Advanced Systems Programming

A Crash Course in Modern C++

Till Smejkal

September 18th, 2026

Technische Universität Dresden

About this course

- One day, four blocks, six exercises.
- The goal is *modern* C++ — the language as it is written in 2026, not C with classes.
- You already know C. Much of what you know still applies; some of it is now the wrong answer.

About this course

- One day, four blocks, six exercises.
- The goal is *modern* C++ — the language as it is written in 2026, not C with classes.
- You already know C. Much of what you know still applies; some of it is now the wrong answer.
- I will say “don’t do that” about things that are perfectly legal. That is the point.

About this course

- One day, four blocks, six exercises.
- The goal is *modern* C++ — the language as it is written in 2026, not C with classes.
- You already know C. Much of what you know still applies; some of it is now the wrong answer.
- I will say “don’t do that” about things that are perfectly legal. That is the point.
- Please interrupt and ask.

About this course

- One day, four blocks, six exercises.
- The goal is *modern* C++ — the language as it is written in 2026, not C with classes.
- You already know C. Much of what you know still applies; some of it is now the wrong answer.
- I will say “don’t do that” about things that are perfectly legal. That is the point.
- Please interrupt and ask.

Key idea

You will not remember all of this. You *will* remember where to look.

What we will not cover

- Threads, `std::atomic`, mutexes, `std::async`, coroutines. → Threads lecture

What we will not cover

- Threads, `std::atomic`, mutexes, `std::async`, coroutines. → Threads lecture
- Exceptions, beyond knowing that they exist and that RAII is what makes them survivable.

What we will not cover

- Threads, `std::atomic`, mutexes, `std::async`, coroutines. → Threads lecture
- Exceptions, beyond knowing that they exist and that RAII is what makes them survivable.
- Every corner of the standard library. It is enormous.

What we will not cover

- Threads, `std::atomic`, mutexes, `std::async`, coroutines. → Threads lecture
- Exceptions, beyond knowing that they exist and that RAII is what makes them survivable.
- Every corner of the standard library. It is enormous.
- Build systems beyond a hand-written Makefile.

What we will not cover

- Threads, `std::atomic`, mutexes, `std::async`, coroutines. → Threads lecture
- Exceptions, beyond knowing that they exist and that RAII is what makes them survivable.
- Every corner of the standard library. It is enormous.
- Build systems beyond a hand-written Makefile.

Nor is this a course on *programming*. It is a course on how to say what you already know how to say, in C++.

The plan for today

Block 1	Types, control flow, and getting something to run	Ex. 1
Block 2	Functions, lambdas, data, and `std::format`	Ex. 2, 3
Block 3	Ownership, value semantics, classes	Ex. 4
Block 4	Templates, concepts, ranges	Ex. 5, 6

The plan for today

Block 1	Types, control flow, and getting something to run	Ex. 1
Block 2	Functions, lambdas, data, and `std::format`	Ex. 2, 3
Block 3	Ownership, value semantics, classes	Ex. 4
Block 4	Templates, concepts, ranges	Ex. 5, 6

Roughly 90 minutes per block.

The plan for today

Block 1	Types, control flow, and getting something to run	Ex. 1
Block 2	Functions, lambdas, data, and `std::format`	Ex. 2, 3
Block 3	Ownership, value semantics, classes	Ex. 4
Block 4	Templates, concepts, ranges	Ex. 5, 6

Roughly 90 minutes per block.

The six exercises build one program: a log-file analyser. By the end of the day it parses, filters, aggregates and prints a real log.

Getting the code

```
$ git clone \
    https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-cpp.git
$ cd sysprog-cpp && make check
```

Getting the code

```
$ git clone \
    https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-cpp.git
$ cd sysprog-cpp && make check
```

You need GCC 14 or newer, or Clang 18 or newer. Check with:

```
$ g++ --version
$ clang++ --version
```

Getting the code

```
$ git clone \
    https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-cpp.git
$ cd sysprog-cpp && make check
```

You need GCC 14 or newer, or Clang 18 or newer. Check with:

```
$ g++ --version
$ clang++ --version
```

All tests will fail. That is the starting position — each exercise turns one test green.

