Module 16 of 20

Modern C++

The standards from C++11 to C++23 and what compiles here; auto, decltype and structured bindings; constexpr, consteval and compile-time tables; the vocabulary types from tuple to span and chrono; concepts, ranges, <=> and std::format, with C++23 as reading.

"Modern C++" is used two ways: as a version — anything from C++11 on — and as a style, the habits that make a program safe and short: RAII and no raw `new`, value semantics, `const` and `constexpr` by default, `auto` where the type is obvious, algorithms and ranges over hand-written loops, vocabulary types in every signature. The earlier modules taught those habits one at a time; this module steps back and shows them as a whole, names the standard each piece came from, and states exactly what this track's runtime — Clang 18, libstdc++ 14, `-std=c++20` — compiles and what it does not.

The five lessons move from history to the future. The standards lesson tables what C++11, 14, 17, 20 and 23 each added, introduces feature-test macros and the values they take on this judge, and fixes the C++23 boundary: `std::print`, `std::expected`, `std::mdspan` and deducing `this` are taught for reading and the quiz only. The deduction lesson settles `auto` (it copies), `decltype` (the parentheses rule), structured bindings over pairs, tuples, structs, arrays and map entries, the `if` / `switch` initialiser, `using` aliases, `nullptr` and the `std::initializer_list` quirk. The compile-time lesson separates `const`, `constexpr`, `consteval` and `constinit`, shows what a C++20 `constexpr` function may contain, builds a lookup table in `std::array` checked by `static_assert`, and explains the one `consteval` rule that stops a first attempt from compiling. The vocabulary lesson covers `std::pair` and `std::tuple` with `std::tie` and `std::apply`, `std::span`, `<chrono>` durations printed deterministically, and `std::byte`. The C++20 lesson puts concepts, ranges, `⟺`, `std::format`, designated initialisers and `std::jthread` side by side, sketches modules and coroutines, and looks at C++23 and how to keep up.

The exercises are whole programs that use each feature for real: a feature-to-standard lookup and a modern rewrite of a C-style program, a stock lookup with an `if` initialiser and bindings over a map, a struct and an array taken apart by structured bindings, a compile-time prime sieve and a `consteval` factorial table both proved by `static_assert`, a leaderboard sorted through `std::tie` and printed through `std::apply`, window sums over `std::span` slices, concept-constrained shapes printed with `std::format`, and a server table built with designated initialisers

and sorted by a defaulted `⟺`. The checkpoint adds a `<chrono>` duration ledger, a day-of-year calculator on a `constexpr` prefix-sum table with `std::optional`, and a concept-constrained statistics function over a `std::span`.

CONTENTS

1	The standards — what each C++ added, and what runs here	12 min
2	auto, decltype and structured bindings	15 min
3	constexpr, consteval and compile-time tables	15 min
4	The vocabulary types — pair, tuple, span, chrono and friends	15 min
5	C++20 and beyond — concepts, ranges, format and what is coming	16 min
✓	Checkpoint — Modern C++	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

The standards — what each C++ added, and what runs here

"C++" names a family of standards, not one language. The C++ of a 2003 textbook has no `auto`, no lambdas, no smart pointers in the library and no move semantics; the C++ this track teaches has all of those and treats them as the default. A version number is therefore the first thing to establish about any C++ code: what it may use, what its compiler must support, and which idioms it should be judged by. This lesson lays out the timeline, what each standard contributed, how a program can ask its own compiler what it supports, exactly what compiles on this track's runtime, and what "modern C++" means when people use it as a compliment.

One language, six editions

C++ was standardised by ISO in 1998, patched in 2003, and then stalled for eight years. C++11 was the reset — Bjarne Stroustrup called it "a new language" — and since then the committee has shipped on a three-year clock: C++14, C++17, C++20, C++23, with C++26 in progress. The releases alternate in size: 11 and 20 each rewrote how idiomatic code looks, while 14, 17 and 23 mostly filled in what the previous release had left rough.

Standard	Year	Character
C++98 / C++03	1998 / 2003	Templates, exceptions, the STL, namespaces — the "classic" language
C++11	2011	The reset: `auto`, lambdas, move semantics, smart pointers, threads
C++14	2014	Generalisations and fixes of C++11
C++17	2017	Vocabulary types, structured bindings, `if constexpr`, filesystem
C++20	2020	Concepts, ranges, modules, coroutines, `std::format`, ⟺
C++23	2023	`std::print`, `std::expected`, `std::mdspan`, deducing `this`

A compiler is not "a version"; it is told one with `-std=c++20`, and GCC, Clang and MSVC each implement the newer standards at their own pace. "Does this compile?" is therefore a

question about a compiler-and-flag pair, never about the language alone.

C++11: the language you actually write

Almost everything this track has called idiomatic arrived in 2011. If a codebase is stuck before it, the code looks like C with classes.

Area	C++11 additions
Declarations	`auto`, `decltype`, `nullptr`, `enum class`, `static_assert`, `constexpr`, `long long`
Functions	Lambdas, rvalue references and move semantics, `= default` / `= delete`, `override` / `final`, `noexcept`, delegating constructors
Loops and init	Range-based `for`, uniform brace initialisation, `std::initializer_list`
Templates	Variadic templates, alias templates (`using`)
Library	`std::unique_ptr` / `shared_ptr` / `weak_ptr`, `std::array`, `std::tuple`, `std::function`, `std::unordered_map`, `<chrono>`, `<random>`, `<regex>`, `std::thread` / `mutex` / `atomic`, `std::to_string`

The three that changed the most code: `auto` (declarations stopped repeating types), lambdas (algorithms became usable in place), and move semantics (returning a `std::vector` by value became free, so functions started returning values instead of filling output parameters).

C++14 and C++17: filling in

C++14 relaxed what C++11 had allowed only narrowly: `constexpr` functions may contain loops and locals, lambdas may take `auto` parameters (generic lambdas) and capture by initialiser (`[p = std::move(p)]`), ordinary functions may deduce their return type, and `std::make_unique` — forgotten from C++11 — was added. Binary literals (`0b1010`) and digit separators (`1'000'000`) date from here.

C++17 is the release whose features you meet in every lesson of this module: structured bindings, `if` and `switch` with an initialiser, `if constexpr`, class template argument

deduction (`std::pair p{1, 2.0}`), fold expressions, `inline` variables, guaranteed copy elision, nested namespace declarations, and the attributes `[[nodiscard]]` , `[[maybe_unused]]` and `[[fallthrough]]` . The library gained its vocabulary types — `std::optional` , `std::variant` , `std::any` , `std::string_view` , `std::byte` — plus `std::filesystem` , `std::from_chars` , `std::apply` , `std::clamp` , `std::gcd` , `std::size` and the parallel algorithm overloads.

C++20: the second big one

C++20 brought four "big" features and a long tail of small ones. Concepts constrain templates and turn a page of template errors into one sentence. Ranges give algorithms a container-level interface and lazy `views` . Coroutines and modules are language-level infrastructure that library and build-system support is still catching up with (lesson 5). The tail is where daily code changed: the three-way comparison `⟺` , `std::format` , `std::span` , `std::jthread` , designated initialisers, `consteval` and `constinit` , `constexpr` `std::vector` and `std::string` , `[[likely]]` , `using enum` , `<bit>` , `<numbers>` , `std::source_location` , `contains()` on the associative containers, `starts_with` / `ends_with` , `std::erase_if` , `std::midpoint` , and calendars and time zones in `<chrono>` .

C++23: reading only on this track

C++23 added `std::print` and `std::println` (formatted output without `<<`), `std::expected` (a value or an error, without exceptions), `std::mdspan` (a multidimensional view over contiguous memory), *deducing* `this` (explicit object parameters, which make const/non-const overload pairs far shorter), `std::ranges::to` (materialise a view into a container), `std::ranges::fold_left` , `if consteval` , `std::flat_map` , `std::generator` , `std::stacktrace` , monadic `std::optional` and `import std;` . libstdc++ 14 has `std::expected` under `-std=c++23` , but this track compiles with `-std=c++20` , where none of it exists. Everything C++23 in this module is taught for reading and the quiz; no exercise depends on it.

Feature-test macros

A program can ask the compiler what it supports. `__cplusplus` expands to the standard's date: `199711L` for C++98/03, `201103L` , `201402L` , `201703L` , `202002L` , `202302L` . Beyond that,

every feature has a macro: language features are predefined (`__cpp_concepts` , `__cpp_constexpr` , `__cpp_structured_bindings`), and library features (`__cpp_lib_format` , `__cpp_lib_ranges` , `__cpp_lib_print`) are defined by `<version>` , a C++20 header that exists only to publish them.

```
#include <iostream>
#include <version>

int main() {
    std::cout << __cplusplus << '\n';          // 202002 on this track
#ifdef __cpp_lib_format
    std::cout << "format " << __cpp_lib_format << '\n';   // 202110
#endif
#ifndef __cpp_lib_print
    std::cout << "no std::print here\n";       // C++23: absent under -std=c++
#endif
    return 0;
}
```

Measured on this track's judge, the program prints `202002` , `format 202110` and `no std::print here` . The value is a date, so `#if __cpp_lib_format ≥ 202106L` asks for "at least the revision that added X" rather than "is it there at all". Two cautions: MSVC reports `__cplusplus` as `199711L` unless given `/Zc:__cplusplus` , so test `_MSVC_LANG` there; and a macro proves the compiler *claims* the feature, which is not the same as the feature being complete.

What compiles on this track

The study judge is Clang 18 with libstdc++ 14 and `-std=c++20 -O2` . Everything in the C++11 through C++20 rows above compiles and runs — including concepts, ranges and views, `std::format` , `std::span` , `⟺` , `std::jthread` , designated initialisers, `consteval` and `constinit` , `constexpr` containers inside a function, `<bit>` and `<numbers>` . Modules and coroutines compile in principle but need build-system or library scaffolding a single-file judge does not have, so they stay in prose. Nothing from the C++23 row compiles.

"Modern C++" is a style, not a version

Ask what "modern C++" means and the answer is a set of habits rather than a flag. Compare two programs that do the same thing:

```
// The 2003 shape
int* data = new int[n];
for (int i = 0; i < n; i++) std::cin >> data[i];
std::sort(data, data + n);
long long sum = 0;
for (int i = 0; i < n; i++) sum += data[i];
delete[] data;
```

```
// The modern shape
std::vector<int> data(n);
for (auto& x : data) std::cin >> x;
std::ranges::sort(data);
const long long sum = std::accumulate(data.begin(), data.end(), 0LL);
```

The second is shorter, but that is not the point. It cannot leak (the container owns its memory — Module 7), cannot walk off the end of the buffer (range-`for` — Module 3), cannot be sorted with the wrong length, and states its intent — sort, accumulate — instead of spelling out the loops. The habits, in the order the track taught them: RAII and no raw `new`; value semantics and return by value; `const` and `constexpr` by default; `auto` where the type is obvious or unnameable; algorithms and ranges over hand-written loops; `std::string_view` and `std::span` for read-only parameters; `enum class`; `nullptr`; brace initialisation; and explicit `std::`. A C++20 compiler running C++03 habits is not modern; a C++17 compiler running these habits is.

Pitfalls

Mistake	What happens
Writing C++23 (`std::print` , `std::expected`) for this judge	Compile error: the headers or names do not exist under `-std=c++20`
Trusting `__cplusplus` on MSVC	`199711L` regardless of the real mode unless `/Zc:__cplusplus` is set
`#ifdef __cpp_lib_format` without `<version>` (or `<format>`)	The macro is never defined, so the fallback path always compiles
Reading "C++11 is old" as "C++11 features are legacy"	`auto` , lambdas and smart pointers are the core of every later standard
Forgetting `-std=` on the command line	GCC 14 defaults to C++17; C++20 features fail to compile with confusing errors

Key takeaways

C++ ships every three years: 11 and 20 were the large releases, 14, 17 and 23 the refinements.

C++11 brought `auto` , lambdas, moves, smart pointers and threads; C++17 the vocabulary types and structured bindings; C++20 concepts, ranges, `std::format` , `std::span` and `⟺` .

`__cplusplus` and the `__cpp_*` macros from `<version>` let a program test its own compiler; the values are dates.

This track compiles with Clang 18, libstdc++ 14, `-std=c++20` : everything through C++20 runs, and C++23 is reading only.

Modern C++ is a style — RAII, value semantics, `const` , `auto` , algorithms, no raw `new` — not a compiler flag.

EXERCISE 1

Which standard introduced it

The starter holds a table mapping feature names to the standard that introduced them, as the two-digit year (`"lambdas"` → `11` , `"concepts"` → `20`). Read an integer `n` , then `n` feature names, one per line, and for each print `<name>: C++<yy>` when it is in the table and `<name>: not in the table` otherwise.

