void f(const std::string& s);` is called as `f("a fairly long literal string")`. What happens at the call?

A. Nothing is copied; the reference binds to the literal

B. A temporary `std::string` is constructed from the literal, allocating on the heap because it exceeds the small-string buffer

C. A compile error — a literal cannot bind to a reference

D. The literal is converted to a `std::string_view`

ANSWER KEY · CHECKPOINT — STRINGS

1. A `(3, 'x')` is the count-and-character constructor, and `std::to_string(7)` is the text `"7"`, appended as characters. `s += 7` (no `to_string`) would append the control character with code 7.

2. B Both operands are `const char*`, and `<` on pointers compares addresses. Wrap one side in `std::string` (or `std::string_view`) to compare characters.

3. B `npos` is `static_cast<std::size_t>(-1)` — the maximum unsigned value. Because it is unsigned, `find(...) ⩾ 0` is always true and the only valid test is `≠ std::string::npos`.

4. A `rfind('.')` finds the last dot at index 3; `substr(4)` returns everything after it. `find` would give `b.c`.

5. A Matches at 0 and 2 — non-overlapping. Restarting at `pos + 1` would also count the match at 1, giving 3.

6. A `'a' + 2` is the `int` 99; casting back to `char` gives the character with code 99, `'c'`. Without the cast, streaming the `int` prints `99`.

7. C `ec == std::errc()` says a number was parsed; `ptr == last` says nothing was left over. `"12abc"` satisfies the first but not the second.

8. B No digits could be parsed, which is `std::invalid_argument`. `std::out_of_range` is for digits whose value does not fit. Uncaught, either ends the program with a runtime error.

9. B A delimiter immediately followed by another yields an empty field between them. Only a *trailing* delimiter loses its empty field with `getline`.

10. B A failed extraction zeroes the target (since C++11) and puts the stream in a failed state that persists. Testing the read — `if (!(iss >> n))` — is how a bad token is detected.

11. C Width is reset to 0 after every insertion; precision, floating mode, fill and alignment all persist until changed.

12. A `.3f` rounds to three decimals (`3.142`), and `>8` right-aligns it in a field of eight characters with three leading spaces.

13. B A view is a pointer and a length into someone else's characters. Views of temporaries, of locals returned from a function, or of a container that then reallocates all dangle — undefined behaviour that often appears to work.

14. B A `const std::string&` can only bind to a `std::string`, so the compiler materialises one from the literal — an allocation for anything past 15 characters. Taking `std::string_view` by value avoids it.