Getting the code

```
$ git clone \
    https://os.inf.tu-dresden.de/Studium/SysProg/SS2026/sysprog-cpp.git
$ cd sysprog-cpp && make check
```

You need GCC 14 or newer, or Clang 18 or newer. Check with:

```
$ g++ --version
$ clang++ --version
```

All tests will fail. That is the starting position — each exercise turns one test green. Solutions live in the same repository, one commit per exercise:

```
$ git show ex3
$ git diff master ex3
```

BLOCK 1 / 4

Getting started

- Hello, C++
- Types and values
- Control flow

Exercise 1

Three ways to print a number

```c
#include <cstdio>
printf("%d items\n", n);            // C: fast, terse, and a loaded gun
```

Three ways to print a number

```c
#include <cstdio>
printf("%d items\n", n);            // C: fast, terse, and a loaded gun
```

C

```cpp
#include <iostream>
std::cout << n << " items" << std::endl;   // C++98: safe, but shouting
```

C++98

Three ways to print a number

```cpp
#include <cstdio>
printf("%d items\n", n);            // C: fast, terse, and a loaded gun
```

C

```cpp
#include <iostream>
std::cout << n << " items" << std::endl;   // C++98: safe, but shouting
```

C++98

```cpp
#include <print>
std::println("{} items", n);        // C++23: what we will use
```

Three ways to print a number

```cpp
#include <cstdio>
printf("%d items\n", n);            // C: fast, terse, and a loaded gun
```

C

```cpp
#include <iostream>
std::cout << n << " items" << std::endl;   // C++98: safe, but shouting
```

C++98

```cpp
#include <print>
std::println("{} items", n);        // C++23: what we will use
```

Key idea

`std::println` is type-safe like `cout` and readable like `printf`.

Why not printf?

```c
printf("%d\n", 3.5);        // undefined behaviour, compiles fine
printf("%s\n", 42);         // undefined behaviour, compiles fine
```


The format string and the arguments are checked by nothing. Modern compilers warn for *literal* format strings, and can do nothing at all once the string comes from a variable.

Why not printf?

```c
printf("%d\n", 3.5);        // undefined behaviour, compiles fine
printf("%s\n", 42);         // undefined behaviour, compiles fine
```


The format string and the arguments are checked by nothing. Modern compilers warn for *literal* format strings, and can do nothing at all once the string comes from a variable.

```c
std::println("{}", 3.5);    // prints 3.5
std::println("{}", 42);     // prints 42
std::println("{:d}", 3.5);  // compile error, not a runtime surprise
```

Why not printf?

```c
printf("%d\n", 3.5);        // undefined behaviour, compiles fine
printf("%s\n", 42);         // undefined behaviour, compiles fine
```


The format string and the arguments are checked by nothing. Modern compilers warn for *literal* format strings, and can do nothing at all once the string comes from a variable.

```c
std::println("{}", 3.5);    // prints 3.5
std::println("{}", 42);     // prints 42
std::println("{:d}", 3.5);  // compile error, not a runtime surprise
```

`{}` means “print the next argument, however that type wants to be printed”. We will learn later how to print our own types. → Block 2

A whole program

```
#include <print>
int main()
{
    std::println("hello, {}", "world");
    return 0;
}
```

A whole program

```cpp
#include <print>
int main()
{
    std::println("hello, {}", "world");
    return 0;
}
```

```
$ g++ -std=c++23 -Wall -Wextra -o hello hello.cc
$ ./hello
```

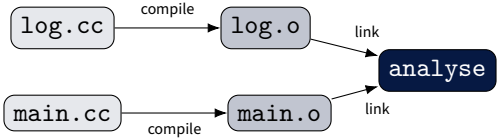
A whole program

```cpp
#include <print>
int main()
{
    std::println("hello, {}", "world");
    return 0;
}
```

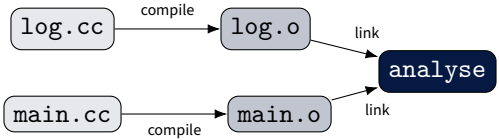
```
$ g++ -std=c++23 -Wall -Wextra -o hello hello.cc
$ ./hello
```

- `-std=c++23` — not the default. Always set it.
- `-Wall -Wextra` — not on by default either. Always set these too.
- `main` returning `int`; the `return 0` is optional in `main` alone.