Look each name up **once**, with an `if` that declares the iterator in its initialiser (`if (auto it = table.find(name); it ≠ table.end())`) — not `count()` followed by `[]`.

Input: `n`, then `n` names (lower-case, hyphenated, no spaces). **Output:** `n` lines.

```
4
lambdas
concepts
structured-bindings
goto
```

prints

```
lambdas: C++11
concepts: C++20
structured-bindings: C++17
goto: not in the table
```

INPUT 1

```
4
lambdas
concepts
structured-bindings
goto
```

OUTPUT 1

```
lambdas: C++11
concepts: C++20
structured-bindings: C++17
goto: not in the table
```

INPUT 2

```
3
print
format
optional
```

OUTPUT 2

```
print: C++23
format: C++20
optional: C++17
```

REFERENCE SOLUTION

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

int main() {
    // Feature name → the standard that introduced it (its two-digit year).
    const std::map<std::string, int> introduced{
        {"auto", 11}, {"lambdas", 11}, {"range-for", 11}, {"nullptr", 11}, {"move-se
        {"unique-ptr", 11}, {"shared-ptr", 11}, {"constexpr", 11}, {"enum-class", 11
        {"make-unique", 14}, {"generic-lambdas", 14}, {"binary-literals", 14}, {"ret
        {"optional", 17}, {"variant", 17}, {"string-view", 17}, {"structured-binding
        {"if-constexpr", 17}, {"fold-expressions", 17}, {"filesystem", 17},
        {"concepts", 20}, {"ranges", 20}, {"format", 20}, {"span", 20}, {"spaceship'
        {"designated-initialisers", 20}, {"coroutines", 20}, {"modules", 20}, {"cons
        {"print", 23}, {"expected", 23}, {"mdspan", 23}, {"deducing-this", 23},
    };
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        std::string feature;
        std::cin >> feature;
        if (auto it = introduced.find(feature); it ≠ introduced.end()) {
            std::cout << feature << ": C++" << it→second << '\n';
        } else {
            std::cout << feature << ": not in the table\n";
        }
    }
    return 0;
}
```

EXERCISE 2

A modern rewrite

This is a working program in the 2003 style:

```cpp
int n;
std::cin >> n;
int* data = new int[n];
for (int i = 0; i < n; i++) std::cin >> data[i];
std::sort(data, data + n);
long long sum = 0;
for (int i = 0; i < n; i++) sum += data[i];
// … print, then
delete[] data;
```

Rewrite it in the modern style: a `std::vector<int>` sized from `n` (no `new`), a range-based `for` with `auto&` to read the values, `std::ranges::sort`, and `std::accumulate` with a `0LL` initial value for the sum. Then print four lines: `sorted:` followed by the values in ascending order each preceded by one space, `sum: <sum>`, `min: <smallest>` and `max: <largest>`. When `n` is 0 print `sorted:` with nothing after it, `sum: 0`, `min: none` and `max: none`.

Input: `n`, then `n` integers (any whitespace). **Output:** four lines.

Example: input `4` / `5 3 9 1` →

```
sorted: 1 3 5 9
sum: 18
min: 1
max: 9
```

INPUT 1

```
4
5 3 9 1
```

OUTPUT 1

```
sorted: 1 3 5 9
sum: 18
min: 1
max: 9
```

INPUT 2

```
3
-2 -7 4
```

OUTPUT 2

```
sorted: -7 -2 4
sum: -5
min: -7
max: 4
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::vector<int> values(n > 0 ? n : 0);
    for (auto& v : values) std::cin >> v;
    std::ranges::sort(values);
    std::cout << "sorted:";
    for (const int v : values) std::cout << ' ' << v;
    std::cout << '\n';
    const long long sum = std::accumulate(values.begin(), values.end(), 0LL);
    std::cout << "sum: " << sum << '\n';
    if (values.empty()) {
        std::cout << "min: none\nmax: none\n";
    } else {
        std::cout << "min: " << values.front() << '\n';
        std::cout << "max: " << values.back() << '\n';
    }
    return 0;
}
```

QUIZ

1. Which standard introduced `auto` type deduction, lambdas, range-based `for`, `nullptr` and move semantics?

A. C++98

B. C++11

C. C++14

D. C++17

2. On this track's runtime (Clang 18, `-std=c++20`), what does `std::cout << __cplusplus;` print?

A. `20`

B. `201703`

C. `202002`

D. `202302`

3. Which of these compiles on the study judge?

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

B. `std::expected<int, std::string> r;`

C. `std::cout << std::format("{}", 42);`

D. `auto v = xs | std::views::filter(f) | std::ranges::to<std::vector>();`

4. What does `#ifdef __cpp_lib_format` test, and where is the macro defined?

A. Whether the compiler flag was `-std=c++20`; it is predefined by the compiler

B. Whether the standard library implements `std::format`; it is defined by `<version>` (and `<format>`)

C. Whether formatting is enabled at run time; it is set by `std::locale`

D. Whether the program has included `<iostream>`

5. Which line is *not* in the modern C++ style?

A. `auto p = std::make_unique<Widget>();`

B. `int* buf = new int[n]; /* … */ delete[] buf;`

C. `for (const auto& item : items)`

D. `std::ranges::sort(v);`

6. Which of these arrived in C++14?

A. `std::make_unique`

B. `std::make_shared`

C. `std::unique_ptr`

D. `std::optional`

7. Since C++11, how often does a new C++ standard ship?

A. Every year

B. Every two years

C. Every three years

D. Whenever the planned features are complete

ANSWER KEY · THE STANDARDS — WHAT EACH C++ ADDED, AND WHAT RUNS HERE

1. B All five arrived together in C++11, the release that reset the language. C++14 generalised them (generic lambdas, relaxed `constexpr`); C++17 added the vocabulary types and structured bindings on top.

2. C `__cplusplus` is the standard's date: `201703L` for C++17, `202002L` for C++20, `202302L` for C++23. The judge compiles with `-std=c++20`, so it prints `202002`.

3. C `std::format` is C++20 and libstdc++ 14 implements it. `std::print`, `std::expected` and `std::ranges::to` are C++23 and do not exist under `-std=c++20`.

4. B Library feature-test macros are defined by the library headers, and `<version>` exists to publish all of them. The value is a date (`202110` here), so `#if __cpp_lib_format >= 202106L` can ask for a minimum revision. Without including `<version>` or `<format>` the macro is never defined.

5. B Owning raw memory through `new[]` / `delete[]` is exactly what the modern style replaces with containers and smart pointers: a `std::vector<int>` cannot leak and carries its own size. The other three are the idioms the track has taught.

6. A `std::unique_ptr` and `std::make_shared` are C++11; `std::make_unique` was left out and added in C++14. `std::optional` is C++17.

7. C The committee moved to a fixed three-year train: 2011, 2014, 2017, 2020, 2023, with 2026 in progress. A feature that misses the train waits for the next one rather than delaying the release.

LESSON 2 · 15 MIN

auto, decltype and structured bindings

Type deduction is where modern C++ stops making you repeat yourself. `auto` lets the compiler write the type it already knows, `decltype` names the type of an expression you cannot spell, and structured bindings take a pair, a tuple, a struct or a map entry apart into named parts in one line. Each has rules that are simple to state and easy to get wrong: `auto` drops references and top-level `const`, `decltype` cares whether you wrote parentheses, and a binding is a copy unless you ask for a reference. This lesson settles those rules, then covers the C++17 `if` / `switch` initialiser, `using` aliases, `nullptr` and the brace-initialisation quirks.

---

auto: deduction, exactly

`auto` deduces a type from the initialiser using the same rules as a template parameter. Two rules do almost all the work: **references are dropped**, and **top-level `const` is dropped**.

```
const std::string name = "ada";
auto a = name;          // std::string — a copy, not const
auto& b = name;         // const std::string& — a reference keeps its const
const auto& c = name;   // const std::string& — the safe read-only default
auto* p = &name;        // const std::string*
auto&& d = name;        // const std::string& (a forwarding reference binds to
```

The consequence that bites is the silent copy. `auto row = table[i];` copies the row; `auto& row = table[i];` aliases it. In a range-`for`, `for (auto x : v)` copies every element and `for (auto& x : v)` does not (Module 3, lesson 4). The rule of thumb: `const auto&` to read, `auto&` to modify in place, plain `auto` when you want your own copy or the value is small.

`auto` earns its keep on return values: `auto it = m.find(k);` instead of `std::map<std::string, std::vector<int>>::const_iterator it`. A lambda's type has no name at all, so `auto f = [](int x) { return x * 2; };` is the only way to store one without `std::function`. The other side: `auto n = v.size();` is `std::size_t`, not `int`, and `auto x = 1 / 2;` is `int` — deduction follows the expression, not your intention.

Brace initialisers have one special case. `auto x{1};` is `int` (since C++17, a single element in braces deduces the element type), but `auto y = {1};` and `auto z = {1, 2};` are `std::initializer_list<int>`, and `auto w{1, 2};` does not compile. Use `=` or parentheses with `auto` unless you specifically want the list.

decltype and decltype(auto)

`decltype(expr)` yields the declared type of an expression without evaluating it — the tool for "the type of *that*", where `auto` would strip something you need.

```
int i = 0;
decltype(i) a = 1;         // int — the declared type of the name
decltype((i)) b = i;       // int& — a parenthesised name is an lvalue expressi
std::vector<int> v;
decltype(v)::value_type e = 3;    // int
```

The parentheses rule is the trap: `decltype(name)` is the type the name was declared with; `decltype((name))` is `T&` for an lvalue. Templates use `decltype` to name return types that depend on parameters — `auto add(const A& a, const B& b) → decltype(a + b)` — and to inspect expressions in a `requires` clause (Module 12).

`decltype(auto)` deduces like `decltype` of the initialiser: it keeps references and `const`. Its use is returning exactly what an inner call returns:

```
decltype(auto) first(std::vector<int>& v) { return v[0]; }    // int& — a plai
```

Read `decltype(auto)` as "forward the type"; write it only when a reference must survive the deduction.

Structured bindings

A structured binding declares several names from one object in a single declaration:

```cpp
std::pair<std::string, int> p{"ada", 31};
auto [name, age] = p;                    // copies p, then names its two membe

std::tuple<int, int, int> rgb{255, 128, 0};
auto [r, g, b] = rgb;

struct Point { int x; int y; };
Point pt{3, 4};
auto [x, y] = pt;                        // any struct whose non-static data m

int arr[3] = {1, 2, 3};
auto [a0, a1, a2] = arr;                 // built-in arrays too (not pointers)

std::map<std::string, int> ages{{"ada", 31}, {"bob", 25}};
for (const auto& [who, years] : ages) std::cout << who << ' ' << years << '\n
```

What is happening underneath matters. `auto [name, age] = p;` creates a hidden copy of `p`, and `name` and `age` are names for that copy's members — not new variables, which is why you cannot give them their own types. `auto& [name, age] = p;` binds the hidden object by reference, so writing through `age` modifies `p`; `const auto& [k, v]` is the read-only form and the right one for iterating a map, where the key is `const` inside the pair anyway. The number of names must equal the number of members exactly; there is no `_` to skip one until C++26.

Where bindings shine is the functions that return several values. A `std::map::insert` returns a `std::pair<iterator, bool>`:

```cpp
if (auto [where, inserted] = ages.insert({"cy", 40}); !inserted)
    std::cout << "already had " << where->first << '\n';
```

And a function that returns a struct reads best of all: `auto [lo, hi, sum] = summarise(values);` says what the three results are, where `std::get<2>(result)` says nothing.

---

if and switch with an initialiser

C++17 lets `if` and `switch` declare a variable whose scope is the statement:

```cpp
if (auto it = ages.find("bob"); it ≠ ages.end()) {
    std::cout << it→second << '\n';
} else {
    std::cout << "unknown\n";
}
// it does not exist here

switch (const auto cmd = readCommand(); cmd) {
    case Command::Quit: return 0;
    default: break;
}
```

The variable lives in both branches and nowhere after. It fixes two things: a lookup no longer happens twice (`if (m.count(k)) use(m[k])` searches twice; this searches once), and a name that exists only to be tested no longer leaks into the rest of the function. A `std::lock_guard` in the initialiser holds a mutex for exactly the branch (Module 17).

---

using aliases and alias templates

`using Name = Type;` replaces `typedef` and reads left to right:

```cpp
using Scores = std::map<std::string, std::vector<int>>;
using Callback = void (*)(int, int);           // a function pointer, readable

template <typename T>
using Grid = std::vector<std::vector<T>>;       // alias templates: typedef co
Grid<double> heights(rows, std::vector<double>(cols));
```

A container type that appears three times deserves an alias; one that appears once does not. Aliases do not create a new type — `Scores` *is* `std::map<…>` — so they change nothing at run time and cannot be overloaded on.

---

nullptr

`nullptr` is a keyword of type `std::nullptr_t` that converts to any pointer type and to nothing else. `NULL` and `0` are integers that happen to convert to pointers, which is why `f(NULL)` with overloads `f(int)` and `f(const char*)` picks `f(int)` (or is ambiguous,

depending on how the platform defines `NULL`) while `f(nullptr)` picks the pointer overload. Write `nullptr`; treat `NULL` in new code as a review comment.

Uniform initialisation and the initializer_list quirk

Braces initialise anything — aggregates, classes with constructors, built-in values — and reject narrowing conversions (`int x{2.5};` is an error; `int x = 2.5;` silently truncates). `T{}` value-initialises, so `int n{};` is `0` and never an uninitialised read. The quirk: when a class has a constructor taking `std::initializer_list`, braces prefer it over every other constructor.

