Module 1 of 20

C++ and the toolchain

Where C++ came from and the zero-overhead principle, the preprocess-compile-assemble-link pipeline, the anatomy of a program, the console I/O patterns every exercise uses, and how to read what the compiler says.

C++ has no virtual machine and no runtime that catches mistakes for you. Between pressing Run and seeing output, a preprocessor pastes your headers into a translation unit, a compiler turns it into machine code for one CPU, and a linker stitches the pieces into an executable — and the shape of the language (no garbage collector, values that copy, a category of errors the standard leaves undefined) follows from one promise: you do not pay for what you do not use. Before writing C++ well you need that map.

This module draws it. You will see where C++ came from and what "C++20" means, walk a two-file program through the four build stages and learn the vocabulary — translation unit, declaration versus definition, the one-definition rule, undefined reference — take apart the anatomy of a program (includes, `main`, `std::`, statements, scope, the exit code), master the console input and output idioms every exercise in this track uses, and learn to read a compiler diagnostic, a warning, a template error and a linker error, with sanitizers and a debugger in outline.

The exercises are small whole programs whose point is exact input and output: greet a name read with `std::getline`, map years to standards, print the build commands for a list of sources and simulate a linker's symbol resolution, compute through functions and namespaces, read N lines after a count and sum tokens until the input ends, summarise compiler diagnostics and apply `-Werror`. The checkpoint's three programs — a multiplication table, a word count and a student report — combine every reading pattern the module taught.

CONTENTS

1	What C++ is, and why it looks the way it does	12 min
2	From source to executable	14 min
3	Anatomy of a C++ program	13 min
4	Console input and output	16 min
5	Compilers, warnings and reading error messages	14 min

✓ Checkpoint — C++ and the toolchain 25 min

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

LESSON 1 · 12 MIN

What C++ is, and why it looks the way it does

C++ is a statically typed, compiled, multi-paradigm language built on one promise: you do not pay for what you do not use, and what you do use costs no more than the hand-written equivalent. Almost everything that makes C++ different from Java or Python — no garbage collector, no bounds checks unless you ask for them, values that are copied rather than shared, a whole category of mistakes the standard refuses to define — follows from that promise. This lesson is the map of those decisions; the other nineteen modules are the territory.

Where it came from

In 1979 Bjarne Stroustrup, at Bell Labs, wanted Simula's classes with C's speed and started "C with Classes"; the name became C++ in 1983 (`++` is C's increment operator — one more than C). The first compiler, Cfront, translated C++ into C, which is why early C++ ran wherever C did. The language grew for fifteen years before its first ISO standard and has been revised on a three-year rhythm since 2011:

Standard	Year	What it brought
C++98 / C++03	1998 / 2003	The first standard: templates, exceptions, namespaces, the STL; 03 was a bug-fix release
C++11	2011	The "modern C++" reset: `auto`, lambdas, move semantics, `std::unique_ptr`, range-based `for`, `std::thread`
C++14	2014	Polish: generic lambdas, return-type deduction, digit separators
C++17	2017	Structured bindings, `std::optional` / `variant` / `string_view`, `if constexpr`, guaranteed copy elision
C++20	2020	Concepts, ranges, `std::format`, `<=>`, `std::span`, modules and coroutines
C++23	2023	`std::print`, `std::expected`, deducing `this`, `std::mdspan`

"Modern C++" means the language as it has been since C++11. The idioms this track teaches — RAII, smart pointers, value semantics, algorithms over hand-written loops — are the C++11-

and-later way, and much of what you will read online about C++ predates them.

The zero-overhead principle

Stroustrup states it as two rules: *what you don't use, you don't pay for*, and *what you do use, you couldn't hand-code any better*. A `std::vector<int>` is a pointer, a size and a capacity — the same three words a hand-managed growable array needs — and indexing it is one multiply and one add, exactly the machine code a C array produces. A class with no virtual functions carries no hidden pointer; a destructor that does nothing costs nothing. The principle is why C++ has no garbage collector (you would pay for it whether or not you wanted it) and why `operator[]` on a vector does not check bounds (call `at()` when you want the check).

Compiled to native code

There is no virtual machine. A C++ compiler reads your source and emits machine instructions for one CPU and one operating system; the result is an executable the OS loads directly.

```
$ g++ -std=c++20 -O2 -Wall hello.cpp -o hello
$ ./hello
Hello, world
```

The program starts in microseconds and runs at the speed of the hardware; the compiler sees every type and can inline, unroll and fold across function boundaries; and the binary you built on Linux x86-64 will not run on an ARM Mac — you rebuild per platform. Lesson 2 walks through the pipeline stage by stage.

Static typing

Every variable, parameter and expression has a type the compiler knows before the program runs, and the compiler refuses a program whose types do not fit. `int n = "seven";` is not a run-time error to be caught; it is a program that does not exist.

```
int count = 3;
double ratio = count / 2;    // integer division: 1, then converted to 1.0
std::string s = count;       // error: no viable conversion from int to std::st
```

The second line compiles with a surprise — integer division — because C++ inherited C's implicit conversions among the arithmetic types; the third does not compile at all. Static typing is why so many bugs surface as a compile error rather than a 3 a.m. page. Module 2, Fundamental types, covers the types and their conversions.

Value semantics

In Java, `b = a` makes two references to one object. In C++, `b = a` makes a second object with the same contents:

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

int main() {
    std::string a = "hello";
    std::string b = a;        // a copy: separate characters, separate lifeti
    b += " world";
    std::cout << a << '\n';   // hello
    std::cout << b << '\n';   // hello world
    return 0;
}
```

Variables *are* objects; they are not handles to objects. Passing an argument copies it unless you ask for a reference (`const std::string&`), and assigning copies; sharing is explicit in the type. This is what lets a destructor run at a predictable moment — the object dies with its variable — and it is why Module 9, Copies, moves and the rule of five, exists: when copying is the default, what a copy *means* for a type that owns a resource has to be defined.

Undefined behaviour as a design decision

The C++ standard leaves some operations **undefined**: signed integer overflow, indexing past the end of an array, reading an uninitialised variable, dereferencing a null pointer, using an object after it has been destroyed. Undefined does not mean "crashes"; it means the standard makes no promise at all, and the optimiser is allowed to assume it never happens. A

bounds check you wrote *after* an out-of-bounds read may be deleted, because if the read happened the program was already outside the rules.

This is the zero-overhead principle applied to correctness: checking every index and every addition would cost every program something, so the language does not, and the responsibility is yours. Working C++ programmers cope with tools (`-Wall -Wextra` , the sanitizers — lesson 5), with habits (containers instead of raw arrays, brace initialisation, `at()` when the index comes from input) and with knowing the catalogue, which Module 19, Performance and undefined behaviour, lays out in full.

C and C++

C++ began as an extension of C and still compiles most C. It adds classes, templates, references, overloading, namespaces, exceptions, the standard library, and `new` / `delete` in place of `malloc` / `free` . Modern C++ uses almost none of the C parts directly: `std::string` instead of `char` arrays, `std::vector` instead of `malloc` , `std::cout` instead of `printf` . "C/C++" on a job advert names two languages with different idioms, and an interviewer notices a C programmer writing C++ as C.

Where C++ is used

The language is chosen where the cost model matters: game engines, browsers, databases, trading systems, embedded and automotive software, compilers (LLVM, GCC), the cores of machine-learning frameworks and operating-system components. All of them rely on the core this track teaches: the type system, ownership and RAII, the containers and algorithms, and what the compiler does with your code.

What "C++20" means, and what this track runs on

A standard is a document; a compiler implements some version of it, and you ask for one with a flag. The three compilers that matter are GCC (`g++`), Clang (`clang++`) and Microsoft's MSVC (`cl`). The study judge behind every exercise in this track compiles with **Clang 18, `-std=c++20` , `-O2` , libstdc++ 14, on x86-64 Linux**, with 5 s of CPU and 256 MB of memory per run. Concepts, ranges, `std::format` , `std::span` and `<=>` all compile there; the C++23 additions (`std::print` , `std::expected`) do not, and the track teaches them as reading.

Every exercise is a whole program: `int main()` reads standard input, prints to standard output and returns 0. A non-zero exit code or an uncaught exception is a runtime error, and only standard output is compared.

Pitfalls on day one

Expecting a runtime to catch you. There is no exception for an out-of-bounds index on a plain array; there is undefined behaviour. Use `std::vector` and `at()` until Module 6 shows you when a raw array is right.

Writing C in a `.cpp` file. `char name[100]; scanf("%s", name);` compiles, and it is how buffer overflows are born. `std::string name; std::cin >> name;` is the C++.

`#include <bits/stdc++.h>`. A GCC-specific header that pulls in the entire library: fine in a contest, meaningless to MSVC, and it hides which headers your code needs. Include the real ones.

Key takeaways

C++ (Stroustrup, 1979–83) is standardised every three years; C++11 reset the idioms, and C++20 is what this track compiles.

The zero-overhead principle explains most of the design: no garbage collector, no default bounds checks, abstractions that compile away.

Compiled to native code, statically typed, and values are values: `b = a` copies.

Undefined behaviour is the price of not checking everything; you pay it with tools and habits, not with a slower program.

The judge is Clang 18, `-std=c++20 -O2`, Linux x86-64; every exercise is `int main()` reading stdin and returning 0.

EXERCISE 1

Hello, you

Read one line — a name — from standard input and print `Hello, <name>!`.

Input: a single line containing a name (it may contain spaces). **Output:** `Hello,` followed by the name and `!`.

Use `std::getline` to read the line — `std::cin >> name` would stop at the first space. Print with `std::cout` and end the line with `'\n'`.

Example: input `Ada` → output `Hello, Ada!`

INPUT 1

```
Ada
```

OUTPUT 1

```
Hello, Ada!
```

INPUT 2

```
Bjarne Stroustrup
```

OUTPUT 2

```
Hello, Bjarne Stroustrup!
```

REFERENCE SOLUTION

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

int main() {
    std::string name;
    std::getline(std::cin, name);
    std::cout << "Hello, " << name << "!\n";
    return 0;
}
```

EXERCISE 2

Standard by year

Read an integer `n`, then `n` years (whitespace-separated, possibly across several lines). For each year print the C++ standard published that year — `C++98`, `C++03`, `C++11`, `C++14`, `C++17`, `C++20` or `C++23` for 1998, 2003, 2011, 2014, 2017, 2020 and 2023 — or `no standard` for any other year.