How a program gets built

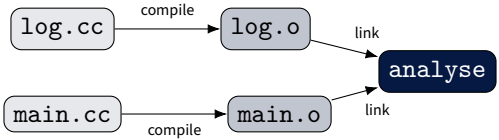


How a program gets built



Each `.cc` file is a *translation unit* and is compiled on its own. A header is not compiled — it is textually pasted into every translation unit that includes it.

How a program gets built



Each `.cc` file is a *translation unit* and is compiled on its own. A header is not compiled — it is textually pasted into every translation unit that includes it.

Consequences you will meet today:

- A function may be *declared* many times, but *defined* once.
- Templates are the exception, and that is why they live in headers.

Headers

```
// log.h
#pragma once                // instead of the #ifndef/#define dance

#include <string_view>

std::string_view to_string(Level level);   // declaration
```

```
// log.cc
#include "log.h"

std::string_view to_string(Level level) { /* ... */ }   // definition
```

Headers

```
// log.h
#pragma once                // instead of the #ifndef/#define dance

#include <string_view>

std::string_view to_string(Level level);   // declaration
```

```
// log.cc
#include "log.h"

std::string_view to_string(Level level) { /* ... */ }   // definition
```

- `#include <...>` for the standard library, `#include "..."` for your own.
- `#pragma once` is not standard, yet every compiler you will meet supports it.

Modules — the future, sort of

C++20 added modules, and C++23 made the whole standard library importable:

```
1 import std;
2
3 int main() { std::println("no #include in sight"); }
```

Modules — the future, sort of

C++20 added modules, and C++23 made the whole standard library importable:

```
1 import std;
2
3 int main() { std::println("no #include in sight"); }
```

One import replaces every `#include`, and it compiles dramatically faster because nothing is re-parsed.

Modules — the future, sort of

C++20 added modules, and C++23 made the whole standard library importable:

```
import std;

int main() { std::println("no #include in sight"); }
```

One import replaces every `#include`, and it compiles dramatically faster because nothing is re-parsed.

Key idea

It works today on GCC 15+ and Clang 17+ — but needs build-system support that is still settling. We will stay with headers.

Namespaces

Every standard-library name lives in namespace `std`. Three ways to reach one:

```
1 std::string name = "alice";   // qualified -- the default
2 using std::string;            // using-declaration: this one name
3 using namespace std;          // using-directive: the whole namespace
```

Namespaces

Every standard-library name lives in namespace `std`. Three ways to reach one:

```
1 std::string name = "alice";   // qualified -- the default
2 using std::string;            // using-declaration: this one name
3 using namespace std;          // using-directive: the whole namespace
```

Define your own:

```
4 namespace logalyzer { struct Entry { /* ... */ }; }
5 logalyzer::Entry entry;
```

Namespaces

Every standard-library name lives in namespace `std`. Three ways to reach one:

```
1 std::string name = "alice";   // qualified -- the default
2 using std::string;            // using-declaration: this one name
3 using namespace std;          // using-directive: the whole namespace
```

Define your own:

```
4 namespace logalyzer { struct Entry { /* ... */ }; }
5 logalyzer::Entry entry;
```

Pitfall

Do not write `using namespace std;`

Thousands of names enter scope at once, and later additions to the standard library start colliding with your own.

Fundamental types

Same as C, with the same guarantees — which is to say, few:

- `bool` — `true` / `false`, not `int`.
- `char`, `short`, `int`, `long`, `long long` — each `signed` or `unsigned`.
- `float`, `double`, `long double`.
- `void` — “no type”.
- `T*` — a pointer to `T`, just as in C. Spell the null pointer `nullptr`.

Fundamental types

Same as C, with the same guarantees — which is to say, few:

- `bool` — `true` / `false`, not `int`.
- `char`, `short`, `int`, `long`, `long long` — each `signed` or `unsigned`.
- `float`, `double`, `long double`.
- `void` — “no type”.
- `T*` — a pointer to `T`, just as in C. Spell the null pointer `nullptr`.