```
std::vector<int> a(3, 5);      // three fives:      5 5 5
std::vector<int> b{3, 5};      // two elements:     3 5
std::vector<int> c{3};         // one element, 3 — not three elements
std::string s(3, 'x');         // "xxx"; std::string t{3, 'x'} is two chars: '
```

The rule: braces mean "these are the elements" for containers; parentheses mean "these are the constructor arguments". And `T t();` declares a function, not a default-constructed `t` — the "most vexing parse", which `T t{};` avoids.

Pitfalls

Mistake	What happens
`auto row = grid[i]; row[0] = 1;`	Modifies a copy; the grid is unchanged. Use `auto&`.
`auto n = v.size(); for (int i = 0; i < n; …)`	Signed/unsigned comparison warning; `n` is `std::size_t`
`decltype((x))` when `decltype(x)` was meant	A reference type; the variable must be initialised
`auto [a, b] = triple;`	Compile error: the count must match
`for (auto [k, v] : m)` on a large map	Copies every entry; use `const auto&`
`std::vector<int> v{10};` to get ten elements	One element, 10. Use parentheses
`Widget w();`	Declares a function; `Widget w{};` or `Widget w;`

Key takeaways

- `auto` copies: it drops references and top-level `const`; `auto&`, `const auto&` and `auto*` keep what you ask for.
- `decltype(name)` is the declared type; `decltype((name))` is a reference; `decltype(auto)` forwards a return type exactly.
- `auto [a, b] = obj;` names the members of a hidden copy; `auto& [a, b]` aliases the original; the count must match.
- `if (init; cond)` and `switch (init; expr)` scope a lookup to the statement and search once.
- Braces reject narrowing and prefer `std::initializer_list`: `vector<int>{3, 5}` is two elements, `(3, 5)` is three.
- `using` aliases and alias templates name types without creating new ones; `nullptr` is the only null.

EXERCISE 1

Stock lookup with bindings

Read an integer `n`, then `n` lines of `name quantity price` into a `std::map<std::string, std::pair<int, double>>` (a later line for the same name replaces the earlier one). Then read an integer `q` and `q` names to look up. For each name print `<name>: <quantity> x <price> = <total>` with the price and total to two decimals, or `<name>: not stocked`. Finally print `stocked:` followed by every stocked name in map order, each preceded by one space.

Requirements: look up with an `if` that declares the iterator in its initialiser; take the pair apart with a structured binding (`const auto& [quantity, price] = it->second;`); iterate the map at the end with `for (const auto& [name, entry] : stock)`.

Input: `n`, `n` stock lines, `q`, `q` names. **Output:** `q` lines, then the `stocked:` line.

```
3
apple 4 0.50
banana 12 0.25
cherry 100 0.10
3
banana
grape
apple
```

prints

```
banana: 12 x 0.25 = 3.00
grape: not stocked
apple: 4 x 0.50 = 2.00
stocked: apple banana cherry
```

INPUT 1

```
3
apple 4 0.50
banana 12 0.25
cherry 100 0.10
3
banana
grape
apple
```

OUTPUT 1

```
banana: 12 x 0.25 = 3.00
grape: not stocked
apple: 4 x 0.50 = 2.00
stocked: apple banana cherry
```

INPUT 2

```
1
pen 3 1.99
2
pen
pen
```

OUTPUT 2

```
pen: 3 x 1.99 = 5.97
pen: 3 x 1.99 = 5.97
stocked: pen
```

REFERENCE SOLUTION

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

int main() {
    int n;
    std::cin >> n;
    std::map<std::string, std::pair<int, double>> stock;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int quantity;
        double price;
        std::cin >> name >> quantity >> price;
        stock[name] = {quantity, price};
    }
    int q;
    std::cin >> q;
    std::cout << std::fixed << std::setprecision(2);
    for (int i = 0; i < q; ++i) {
        std::string name;
        std::cin >> name;
        if (auto it = stock.find(name); it != stock.end()) {
            const auto& [quantity, price] = it->second;
            std::cout << name << ": " << quantity << " x " << price << " = " << quar
        } else {
            std::cout << name << ": not stocked\n";
        }
    }
    std::cout << "stocked:";
    for (const auto& [name, entry] : stock) std::cout << ' ' << name;
    std::cout << '\n';
    return 0;
}
```

EXERCISE 2

Summarise and classify

Complete two functions and bind their results. `summarise` returns a `Summary` struct — the smallest value, the largest value and the total (a `long long`) of a non-empty vector. `classify` returns a `std::array<int, 3>` — how many values are negative, zero and positive, in that order. In `main`, take both results apart with structured bindings (`const auto [lo, hi, sum] =`

`summarise(values);` and the same for the array) and print two lines: `lo=<lo> hi=<hi> sum=<sum>` and `neg=<neg> zero=<zero> pos=<pos>`.

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

Example: input `5` / `-3 0 4 0 -1` →

```
lo=-3 hi=4 sum=0
neg=2 zero=2 pos=1
```

INPUT 1

```
5
-3 0 4 0 -1
```

OUTPUT 1

```
lo=-3 hi=4 sum=0
neg=2 zero=2 pos=1
```

INPUT 2

```
4
1 9 3 5
```

OUTPUT 2

```
lo=1 hi=9 sum=18
neg=0 zero=0 pos=4
```

REFERENCE SOLUTION

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

struct Summary {
    int lo;
    int hi;
    long long sum;
};

// Smallest, largest and total of a non-empty vector.
Summary summarise(const std::vector<int>& values) {
    Summary s{values.front(), values.front(), 0};
    for (const int v : values) {
        if (v < s.lo) s.lo = v;
        if (v > s.hi) s.hi = v;
        s.sum += v;
    }
    return s;
}

// How many values are negative, zero and positive, in that order.
std::array<int, 3> classify(const std::vector<int>& values) {
    std::array<int, 3> counts{};
    for (const int v : values) {
        if (v < 0) ++counts[0];
        else if (v == 0) ++counts[1];
        else ++counts[2];
    }
    return counts;
}

int main() {
    int n;
    std::cin >> n;
    std::vector<int> values(n);
    for (auto& v : values) std::cin >> v;
    const auto [lo, hi, sum] = summarise(values);
    const auto [neg, zero, pos] = classify(values);
    std::cout << "lo=" << lo << " hi=" << hi << " sum=" << sum << '\n';
    std::cout << "neg=" << neg << " zero=" << zero << " pos=" << pos << '\n';
    return 0;
}
```

QUIZ

1. What does this print?

```
const std::string s = "x";
auto a = s;
a += "y";
std::cout << s << a;
```

A. `xxy`

B. `xyxy`

C. Compile error: `a` is const

D. `xy`

2. What are the types of `x` and `y`?

```
auto x{1};
auto y = {1};
```

A. Both `int`

B. Both `std::initializer_list<int>`

C. `x` is `int`, `y` is `std::initializer_list<int>`

D. Compile error on the first line

3. Given `int i = 0;`, what are the types of `a` and `b`?

```
decltype(i) a = 1;
decltype((i)) b = i;
```

A. Both `int`

B. `a` is `int`, `b` is `int&`

C. `a` is `int&`, `b` is `int`

D. Both `int&`

4. What is `m["a"]` after this runs?

```
std::map<std::string, int> m{{"a", 1}};
for (auto& [k, v] : m) v *= 2;
```

A. `1` — the binding is a copy

B. `2`

C. Compile error: `k` is const

D. Undefined behaviour

5. Overloads `void f(int);` and `void f(const char*);` exist. Which overload does `f(nullptr);` call?

A. `f(int)`, because `nullptr` is zero

B. `f(const char*)`

C. Ambiguous — compile error

D. Neither; `nullptr` needs a cast

6. What does this print?

```
std::vector<int> a(3, 5);
std::vector<int> b{3, 5};
std::cout << a.size() << ' ' << b.size();
```

A. `3 3`

B. `2 2`

C. `3 2`

D. `2 3`

7. Does this compile?

```
std::tuple<int, int, int> t{1, 2, 3};
auto [a, b] = t;
```

A. Yes; `a` and `b` take the first two elements

B. Yes; `b` becomes a pair of the last two

C. No: the number of names must equal the number of elements

D. No: structured bindings do not work on tuples

ANSWER KEY · AUTO, DECLTYPE AND STRUCTURED BINDINGS

1. A `auto` drops top-level `const` and makes a copy, so `a` is a mutable `std::string` holding `"x"`; appending gives `"xy"` and `s` is untouched. `const auto& a = s;` would have kept both the const and the alias.

2. C Since C++17 direct brace initialisation of `auto` with one element deduces the element type, so `x` is `int`. Copy-list-initialisation (`= {…}`) still deduces `std::initializer_list`. `auto w{1, 2};` is the form that does not compile.

3. B `decltype(name)` gives the declared type of the name. Parenthesising it makes it an expression — an lvalue of type `int` — and `decltype` of an lvalue expression is a reference, `int&`. That is why `decltype((i)) b;` without an initialiser would not compile.

4. B `auto&` binds the hidden object by reference, so `v` aliases the map's value and the doubling lands in the map. `k` is a `const std::string` but nothing writes to it. With plain `auto [k, v]` each entry would be copied and the map left at `1`.

5. B `nullptr` has type `std::nullptr_t`, which converts to any pointer type and to no integer type, so only the pointer overload is viable. `f(NULL)` or `f(0)` is the version that picks `f(int)` (or is ambiguous), which is why `nullptr` exists.

6. C Parentheses call the (count, value) constructor: three fives. Braces prefer the `std::initializer_list` constructor: the two elements 3 and 5. Braces mean "these are the elements" whenever an initializer-list constructor exists.

7. C A structured binding must name every element — three here. There is no way to skip one (a `_` placeholder arrives only in C++26); write a third name. Tuples, pairs, arrays and all-public structs are all bindable.

LESSON 3 · 15 MIN

constexpr, consteval and compile-time tables

A C++ compiler is also an interpreter. Mark a function `constexpr` and the compiler may run it while compiling, turning a computation into a constant baked into the executable; mark it `consteval` and it *must*. Since C++20 the interpreted subset is most of the language — loops, locals, `std::array`, even `std::vector` and `std::string` inside the function — so lookup tables and unit conversions can be computed before the program starts and *checked* before it starts too, with `static_assert`. This lesson settles what `const` and `constexpr` each promise, what a `constexpr` function may contain, how to force compile-time evaluation, how to build a table in `std::array`, and the one rule about `consteval` that stops most first attempts from compiling.

---

const versus constexpr

`const` means *read-only after initialisation*. The value may come from run time:

```
int n;
std::cin >> n;
const int limit = n * 2;      // fine: const, but a run-time value
```

`constexpr` means *a constant expression*: the value is known at compile time and can be used wherever the language demands one — an array bound, a template argument, a `case` label, a `static_assert`.