Input: `n` (at least 1), then `n` integers. **Output:** `n` lines, one per year, in input order.

Put the year-to-name mapping in a function above `main` that returns a `std::string`, and keep `main` to the reading loop.

Example: input `3 1998 2011 2020` →

```
C++98
C++11
C++20
```

INPUT 1

```
3
1998 2011 2020
```

OUTPUT 1

```
C++98
C++11
C++20
```

INPUT 2

```
2
2003 2024
```

OUTPUT 2

```
C++03
no standard
```

REFERENCE SOLUTION

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

std::string standard_for(int year) {
    if (year == 1998) return "C++98";
    if (year == 2003) return "C++03";
    if (year == 2011) return "C++11";
    if (year == 2014) return "C++14";
    if (year == 2017) return "C++17";
    if (year == 2020) return "C++20";
    if (year == 2023) return "C++23";
    return "no standard";
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int year;
        std::cin >> year;
        std::cout << standard_for(year) << '\n';
    }
    return 0;
}
```

QUIZ

1. What does the zero-overhead principle promise?

A. A C++ program has no run-time cost at all

B. You do not pay for features you do not use, and what you do use costs no more than hand-written code

C. The compiler removes every abstraction, so classes and templates vanish from the binary

D. Memory is reclaimed automatically at no cost

2. Which set of features arrived with C++11?

A. Templates, exceptions and namespaces

B. `auto`, lambdas, move semantics and `std::unique_ptr`

C. Concepts, ranges and `std::format`

D. `std::print` and `std::expected`

3. What does this print?

```
std::string a = "hello";
std::string b = a;
b += " world";
std::cout << a;
```

A. `hello`

B. `hello world`

C. A compile error: `b` is a copy and cannot be modified

D. Undefined behaviour

4. Signed integer overflow (`INT_MAX + 1`) in C++ is…

A. Guaranteed to wrap to `INT_MIN`

B. An exception

C. Undefined behaviour — the compiler may assume it never happens

D. Implementation-defined, but always wraps on x86-64

5. What does `-std=c++20` tell the compiler?

A. To optimise for the 2020 generation of CPUs

B. Which edition of the language standard to accept

C. To link the C++20 version of the standard library

D. To reject any code written before 2020

6. "C++ is a superset of C." Which statement is the accurate version?

A. Every valid C program is a valid C++ program with identical meaning

B. C++ grew out of C and keeps most of its syntax, but valid C can fail to compile as C++ (implicit `void*` conversions, new keywords) and modern C++ avoids the C idioms

C. C++ and C share syntax only by coincidence

D. C is what C++ compiles to

7. Which of these does NOT compile on this track's judge (Clang 18, `-std=c++20`)?

A. `std::format("{}", 42)`

B. A `requires` clause on a template

C. `std::print("{}\n", 42)`

D. `std::span<int>`

ANSWER KEY · WHAT C++ IS, AND WHY IT LOOKS THE WAY IT DOES

1. B Stroustrup's two rules: what you don't use you don't pay for, and what you do use you couldn't hand-code better. It is why there is no garbage collector (everyone would pay for it) and why `operator[]` does not check bounds. Abstractions are not removed — they are designed to compile to the same code you would write by hand.

2. B C++11 is the "modern C++" reset: `auto`, lambdas, rvalue references and moves, smart pointers, range-based `for`, `std::thread`. Templates and exceptions are C++98; concepts, ranges and `std::format` are C++20; `std::print` and `std::expected` are C++23.

3. A Value semantics: `b = a` creates a second, independent string. Appending to `b` does not touch `a`. In Java the same code would print `hello world` only if strings were mutable and `b` aliased `a`; in C++ variables are objects, not references.

4. C The standard leaves signed overflow undefined, and the optimiser exploits that: it may delete a check written after the overflow, or fold the expression to whatever is convenient. Unsigned arithmetic wraps by definition; signed does not, even though the hardware would.

5. B A standard is a document; the flag selects which one the compiler enforces. Without it GCC 14 defaults to C++17, and concepts, ranges and `std::format` would not compile. Optimisation is `-O2`; the library is linked automatically.

6. B C++ started as C with Classes and still compiles most C, but not all: `int* p = malloc(4);` is C, not C++, and `class`, `new` and `template` are identifiers in C. The first Cfront compiler did emit C, but that was an implementation, not the language's definition.

7. C `std::print` is C++23 and the judge compiles C++20. `std::format`, concepts and `std::span` are all C++20 and work. The track teaches the C++23 additions as reading only.

LESSON 2 · 14 MIN

From source to executable

`g++ hello.cpp -o hello` looks like one step. It is four programs run in sequence — the preprocessor, the compiler proper, the assembler and the linker — and every C++ error message you will ever read comes from exactly one of them. Knowing which stage produced a message tells you where to look: a compiler error means a file does not make sense on its own; a linker error means the files made sense individually but do not fit together. This lesson walks a two-file program through the pipeline and settles the vocabulary the rest of the track uses: translation unit, declaration, definition, object file, symbol, the one-definition rule.

A program in two files

```
// mathlib.h — the interface: declarations only
int square(int x);
int cube(int x);
```

```
// mathlib.cpp — the implementation: definitions
#include "mathlib.h"

int square(int x) { return x * x; }
int cube(int x) { return x * square(x); }
```

```
// main.cpp
#include <iostream>
#include "mathlib.h"

int main() {
    std::cout << square(7) << ' ' << cube(3) << '\n';
    return 0;
}
```

Build and run:

```
$ g++ -std=c++20 -O2 -Wall -c mathlib.cpp     # → mathlib.o
$ g++ -std=c++20 -O2 -Wall -c main.cpp        # → main.o
$ g++ mathlib.o main.o -o app                 # link
$ ./app
49 27
```

`-c` means "compile, do not link": each `.cpp` becomes an object file. The last command names no source file at all; it links the two objects into an executable. `g++ mathlib.cpp main.cpp -o app` does all of it in one go, but the three-command form is what a build system (Make, CMake, Ninja) actually runs, and it is why editing `mathlib.cpp` does not force `main.cpp` to be recompiled.

Stage 1: the preprocessor

The preprocessor is a text tool that runs before the compiler has any idea what C++ is. It handles every line that starts with `#`: `#include "mathlib.h"` is replaced by the entire text of that file; `#define N 100` replaces later `N`s with `100`; `#ifdef` / `#endif` keep or drop regions. `g++ -E main.cpp` prints its output — for the `main.cpp` above, tens of thousands of lines, almost all from `<iostream>`, followed by your ten. That output is the **translation unit**: one `.cpp` file after preprocessing, the unit the compiler compiles at once, and the reason a header is included rather than compiled. Angle brackets (`<iostream>`) search the system include directories; quotes (`"mathlib.h"`) search the including file's directory first, then the same places.

Because inclusion is textual, a header included twice is defined twice, and a class defined twice in one translation unit is an error. Every header therefore guards itself:

```
#pragma once            // or the classic #ifndef MATHLIB_H / #define MATHLIB_
int square(int x);
int cube(int x);
```

Stage 2: the compiler

The compiler proper parses the translation unit, checks every type, resolves every name it can and generates assembly for the target CPU (`g++ -S main.cpp` writes `main.s` if you want to read it). This is where `-std=c++20` (which edition of the language to accept), `-O2` (how hard to optimise; `-O0` is not at all, `-O3` is hardest) and `-Wall` (report the common mistakes as

warnings) matter. Everything the compiler knows comes from the one translation unit in front of it: when it sees `square(7)` in `main.cpp` it needs a **declaration** of `square` — a name and a type — to check the call; it does not need, and never sees, the body in `mathlib.cpp`. It emits a call to a symbol named `square(int)` and leaves a note for the linker: *someone else defines this*.

Stage 3: the assembler

The assembler turns the assembly text into machine code and packages it as an **object file** (`main.o`; `.obj` on Windows). An object file is not runnable. It holds code and data, a table of the symbols it *defines* (`main`) and a table of the symbols it *needs* (`square(int)`, `cube(int)`, `std::cout`, `operator<<`). `nm main.o` lists both. C++ symbol names are *mangled* — `square(int)` is stored as `_Z6squarei` — so that overloads and namespaces can coexist; `nm -C` demangles them.

Stage 4: the linker

The linker takes every object file and library, matches each needed symbol to exactly one definition, patches in the addresses and writes the executable. The standard library arrives here too: `std::cout` lives in `libstdc++.so`, which `g++` links automatically. Two things go wrong at this stage, and only at this stage:

```
$ g++ main.cpp -o app          # forgot mathlib.cpp
/usr/bin/ld: main.o: in function `main':
main.cpp:(.text+0x5): undefined reference to `square(int)'
collect2: error: ld returned 1 exit status
```

Undefined reference: something was declared and used, but no object file defines it. The compiler was satisfied — the declaration in the header was all it needed — and the failure surfaces only when the pieces are put together. Causes: a `.cpp` left off the command line, a library not linked, a function declared and never written, or a definition whose signature does not match the declaration (`square(long)` is a different symbol from `square(int)`).

Multiple definition: two object files both define the same non-inline function or variable, and the linker refuses to choose. The usual cause is a function *defined* in a header that two `.cpp` files include.

Declarations, definitions and the one-definition rule

A **declaration** introduces a name and its type: `int square(int x);`, `extern int counter;`, `class Widget;`. A **definition** is a declaration that also provides the thing: the function body, the variable's storage, the class's members. You may declare something as often as you like, and every use needs a declaration visible earlier in the translation unit. The **one-definition rule** (ODR) says a non-inline function or variable must be defined exactly once in the whole program, while a class, a template or an `inline` function may be defined in several translation units as long as every definition is token-for-token identical — which is what lets a class live in a header. Module 4 lesson 6 returns to `inline`, `static` and linkage; Module 8 lesson 6 puts the rule to work in the header/source split.

The flags this track uses