The standard promises only *minimum* widths. When the width matters, say so:

```
#include <cstdint>
std::int32_t   id;       // exactly 32 bits, signed
std::uint64_t  offset;   // exactly 64 bits, unsigned
std::size_t    n;        // big enough for any object size
```

auto

`auto` asks the compiler to work out the type from the initialiser.

```
1 auto n = 42;                    // int
2 auto x = 3.5;                   // double
3 auto name = std::string{"hi"};  // std::string
```

auto

`auto` asks the compiler to work out the type from the initialiser.

```
1 auto n = 42;                    // int
2 auto x = 3.5;                   // double
3 auto name = std::string{"hi"};  // std::string
```

It is not Python's dynamic typing: the type is fixed at compile time, you just did not have to spell it.

auto

`auto` asks the compiler to work out the type from the initialiser.

```
1 auto n = 42;                    // int
2 auto x = 3.5;                   // double
3 auto name = std::string{"hi"};  // std::string
```

It is not Python's dynamic typing: the type is fixed at compile time, you just did not have to spell it.

Use it when the type is obvious or unspeakable:

```
4 for (auto const& [level, count] : counts)   // the alternative is unreadable
5 auto lambda = [](int x) { return x * 2; };  // the type has no name
```

auto

`auto` asks the compiler to work out the type from the initialiser.

```
auto n = 42;                    // int
auto x = 3.5;                   // double
auto name = std::string{"hi"};  // std::string
```

It is not Python's dynamic typing: the type is fixed at compile time, you just did not have to spell it.

Use it when the type is obvious or unspeakable:

```
for (auto const& [level, count] : counts)   // the alternative is unreadable
auto lambda = [](int x) { return x * 2; };  // the type has no name
```

Spell the type out when it is important:

```
std::uint32_t checksum = compute();   // the width matters here
```

decltype

`auto` takes its type from an initialiser. `decltype` asks for the type of an *expression* — without evaluating it:

```
1 auto n = 42;
2 auto x = 3.5;
3
4 decltype(n)     count = 0;    // int    -- same type as n
5 decltype(n * x) total{};      // double -- the type of the result
```

decltype

`auto` takes its type from an initialiser. `decltype` asks for the type of an *expression* — without evaluating it:

```
auto n = 42;
auto x = 3.5;

decltype(n)     count = 0;    // int    -- same type as n
decltype(n * x) total{};      // double -- the type of the result
```

You will rarely write it in everyday code. It earns its keep where a type can only be described by what you do with other values:

- a return type computed from a function's parameters → Block 2
- template metaprogramming — computing with types at compile time.

const

`const` means “I will not modify this”. The compiler enforces it.

```
1 int const limit = 100;        // or: const int limit = 100;
```

const

`const` means “I will not modify this”. The compiler enforces it.

```
1 int const limit = 100;        // or: const int limit = 100;
```

Read a declaration right to left:

```
2 int const*       p;   // pointer to const int      -- *p is fixed
3 int      * const q;   // const pointer to int      -- q is fixed
4 int const* const r;   // const pointer to const int -- both fixed
```

const

`const` means “I will not modify this”. The compiler enforces it.

```
1 int const limit = 100;        // or: const int limit = 100;
```

Read a declaration right to left:

```
2 int const*       p;   // pointer to const int      -- *p is fixed
3 int      * const q;   // const pointer to int      -- q is fixed
4 int const* const r;   // const pointer to const int -- both fixed
```

Best practice

Default to `const`. Drop it only when you find you need to.

`constexpr` **and** `constinit`

`const` says *who* may change it. `constexpr` says *when* it is known.

```
1 int constexpr answer = 42;              // a compile-time constant
2 int constexpr size   = answer * 2;      // also compile time
```

`constexpr` and `constinit`

`const` says *who* may change it. `constexpr` says *when* it is known.

```
1 int constexpr answer = 42;              // a compile-time constant
2 int constexpr size   = answer * 2;      // also compile time
```

A `constexpr` variable can be used where the language demands a constant:

```
3 std::array<int, size> table;            // size must be compile time
4 static_assert(answer == 42);            // checked while compiling
```