```
constexpr int MAX_ITEMS = 64;
int buffer[MAX_ITEMS];                    // needs a constant: constexpr quali
std::array<int, MAX_ITEMS * 2> wide{};
```

`constexpr` on a variable implies `const`. The distinction to remember for interviews: every `constexpr` variable is `const`, but a `const` variable is only usable in a constant expression when its initialiser was one (`const int k = 5;` is, `const int k = n;` is not). Prefer `constexpr` for anything that is genuinely a constant — the compiler then enforces it.

---

constexpr functions: may run at either time

A `constexpr` function is one the compiler is *allowed* to evaluate at compile time. It is still an ordinary function:

```
constexpr long long square(long long x) { return x * x; }

constexpr auto BIG = square(1'000'000);   // evaluated at compile time — the r
int n;
std::cin >> n;
std::cout << square(n) << '\n';           // evaluated at run time, like any j
```

The same function serves both calls. Whether a call happens at compile time depends on the context: a `constexpr` variable's initialiser, a template argument, an array bound or a `static_assert` *require* it; a plain call with run-time arguments simply runs. A plain call with constant arguments (`std::cout << square(12)`) may be folded by the optimiser but is not guaranteed to be.

What may a `constexpr` function contain? C++11 allowed a single `return` . C++14 allowed loops, locals and assignment. C++20 allowed `try` blocks, virtual calls and — the important one — dynamic allocation, provided the memory is freed before the function returns. That is what lets `std::vector` , `std::string` and most of `<algorithm>` work inside a `constexpr` function:

```
constexpr int sumOfSquares(int n) {
    std::vector<int> squares;                 // lives only during the evaluat
    for (int i = 1; i <= n; ++i) squares.push_back(i * i);
    int total = 0;
    for (int s : squares) total += s;
    return total;                             // the vector is destroyed here,
}
static_assert(sumOfSquares(3) == 14);
```

What it may *not* do during constant evaluation: call a non- `constexpr` function (`std::stoi` , anything from `<iostream>`), read a non- `constexpr` global, use `reinterpret_cast` , declare a `static` local (allowed only from C++23), or invoke undefined behaviour. The last is a feature: signed overflow, an out-of-range `std::array` index or a null dereference inside a compile-time evaluation is a **compile error**, not a silent wrong answer. Constant evaluation is the one place C++ has a full UB checker.

Forcing evaluation and proving it

Three contexts force compile-time evaluation, and one of them doubles as a test:

```
constexpr auto TABLE = makeTable();          // 1. a constexpr variable
std::array<int, square(4)> grid{};           // 2. a template argument or array
static_assert(square(12) == 144);            // 3. a static_assert — and a compi
```

`static_assert(cond, "message")` fails the *build* when `cond` is false, costs nothing at run time, and belongs next to every compile-time table: `static_assert(TABLE[20] == 2432902008176640000ULL);` is the cheapest test you will ever write.

Two C++20 keywords remove the "may": `consteval` declares an *immediate function* that can only be evaluated at compile time — a run-time argument is a compile error — and `constinit` declares a static-storage variable that must be initialised at compile time but may still change afterwards, which rules out the static-initialisation-order problems of Module 4, lesson 6.

```
consteval int cube(int x) { return x * x * x; }
constexpr int EIGHT = cube(2);               // fine
int n = 2;
// int bad = cube(n);                        // error: call to consteval functio

constinit int requestCount = cube(0);       // initialised at compile time; ++re
```

Use `consteval` for things that make no sense at run time (a compile-time hash of a string literal, a table builder) and `constinit` for globals whose initialisation must not depend on other globals.

Building a table with std::array

The workhorse pattern: a `constexpr` function fills a `std::array` with a loop and returns it, a `constexpr` variable captures it, and a `static_assert` checks a few entries.

```cpp
#include <array>

constexpr int LIMIT = 100;

constexpr std::array<bool, LIMIT + 1> makeSieve() {
    std::array<bool, LIMIT + 1> isPrime{};          // all false
    for (int i = 2; i ⩽ LIMIT; ++i) isPrime[i] = true;
    for (int i = 2; i * i ⩽ LIMIT; ++i) {
        if (!isPrime[i]) continue;
        for (int j = i * i; j ⩽ LIMIT; j += i) isPrime[j] = false;
    }
    return isPrime;
}

constexpr auto SIEVE = makeSieve();
static_assert(SIEVE[97] && !SIEVE[91] && !SIEVE[1]);

bool isPrime(int k) { return k ⩾ 0 && k ⩽ LIMIT && SIEVE[k]; }   // a run-ti
```

The array is computed once, by the compiler, and stored in the read-only data of the executable; `isPrime` is a bounds check and a load. It is `std::array` rather than `std::vector` because a `constexpr` *variable* may not own heap memory — a `std::vector` may live *inside* a `constexpr` function but not *be* a `constexpr` global. Return the data in an array whose size is itself a constant.

The consteval rule that bites

The natural next step is to make the table builder call a `consteval` helper:

```cpp
consteval unsigned long long factorial(unsigned n) {
    unsigned long long f = 1;
    for (unsigned i = 2; i ⩽ n; ++i) f *= i;
    return f;
}

constexpr std::array<unsigned long long, 21> makeTable() {   // constexpr, not
    std::array<unsigned long long, 21> t{};
    for (unsigned i = 0; i ⩽ 20; ++i) t[i] = factorial(i);  // error!
    return t;
}
```

GCC 14 refuses: *"call to consteval function 'factorial(i)' is not a constant expression … 'i' is not const"*. A `constexpr` function might be called at run time, so its body must be valid at run time, and `factorial(i)` with a loop variable is not a constant expression *in that body* — even though this call would only ever run during constant evaluation. The rule: a `consteval` function may be called with non-constant arguments **only from another `consteval` function** (an "immediate function context"). Fix it by making `makeTable` `consteval` too, or by making `factorial` `constexpr`. `consteval` is contagious upward: use it for the whole chain or not at all.

---

if constexpr

`if constexpr (cond)` needs `cond` to be a constant expression and **discards** the branch not taken — inside a template, the discarded branch is not even instantiated, so it may contain code that would not compile for that type:

```
template <typename T>
std::string describe(const T& value) {
    if constexpr (std::is_integral_v<T>) return "integer " + std::to_string(va
    else if constexpr (std::is_floating_point_v<T>) return "real " + std::to_s
    else return "text " + std::string(value);   // std::string("42") would not
}
```

A plain `if` with a constant condition compiles both branches and picks one; `if constexpr` compiles one. It replaced most uses of tag dispatch and `enable_if` (Module 12, lesson 6). `std::is_constant_evaluated()` (C++20) tells a `constexpr` function which mode it is running in; C++23's `if consteval` is the same test as a statement.

Pitfalls

Mistake	What happens
`const int n = read(); int a[n];`	Not a constant expression; GCC accepts it as a VLA extension, standard C++ does not
`constexpr int x = std::stoi("5");`	`std::stoi` is not `constexpr`: compile error
Calling a `consteval` function with a run-time value	Compile error: not a constant expression
Indexing past the array during constant evaluation	Compile error (UB is rejected) — a run-time call would silently misbehave
`constexpr std::vector<int> V{1, 2};` at namespace scope	Compile error: the allocation would outlive the evaluation; return a `std::array`
A `static` local in a `constexpr` function	Compile error in C++20 (allowed from C++23)

Key takeaways

`const` is read-only; `constexpr` is a compile-time constant and implies `const`.

A `constexpr` function runs at compile time only when the context demands it (a `constexpr` variable, template argument, array bound, `static_assert`); otherwise it is an ordinary function.

C++20 `constexpr` bodies may use loops, `std::array`, `std::vector` and `std::string`, provided nothing allocated escapes; UB inside constant evaluation is a compile error.

`consteval` forces compile time and can take non-constant arguments only from another `consteval` function; `constinit` forces compile-time initialisation of a mutable global.

Build tables with a `constexpr` function returning `std::array`, capture with `constexpr auto`, and check with `static_assert`.

`if constexpr` discards the untaken branch, so a template can hold code that is valid for only some types.

EXERCISE 1

A compile-time prime sieve

Build a sieve of Eratosthenes for 0 to 100 **at compile time**: a `constexpr` function `makeSieve()` that fills and returns a `std::array<bool, 101>` with a loop, captured as `constexpr auto SIEVE = makeSieve();`, and a `static_assert` that checks at least `SIEVE[2]`, `SIEVE[97]`, `!SIEVE[91]` and `!SIEVE[1]`. Then read an integer `q` and `q` integers, and for each print `<k>: prime`, `<k>: not prime` (0 and 1 are not prime) or `<k>: out of range` when `k` is below 0 or above 100.

The table must be built by the compiler: if you compute it in `main`, or leave the `static_assert` out, the program may still pass the cases but it has not done the exercise.

Input: `q`, then `q` integers. **Output:** `q` lines.

Example: input `5` / `2 15 97 1 100` →

```
2: prime
15: not prime
97: prime
1: not prime
100: not prime
```

INPUT 1

```
5
2
15
97
1
100
```

OUTPUT 1

```
2: prime
15: not prime
97: prime
1: not prime
100: not prime
```

INPUT 2

```
3
0
101
-5
```

OUTPUT 2

```
0: not prime
101: out of range
-5: out of range
```

REFERENCE SOLUTION

```cpp
#include <array>
#include <iostream>

constexpr int LIMIT = 100;

// true at every prime index, false elsewhere — built by the compiler.
constexpr std::array<bool, LIMIT + 1> makeSieve() {
    std::array<bool, LIMIT + 1> isPrime{};
    for (int i = 2; i <= LIMIT; ++i) isPrime[i] = true;
    for (int i = 2; i * i <= LIMIT; ++i) {
        if (!isPrime[i]) continue;
        for (int j = i * i; j <= LIMIT; j += i) isPrime[j] = false;
    }
    return isPrime;
}

constexpr auto SIEVE = makeSieve();
static_assert(SIEVE[2] && SIEVE[97] && !SIEVE[91] && !SIEVE[1] && !SIEVE[0]);

constexpr int countPrimes() {
    int count = 0;
    for (bool p : SIEVE) count += p ? 1 : 0;
    return count;
}
static_assert(countPrimes() == 25);

int main() {
    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        int k;
        std::cin >> k;
        if (k < 0 || k > LIMIT) {
            std::cout << k << ": out of range\n";
        } else {
            std::cout << k << (SIEVE[k] ? ": prime\n" : ": not prime\n");
        }
    }
    return 0;
}
```

EXERCISE 2

A consteval factorial table

Write `consteval unsigned long long factorial(unsigned n)` — an immediate function that computes `n!` with a loop — and `consteval std::array<unsigned long long, 21> makeTable()`

that fills a table with `factorial(0)` to `factorial(20)` by calling it in a loop. Capture it as `constexpr auto FACTORIALS = makeTable();` and add a `static_assert` that `FACTORIALS[20] == 2432902008176640000ULL` and `FACTORIALS[0] == 1`.

Both functions must be `consteval`: the lesson explains why the table builder cannot be a plain `constexpr` function when it calls a `consteval` one with a loop variable.

Then read an integer `q` and `q` integers (they may be negative or very large — read them as `long long`). For each print `<n>! = <value>` from the table, `<n>! = too large` when `n` is above 20, or `<n>: invalid` when `n` is negative.

Input: `q`, then `q` integers. **Output:** `q` lines.

Example: input `4` / `0 5 20 21` →

```
0! = 1
5! = 120
20! = 2432902008176640000
21! = too large
```

INPUT 1

```
4
0
5
20
21
```

OUTPUT 1

```
0! = 1
5! = 120
20! = 2432902008176640000
21! = too large
```

INPUT 2

```
2
-1
10
```

OUTPUT 2

```
-1: invalid
10! = 3628800
```

REFERENCE SOLUTION

```cpp
#include <array>
#include <iostream>

// n! for 0 ⩽ n ⩽ 20 — an immediate function: it can only run at compile time.
consteval unsigned long long factorial(unsigned n) {
    unsigned long long result = 1;
    for (unsigned i = 2; i ⩽ n; ++i) result *= i;
    return result;
}

// All twenty-one values, built at compile time. consteval, not constexpr: a constex
// function might run at run time, so it may not call factorial(i) with a loop varia
consteval std::array<unsigned long long, 21> makeTable() {
    std::array<unsigned long long, 21> table{};
    for (unsigned i = 0; i ⩽ 20; ++i) table[i] = factorial(i);
    return table;
}

constexpr auto FACTORIALS = makeTable();
static_assert(FACTORIALS[0] == 1 && FACTORIALS[1] == 1 && FACTORIALS[5] == 120);
static_assert(FACTORIALS[20] == 2432902008176640000ULL);

int main() {
    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        long long n;
        std::cin >> n;
        if (n < 0) {
            std::cout << n << ": invalid\n";
        } else if (n > 20) {
            std::cout << n << "! = too large\n";
        } else {
            std::cout << n << "! = " << FACTORIALS[static_cast<std::size_t>(n)] << '
        }
    }
    return 0;
}
```

QUIZ

1. Which declaration is a compile error?

A. `const int n = std::stoi("5");`

B. `constexpr int n = 5 * 2;`

C. `constexpr int n = std::stoi("5");`

D. `constexpr auto n = std::array{1, 2}.size();`

2. Input is `7`. What does this print?

```
constexpr int sq(int x) { return x * x; }
int main() {
    int n;
    std::cin >> n;
    std::cout << sq(n);
}
```

A. `49`

B. Compile error: `n` is not a constant expression

C. Undefined behaviour

D. `0`

3. Does this compile?