```
-std=c++20     the language edition; without it GCC 14 defaults to C++17
-O2            optimise; the judge uses it, so undefined behaviour is folded,
-Wall -Wextra  the warnings worth reading (lesson 5)
-c             compile to an object file, do not link
-o name        name the output
-g             debug information for gdb and the sanitizers
```

Pitfalls

A definition in a header. `int helper() { ... }` in `util.h` included by two files is a multiple-definition link error. Declare in the header, define in one `.cpp` — or mark it `inline`.

Reading a linker error as a compiler error. No line and column in *your* file, and the words `undefined reference` or `ld returned`, mean the compile succeeded; look at what you linked, not at your syntax.

Mismatched declaration and definition. `void print(const std::string&)` declared and `void print(std::string)` defined are two overloads; the call binds to the declared one and the link fails.

Trusting `#include` to be more than text. A header that uses `std::string` without including `<string>` works in one file and fails in the next.

Key takeaways

Preprocess (text: `#include`, `#define`) → compile (one translation unit → assembly) → assemble (object file) → link (resolve symbols → executable).

A translation unit is a `.cpp` after preprocessing; headers are included, never compiled on their own.

A declaration names a thing; a definition provides it. Declare in headers, define once in a `.cpp`.

Compiler errors are per file and carry a line and column; `undefined reference` is the linker, and means no object file defines the symbol.

The one-definition rule: one definition per program for functions and variables; identical definitions per translation unit for classes, templates and `inline`.

EXERCISE 1

Build lines

Print the commands a build system would run for a small project. The first line of input is the program name. Every following token (whitespace-separated, across any number of lines, until the input ends) is a `.cpp` source file. For each source print the compile line `g++ -std=c++20 -O2 -Wall -c <source> -o <object>`, where the object file name is the source with its `.cpp` replaced by `.o` (`src/util.cpp` becomes `src/util.o`). Then print the link line `g++ <every object, in order> -o <program>`.

Input: the program name on the first line, then the sources. **Output:** one compile line per source, then the link line.

Read the program name with `std::getline` and the sources with `while (std::cin >> source)`.

Example: input

```
app
main.cpp util.cpp
```

→

```
g++ -std=c++20 -O2 -Wall -c main.cpp -o main.o
g++ -std=c++20 -O2 -Wall -c util.cpp -o util.o
g++ main.o util.o -o app
```

INPUT 1

```
app
main.cpp util.cpp
```

OUTPUT 1

```
g++ -std=c++20 -O2 -Wall -c main.cpp -o
main.o
g++ -std=c++20 -O2 -Wall -c util.cpp -o
util.o
g++ main.o util.o -o app
```

INPUT 2

```
hello
hello.cpp
```

OUTPUT 2

```
g++ -std=c++20 -O2 -Wall -c hello.cpp -o
hello.o
g++ hello.o -o hello
```

REFERENCE SOLUTION

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

// "src/util.cpp" → "src/util.o": drop the four characters of ".cpp", append ".o".
std::string object_name(const std::string& source) {
    return source.substr(0, source.size() - 4) + ".o";
}

int main() {
    std::string program;
    std::getline(std::cin, program);
    std::string objects;
    std::string source;
    while (std::cin >> source) {
        std::string object = object_name(source);
        std::cout << "g++ -std=c++20 -O2 -Wall -c " << source << " -o " << object <<
        objects += " " + object;
    }
    std::cout << "g++" << objects << " -o " << program << '\n';
    return 0;
}
```

EXERCISE 2

A linker in miniature

Simulate the linker's symbol resolution. Read an integer `n`, then `n` lines, each `def <symbol>` (an object file defines the symbol) or `ref <symbol>` (an object file needs it). Then report, in this order:

- `multiple definition of '<symbol>'` — once for every `def` of a symbol that was already defined, in input order;

`undefined reference to '<symbol>'` — for every referenced symbol that no `def` line defines, in the order of first reference. The runtime's start-up code always references `main` before anything else, so a missing `main` is reported first, and a `ref main` line adds nothing new;

`link ok` if nothing was reported, otherwise `ld returned 1 exit status`.

Input: `n`, then `n` lines of `def` / `ref` and a symbol (no spaces in symbols). **Output:** the report described above.

The starter gives you a `contains` helper and three `std::vector<std::string>` lists; a symbol is referenced once however many `ref` lines name it.

Example: input

```
4
def main
ref compute
def helper
ref helper
```

→

```
undefined reference to 'compute'
ld returned 1 exit status
```

INPUT 1

```
4
def main
ref compute
def helper
ref helper
```

OUTPUT 1

```
undefined reference to 'compute'
ld returned 1 exit status
```

INPUT 2

```
3
def main
def compute
ref compute
```

OUTPUT 2

```
link ok
```

REFERENCE SOLUTION

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

bool contains(const std::vector<std::string>& names, const std::string& name) {
    for (const std::string& n : names) {
        if (n == name) return true;
    }
    return false;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<std::string> defined;
    std::vector<std::string> referenced{"main"};
    std::vector<std::string> duplicated;
    for (int i = 0; i < n; ++i) {
        std::string kind, symbol;
        std::cin >> kind >> symbol;
        if (kind == "def") {
            if (contains(defined, symbol)) duplicated.push_back(symbol);
            else defined.push_back(symbol);
        } else if (!contains(referenced, symbol)) {
            referenced.push_back(symbol);
        }
    }
    bool failed = false;
    for (const std::string& symbol : duplicated) {
        std::cout << "multiple definition of '" << symbol << "'\n";
        failed = true;
    }
    for (const std::string& symbol : referenced) {
        if (!contains(defined, symbol)) {
            std::cout << "undefined reference to '" << symbol << "'\n";
            failed = true;
        }
    }
    std::cout << (failed ? "ld returned 1 exit status" : "link ok") << '\n';
    return 0;
}
```

QUIZ

1. Which stage of the build replaces `#include <iostream>` with the header's text?

A. The linker

B. The preprocessor

C. The assembler

D. The loader, at run time

2. `undefined reference to 'compute(int)'` comes from…

A. The compiler: `compute` was never declared

B. The linker: `compute` was declared and called, but no object file defines it

C. The preprocessor: the header that declares `compute` was not found

D. The run-time loader

3. Which of these is a *definition*?

A. `int add(int, int);`

B. `extern int counter;`

C. `int add(int a, int b) { return a + b; }`

D. `class Widget;`

4. What does `g++ -c main.cpp` produce?

A. An executable named `a.out`

B. `main.o`, an object file that is not yet runnable

C. `main.s`, the assembly text

D. `main.i`, the preprocessed source

5. Two `.cpp` files each contain `int helper() { return 1; }` (not `inline`). Linking both into one program…

A. Works; the linker uses the first definition it finds

B. Fails with a multiple-definition error — the one-definition rule

C. Fails in the compiler, in whichever file is compiled second

D. Works, but calling `helper` is undefined behaviour

6. Where should the definition of an ordinary (non-inline, non-template) function go?

A. In the header, so every file that needs it can see the body

B. In exactly one `.cpp` file, with a declaration in the header

C. In every `.cpp` file that calls it

D. In `main.cpp`, always

7. What is a translation unit?

A. A header file

B. One `.cpp` file after preprocessing — the unit the compiler compiles at once

C. A function and everything it calls

D. The symbol table of an object file

ANSWER KEY · FROM SOURCE TO EXECUTABLE

1. B The preprocessor is a text tool that handles every `#` line before the compiler sees anything: `#include` is a paste, `#define` a substitution. Nothing about headers survives to the linker, and nothing is loaded at run time.

2. B The compiler was satisfied by the declaration and emitted a call to the symbol; the linker looked for a definition among the object files and libraries and found none. If `compute` had never been declared, the compiler would have stopped with `was not declared in this scope`.

3. C A definition provides the thing — the body, the storage, the members. The other three only introduce a name and a type: a function prototype, an `extern` variable declaration and a forward declaration of a class.

4. B `-c` means compile and assemble but do not link. `-S` stops at assembly and `-E` at preprocessing; with no flag at all, `g++` runs every stage and links to `a.out`.

5. B A non-inline function must be defined exactly once in the whole program. Each file compiles fine on its own — the compiler sees one translation unit at a time — and the clash appears when the linker finds two definitions of the same symbol.

6. B Declare in the header (any number of translation units may see a declaration), define once in a source file. A body in a header that two files include is a multiple-definition link error unless the function is `inline`.

7. B After `#include`s have been pasted in and macros expanded, the compiler compiles the resulting text as a single unit, knowing nothing about any other file. Headers are part of whichever translation units include them; they are never compiled on their own.

LESSON 3 · 13 MIN

Anatomy of a C++ program

Every C++ program you will write in this track has the same skeleton: some `#include` lines, perhaps a few functions, and `int main()`, inside which statements run from the first brace to the last. This lesson takes that skeleton apart — what an include is, what `main` must look like, why `std::` is everywhere and when `using` is acceptable, what counts as a statement, how blocks make scopes, and what the number `main` returns means to the world outside the program.

The smallest complete program

```
#include <iostream>

int main() {
    std::cout << "Hello, world\n";
    return 0;
}
```

Line by line. `#include <iostream>` pastes in the declarations of the standard streams (lesson 2 explained why it is a paste). `int main()` is the function the operating system calls; every program must define it, and its name and return type are fixed. `std::cout << "Hello, world\n";` is a statement: `std::cout` is the standard output stream, `<<` sends the string to it, and the `\n` inside the literal is a newline. `return 0;` hands the exit code back to whoever ran the program.

`main` has two legal signatures — `int main()` and `int main(int argc, char* argv[])`, the second receiving the command-line arguments as an array of C strings — and it must return `int`; `void main()` is a compiler extension the standard rejects. Uniquely, `main` may omit its `return`: falling off the closing brace returns 0. Any other non-void function that falls off its end is undefined behaviour, which is why `-Wall` warns about it (`-Wreturn-type`).

Headers you will include

Each standard facility lives in a header, and nothing is available until it is included:

Header	Provides
`<iostream>`	`std::cin`, `std::cout`, `std::cerr`
`<string>`	`std::string`, `std::getline`, `std::to_string`
`<vector>`, `<array>`, `<map>`, `<set>`	the containers
`<algorithm>`, `<numeric>`	`std::sort`, `std::max`, `std::accumulate`, …
`<iomanip>`	`std::setw`, `std::setprecision`, `std::fixed`
`<cmath>`	`std::sqrt`, `std::pow`, `std::abs` on doubles

Include what you use, even when it happens to compile without: one standard library's `<iostream>` pulls in `<string>` and another's does not, and code that depended on the accident breaks when the compiler changes. The C headers are spelt with a `c` prefix and no `.h` (`<cmath>`, `<cstdio>`), which puts their names inside `std::`.

Namespaces and `std::`

Every name in the standard library — `cout`, `string`, `vector`, `sort` — sits inside the namespace `std`, and `std::cout` reads as "the `cout` inside `std`". `::` is the scope-resolution operator. A namespace is a named scope whose only purpose is to keep names apart: your `max` and the library's `std::max` can coexist, and so can two libraries that both define a `Logger`.

```
namespace geometry {
    double area(double w, double h) { return w * h; }
}

namespace finance {
    double area(double principal, double rate) { return principal * rate; }
}

int main() {
    double a = geometry::area(3.0, 4.0);     // 12
    double b = finance::area(100.0, 0.05);   // 5
    return 0;
}
```

There are two ways to shorten `std::`:

```
using std::cout;        // a using-declaration: this one name, in this scope
using namespace std;    // a using-directive: every name in std, in this scope
```

The declaration is precise and safe. The directive is the one you will see in tutorials and contest code, and it works — until it does not. `std` contains `count`, `size`, `max`, `min`, `swap`, `left`, `right` and `hex`; a variable of your own called `count` then either hides the library's or, when both are visible, is *ambiguous*, a compile error that appears only when you add the include that brings the clash in. Inside a header the directive is unforgivable: it forces itself on every file that includes the header. This track writes `std::` explicitly everywhere except Module 20, which explains the contest trade-off. Five characters is a small price for never guessing where a name came from.

Statements and expressions

An **expression** computes a value: `x + 1`, `square(7)`, `a = b` (assignment is an expression whose value is the assigned value, which is why `a = b = 0` works). A **statement** is a unit of execution; it ends in `;` or is a block. Most statements are one of:

- an *expression statement* — any expression followed by `;`: `total += x;`, `std::cout << x;`, `square(7);` (result discarded), and even `x + 1;`, which is legal, does nothing and earns a `-Wunused-value` warning;
- a *declaration statement* — `int n = 0;`, `std::string line;`;
- a *compound statement* or **block** — zero or more statements in braces;
- a control statement — `if`, `for`, `while`, `switch`, `return` (Module 3).

`;` on its own is the empty statement, which is how `for (...);` becomes a loop with no body.

Blocks and scope

A name is visible from its declaration to the end of the block that contains it. Blocks nest, and an inner declaration of the same name **shadows** the outer one until the inner block closes:

```cpp
#include <iostream>

int main() {
    int x = 1;
    {
        int x = 2;            // a different x, shadowing the outer one
        std::cout << x;       // 2
    }                         // the inner x is destroyed here
    std::cout << x << '\n';   // 1 → the program prints "21"
    return 0;
}
```

Loop variables, `if`-initialiser variables and function parameters follow the same rule: they live in the block they belong to and die at its end. Declare a variable at the latest point you can, in the smallest block that needs it; it is the C++ habit that makes lifetimes obvious, and Module 7 shows how much depends on an object's lifetime being exactly its block. `-Wshadow` (not part of `-Wall`) reports shadowing when you want it flagged.

Functions before use

The compiler reads a translation unit top to bottom and must see a declaration of a function before a call to it. Define helpers above `main`, or declare them above and define them below:

```cpp
#include <iostream>

long long square(long long x);   // declaration (a prototype)

int main() {
    std::cout << square(1'000'000) << '\n';   // 1000000000000: long long, or
    return 0;
}

long long square(long long x) { return x * x; }   // definition
```

`1'000'000` uses a digit separator (C++14); the compiler ignores the apostrophes. Module 4 covers functions properly.

Comments

`//` runs to the end of the line; `/* ... */` spans lines and does not nest. Comments explain *why* — the invariant a loop keeps, the reason for a limit — not what the code visibly does. `// increment i` is noise; `// n is at most 10^5, so i * i fits in int` is a comment the next reader will thank you for.

What `main` returns

The `int` from `main` is the process's **exit code**. Zero means success; anything else means failure, and shells, build tools and the study judge all read it: a program that prints the right answer and then executes `return 1;` is marked as a runtime error, because the judge trusts the exit code before it trusts the output. `<cstdlib>` names the conventional values `EXIT_SUCCESS` and `EXIT_FAILURE`. An uncaught exception ends the program with a non-zero code as well (Module 15). For everything in this track: compute, print, `return 0`.

Pitfalls

`using namespace std;` **in a header**, or above code that defines its own `count`, `size` or `max`.

A missing `return` in a non-void function. Legal for `main`, undefined behaviour anywhere else, and the optimiser will happily produce nonsense. Read the `-Wreturn-type` warning.

A stray semicolon. `if (x > 0);` and `for (int i = 0; i < n; ++i);` both compile and do nothing useful.

Calling a function defined lower in the file with no declaration above the call: `'square' was not declared in this scope`.

Key takeaways

A program is includes, functions and `int main()`; `main` returns the exit code, and it alone may omit `return` (yielding 0).

Library names live in `std`; write `std::`, and reserve `using namespace std;` for contest scratch, never for headers.

Expressions produce values; statements execute; `x + 1;` compiles and does nothing.

Names live from their declaration to the end of their block; inner declarations shadow outer ones; declare before use.

A non-zero exit code means failure everywhere, including on the judge.

EXERCISE 1

Area and perimeter

Read the width and height of a rectangle and print its area and perimeter.

Input: one line with two integers, `w` and `h` (each between 1 and 2 000 000 000). **Output:** two lines, `area=<w*h>` and `perimeter=<2*(w+h)>`.

Write two functions above `main` — `area` and `perimeter` — and call them from `main`. Use `long long` throughout: the largest inputs overflow `int`.

Example: input `3 4` →

```
area=12
perimeter=14
```

INPUT 1

```
3 4
```

OUTPUT 1

```
area=12
perimeter=14
```

INPUT 2

```
100000 100000
```

OUTPUT 2

```
area=10000000000
perimeter=400000
```

REFERENCE SOLUTION

```cpp
#include <iostream>

long long area(long long w, long long h) {
    return w * h;
}

long long perimeter(long long w, long long h) {
    return 2 * (w + h);
}

int main() {
    long long w, h;
    std::cin >> w >> h;
    std::cout << "area=" << area(w, h) << '\n';
    std::cout << "perimeter=" << perimeter(w, h) << '\n';
    return 0;
}
```

EXERCISE 2

Two namespaces, one name

Convert temperatures to kelvin through two namespaces that share a function name. Define `namespace celsius` with `double to_kelvin(double c)` (kelvin = c + 273.15) and `namespace fahrenheit` with `double to_kelvin(double f)` (kelvin = (f − 32) × 5 / 9 + 273.15). Each input line is a scale letter, `C` or `F`, and a value; call the matching namespace's function with the scope-resolution operator and print the kelvin value with exactly two decimals.

Input: lines of `<C|F> <value>` until the input ends. **Output:** one line per input line: the temperature in kelvin, fixed to two decimals.

Read with `while (std::cin >> scale >> value)` where `scale` is a `char`. Set `std::fixed` and `std::setprecision(2)` once, before the loop.

Example: input

```
C 25
F 77
```

→

```
298.15
298.15
```

INPUT 1

```
C 25
F 77
```

OUTPUT 1

```
298.15
298.15
```

INPUT 2

```
C 100
```

OUTPUT 2

```
373.15
```

REFERENCE SOLUTION

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

namespace celsius {
    double to_kelvin(double c) {
        return c + 273.15;
    }
}

namespace fahrenheit {
    double to_kelvin(double f) {
        return (f - 32.0) * 5.0 / 9.0 + 273.15;
    }
}

int main() {
    std::cout << std::fixed << std::setprecision(2);
    char scale;
    double value;
    while (std::cin >> scale >> value) {
        double kelvin = (scale == 'C') ? celsius::to_kelvin(value) : fahrenheit::to_
        std::cout << kelvin << '\n';
    }
    return 0;
}
```

QUIZ

1. Why is `using namespace std;` in a header file a problem?

A. It slows compilation of that header

B. It forces the directive on every file that includes the header, so names like `count` or `size` can clash there without warning

C. It is a syntax error outside a `.cpp` file

D. It disables the `std::` prefix for the rest of the program

2. Given `int x = 1;`, which of these lines is a compile error?

A. `x + 1;`

B. `x = 1;`

C. `1 = x;`

D. `x;`

3. `int main()` reaches its closing brace with no `return` statement. What happens?

A. Undefined behaviour, as for any non-void function

B. The program returns 0 — `main` is the one function allowed to omit `return`

C. A compile error: missing return statement

D. The exit code is whatever was left in the return register

4. What does this print?

```
int x = 1;
{
    int x = 2;
    std::cout << x;
}
std::cout << x;
```

A. `12`

B. `21`

C. `22`

D. A compile error: `x` is redeclared

5. `using std::cout;` versus `using namespace std;`:

A. They are identical

B. The first brings the single name `cout` into scope; the second brings every name in `std`

C. The first is slower because it is resolved at run time

D. Only the second allows `cout` to be used without `std::`

6. A program prints the correct answer and then executes `return 1;` from `main`. The judge reports…

A. Accepted — only standard output is compared

B. A runtime error — a non-zero exit code means failure, whatever was printed

C. Accepted, with `1` appended to the output

D. A compile error

7. What is the difference between `#include <iostream>` and `#include "mathlib.h"`?

A. None; the two spellings are interchangeable

B. Angle brackets search the system include directories; quotes search the including file's directory first, then the same places

C. Quotes are for C headers and angle brackets for C++ headers

D. Angle-bracket includes are resolved by the linker

ANSWER KEY · ANATOMY OF A C++ PROGRAM

1. B A using-directive applies to the scope it appears in — for a header, every translation unit that includes it. Those files did not ask for every `std` name and may already have their own `count`, `max` or `left`; the resulting ambiguity error appears far from its cause.

2. C Assignment needs something assignable on the left; the literal `1` is not. `x + 1;` and `x;` are legal expression statements that do nothing (GCC warns with `-Wunused-value`). Unlike Java, C++ allows any expression to stand as a statement.

3. B The standard makes a special case for `main`: falling off the end is `return 0;`. Every other non-void function that reaches its end without returning is undefined behaviour, which `-Wreturn-type` warns about.

4. B The inner block declares a new `x` that shadows the outer one until the block closes, so the first print sees 2 and the second sees the outer 1. Redeclaring in an inner block is legal (and `-Wshadow` can flag it); redeclaring in the *same* block would be the error.