```
consteval int f(int x) { return x + 1; }
int main() {
    int n = 3;
    return f(n);
}
```

A. Yes, and returns 4

B. Yes, but `f` runs at run time

C. No: a `consteval` function needs a constant argument

D. No: `consteval` functions cannot return `int`

4. What does `constinit` guarantee?

A. The variable is `const`

B. A static-storage variable is initialised at compile time; it may still be modified afterwards

C. The variable is thread-local

D. The initialiser is evaluated lazily on first use

5. Which of these is *not* allowed inside a `constexpr` function under C++20?

A. A `for` loop

B. A local `std::vector` that is destroyed before the function returns

C. A `try` block

D. A `static` local variable

6. `makeSieve()` accidentally writes to index 101 of a `std::array<bool, 101>`. What happens with `constexpr auto SIEVE = makeSieve();`?

A. The program compiles and behaves unpredictably at run time

B. A compile error: undefined behaviour is not a constant expression

C. The write is silently ignored

D. A run-time exception

7. Inside a template, what is the difference between `if (std::is_integral_v<T>)` and `if constexpr (std::is_integral_v<T>)`?

A. None — the optimiser removes the dead branch either way

B. `if constexpr` discards the untaken branch, so it may contain code that would not compile for `T`

C. `if constexpr` is evaluated at run time, the plain `if` at compile time

D. `if constexpr` can only appear in `constexpr` functions

1. C `constexpr` demands a constant expression and `std::stoi` is not a `constexpr` function. `const` accepts a run-time initialiser, and `std::array::size` is `constexpr`, so the other three are fine.

2. A A `constexpr` function is still an ordinary function; it is evaluated at compile time only where the context requires a constant. With a run-time argument it simply runs at run time. A `consteval` function would have been the compile error.

3. C `consteval` makes `f` an immediate function — every call must be a constant expression. `n` is a mutable local, so `f(n)` is rejected. `f(3)` or `constexpr int n = 3;` would compile.

4. B `constinit` forces constant initialisation of a global or `static` — ruling out the static-initialisation-order problem — without making the variable `const`. `constexpr` on a variable implies `const`; `constinit` does not.

5. D C++14 allowed loops, C++20 allowed `try` blocks and transient allocation (so a local `std::vector` is fine as long as nothing allocated escapes). A `static` local inside a `constexpr` function is rejected until C++23.

6. B During constant evaluation the compiler must detect undefined behaviour and refuse — an out-of-bounds write, signed overflow or a null dereference all become compile errors. The same code called at run time would be silent UB, which is why compile-time tables are safer than run-time ones.

7. B A plain `if` compiles both branches for every `T`, so `std::string(value)` in the non-integral branch is an error when `T` is `int`. `if constexpr` does not instantiate the discarded branch inside a template, which is what makes per-type code possible without overloads or specialisation.

LESSON 4 · 15 MIN

The vocabulary types — pair, tuple, span, chrono and friends

A vocabulary type is one that appears in function signatures across a whole codebase because everyone agrees what it means: "two things" is a `std::pair`, "maybe a value" is a `std::optional`, "read-only text" is a `std::string_view`, "a length of time" is a `std::chrono::duration`. Using them instead of home-grown equivalents means two libraries can hand values to each other without conversion, and a reader knows the semantics before reading the body. This lesson covers the ones that have not had a module of their own — `std::pair` and `std::tuple` with `std::tie` and `std::apply`, `std::span`, `<chrono>` and `std::byte` — and places `std::optional`, `std::variant` and `std::string_view` beside them by use, so the whole set reads as one toolkit.

---

pair and tuple

`std::pair<A, B>` holds two members named `first` and `second`; `std::tuple<Ts...>` holds any number, accessed by index or by type. Both compare lexicographically, which is why a pair sorts by `first` then `second` without a comparator.

```
#include <tuple>
#include <utility>

std::pair<std::string, int> p{"ada", 31};       // or: std::pair p{"ada"s, 31}
std::tuple<std::string, int, double> row{"bob", 25, 88.5};

std::cout << p.first << ' ' << std::get<1>(row) << ' ' << std::get<double>(row
auto [name, age, score] = row;                  // structured bindings (lesson
```

Two helpers make tuples do work. `std::tie(a, b, c)` builds a tuple of *references* to existing variables, which gives an assignment target and a comparison key in one idiom:

```cpp
int lo, hi;
std::tie(lo, hi) = std::minmax(3, 9);           // assign both results at once

struct Version { int major, minor, patch; };
bool operator<(const Version& a, const Version& b) {
    return std::tie(a.major, a.minor, a.patch) < std::tie(b.major, b.minor, b.
}
```

`std::apply(f, t)` calls `f` with the tuple's elements as separate arguments — the bridge from "a tuple of values" to "a function of N parameters":

```cpp
auto print = [](const std::string& n, int a, double s) { std::cout << n << " (
std::apply(print, row);                         // print("bob", 25, 88.5)
```

When should a function return a tuple? Rarely. `std::pair` is fine for a natural two-ness (`insert`'s iterator-and-bool, a `minmax`), and a tuple is fine at the boundary of generic code, but three values with meaning deserve a struct: `Summary{lo, hi, sum}` bound as `auto [lo, hi, sum]` reads the same as a tuple and `result.sum` reads far better than `std::get<2>(result)`.

---

optional, variant and string_view by use

Module 15 taught `std::optional` and `std::variant` in depth and Module 5 taught `std::string_view`; here is the placement rule for each. `std::optional<T>` is "a `T` or nothing" — a lookup that can miss, a parse that can fail, a field that may be unset; test it with `if (opt)` or `has_value()`, read with `*opt` or `value_or(default)`. `std::variant<A, B, C>` is "exactly one of these" — a token that is a number or a word, a message that is one of several commands; take it apart with `std::visit` and an overloaded lambda set. `std::string_view` is "read-only text I do not own" — the parameter type for any function that only reads, at the cost of the lifetime rule: never keep a view past the string it looks into. The three share a design: no heap allocation, value semantics, and a type that states what the value *can* be.

---

std::span

`std::span<T>` (C++20) is `std::string_view` for anything contiguous: a pointer and a length, viewing elements owned by someone else. It replaces the `(T* data, std::size_t n)`

parameter pair and the "which container?" question at once — a `std::span<const int>` accepts a `std::vector<int>`, a `std::array<int, N>`, a built-in array or a pointer-plus-size, and the function does not care which.

```
#include <span>

long long total(std::span<const int> xs) {          // const: we only read
    long long sum = 0;
    for (int x : xs) sum += x;
    return sum;
}

std::vector<int> v{4, -2, 7, 1, 9};
int raw[] = {1, 2, 3};
total(v);  total(raw);  total({v.data() + 1, 3});    // all fine

std::span<const int> all{v};
auto middle = all.subspan(1, 3);                    // elements 1, 2, 3 — no c
auto head = all.first(2);  auto tail = all.last(2);
std::cout << middle.size() << ' ' << middle[0] << ' ' << middle.back() << '\n
```

A span is cheap to copy (two words) and should be passed by value. Its element type says whether the callee may write: `std::span<int>` may modify the caller's elements in place; `std::span<const int>` may not. `subspan`, `first` and `last` are O(1) and produce more spans. Two things a span does *not* do: it does not own (the container must outlive every span into it, and a `push_back` that reallocates leaves the span dangling — Module 13's invalidation rules apply), and it does not bounds-check (`s[i]` past `size()` is UB, exactly like a raw array). A span with a compile-time size, `std::span<int, 4>`, exists for fixed-shape data and is rarely needed.

---

chrono: durations and time points

`<chrono>` separates three ideas that untyped `int milliseconds` code muddles: a **duration** is a count of ticks in a unit, a **time point** is a duration since a clock's epoch, and a **clock** says what "now" means. Durations are the part every program uses:

```cpp
#include <chrono>
using namespace std::chrono_literals;

auto d = 90s + 5min + 2h;                          // std::chrono::seconds — t
std::cout << d.count() << '\n';                    // 7590 (ticks in d's unit)
auto m = std::chrono::duration_cast<std::chrono::minutes>(d);   // 126 — trunc
std::chrono::milliseconds ms = 3s;                 // widening is implicit: 30
// std::chrono::seconds s = 1500ms;                // error: lossy — spell it

auto hours = std::chrono::duration_cast<std::chrono::hours>(d);
auto rest = d - hours;                             // 390s: durations subtract
std::cout << hours.count() << "h " << (rest / 60).count() << "m " << (rest % 6
```

`.count()` is how a duration is printed deterministically — it is the integer you would have kept by hand, now with its unit checked by the type system. (`std::format("{}", d)` prints `7590s` with the unit and `std::chrono::hh_mm_ss{d}` splits it into fields; both exist on libstdc++ 14, and `.count()` is the portable core.) The compile error on the narrowing assignment is the point: a function taking `std::chrono::milliseconds` can be handed `2s` or `500ms` and never a bare `2` that someone meant as seconds.

Time points come from clocks. `std::chrono::steady_clock::now()` is for measuring elapsed time (it never jumps backward); `std::chrono::system_clock::now()` is wall-clock time, convertible to a calendar date. Subtracting two time points gives a duration:

```cpp
const auto start = std::chrono::steady_clock::now();
work();
const auto elapsed = std::chrono::steady_clock::now() - start;
std::cerr << std::chrono::duration_cast<std::chrono::milliseconds>(elapsed).co
```

Note `std::cerr`: a judged program must never print a measured time or the current date to standard output, because no expected output can match it (Module 19 covers measurement properly). Durations are pure arithmetic and print the same every run; clocks are not.

---

std::byte

`std::byte` (C++17, `<cstddef>`) is a byte that is *not a number*: an `enum class` over `unsigned char` with only the bitwise operators. `std::byte b{0x2A}; b ⊨ std::byte{0x01};` works; `b + 1` and `std::cout << b` do not, and `std::to_integer<int>(b)` is the explicit way out. Use it for buffers of raw memory — file contents, network packets, serialisation — where `char`

would invite arithmetic and string operations that have no meaning on the data. When the bytes *are* text, keep `char`.

Pitfalls

Mistake	What happens
Returning `std::tuple<int, int, long long>` for min, max, sum	Compiles; every caller must remember which index is which. Return a struct.
`std::tie(a, b) = f();` when `f` returns a struct	Compile error; `std::tie` needs a tuple-like. Use `auto [a, b] = f();`
Holding a `std::span` across a `push_back`	Dangling after reallocation: UB
`std::span<int>` for a read-only parameter	Compiles, but claims write access; write `std::span<const int>`
`std::chrono::seconds s = 1500ms;`	Compile error: lossy; `duration_cast` truncates it to `1s` explicitly
Printing `steady_clock::now()` or an elapsed time to stdout in a judged program	Different every run; the case can never pass
`std::byte b = 5;`	Compile error: no implicit conversion; `std::byte{5}`

Key takeaways

`std::pair` / `std::tuple` hold values positionally and compare lexicographically; `std::tie` makes reference tuples for assignment and comparison keys, `std::apply` unpacks a tuple into a call.

Three named results are a struct with structured bindings, not a tuple.

`std::span<const T>` is the parameter type for "any contiguous sequence"; it is a view — cheap to copy, never owning, never bounds-checked.

`<chrono>` durations carry their unit: widening converts implicitly, narrowing needs `duration_cast`, and `.count()` prints deterministically.

`steady_clock` measures, `system_clock` tells the date; neither belongs in judged output.

`std::byte` is raw memory with bitwise operators only; `std::to_integer` converts out.

EXERCISE 1

Leaderboard with tuples

Read an integer `n` (at least 1), then `n` lines of `name age score` (`score` is a decimal number) into a `std::vector<std::tuple<std::string, int, double>>`. Sort by score **descending**, then by name ascending — write the comparator with `std::tie` so the two-key comparison is one expression. Print each row numbered from 1 as `<rank>. <name> (<age>) <score>` with the score to one decimal, using `std::apply` to hand the tuple's elements to a lambda that prints them. Finally print `best: <name of the first row>`.

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

```
3
ada 31 92.5
bob 25 88.0
cy 40 92.5
```

prints

```
1. ada (31) 92.5
2. cy (40) 92.5
3. bob (25) 88.0
best: ada
```

INPUT 1

```
3
ada 31 92.5
bob 25 88.0
cy 40 92.5
```

OUTPUT 1

```
1. ada (31) 92.5
2. cy (40) 92.5
3. bob (25) 88.0
best: ada
```

INPUT 2

```
2
zed 20 10
abe 22 20
```

OUTPUT 2

```
1. abe (22) 20.0
2. zed (20) 10.0
best: abe
```

REFERENCE SOLUTION

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

using Row = std::tuple<std::string, int, double>;

int main() {
    int n;
    std::cin >> n;
    std::vector<Row> rows;
    for (int i = 0; i < n; ++i) {
        std::string name;
        int age;
        double score;
        std::cin >> name >> age >> score;
        rows.emplace_back(name, age, score);
    }
    std::ranges::sort(rows, [](const Row& a, const Row& b) {
        // Score descending, name ascending: compare (b.score, a.name) against (a.sc
        return std::tie(std::get<2>(b), std::get<0>(a)) < std::tie(std::get<2>(a), s
    });
    int rank = 0;
    for (const Row& row : rows) {
        ++rank;
        std::apply([rank](const std::string& name, int age, double score) {
            std::cout << std::format("{}. {} ({}) {:.1f}\n", rank, name, age, score)
        }, row);
    }
    std::cout << "best: " << std::get<0>(rows.front()) << '\n';
    return 0;
}
```

EXERCISE 2

Window sums over a span

Write `long long total(std::span<const int> window)` and `int largest(std::span<const int> window)`. Read an integer `n` and `n` integers into a `std::vector<int>`, then an integer `q` and `q` queries of `offset count`. For each query take the window with `subspan(offset, count)` on a `std::span<const int>` over the vector — no copying — and print `sum=<total> max=<largest>`, or `invalid` when the window does not fit (`offset` below 0, `count` below 1, or `offset + count` beyond `n`).