5. B A using-declaration names one entity and is precise; a using-directive opens the whole namespace. Both are compile-time; neither costs anything at run time. Prefer the declaration when you shorten anything at all.

6. B The exit code is the process's verdict on itself: zero is success, anything else is failure, and the judge (like every shell and build tool) reads it before the output. Return 0 from every program in this track.

7. B Both are textual pastes by the preprocessor; only the search path differs. The convention is angle brackets for the standard library and installed libraries, quotes for your own project's headers.

LESSON 4 · 16 MIN

Console input and output

Every exercise in this track — and every coding-round harness, contest judge and command-line tool — is a program that reads standard input and writes standard output. The C++ streams do this well once you know three facts: `>>` reads whitespace-separated tokens and skips the whitespace around them; `std::getline` reads a whole line, spaces included, and consumes the newline; and switching from the first to the second leaves a newline behind that you must throw away. This lesson fixes those facts, the read-until-the-input-ends loops, and the output habits that keep a judged program fast and its output exact.

The three streams

`<iostream>` declares `std::cin` (standard input), `std::cout` (standard output) and `std::cerr` (standard error). The judge feeds a test case to `cin` and compares only what you write to `cout`; `cerr` is shown to you as diagnostics and never marked wrong, so it is the place for debugging prints.

`>>` reads tokens

```
#include <iostream>

int main() {
    int n;
    std::cin >> n;
    long long total = 0;
    for (int i = 0; i < n; ++i) {
        long long x;
        std::cin >> x;
        total += x;
    }
    std::cout << total << '\n';
    return 0;
}
```

`std::cin >> x` skips any leading whitespace (spaces, tabs, newlines — any amount, across lines), then reads the characters that can belong to a value of `x`'s type and stops at the first

that cannot. For an `int` that is an optional sign and digits; for a `std::string`, everything up to the next whitespace; for a `char`, exactly one non-space character. The input `3 10 20 30` and the input `3⏎10⏎20⏎30⏎` read identically, which is why "N then N numbers" never needs to know the line layout. Chains read left to right: `std::cin >> name >> age;`.

When the next characters cannot form the value (`abc` into an `int`), or there are none left, the extraction **fails**: the stream sets its fail bit, the variable is set to 0, and every later `>>` does nothing until you `std::cin.clear()` it. In exercises the input is well-formed, so a failed read means your reading order is wrong, not the input.

`std::getline` reads lines

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

int main() {
    std::string line;
    std::getline(std::cin, line);
    std::cout << "[" << line << "]\n";
    return 0;
}
```

`std::getline(std::cin, line)` reads everything up to the next newline into `line`, discards the newline and returns the stream. Leading spaces are kept, an empty line yields an empty string, and a name with a space in it arrives whole — which `>>` could never manage. A third argument changes the delimiter: `std::getline(stream, field, ',')` reads up to the next comma, the basis of every CSV parser (Module 5 lesson 4).

Mixing them: the leftover newline

The most common input bug in C++:

```
int n;
std::cin >> n;                    // input "2⏎Ada Lovelace⏎": reads 2, leaves
std::string name;
std::getline(std::cin, name);     // reads up to the first newline: name is "
```

`>>` stops at the newline after `2` and leaves it in the buffer; `getline` meets that newline immediately and returns an empty line. The fix is to discard the rest of the current line before the first `getline`:

```
int n;
std::cin >> n;
std::cin.ignore();                // throw away one character: the newline
std::string name;
std::getline(std::cin, name);     // "Ada Lovelace"
```

`std::cin.ignore()` skips one character. When the line might carry trailing spaces after the number, the robust form is `std::cin.ignore(std::numeric_limits<std::streamsize>::max(), '\n');` (from `<limits>`), which skips everything up to and including the next newline. A third form, `std::getline(std::cin >> std::ws, name)`, skips *all* whitespace first — including empty lines, which is wrong when an empty line is meaningful. The rule: after `>>`, before `getline`, ignore the rest of the line.

Reading until the end of input

`std::cin >> x` returns the stream, and a stream converts to `false` once a read has failed. That gives the two loops every "read everything" program uses:

```
long long x;
while (std::cin >> x) {                    // one token at a time, any layout,
    total += x;
}

std::string line;
while (std::getline(std::cin, line)) {     // one line at a time, until the in
    ++lines;
}
```

There is no count to read first and no sentinel to look for: the loop ends when the input does. Do not write `while (!std::cin.eof())` — `eof()` becomes true only *after* a read has hit the end, so that loop processes the last value twice.

A line, then the tokens inside it

When a line's *content* is a list of tokens and the line boundary matters, read the line and then treat it as a stream:

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

int main() {
    std::string line;
    while (std::getline(std::cin, line)) {
        std::istringstream tokens(line);
        std::string word;
        int count = 0;
        while (tokens >> word) ++count;
        std::cout << count << '\n';
    }
    return 0;
}
```

`std::istringstream` (from `<sstream>`) gives a string the same `>>` interface as `cin`, so "how many words on each line" needs no hand parsing. Module 5 lesson 4 builds the full parsing toolkit on this idea.

`<<` writes

`std::cout << a << b << c` writes each value in turn — numbers in decimal, `bool` as `1` / `0` (`std::boolalpha` switches to `true` / `false`), a `char` as its character (`'A'` prints `A`; `int('A')` prints `65`), a `std::string` or literal as its text. Nothing is inserted between values: `std::cout << 1 << 2` prints `12`, and separating spaces are your job.

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

int main() {
    double ratio = 200.0 / 3;
    std::cout << ratio << '\n';                                    // 6
    std::cout << std::fixed << std::setprecision(2) << ratio << '\n';    // 6
    std::cout << std::setw(8) << 42 << '|' << std::left << std::setw(8) << "al
    return 0;
}
```

Floating-point output must always be pinned with `std::fixed` and `std::setprecision(k)` before it is judged; the default format prints six significant digits, drops trailing zeros and switches to scientific notation for large values, none of which a test case can rely on. `std::fixed`, `setprecision` and `std::left` are *sticky* — set once, they stay; `std::setw` applies to the next value only. Module 5 lesson 5 covers formatting in full, including `std::format`.

`'\n'` versus `std::endl`

Both end the line. `std::endl` also **flushes** the stream — forces the buffered text to be written out now — and a flush is a system call. In a loop printing a hundred thousand lines, `std::endl` costs a hundred thousand system calls and turns an accepted solution into a timeout. Write `'\n'`. The stream flushes on its own when its buffer fills and when the program ends, so nothing is lost.

Fast I/O

```
int main() {
    std::ios::sync_with_stdio(false);
    std::cin.tie(nullptr);
    // ... read and write as usual ...
    return 0;
}
```

By default the C++ streams are synchronised with C's `stdio`, so that `printf` and `std::cout` can be mixed, and `cin` flushes `cout` before every read. The first line switches the synchronisation off — after it, do not mix `scanf` / `printf` with the streams — and the second stops `cin` flushing `cout` on every read. Together they make the streams several times faster on large inputs, and they are the first two lines of every contest template (Module 20).

The patterns, in one place

Input format	Read it with
N, then N numbers in any layout	`std::cin >> n;` then a `for` of `std::cin >> x`
One line of text with spaces	`std::getline(std::cin, line)`
N, then N lines of text	`std::cin >> n; std::cin.ignore();` then N `getline`s
Numbers until the input ends	`while (std::cin >> x)`
Lines until the input ends	`while (std::getline(std::cin, line))`
A line whose tokens matter	`getline`, then `std::istringstream` and `>>`

Pitfalls

`getline` **right after** `>>` returns an empty line. `std::cin.ignore()` first.

`std::endl` **in a loop.** Use `'\n'`.

Unpinned floating-point output. Set `std::fixed << std::setprecision(k)` once, before printing.

`while (!std::cin.eof())` processes the last item twice; test the read itself.

Key takeaways

`>>` reads whitespace-separated tokens and ignores line boundaries; `std::getline` reads a whole line and eats the newline.

After `>>`, before `getline`: `std::cin.ignore()`.

`while (std::cin >> x)` and `while (std::getline(std::cin, line))` read until the input ends; never test `eof()` first.

`'\n'`, not `std::endl`; `std::fixed << std::setprecision(k)` for every floating-point value that is judged.

`std::ios::sync_with_stdio(false); std::cin.tie(nullptr);` for big inputs, and no C stdio after it.

EXERCISE 1

N, then N lines

Read an integer `n`, then `n` lines of text (a line may contain spaces and may be empty). Print each line as `<i>: <line>` with `i` counting from 1, then a final line `chars=<total>`, the sum of the lines' lengths.

Input: `n` (0 or more) on its own line, then `n` lines. **Output:** `n` numbered lines, then `chars=<total>`.

This is the mixed pattern: `std::cin >> n` leaves the newline behind, so you must `std::cin.ignore()` before the first `std::getline`. Read the lines with `std::getline` — nothing else keeps the spaces.

Example: input

```
2
hello world
C++ is fun
```

→

```
1: hello world
2: C++ is fun
chars=21
```

INPUT 1

```
2
hello world
C++ is fun
```

OUTPUT 1

```
1: hello world
2: C++ is fun
chars=21
```

INPUT 2

```
3
a
bb
ccc
```

OUTPUT 2

```
1: a
2: bb
3: ccc
chars=6
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore();   // the newline after n is still in the buffer; getline woul
    long long total = 0;
    for (int i = 1; i <= n; ++i) {
        std::string line;
        std::getline(std::cin, line);
        std::cout << i << ": " << line << '\n';
        total += static_cast<long long>(line.size());
    }
    std::cout << "chars=" << total << '\n';
    return 0;
}
```

EXERCISE 2

Numbers until the input ends

Read integers until the end of input — there is no count and no sentinel — and print `count=<n> sum=<s> min=<lo> max=<hi>` on one line. If there were no numbers at all, print `count=0`.

Input: zero or more integers in any layout (any number per line, any number of lines). Each fits in a 32-bit `int`; their sum may not. **Output:** one line.

Use `while (std::cin >> x)` — it is false the moment a read fails, which is how the loop ends at the end of input. Keep the sum in a `long long`.

Example: input `3 1 4 1 5` → `count=5 sum=14 min=1 max=5`

INPUT 1

```
3 1 4 1 5
```

OUTPUT 1

```
count=5 sum=14 min=1 max=5
```

INPUT 2

```
10
-20
30
```

OUTPUT 2

```
count=3 sum=20 min=-20 max=30
```

REFERENCE SOLUTION

```cpp
#include <iostream>

int main() {
    long long count = 0;
    long long sum = 0;
    long long min = 0;
    long long max = 0;
    long long x;
    while (std::cin >> x) {
        if (count == 0 || x < min) min = x;
        if (count == 0 || x > max) max = x;
        sum += x;
        ++count;
    }
    if (count == 0) {
        std::cout << "count=0\n";
    } else {
        std::cout << "count=" << count << " sum=" << sum << " min=" << min << " max=
    }
    return 0;
}
```

QUIZ

1. The input is `30⏎Ada⏎`. The code runs `int age; std::cin >> age; std::string name; std::getline(std::cin, name);`. What is `name`?

A. `"Ada"`

B. `"30"`

C. `""` (empty)

D. Undefined behaviour

2. `while (std::cin >> x)` stops when…

A. `x` is read as 0

B. The extraction fails — at the end of the input, or on a token that is not a number

C. A newline is read

D. It never stops without a `break`

3. What is the difference between `std::cout << x << '\n';` and `std::cout << x << std::endl;`?

A. None — they are interchangeable

B. `std::endl` also flushes the stream, which is a system call; in a loop of 100 000 lines it is dramatically slower

C. `'\n'` does not end the line on Linux

D. `std::endl` is required for the judge to see the output

4. What does `std::cout << 1 << 2 << '\n';` print?

A. `3`

B. `12`

C. `1 2`

D. A compile error

5. The next input characters are `3.7`. After `int x; std::cin >> x;` …

A. `x` is 3 and `.7` remains in the buffer for the next read

B. `x` is 4 (rounded)

C. The read fails and `x` is 0

D. `x` is 3.7 — the stream converts as needed

6. What do `std::ios::sync_with_stdio(false); std::cin.tie(nullptr);` do?

A. Make the streams faster on large inputs, at the price of never mixing `printf` / `scanf` with them afterwards

B. Disable output buffering so that every `<<` reaches the terminal immediately

C. Are required before `std::getline` can be used

D. Turn off all input validation

7. Which line prints exactly `66.67` for `double r = 200.0 / 3;` ?

A. `std::cout << r;`

B. `std::cout << std::fixed << std::setprecision(2) << r;`

C. `std::cout << std::setprecision(2) << r;`

D. `std::cout << std::setw(5) << r;`

ANSWER KEY · CONSOLE INPUT AND OUTPUT

1. C `>>` consumes `30` and stops at the newline, leaving it in the buffer; `getline` reads up to that newline immediately and returns the empty rest of the first line. `std::cin.ignore()` between the two reads discards the leftover newline.

2. B `>>` returns the stream, and a stream converts to `false` once its fail bit is set. End of input and a malformed token both set it. Line boundaries are just whitespace to `>>` and never end the loop.

3. B Both write a newline; `endl` additionally forces the buffer out. The stream flushes on its own when the buffer fills and at program exit, so nothing is lost with `'\n'`, and the judge sees everything.

4. B `<<` writes each value in turn with nothing between them; separating spaces are the programmer's job. It is not addition — the values are written as text, one after the other.

5. A `>>` for an `int` reads a sign and digits and stops at the first character that cannot belong to an integer. `3` is consumed, `.7` is left, and the next `>>` will meet it — a `double` would read it as 0.7, an `int` would fail.

6. A The first line stops the C++ streams from staying in step with C's `stdio`; the second stops `cin` from flushing `cout` before every read. Together they are several times faster on big inputs. After them, mixing `printf` and `std::cout` can interleave output unpredictably.

7. B `std::fixed` makes `setprecision` count digits after the decimal point. Without `fixed`, `setprecision(2)` means two *significant* digits and prints `67`; the default prints six significant digits, `66.6667`; `setw` only pads.

LESSON 5 · 14 MIN

Compilers, warnings and reading error messages

A C++ compiler is the most thorough reviewer your code will ever have, and it works for free — if you read what it says. This lesson teaches the shape of a diagnostic so that you can read it like a stack trace: which file, which line and column, what kind of message, and what the notes underneath are pointing at. It also covers the warning flags worth turning on, what a template error and a linker error look like, and the two run-time tools — sanitizers and a debugger — that catch what the compiler cannot.

The three compilers

Compiler	Driver	Where	Notes
GCC	`g++`	Linux default, MinGW on Windows	Ships libstdc++, the standard library most Linux C++ uses
Clang	`clang++`	macOS default (Apple Clang), Linux, Windows	LLVM-based; famously readable diagnostics; uses libstdc++ or its own libc++
MSVC	`cl`	Windows, Visual Studio	Different flag syntax (`/W4` , `/std:c++20`) and a different message format

GCC and Clang accept the same flags and produce diagnostics in the same `file:line:col: kind: message` shape, so what you learn on one transfers. The study judge is **Clang 18 with libstdc++ 14**, `-std=c++20 -O2` ; the local gate this track was written against is GCC 14 with `-Wall -Wextra` . Code that is clean on either compiles on the other unless it leans on a compiler extension — variable-length arrays, `__int128` , `#include <bits/stdc++.h>` — which is why the track avoids them.

Anatomy of a compile error

```cpp
#include <iostream>

int main() {
    int total = 0;
    for (int i = 0; i < 3; ++i) {
        totl += i;
    }
    std::cout << total << '\n';
}
```

```
main.cpp:6:9: error: use of undeclared identifier 'totl'; did you mean 'total
    6 |         totl += i;
      |         ^~~~
      |         total
main.cpp:4:9: note: 'total' declared here
    4 |     int total = 0;
      |         ^
1 error generated.
```

Read it left to right. `main.cpp:6:9` — file, line 6, column 9; every editor can jump there. `error` — the severity: an *error* stops the build, a *warning* does not, a *note* is not a problem at all but context for the message above it. Then the message, often with a suggestion. The source line and the caret show exactly which token, and the `note:` says where the related declaration is. `1 error generated.` is Clang's summary; GCC ends with nothing.

Two habits: **fix the first error first**, because one missing semicolon or unknown type cascades into dozens of nonsense errors below it; and **read the notes**, because for an overload or template failure the note chain (`candidate function not viable: ...`) is the actual explanation.

Warnings, and which flags to turn on

A warning is the compiler saying "this is legal and probably wrong". `-Wall` enables the ones almost always worth fixing; `-Wextra` adds a second tier; `-Wpedantic` complains about extensions the standard does not allow; `-Werror` turns every warning into an error, so a build with warnings fails — which is how teams keep the count at zero.

Warning (flag)	What it catches	Why it matters
unused variable (`-Wunused-variable`)	a name declared and never read	usually a typo elsewhere, or dead code
may be used uninitialised (`-Wuninitialized` , `-Wmaybe-uninitialized`)	a read before any write	undefined behaviour; initialise with `{}`
control reaches end of non-void function (`-Wreturn-type`)	a missing `return` on some path	undefined behaviour, and `-O2` makes it bite
comparison of integers of different signs (`-Wsign-compare`)	`i < v.size()` with `int i`	the `int` converts to unsigned; `-1 < v.size()` is false
unused value (`-Wunused-value`)	`x + 1;`	a statement with no effect
`-Wshadow` (not in `-Wall`)	an inner variable hiding an outer one	the bug from lesson 3
`-Wconversion` (not in `-Wall`)	`int x = 3.7;` , `long` into `int`	silent narrowing

Build with `-Wall -Wextra` from day one. A warning you have decided is harmless is noise that hides the next real one; fix it, or silence that one case deliberately, never by lowering the flags.

A template error

Template errors are long because the compiler reports the failure where it happened — deep in the library — and then the chain of instantiations that led there. Sorting a vector of a type that has no `<` :

```
/usr/include/c++/14/bits/predefined_ops.h:45:23: error: no match for 'operator
   45 |       { return *__it1 < *__it2; }
      |                ~~~~~~~^~~~~~~~
main.cpp:12:14:   required from here
   12 |     std::sort(points.begin(), points.end());
      |     ~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Read the **first line** for the actual problem (`no match for 'operator<'` on two `Point`s) and the `required from here` line (Clang says `in instantiation of ... requested here`) for *your* line that triggered it. Everything in between is the library's plumbing. C++20 concepts (Module 12 lesson 5) exist largely to turn this wall of text into one line: `constraints not satisfied`.

A linker error

```
/usr/bin/ld: main.o: in function `main':
main.cpp:(.text+0x1a): undefined reference to `compute(int)'
collect2: error: ld returned 1 exit status
```

No `error:` with a column, `ld` or `collect2` in the text, and `undefined reference` (or `undefined symbol` from Clang's `lld`): the compile stage passed, and the linker cannot find a definition. Lesson 2 listed the causes — a file not on the command line, a library not linked, a declaration whose definition was never written or does not match.

Sanitizers: the checks the compiler cannot do

Undefined behaviour is invisible at compile time by design. The sanitizers add the checks at run time:

```
$ g++ -std=c++20 -g -O1 -fsanitize=address,undefined main.cpp -o main
$ ./main
main.cpp:7:14: runtime error: signed integer overflow: 2147483647 + 1 cannot b
=================================================================
==4242==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x60200000003
READ of size 4 at 0x602000000034 thread T0
    #0 0x55d4 in main main.cpp:9
```

AddressSanitizer (ASan) reports out-of-bounds reads and writes, use-after-free, double free and leaks, with a stack trace naming your line. **UndefinedBehaviorSanitizer** (UBSan) reports signed overflow, bad shifts, null dereference and misaligned access. `-g` gives the reports line numbers. The program runs two to three times slower, so these are development flags, never release ones — and the judge, which compiles with plain `-O2`, cannot tell you *why* a run failed, only that it did. Module 7 lesson 6 and Module 19 use them on the memory-error catalogue.

A debugger, in outline

Build with `-g -O0` (debug information, no optimisation, so variables and lines are where you expect them), then:

```
$ gdb ./main
(gdb) break main.cpp:9        # stop at line 9
(gdb) run < input.txt         # start, feeding the file as stdin
(gdb) next                    # execute one line, stepping over calls
(gdb) step                    # execute one line, stepping into calls
(gdb) print total             # show a variable
(gdb) backtrace               # the call stack: where am I, and how did I get
(gdb) continue                # run to the next breakpoint
```

`lldb` (Clang's debugger) has the same commands with slightly different spellings, and every IDE wraps one of the two. The one command to know from memory is `backtrace` after a crash: it names the line that failed and the calls that led to it.