Both functions must take `std::span<const int>`, so they work unchanged for a vector, a `std::array` or a built-in array.

Input: `n`, `n` integers, `q`, then `q` lines of `offset count`. **Output:** `q` lines.

```
5
4 -2 7 1 9
3
0 5
1 2
2 3
```

prints

```
sum=19 max=9
sum=5 max=7
sum=17 max=9
```

INPUT 1

```
5
4 -2 7 1 9
3
0 5
1 2
2 3
```

OUTPUT 1

```
sum=19 max=9
sum=5 max=7
sum=17 max=9
```

INPUT 2

```
3
1 2 3
2
2 1
3 1
```

OUTPUT 2

```
sum=3 max=3
invalid
```

REFERENCE SOLUTION

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

long long total(std::span<const int> window) {
    long long sum = 0;
    for (const int x : window) sum += x;
    return sum;
}

int largest(std::span<const int> window) {
    return *std::ranges::max_element(window);
}

int main() {
    int n;
    std::cin >> n;
    std::vector<int> values(n);
    for (auto& v : values) std::cin >> v;
    const std::span<const int> all{values};
    int q;
    std::cin >> q;
    for (int i = 0; i < q; ++i) {
        long long offset, count;
        std::cin >> offset >> count;
        if (offset < 0 || count < 1 || offset + count > static_cast<long long>(all.s
            std::cout << "invalid\n";
            continue;
        }
        const auto window = all.subspan(static_cast<std::size_t>(offset), static_cas
        std::cout << "sum=" << total(window) << " max=" << largest(window) << '\n';
    }
    return 0;
}
```

QUIZ

1. What does this print?

```
std::tuple<int, std::string> t{1, "a"};
auto [n, s] = t;
s += "b";
std::cout << std::get<1>(t);
```

A. `a`

B. `ab`

C. Compile error

D. Undefined behaviour

2. What does this print?

```
int a = 1, b = 2;
std::tie(a, b) = std::make_pair(b, a);
std::cout << a << ' ' << b;
```

A. `2 1`

B. `1 2`

C. `2 2`

D. Undefined behaviour

3. What is the state of `s` after this?

```
std::vector<int> v{1, 2, 3};
std::span<const int> s{v};
v.push_back(4);
```

A. `s` still views the three original elements

B. `s` now views all four elements

C. `s` may be dangling: reading through it is undefined behaviour if the vector reallocated

D. Compile error: a span cannot view a vector

4. Does this compile?

```
using namespace std::chrono_literals;
std::chrono::seconds s = 1500ms;
```

A. Yes; `s` is 1 second

B. Yes; `s` is 2 seconds

C. No: converting milliseconds to seconds loses precision and needs `duration_cast`

D. Yes; `s` is 1500 seconds

5. What does `std::apply(f, t)` do for a tuple `t` with three elements?

A. Calls `f(t)` with the tuple as one argument

B. Calls `f` once per element

C. Calls `f(std::get<0>(t), std::get<1>(t), std::get<2>(t))`

D. Returns a new tuple with `f` applied to each element

6. Which parameter type says "read-only access to any contiguous sequence of `int`" and accepts a `std::vector<int>`, a `std::array<int, 4>` and a built-in array alike?

A. `const std::vector<int>&`

B. `const int*`

C. `std::span<const int>`

D. `std::initializer_list<int>`

7. Given `std::byte b{5};`, which expression compiles?

A. `b + 1`

B. `std::cout << b`

C. `std::to_integer<int>(b) + 1`

D. `int n = b;`

ANSWER KEY · THE VOCABULARY TYPES — PAIR, TUPLE, SPAN, CHRONO AND FRIENDS

1. A `auto [n, s] = t;` copies the tuple and binds names to the copy's elements, so appending to `s` leaves `t` unchanged. `auto& [n, s] = t;` would have printed `ab`.

2. A `std::make_pair(b, a)` copies the current values into a temporary pair `(2, 1)` first; assigning it to the tuple of references then stores 2 in `a` and 1 in `b`. The idiom swaps without a temporary variable.

3. C A span is a pointer and a length into memory it does not own. `push_back` may reallocate, moving the elements and freeing the old block the span still points at. The same invalidation rules as iterators apply (Module 13).

4. C Durations convert implicitly only when no precision is lost (seconds to milliseconds). The lossy direction must be spelled out: `std::chrono::duration_cast<std::chrono::seconds>(1500ms)` gives `1s` by truncation.

5. C `std::apply` unpacks the tuple into separate arguments — the bridge from "a tuple of values" to "a function of N parameters". Per-element application would need a fold or a loop over a homogeneous container.

6. C `std::span<const int>` is a non-owning view over any contiguous storage and carries its length; `const int*` loses the length, `const std::vector<int>&` accepts only vectors, and `std::initializer_list` is for literal lists.

7. C `std::byte` deliberately has no arithmetic, no stream output and no implicit conversion — only bitwise operators. `std::to_integer<int>(b)` is the explicit way out, and `b |= std::byte{1}` the way to manipulate it.

LESSON 5 · 16 MIN

C++20 and beyond — concepts, ranges, format and what is coming

C++20 is the standard this track compiles against, and it is the largest since C++11. Some of its features have already had a lesson of their own — concepts in Module 12, ranges in Module 14, `⟺` in Module 11 — so this lesson does not re-teach them; it puts them side by side as the *shape* of C++20 code, adds the smaller pieces that make daily programs shorter (`std::format`, designated initialisers, `std::jthread`, `[[likely]]`), sketches modules and coroutines in outline, and then looks past the fence at C++23, which does not compile here but which you will meet in code and in interviews. It closes with how to keep up without reading the standard.

Concepts: constraints you can read

A template used to accept anything and fail deep inside its body; a concept states up front what a type must provide, and a call that does not satisfy it fails at the call site, in one line.

```
#include <concepts>

template <typename T>
concept HasArea = requires(const T& t) {
    { t.area() } -> std::convertible_to<double>;
};

template <HasArea S>                                  // the constrained parameter
double doubled(const S& s) { return 2 * s.area(); }

void show(const HasArea auto& s) { std::cout << s.area() << '\n'; }   // const

template <std::integral T> T twice(T x) { return x * 2; }   // a standard conc
// twice(2.5);  → error: constraints not satisfied — 'double' does not satisfy
```

The three spellings — `template <Concept T>`, `requires Concept<T>` after the parameter list, and `Concept auto` in a parameter — are equivalent; use the shortest that fits. `<concepts>` supplies `std::integral`, `std::floating_point`, `std::same_as`, `std::convertible_to`, `std::invocable`, `std::equality_comparable`, `std::totally_ordered` and more. A constrained overload beats an unconstrained one, and a more specific constraint beats a

more general one (subsumption), which is how ranges algorithms pick the fast path for random-access iterators.

Ranges and views: algorithms without begin/end

```
#include <algorithm>
#include <ranges>

std::vector<int> v{5, 3, 9, 1, 4};
std::ranges::sort(v);                                          // no v.beg
auto odd = v | std::views::filter([](int x) { return x % 2; })
             | std::views::transform([](int x) { return x * 10; });
for (int x : odd) std::cout << x << ' ';                       // 10 30 50

struct Person { std::string name; int age; };
std::vector<Person> people = /* … */;
std::ranges::sort(people, {}, &Person::age);                   // {} = def
auto youngest = std::ranges::min_element(people, {}, &Person::age);
```

A **range** is anything with `begin()` and `end()`; the `std::ranges::` algorithms take one instead of a pair of iterators and check their arguments with concepts. A **view** is a lazy, non-owning range: `filter`, `transform`, `take`, `drop`, `reverse`, `iota`, `split`, `keys` / `values`, composed with `|`, nothing computed until iterated. A **projection** — the third argument above — tells an algorithm which part of each element to look at, which removes most one-line comparator lambdas. Materialising a view into a container is `std::vector<int> out(odd.begin(), odd.end())` in C++20; `std::ranges::to<std::vector>()` is C++23.

<=>: one operator, six comparisons

```
#include <compare>

struct Version {
    int major, minor, patch;
    auto operator<=>(const Version&) const = default;   // also generates ==
};
Version{1, 2, 0} < Version{1, 10, 0};                    // true: rewritten as
std::set<Version> releases;                              // works: std::less u
```

Defaulting `operator<=>` compares members in declaration order and gives all six comparisons; its return type — `std::strong_ordering`, `weak_ordering` or `partial_ordering` — states whether equal means substitutable, and `partial_ordering` admits unordered values (`NaN`). Hand-written `<=>` is for types whose ordering is not memberwise (Module 11, lesson 2).

---

std::format: the formatting mini-language

```
#include <format>

std::cout << std::format("{:<10}{:>6}{:>9}\n", "NAME", "PORT", "THREADS");
std::cout << std::format("{:<10}{:>6}{:>9}\n", "api", 9000, 8);
std::cout << std::format("{:.2f} | {:04d} | {:x} | {:#b} | {:>8.3f}|\n", 3.141
//  3.14 | 0042 | ff | 0b101 |    2.500|
std::cout << std::format("{0} then {1} then {0}\n", "a", "b");    // positional
std::string s = std::format("{}", 3.0);                           // "3" — no
```

A replacement field is `{index:spec}`; the spec is `[fill][align][sign][#][0][width][.precision][type]` — `<`, `>`, `^` align; `0` pads numbers; `.2f` fixes decimals; `d`, `x`, `b`, `o` choose the base; `#` adds the prefix. It is type-safe (a wrong type is a compile error, not a crash), locale-independent, and returns a `std::string`; `std::format_to` writes into an iterator instead. A user type joins in by specialising `std::formatter` (Module 11, lesson 3). `std::format` is the C++20 half of what became `std::print` in C++23.

The smaller pieces

Designated initialisers name the members of an aggregate, in declaration order, and leave the rest at their defaults:

```
struct Server { std::string name; int port = 8080; int threads = 4; bool tls =
Server s{.name = "api", .port = 9000};      // threads 4, tls false
// Server t{.port = 1, .name = "x"};        // error: out of declaration order
```

`std::jthread` is `std::thread` that joins in its destructor and carries a `std::stop_token`, so forgetting `join()` is no longer a call to `std::terminate` (Module 17). `[[likely]]` / `[[unlikely]]` on a branch hint the optimiser's code layout and change no semantics. `using enum Color;` inside a `switch` lets the cases read `case Red:`. `<bit>` gives `std::popcount`,

`std::bit_width`, `std::has_single_bit`, `std::rotl`; `<numbers>` gives `std::numbers::pi`, `e`, `sqrt2` as `constexpr` doubles. `std::span` and `<chrono>` calendars were lesson 4; `contains()`, `starts_with`, `std::erase_if`, `std::midpoint`, `std::ssize`, `std::to_array` and `std::source_location` round out the list.

Modules and coroutines, in outline

A **module** replaces headers: `export module geometry;` in one file, `import geometry;` in another, no macros leaking across the boundary and no re-parsing the same text into every translation unit. `import std;` (C++23) imports the whole standard library in one line. Compilers support modules; build systems and package managers are still catching up, which is why most code you read still uses `#include`, and why a single-file judge cannot demonstrate them.

A **coroutine** is a function that can suspend and resume — it contains `co_await`, `co_yield` or `co_return`. C++20 shipped the *machinery* (promise types, awaiters, `std::coroutine_handle`) but no ready-made generator or task type; `std::generator` arrived in C++23, and asynchronous frameworks bring their own. Know the vocabulary: a coroutine's frame lives on the heap, suspension is cheap, and the mechanism underlies async I/O libraries the way `std::thread` underlies threading.

C++23: reading only

None of this compiles under `-std=c++20`, and all of it will be in code you read:

Feature	What it is
`std::print("{} {}\n", a, b)` / `std::println`	`std::format` straight to stdout; Unicode-aware, faster than `<<`
`std::expected<T, E>`	A `T` or an error `E`, without exceptions; `has_value()`, `value()`, `error()`, monadic `and_then` / `or_else`
`std::mdspan`	A multidimensional view over contiguous memory — `md(i, j)` over one buffer
Deducing `this`	`void f(this Self&& self)` — one function replaces const/non-const overload pairs, and CRTP without the template
`std::ranges::to<C>()`	Materialise a view: `v | views::filter(f) | ranges::to<std::vector>()`
`std::ranges::fold_left`, `views::zip`, `views::enumerate`, `views::chunk`	The ranges pieces C++20 ran out of time for
`if consteval`, `static operator()`, `std::flat_map`, `std::generator`, `std::stacktrace`, `std::optional::and_then`	Smaller additions

C++26 is in progress: static reflection, contracts, `std::execution` (senders and receivers), `std::inplace_vector` and the `_` placeholder for structured bindings are the headlines.

Keeping up

You do not read the standard. Three sources are enough: cppreference's compiler-support page (a matrix of feature × compiler version — the answer to "can I use this at work?"), cppreference's per-feature pages (which carry the `__cpp_*` macro and the version each feature arrived in), and the feature-test macros from lesson 1, which let the code itself say what it needs. A good habit is to know your project's `-std=` and the oldest compiler it must build with, and to treat "it is in the standard" as necessary but not sufficient.

Pitfalls

Mistake	What happens
`template <std::integral T> … twice(2.5)`	Compile error at the call: constraints not satisfied
`auto r = v | std::views::filter(f); v.push_back(1);` then iterating `r`	The view holds an iterator into `v`; reallocation dangles it
`std::format("{:d}", 2.5)`	Compile error (a format-spec mismatch is checked at compile time for literal format strings)
Designated initialisers out of order or mixed with positional	Compile error
Storing a `std::jthread` in a container and expecting output order	Threads interleave; join, then print aggregated results (Module 17)
Writing `std::print` or `std::expected` here	Does not exist under `-std=c++20`
`std::ranges::sort(v, &Person::age)`	The second argument is the *comparator*; the projection is the third: `sort(v, {}, &Person::age)`

Key takeaways

Concepts constrain templates at the call site; ranges take containers, compose lazy views with `|` and accept projections; `⟺` defaulted gives six comparisons in one line.

`std::format` is type-safe formatting with `{index:spec}`; it became `std::print` in C++23.

Designated initialisers name aggregate members in order; `std::jthread` joins itself; `[[likely]]` is a hint, not a rule.

Modules replace headers and coroutines add suspension, but both need tooling a single-file judge lacks.

C++23 (`std::print`, `std::expected`, `std::mdspan`, deducing `this`, `ranges::to`) is reading only on this track.

cppreference's compiler-support matrix plus feature-test macros answer "can I use this?" for any compiler.

EXERCISE 1

Shapes that satisfy a concept

Define a concept `HasArea` that is satisfied by any type whose `area()` const member function returns something convertible to `double`, three structs that satisfy it — `Circle{r}` (area πr², use `std::numbers::pi`), `Rect{w, h}` and `Square{side}` — and a function template `report` constrained by the concept (`template <HasArea S>`) that prints `<label> area=<area>` with the area to two decimals through `std::format` and returns the area.

Read an integer `n`, then `n` lines: `circle <r>`, `rect <w> <h>`, `square <side>`, or something else, which prints `unknown shape`. After the lines print `total=<sum of the reported areas>` to two decimals.

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

```
3
circle 2
rect 3 4
square 2
```

prints

```
circle area=12.57
rect area=12.00
square area=4.00
total=28.57
```

INPUT 1

```
3
circle 2
rect 3 4
square 2
```

OUTPUT 1

```
circle area=12.57
rect area=12.00
square area=4.00
total=28.57
```

INPUT 2

```
2
triangle 3 4
circle 1
```

OUTPUT 2

```
unknown shape
circle area=3.14
total=3.14
```

REFERENCE SOLUTION

```cpp
#include <concepts>
#include <format>
#include <iostream>
#include <numbers>
#include <sstream>
#include <string>
#include <string_view>

template <typename T>
concept HasArea = requires(const T& t) {
    { t.area() } -> std::convertible_to<double>;
};

struct Circle {
    double r;
    double area() const { return std::numbers::pi * r * r; }
};

struct Rect {
    double w;
    double h;
    double area() const { return w * h; }
};

struct Square {
    double side;
    double area() const { return side * side; }
};

template <HasArea S>
double report(std::string_view label, const S& shape) {
    const double a = shape.area();
    std::cout << std::format("{} area={:.2f}\n", label, a);
    return a;
}

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore();
    double sum = 0.0;
    for (int i = 0; i < n; ++i) {
        std::string line;
        std::getline(std::cin, line);
        std::istringstream in(line);
        std::string kind;
        in >> kind;
        if (kind == "circle") {
            double r;
```

```
            in >> r;
            sum += report("circle", Circle{r});
        } else if (kind == "rect") {
            double w, h;
            in >> w >> h;
            sum += report("rect", Rect{w, h});
        } else if (kind == "square") {
            double side;
            in >> side;
            sum += report("square", Square{side});
        } else {
            std::cout << "unknown shape\n";
        }
    }
    std::cout << std::format("total={:.2f}\n", sum);
    return 0;
}
```

EXERCISE 2

A server table

A `Server` is an aggregate with defaults — `name`, `port` (default 8080), `threads` (default 4) and `tls` (default `false`) — and a defaulted `operator<=>` so servers sort by name, then port, then threads, then tls. Read an integer `n`, then `n` lines of `name port [threads] [tls|plain]`, where the last two tokens are optional. Build each server with a **designated initialiser** naming only `name` and `port` (`Server s{.name = name, .port = port};`), then overwrite `threads` and `tls` only when the line provides them. Sort with `std::ranges::sort` (no comparator — the spaceship supplies it) and print a table with `std::format`: the header `{:<10}{:>5}{:>8} {:>3}` over `NAME`, `PORT`, `THREADS`, `TLS`, then one row per server in the same widths, with `yes` or `no` in the last column.

Input: `n`, then `n` lines. **Output:** the header, then `n` rows.

```
3
web 80
api 9000 8 tls
db 5432 2
```

prints

```
NAME       PORT THREADS TLS
api        9000       8 yes
db         5432       2  no
web          80       4  no
```

INPUT 1

```
3
web 80
api 9000 8 tls
db 5432 2
```

OUTPUT 1

```
NAME       PORT THREADS TLS
api        9000       8 yes
db         5432       2  no
web          80       4  no
```

INPUT 2

```
2
x 1 1 tls
x 1 1 plain
```

OUTPUT 2

```
NAME       PORT THREADS TLS
x             1       1  no
x             1       1 yes
```

REFERENCE SOLUTION

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

struct Server {
    std::string name;
    int port = 8080;
    int threads = 4;
    bool tls = false;
    auto operator<=>(const Server&) const = default;
};

int main() {
    int n;
    std::cin >> n;
    std::cin.ignore();
    std::vector<Server> servers;
    for (int i = 0; i < n; ++i) {
        std::string line;
        std::getline(std::cin, line);
        std::istringstream in(line);
        std::string name;
        int port;
        in >> name >> port;
        Server server{.name = name, .port = port};
        if (int threads; in >> threads) server.threads = threads;
        if (std::string mode; in >> mode) server.tls = (mode == "tls");
        servers.push_back(server);
    }
    std::ranges::sort(servers);
    std::cout << std::format("{:<10}{:>5}{:>8} {:>3}\n", "NAME", "PORT", "THREADS",
    for (const auto& s : servers) {
        std::cout << std::format("{:<10}{:>5}{:>8} {:>3}\n", s.name, s.port, s.threa
    }
    return 0;
}
```

QUIZ

1. What happens here?

```
template <std::integral T> T twice(T x) { return x * 2; }
auto r = twice(2.5);
```

A. `r` is `5.0`

B. `r` is `4` — the argument is truncated to `int`

C. Compile error at the call: the constraint `std::integral<double>` is not satisfied

D. Undefined behaviour

2. What does this print?

```
auto evens = std::views::iota(1)
           | std::views::filter([](int x) { return x % 2 == 0; })
           | std::views::take(3);
for (int x : evens) std::cout << x << ' ';
```

A. `2 4 6`

B. `1 2 3`

C. It never terminates: `iota(1)` is infinite

D. Compile error: `filter` cannot follow an infinite range

3. Given

```
struct P { int x, y; auto operator⟺(const P&) const = default; };
```

what is `P{1, 2} < P{1, 3}`?

A. `true`

B. `false`

C. Compile error: no `operator<` was declared

D. Compiles, but `==` is not available

4. What does `std::format("{:>6.2f}|{:04}", 3.14159, 7)` return?

A. `3.14|0007`

B. `3.14 |7000`

C. `3.1416|0007`

D. `3.14| 7`

5. How does `std::jthread` differ from `std::thread`?

A. It runs on a real-time scheduler

B. It joins in its destructor and can be asked to stop through a `std::stop_token`

C. It detaches automatically when it goes out of scope

D. It cannot run a lambda

6. Which of these does *not* compile on this track's runtime?

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

B. `std::span<const int> s{v};`

C. `std::println("{}", x);`

D. `std::jthread t{[] { work(); }};`

7. Does this compile?

```
struct S { int a; int b; };
S s{.b = 1, .a = 2};
```

A. Yes; `a` is 2 and `b` is 1

B. No: designators must appear in declaration order

C. Yes; `a` is 1 and `b` is 2

D. No: designated initialisers need a constructor

1. C A concept is checked at the call site during overload resolution; `double` is not integral, so no viable `twice` exists and the compiler says which constraint failed. Without the concept the same template would have compiled and returned `5.0`.

2. A Views are lazy: `iota(1)` produces numbers only as they are pulled, `filter` passes the even ones, and `take(3)` stops the pipeline after three. Nothing is computed until the loop asks, so the infinite source is fine.

3. A A defaulted `<=>` compares members in declaration order — `x` ties, `y` decides — and the compiler rewrites `a < b` as `(a <=> b) < 0`. Defaulting `<=>` also declares a defaulted `==`, so all six comparisons work.

4. A `>6.2f` right-aligns a fixed two-decimal number in six columns; `04` zero-pads the integer to four digits. Alignment defaults to right for numbers and left for strings, so the zero padding needs the explicit `0`.

5. B A `std::thread` that is destroyed while joinable calls `std::terminate`; a `std::jthread` requests a stop and joins instead, which makes it RAII-correct. Both take any callable, lambdas included.

6. C `std::print` and `std::println` are C++23 (`<print>`) and absent under `-std=c++20`. `std::format`, `std::span` and `std::jthread` are C++20 and available.

7. B C++20 designated initialisers must name members in the order they are declared (C allows any order; C++ does not), and they work only on aggregates — no constructor involved. `S s{.a = 2, .b = 1};` is the fix.

CHECKPOINT · 25 MIN

Checkpoint — Modern C++

This checkpoint covers the whole module: the standards and what compiles here, `auto`, `decltype` and structured bindings, `constexpr` / `consteval` / `constinit` and compile-time tables, the vocabulary types from `std::tuple` to `std::span` and `<chrono>`, and the C++20 set — concepts, ranges, `⟺`, `std::format` — with C++23 as reading.

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:

- What does `auto` drop from the initialiser's type, and how do `auto&` and `const auto&` differ?
- Why is `auto [k, v] = *it;` a copy, and what makes it a reference?
- What is the difference between `const`, `constexpr`, `consteval` and `constinit`?
- Why can a `consteval` function not be called with a loop variable from a `constexpr` function?
- Why does `std::chrono::seconds s = 1500ms;` fail to compile, and what does `duration_cast` do instead?
- Which of `std::format`, `std::print`, `std::expected` and `std::span` compile under `-std=c++20`?

The three programs are a duration ledger that sums `<chrono>` durations, a day-of-year calculator on a `constexpr` prefix-sum table and `std::optional`, and a concept-constrained statistics function over a `std::span` printed with `std::format`. Read the input format carefully before writing the parsing.

EXERCISE 1

Duration ledger

Read an integer `n`, then `n` durations, one per line, each an integer immediately followed by a unit: `ms`, `s`, `min` or `h` (`250ms`, `90s`, `5min`, `2h`). Add them all up as a `std::chrono::milliseconds` — convert each token into the matching `<chrono>` type and let the implicit widening conversion do the rest; never multiply by 1000 by hand. Then split the total with `duration_cast` and duration

subtraction and print `total: <h>h <m>m <s>s <ms>ms`, followed by `seconds: <total in whole seconds>`.