Pitfalls

Reading only the first word of the first error and guessing. Read the location, the message and the note.

Fixing the last error first. The first is the cause; the rest are often its consequences.

A warning-free build with `-w`. That flag suppresses all warnings; it is how bugs become invisible.

Debugging optimised code. At `-O2` variables vanish and lines reorder; build with `-O0 -g` to debug.

Key takeaways

A diagnostic is `file:line:col: severity: message`, followed by the source line, a caret and notes; errors stop the build, warnings do not, notes explain.

Fix the first error first and read the notes; for a template error, the first line and `required from here`.

`-Wall -Wextra` always; `-Werror` to keep a codebase at zero warnings; never `-w`.

`undefined reference` with `ld` in the text is the linker, not the compiler.

`-fsanitize=address,undefined` with `-g` finds memory errors and UB at run time; the judge (Clang 18, `-std=c++20 -O2`) has no sanitizers, so test locally with them.

EXERCISE 1

Diagnostic summary

Summarise a compiler's output. Read lines until the input ends. A line is a diagnostic when it contains `: error:` , `: warning:` or `: note:` ; everything before that marker is its location (`file:line:col`). Other lines (source excerpts, carets, `1 error generated.`) are ignored.

Input: zero or more lines. **Output:** two lines: `errors=<e> warnings=<w> notes=<n>` , then `first error: <location>` for the first error line, or `first error: none` .

Read with `while (std::getline(std::cin, line))` ; `line.find(": error: ")` returns the position of the marker or `std::string::npos` when it is absent, and `line.substr(0, pos)` is the location.

Example: input

```
main.cpp:12:5: error: use of undeclared identifier 'x'
main.cpp:3:10: note: 'y' declared here
main.cpp:20:9: warning: unused variable 'z' [-Wunused-variable]
util.cpp:4:1: error: expected ';' after expression
```

→

```
errors=2 warnings=1 notes=1
first error: main.cpp:12:5
```

INPUT 1

```
main.cpp:12:5: error: use of undeclared
identifier 'x'
main.cpp:3:10: note: 'y' declared here
main.cpp:20:9: warning: unused variable
'z' [-Wunused-variable]
util.cpp:4:1: error: expected ';' after
expression
```

OUTPUT 1

```
errors=2 warnings=1 notes=1
first error: main.cpp:12:5
```

INPUT 2

```
In file included from main.cpp:1:
util.h:7:3: warning: comparison of
integers of different signs [-Wsign-
compare]
```

OUTPUT 2

```
errors=0 warnings=1 notes=0
first error: none
```

REFERENCE SOLUTION

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

int main() {
    int errors = 0;
    int warnings = 0;
    int notes = 0;
    std::string first_error;
    std::string line;
    while (std::getline(std::cin, line)) {
        std::size_t at = line.find(": error: ");
        if (at != std::string::npos) {
            ++errors;
            if (first_error.empty()) first_error = line.substr(0, at);
        } else if (line.find(": warning: ") != std::string::npos) {
            ++warnings;
        } else if (line.find(": note: ") != std::string::npos) {
            ++notes;
        }
    }
    std::cout << "errors=" << errors << " warnings=" << warnings << " notes=" << not
    if (first_error.empty()) {
        std::cout << "first error: none\n";
    } else {
        std::cout << "first error: " << first_error << '\n';
    }
    return 0;
}
```

EXERCISE 2

Warnings as errors

Apply `-Werror` to a compiler's output. The first line of input is the set of flags the build used, for example `-Wall -Werror` or `-O2 -Wall`. Every following line is compiler output. When the flags contain `-Werror`, every `: warning:` marker is rewritten to `: error:` before the line is printed; otherwise lines are printed unchanged. Every line is echoed. After the last line print Clang's summary: `<w> warning(s) and <e> error(s) generated.` with correct singular/plural forms and only the non-zero parts — `1 warning generated.`, `2 errors generated.`, `1 warning and 2 errors generated.` — or `build ok` when there were neither.

Input: the flags line, then zero or more lines of compiler output. **Output:** every output line (promoted when `-Werror` is on), then the summary line.

`line.find(": warning: ")` locates the marker and `line.replace(pos, len, ": error: ")` rewrites it in place.

Example: input

```
-Wall -Werror
main.cpp:4:9: warning: unused variable 'x' [-Wunused-variable]
main.cpp:9:1: error: expected ';' after expression
```

→

```
main.cpp:4:9: error: unused variable 'x' [-Wunused-variable]
main.cpp:9:1: error: expected ';' after expression
2 errors generated.
```

INPUT 1

```
-Wall -Werror
main.cpp:4:9: warning: unused variable
'x' [-Wunused-variable]
main.cpp:9:1: error: expected ';' after
expression
```

OUTPUT 1

```
main.cpp:4:9: error: unused variable 'x'
[-Wunused-variable]
main.cpp:9:1: error: expected ';' after
expression
2 errors generated.
```

INPUT 2

```
-Wall
main.cpp:4:9: warning: unused variable
'x' [-Wunused-variable]
```

OUTPUT 2

```
main.cpp:4:9: warning: unused variable
'x' [-Wunused-variable]
1 warning generated.
```

REFERENCE SOLUTION

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

int main() {
    std::string flags;
    std::getline(std::cin, flags);
    bool werror = flags.find("-Werror") != std::string::npos;
    const std::string warning_marker = ": warning: ";
    int errors = 0;
    int warnings = 0;
    std::string line;
    while (std::getline(std::cin, line)) {
        std::size_t at = line.find(warning_marker);
        if (werror && at != std::string::npos) {
            line.replace(at, warning_marker.size(), ": error: ");
        }
        if (line.find(": error: ") != std::string::npos) ++errors;
        else if (line.find(": warning: ") != std::string::npos) ++warnings;
        std::cout << line << '\n';
    }
    if (errors == 0 && warnings == 0) {
        std::cout << "build ok\n";
        return 0;
    }
    if (warnings > 0) std::cout << warnings << (warnings == 1 ? " warning" : " warni
    if (warnings > 0 && errors > 0) std::cout << " and ";
    if (errors > 0) std::cout << errors << (errors == 1 ? " error" : " errors");
    std::cout << " generated.\n";
    return 0;
}
```

QUIZ

1. In `main.cpp:12:5: error: use of undeclared identifier 'x'`, what are `12` and `5`?

A. The error code and its severity

B. The line and the column

C. The column and the line

D. The number of errors so far and the number of notes

2. What does `-Werror` do?

A. Enables every warning

B. Turns warnings into errors, so a build with any warning fails

C. Suppresses warnings and shows only errors

D. Prints more detail for each error

3. A call to `std::sort` on a vector of your own type produces 300 lines of errors. Where do you look first?

A. The last line — that is where the compiler gave up

B. The first error line for the actual problem, and the `required from here` (or `in instantiation of … requested here`) line that names your call

C. The middle of the output, where the library code is

D. Nowhere — add `-w` and try again

4. What does `-fsanitize=address,undefined` do?

A. Makes the program run faster by removing checks

B. Instruments the compiled program to report memory errors and undefined behaviour at run time, naming the line

C. Fixes the bugs it finds

D. Is a linker flag that removes unused symbols

5. Which of these can NO warning flag catch at compile time?

A. A variable that is declared and never used

B. A non-void function with a path that reaches its end without `return`

C. Indexing a vector with a value read from input that turns out to be out of range

D. Comparing a signed `int` with an unsigned `size_t`

6. What does the study judge compile every exercise with?

A. GCC 14 with `-std=c++17` and no optimisation

B. Clang 18 with `-std=c++20 -O2` and libstdc++ 14, on x86-64 Linux

C. MSVC with `/std:c++20`

D. An interpreter that runs C++ line by line

7. Which output tells you the problem is in the *link* stage, not the compiler?

A. `main.cpp:8:5: error: expected ';' after expression`

B. `undefined reference to 'compute(int)'` followed by `collect2: error: ld returned 1 exit status`

C. `main.cpp:3:1: warning: unused variable 'n'`

D. `1 error generated.`

ANSWER KEY · COMPILERS, WARNINGS AND READING ERROR MESSAGES

1. B GCC and Clang diagnostics are `file:line:column: severity: message`. Line 12, column 5 is where the caret under the source line points; every editor can jump there from the text.

2. B `-Wall` and `-Wextra` enable warnings; `-Werror` changes their severity so the compiler refuses to produce output. Teams use it to keep the warning count at zero. `-w` (lower case) is the flag that suppresses warnings — and hides bugs.

3. B A template error is reported where the instantiation failed, deep in the library, followed by the chain back to the line that triggered it. The first line says what is missing (here, an `operator<` for your type); the `required from here` line says which of your lines caused it. The rest is plumbing.

4. B AddressSanitizer catches out-of-bounds accesses, use-after-free and leaks; UndefinedBehaviorSanitizer catches signed overflow, bad shifts and null dereferences. The program runs slower and prints a report when it hits one; nothing is fixed for you.

5. C The compiler cannot know what the input will be; an index it cannot see is exactly what AddressSanitizer or `at()` catches at run time. The other three are static properties of the code: `-Wunused-variable`, `-Wreturn-type` and `-Wsign-compare` report them.

6. B Clang 18, C++20, optimised, with the same standard-library generation as GCC 14. There are no sanitizers on the judge, so a run that fails there tells you *that* it failed, not why — test locally with `-fsanitize=address,undefined` when a hidden case fails.

7. B `ld` is the linker and `collect2` is GCC's wrapper around it; `undefined reference` means every file compiled but no object file defines the symbol. The other three are compiler messages with a file, line and column.

CHECKPOINT · 25 MIN

Checkpoint — C++ and the toolchain

This checkpoint covers the whole module: what C++ is and the decisions behind it, the four-stage pipeline from source to executable, the anatomy of a program, console input and output, and reading what the compiler says.

How it works. Fifteen 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:

- Which stage of the build turns `#include` into text, and which one reports `undefined reference`?
- What is a translation unit, and why is a header never compiled on its own?
- What does `main` return when it falls off its closing brace, and what does a non-zero exit code mean to the judge?
- Why does `std::getline` right after `std::cin >> n` return an empty line, and what is the fix?
- How does `while (std::cin >> x)` know when to stop?
- What is the difference between `'\n'` and `std::endl`?

The three programs use lesson 4's reading patterns — a token, lines until the input ends, and a count followed by alternating lines and tokens. Read each input format carefully; most failed submissions here are reading problems, not logic problems.

EXERCISE 1

Multiplication table

Read an integer `n` (1 ≤ n ≤ 12) and print the multiplication table for 1 through `n`: row `i` contains `i×1, i×2, …, i×n`, each number right-aligned in a field of width 4, with no trailing text.

Input: `n`. **Output:** `n` lines of `n` numbers.

Use `std::setw(4)` from `<iomanip>` before each number; it applies to the next value only.

Example: `n = 3` →

```
   1   2   3
   2   4   6
   3   6   9
```

INPUT 1

```
3
```

OUTPUT 1

```
   1   2   3
   2   4   6
   3   6   9
```

INPUT 2

```
1
```

OUTPUT 2

```
   1
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= n; ++j) {
            std::cout << std::setw(4) << i * j;
        }
        std::cout << '\n';
    }
    return 0;
}
```

EXERCISE 2

Word count

Implement a tiny `wc`: read all of standard input until it ends and print three lines — the number of lines, the number of words (whitespace-separated tokens) and the number of characters, counting every character including spaces and the newline that ends each line.

Input: zero or more lines, each ending in a newline. **Output:** `lines=<l>`, `words=<w>`, `chars=<c>`.

Use `while (std::getline(std::cin, line))`; every line read counts as one line and contributes its length plus one. Count the words on a line by wrapping it in a `std::istringstream` and extracting with `>>` until that fails.

Example: input

```
the quick brown
fox
```

→

```
lines=2
words=4
chars=20
```

INPUT 1

```
the quick brown
fox
```

OUTPUT 1

```
lines=2
words=4
chars=20
```

INPUT 2

```
one

two  three
```

OUTPUT 2

```
lines=3
words=3
chars=16
```

REFERENCE SOLUTION

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

int main() {
    long long lines = 0;
    long long words = 0;
    long long chars = 0;
    std::string line;
    while (std::getline(std::cin, line)) {
        ++lines;
        chars += static_cast<long long>(line.size()) + 1;   // + 1 for the newline g
        std::istringstream tokens(line);
        std::string word;
        while (tokens >> word) ++words;
    }
    std::cout << "lines=" << lines << '\n';
    std::cout << "words=" << words << '\n';
    std::cout << "chars=" << chars << '\n';
    return 0;
}
```

EXERCISE 3

Student report

Read an integer `n`, then `n` students. Each student takes two lines: the name (which may contain spaces), then a count `k` (at least 1) followed by `k` integer marks on the same line. Print one line per student, `<name>: <sum> (<average>)` with the average fixed to two decimals, then `top: <name>` for the student with the highest average — the first such student if several tie.

Input: `n`, then `n` pairs of lines. **Output:** `n` report lines, then the `top:` line.

This mixes every reading pattern: `>>` for `n`, `std::cin.ignore()` before the first `std::getline`, `getline` for the name, `>>` for the marks, and `ignore()` again before the next name.

Example: input

```
2
Ada Lovelace
3 90 80 70
Alan Turing
2 100 95
```

→

```
Ada Lovelace: 240 (80.00)
Alan Turing: 195 (97.50)
top: Alan Turing
```

INPUT 1

```
2
Ada Lovelace
3 90 80 70
Alan Turing
2 100 95
```

OUTPUT 1

```
Ada Lovelace: 240 (80.00)
Alan Turing: 195 (97.50)
top: Alan Turing
```

INPUT 2

```
1
Grace Hopper
4 100 100 100 99
```

OUTPUT 2

```
Grace Hopper: 399 (99.75)
top: Grace Hopper
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore();
    std::cout << std::fixed << std::setprecision(2);
    std::string best_name;
    double best_average = -1.0;
    for (int i = 0; i < n; ++i) {
        std::string name;
        std::getline(std::cin, name);
        int k;
        std::cin >> k;
        long long sum = 0;
        for (int j = 0; j < k; ++j) {
            long long mark;
            std::cin >> mark;
            sum += mark;
        }
        std::cin.ignore();   // the newline after the last mark, before the next nam
        double average = static_cast<double>(sum) / k;
        std::cout << name << ": " << sum << " (" << average << ")\n";
        if (average > best_average) {
            best_average = average;
            best_name = name;
        }
    }
    std::cout << "top: " << best_name << '\n';
    return 0;
}
```

CHECKPOINT QUESTIONS

1. Who created C++, and where?

A. Dennis Ritchie, at Bell Labs, in 1972

B. Bjarne Stroustrup, at Bell Labs, starting in 1979

C. James Gosling, at Sun Microsystems, in 1995

D. An ISO committee, in 1998

2. The four build stages, in order:

A. Compile, preprocess, assemble, link

B. Preprocess, compile, assemble, link

C. Link, compile, preprocess, run

D. Preprocess, link, compile, assemble

3. Which flag selects the edition of the language the compiler accepts?

A. `-O2`

B. `-std=c++20`

C. `-Wall`

D. `-c`

4. What does this print?

```cpp
#include <iostream>
int main() {
    int a = 5;
    int b = a;
    b = 7;
    std::cout << a << b << '\n';
}
```

A. `57`

B. `77`

C. `55`

D. A compile error: `main` has no `return`

5. An `#include` directive is…

A. A request to load a library at run time

B. A textual insertion of the file by the preprocessor, before compilation

C. An instruction to the linker

D. A namespace import, like Java's `import`

6. `error: 'compute' was not declared in this scope` is reported by…

A. The linker

B. The compiler — the name is unknown in this translation unit; a declaration or include is missing

C. The preprocessor

D. The operating system

7. `using namespace std;` at the top of a single small `.cpp` file is…

A. A compile error

B. Legal and common in scratch and contest code, but it opens every `std` name to clashes, so this track writes `std::` explicitly

C. Required for `std::cout` to work

D. A run-time cost

8. The input is `4⏎abc⏎`. After `int n; std::cin >> n; std::string line; std::getline(std::cin, line);`, `line` is…

A. `"abc"`

B. `""` (empty)

C. `"4"`

D. `"4abc"`

9. Which single call fixes the previous problem?

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

B. `std::cin.ignore();` after reading `n`

C. `std::cout.flush();`

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

10. A solution reads a million integers and times out. Which two lines are the standard speed-up?

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

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

C. `std::cin.ignore(); std::cin.clear();`

D. `std::cout << std::endl;` after every read

11. What does `std::cout << 7 / 2 << ' ' << 7.0 / 2 << '\n';` print?

A. `3 3.5`

B. `3.5 3.5`

C. `3 3`

D. `4 3.5`

12. Which statement about undefined behaviour is correct?

A. The program always crashes at the offending line

B. The compiler may assume it never happens, so the program may do anything — including appear to work

C. It is reported as a compile error

D. Sanitizers turn it into defined behaviour

13. The build printed `warning: unused variable 'x' [-Wunused-variable]` and nothing else. What happened?

A. The build failed

B. The build succeeded; the bracketed text names the flag that raised the warning, and `-Werror` would have turned it into a failure

C. `x` was read before being initialised

D. `x` is declared twice

14. Which of these makes the judge report a runtime error?

A. Printing to `std::cerr`

B. `return 1;` from `main`, or an uncaught exception

C. Printing a trailing space at the end of a line

D. Using `'\n'` instead of `std::endl`

15. What does `-O2` do, and why does it matter on the judge?

A. Enables the second tier of warnings

B. Asks for optimisation; the judge compiles with it, so code relying on undefined behaviour may behave differently from an unoptimised local build

C. Selects the second version of the language standard

D. Links the program twice for safety

ANSWER KEY · CHECKPOINT — C++ AND THE TOOLCHAIN

1. B Stroustrup's "C with Classes" began in 1979 and was renamed C++ in 1983; ISO standardised it in 1998. Ritchie created C; Gosling created Java.

2. B The preprocessor pastes and substitutes text, the compiler turns the translation unit into assembly, the assembler produces an object file, and the linker resolves symbols across object files into an executable.

3. B `-std=` names the standard. `-O2` is optimisation, `-Wall` enables warnings, `-c` compiles to an object file without linking.

4. A `b = a` copies the value; changing `b` afterwards leaves `a` at 5, and `<<` writes the two numbers with nothing between them. `main` may omit its `return` and yields 0.

5. B The preprocessor pastes the file's text in place; the compiler then sees one large translation unit. Nothing is loaded at run time, and — unlike `import` — the compiler has no notion of the header as a unit afterwards.

6. B The compiler needs a declaration to check a call, and it has none. Contrast `undefined reference`, which is the linker saying a declared symbol has no definition anywhere.

7. B The directive is fine where nothing else can include the file. It becomes a liability in headers and in any file that defines its own `count`, `size` or `max`. The explicit prefix costs five characters and removes the doubt.

8. B `>>` leaves the newline after `4` in the buffer and `getline` returns the empty remainder of that line. The `abc` is still unread.

9. B `ignore()` discards the leftover newline so the next `getline` starts on the next line. `clear()` resets error flags (there are none here), `flush` concerns output, and `sync()` has no defined effect on a standard input stream.

10. A Unsynchronising from C stdio and untying `cin` from `cout` makes the streams several times faster. The include and the directive change nothing at run time; `endl` in a loop makes things slower, not faster.

11. A `7 / 2` is integer division and truncates to 3; `7.0 / 2` has a `double` operand, so the division is floating-point and prints `3.5` in the default format.

12. B The standard makes no promise, and the optimiser exploits that freedom: checks can vanish and results can be nonsense that looks fine in testing. Sanitizers *detect* UB at run time; they do not define it.

13. B A warning does not stop the build. The `[-W…]` suffix names the specific flag, which is how you find the documentation or silence one case deliberately. Under `-Werror` the same line would be an error.

14. B The judge checks the exit code first: non-zero is failure. Standard error is ignored, trailing whitespace is normalised away, and `'\n'` is the recommended way to end a line.

15. B Optimisation levels change what the compiler assumes and reorders; undefined behaviour that happened to "work" at `-O0` can be folded into something else at `-O2`. Warnings are `-Wall` / `-Wextra`; the standard is `-std=`.