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

Example: input `3` / `90s` / `5min` / `2h` →

```
total: 2h 6m 30s 0ms
seconds: 7590
```

INPUT 1

```
3
90s
5min
2h
```

OUTPUT 1

```
total: 2h 6m 30s 0ms
seconds: 7590
```

INPUT 2

```
2
250ms
1s
```

OUTPUT 2

```
total: 0h 0m 1s 250ms
seconds: 1
```

REFERENCE SOLUTION

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

int main() {
    using namespace std::chrono;
    int n;
    std::cin >> n;
    milliseconds total{0};
    for (int i = 0; i < n; ++i) {
        std::string token;
        std::cin >> token;
        std::size_t split = 0;
        while (split < token.size() && std::isdigit(static_cast<unsigned char>(toker
        const long long amount = std::stoll(token.substr(0, split));
        const std::string unit = token.substr(split);
        if (unit == "ms") total += milliseconds{amount};
        else if (unit == "s") total += seconds{amount};
        else if (unit == "min") total += minutes{amount};
        else if (unit == "h") total += hours{amount};
    }
    const auto h = duration_cast<hours>(total);
    const auto m = duration_cast<minutes>(total - h);
    const auto s = duration_cast<seconds>(total - h - m);
    const auto ms = total - h - m - s;
    std::cout << "total: " << h.count() << "h " << m.count() << "m " << s.count() <<
    std::cout << "seconds: " << duration_cast<seconds>(total).count() << '\n';
    return 0;
}
```

EXERCISE 2

Day of the year at compile time

Using a `constexpr std::array<int, 12>` of month lengths for a non-leap year, build a second table `START` of thirteen prefix sums with a `constexpr` function — `START[m]` is the number of days before month `m + 1`, so `START[0] == 0`, `START[2] == 59` and `START[12] == 365` — and prove those three with a `static_assert`. Then write `std::optional<int> dayOfYear(int month, int day)` that returns `std::nullopt` when the month is not 1–12 or the day is not valid for that month, and otherwise the day's number in the year.

Read an integer `n`, then `n` lines of `month day`, and print `<month>/<day> → <day of year>` or `<month>/<day> → invalid`. Use an `if` with initialiser to test the optional.

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

Example: input `3` / `1 1` / `3 1` / `12 31` →

```
1/1 → 1
3/1 → 60
12/31 → 365
```

INPUT 1

```
3
1 1
3 1
12 31
```

OUTPUT 1

```
1/1 → 1
3/1 → 60
12/31 → 365
```

INPUT 2

```
2
2 29
13 1
```

OUTPUT 2

```
2/29 → invalid
13/1 → invalid
```

REFERENCE SOLUTION

```cpp
#include <array>
#include <iostream>
#include <optional>

constexpr std::array<int, 12> DAYS{31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

// START[m] = days before month m + 1; START[12] = 365.
constexpr std::array<int, 13> makeStarts() {
    std::array<int, 13> starts{};
    for (std::size_t m = 0; m < DAYS.size(); ++m) starts[m + 1] = starts[m] + DAYS[m
    return starts;
}

constexpr auto START = makeStarts();
static_assert(START[0] == 0 && START[2] == 59 && START[12] == 365);

std::optional<int> dayOfYear(int month, int day) {
    if (month < 1 || month > 12) return std::nullopt;
    const std::size_t index = static_cast<std::size_t>(month - 1);
    if (day < 1 || day > DAYS[index]) return std::nullopt;
    return START[index] + day;
}

int main() {
    int n;
    std::cin >> n;
    for (int i = 0; i < n; ++i) {
        int month, day;
        std::cin >> month >> day;
        std::cout << month << '/' << day << " -> ";
        if (const auto d = dayOfYear(month, day); d.has_value()) {
            std::cout << *d << '\n';
        } else {
            std::cout << "invalid\n";
        }
    }
    return 0;
}
```

EXERCISE 3

Constrained statistics

Define a concept `Number` satisfied by any integral or floating-point type, and a function template `summarise` constrained by it that takes a `std::span<const T>` and returns a `std::tuple<T, T, double>` of the smallest value, the largest value and the mean. Accumulate the sum in a `long long`

when `T` is integral and in a `double` otherwise — `if constexpr` or `std::conditional_t` — so integer input never overflows.

The first token of the input is `int` or `double`; then comes `n` (at least 1) and `n` numbers of that kind. Read them into a `std::vector` of the matching type, call `summarise` on a span over it, bind the result with a structured binding, and print one line with `std::format`: for `int`, `lo=<lo> hi=<hi> mean=<mean>`; for `double`, the same with `lo` and `hi` to two decimals. The mean always has two decimals.

Input: the kind, `n`, then `n` numbers. **Output:** one line.

Example: input `int 4 1 9 3 5` → `lo=1 hi=9 mean=4.50`; input `double 3 1.5 9.25 2.75` → `lo=1.50 hi=9.25 mean=4.50`.

INPUT 1

```
int
4
1 9 3 5
```

OUTPUT 1

```
lo=1 hi=9 mean=4.50
```

INPUT 2

```
double
3
1.5 9.25 2.75
```

OUTPUT 2

```
lo=1.50 hi=9.25 mean=4.50
```

REFERENCE SOLUTION

```cpp
#include <concepts>
#include <format>
#include <iostream>
#include <span>
#include <string>
#include <tuple>
#include <type_traits>
#include <vector>

template <typename T>
concept Number = std::integral<T> || std::floating_point<T>;

// Smallest, largest and mean of a non-empty sequence. The running sum is a long lon
// for integers (so int input cannot overflow) and a double otherwise.
template <Number T>
std::tuple<T, T, double> summarise(std::span<const T> xs) {
    using Sum = std::conditional_t<std::integral<T>, long long, double>;
    T lo = xs[0];
    T hi = xs[0];
    Sum sum = 0;
    for (const T x : xs) {
        if (x < lo) lo = x;
        if (x > hi) hi = x;
        sum += x;
    }
    return {lo, hi, static_cast<double>(sum) / static_cast<double>(xs.size())};
}

template <typename T>
void run() {
    int n;
    std::cin >> n;
    std::vector<T> values(n);
    for (auto& v : values) std::cin >> v;
    const auto [lo, hi, mean] = summarise(std::span<const T>{values});
    if constexpr (std::integral<T>) {
        std::cout << std::format("lo={} hi={} mean={:.2f}\n", lo, hi, mean);
    } else {
        std::cout << std::format("lo={:.2f} hi={:.2f} mean={:.2f}\n", lo, hi, mean);
    }
}

int main() {
    std::string kind;
    std::cin >> kind;
    if (kind == "int") run<int>();
    else run<double>();
```

```
    return 0;
}
```

CHECKPOINT QUESTIONS

1. Structured bindings, `std::optional`, `std::string_view` and `if` with an initialiser all arrived in which standard?

A. C++11

B. C++14

C. C++17

D. C++20

2. Which header exists solely to define the library feature-test macros such as `__cpp_lib_format`?

A. `<format>`

B. `<version>`

C. `<cstddef>`

D. `<limits>`

3. What does this print?

```
std::vector<int> v{1, 2};
auto w = v;
w.push_back(3);
std::cout << v.size();
```

A. `2`

B. `3`

C. Compile error

D. Undefined behaviour

4. In `for (const auto& [k, v] : m)` over a `std::map<std::string, int>`, what is the type of `k`?

A. `std::string`

B. `const std::string`

C. `std::string&`

D. `int`

5. What does `static_assert(SIEVE[91] == false);` beside `constexpr auto SIEVE = makeSieve();` guarantee?

A. The sieve is rebuilt at run time if the check fails

B. The check runs on every execution and prints a message

C. The build fails if the entry is wrong; the check costs nothing at run time

D. `SIEVE` becomes `const` for the rest of the program

6. A function computes a minimum, a maximum and a total. Which return type is the idiomatic choice?

A. `std::tuple<int, int, long long>`

B. A struct with named members `lo`, `hi`, `sum`

C. Three output parameters by reference

D. `std::vector<long long>` with three elements

7. `std::span<const int> s{v};` where `v` is `{10, 20, 30, 40, 50}`. Which elements does `s.subspan(2, 3)` view?

A. `20, 30`

B. `30, 40, 50`

C. `30, 40`

D. `40, 50`

8. What does `std::chrono::duration_cast<std::chrono::minutes>(std::chrono::seconds{150}).count()` return?

A. `2`

B. `2.5`

C. `3`

D. Compile error

9. What is the main benefit of constraining a template with a concept rather than leaving it unconstrained?

A. The generated code runs faster

B. A bad argument fails at the call site with a message naming the unmet requirement, instead of deep inside the body

C. The template can be defined in a `.cpp` file

D. It allows run-time type checks

10. In `std::ranges::sort(people, {}, &Person::age)`, what is the third argument?

A. A comparator

B. A projection: sort by each element's `age`

C. A predicate that filters which elements are sorted

D. The initial value

11. What does `std::format("{} {:.1f}", 3.0, 3.0)` return?

A. `3 3.0`

B. `3.0 3.0`

C. `3.000000 3.0`

D. `3 3`

12. Which C++23 type represents "either a value or an error" without exceptions?

A. `std::optional<T>`

B. `std::variant<T, E>`

C. `std::expected<T, E>`

D. `std::error_code`

13. What does `[[likely]]` on a branch change?

A. The branch is evaluated first

B. The compiler's code layout for that branch; program behaviour is unchanged

C. The branch is checked at compile time

D. The branch can no longer throw

14. Why must a judged program that uses `std::jthread` still print its results only after the threads have finished?

A. Because `std::jthread` does not join automatically

B. Because output from several threads interleaves unpredictably, so the expected output could never match

C. Because `std::cout` cannot be used from a thread

D. Because the judge runs single-core

15. Which line is a compile error?

```
auto a = {1, 2};        // 1
auto b{1};              // 2
auto c{1, 2};           // 3
auto d = 1;             // 4
```

A. Line 1

B. Line 2

C. Line 3

D. Line 4

ANSWER KEY · CHECKPOINT — MODERN C++

1. C C++17 is the vocabulary-types release. C++11 brought `auto` and lambdas, C++14 refined them, and C++20 added concepts, ranges and `std::format`.

2. B `<version>` (C++20) defines every `__cpp_lib_*` macro in one place. Each feature's own header also defines its macro, but only `<version>` defines all of them without pulling in the feature.

3. A `auto w = v;` deduces `std::vector<int>` and copies — value semantics, not a reference. `auto& w = v;` would have made `w` an alias and printed `3`.

4. B A map's value type is `std::pair<const std::string, int>`: the key is `const` inside the pair so it can never be changed under the tree. The binding names that `const std::string`, seen through a const reference.

5. C `static_assert` is evaluated by the compiler; a false condition is a compile error and a true one leaves no trace in the executable. It is the natural unit test for compile-time tables.

6. B A struct gives the results names — `result.sum` rather than `std::get<2>(result)` — and still works with structured bindings. Tuples suit generic code and pairs suit natural two-ness; output parameters are the pre-C++11 shape.

7. B `subspan(offset, count)` starts at index `offset` and takes `count` elements: indices 2, 3, 4. It is O(1) and copies nothing — a new pointer-and-length over the same memory.

8. A `duration_cast` truncates toward zero, so 150 seconds is 2 whole minutes. The implicit conversion is refused precisely because it would lose the 30 seconds; `duration_cast` is the explicit acknowledgement.

9. B Constraints are checked during overload resolution, so the error says "`double` does not satisfy `std::integral`" at the call rather than pages of instantiation notes. Code generation is unchanged; templates still live in headers.

10. B Ranges algorithms take an optional comparator (here `{}`, the default `<`) and then a projection applied to each element before comparing. `sort(people, &Person::age)` would pass the member pointer as the comparator and fail to compile.

11. A An empty spec prints the shortest representation that round-trips — `3` for `3.0` — while `.1f` asks for fixed notation with one decimal. This differs from `std::cout`, which would print `3` for both without manipulators.

12. C `std::expected<T, E>` holds a `T` or an `E` with `has_value()`, `value()` and `error()`. `std::optional` has no error payload, `std::variant` has no success/failure meaning, and `std::error_code` is only the error half. It is C++23 and reading only on this track.

13. B It is an optimisation hint about which outcome to expect, affecting instruction placement and nothing observable. A wrong hint costs a little speed, never correctness.

14. B `std::jthread` does join in its destructor, but that only guarantees the thread has ended by then. Anything the threads print themselves arrives in an order the scheduler chooses; a deterministic program collects results, joins, then prints once.

15. C Direct-list-initialisation of `auto` requires exactly one element (line 2 gives `int`); line 3 is rejected. Line 1 is copy-list-initialisation and deduces `std::initializer_list<int>`; line 4 is plain `int`.