`constexpr` and `constinit`

`const` says *who* may change it. `constexpr` says *when* it is known.

```
1 int constexpr answer = 42;              // a compile-time constant
2 int constexpr size   = answer * 2;      // also compile time
```

A `constexpr` variable can be used where the language demands a constant:

```
3 std::array<int, size> table;            // size must be compile time
4 static_assert(answer == 42);            // checked while compiling
```

`constinit` is the rarer cousin: initialised at compile time, mutable afterwards.

```
5 constinit int counter = 0;   // no static-initialisation-order surprise
```

References

A reference is another name for an existing object.

```
1 int  value = 10;
2 int& alias = value;    // alias IS value, not a copy of it
3 alias = 20;            // value is now 20
```

References

A reference is another name for an existing object.

```
1 int  value = 10;
2 int& alias = value;    // alias IS value, not a copy of it
3 alias = 20;            // value is now 20
```

Unlike a pointer, a reference

- must be bound when it is created,
- can never be re-bound to something else,
- can never be null,
- needs no $*$ or `->` to use.

References

A reference is another name for an existing object.

```
int  value = 10;
int& alias = value;    // alias IS value, not a copy of it
alias = 20;            // value is now 20
```

Unlike a pointer, a reference

- must be bound when it is created,
- can never be re-bound to something else,
- can never be null,
- needs no $*$ or -> to use.

Best practice

Reach for a reference first. Use a pointer when you genuinely need “possibly nothing” or “re-assignable”.

`const&` — the workhorse

```
1 void print(std::string const& text);   // no copy, cannot modify
```

This is the most common parameter type in C++. It gives you C's "pass a pointer to avoid the copy" without the null check and without the callee modifying your data.

`const&` — the workhorse

```
1 void print(std::string const& text);   // no copy, cannot modify
```

This is the most common parameter type in C++. It gives you C's "pass a pointer to avoid the copy" without the null check and without the callee modifying your data.

You want	Write
a cheap copy (`int`, `double`)	`T`
to read a big object	`T const&`
to modify the caller's object	`T&`
to take ownership	`T&&` (move semantics) → Block 3

enum class

C's enums leak their names and convert to `int` behind your back:

```
1 enum Level { Debug, Info, Warn, Error };   // unscoped
2 int x = Warn;                              // compiles. why?
```


C-style enum

enum class

C's enums leak their names and convert to `int` behind your back:

```
1 enum Level { Debug, Info, Warn, Error };   // unscoped
2 int x = Warn;                              // compiles. why?
```


C-style enum

Scoped enums fix both problems:

```
3 enum class Level { Debug, Info, Warn, Error };
4
5 Level level = Level::Warn;     // must say Level::
6 int x = level;                 // compile error, good
```

enum class

C's enums leak their names and convert to `int` behind your back:

```
1 enum Level { Debug, Info, Warn, Error };   // unscoped
2 int x = Warn;                              // compiles. why?
```


C-style enum

Scoped enums fix both problems:

```
3 enum class Level { Debug, Info, Warn, Error };
4
5 Level level = Level::Warn;     // must say Level::
6 int x = level;                 // compile error, good
```

They still compare, and in declaration order:

```
7 if (entry.level >= Level::Warn) { /* warnings and errors */ }
```

enum class

C's enums leak their names and convert to `int` behind your back:

```
1 enum Level { Debug, Info, Warn, Error };   // unscoped
2 int x = Warn;                              // compiles. why?
```

C-style enum

Scoped enums fix both problems:

```
3 enum class Level { Debug, Info, Warn, Error };
4
5 Level level = Level::Warn;     // must say Level::
6 int x = level;                 // compile error, good
```

They still compare, and in declaration order:

```
7 if (entry.level >= Level::Warn) { /* warnings and errors */ }
```

In the exercise

That ordering is why `Level` is declared least-severe-first.

Strings

`std::string` owns its characters, grows on demand, and frees itself.

```
1 std::string name = "alice";
2 name += " smith";       // no malloc, no strcat, no overflow
```

Strings

`std::string` owns its characters, grows on demand, and frees itself.

```
std::string name = "alice";
name += " smith";      // no malloc, no strcat, no overflow
```

`std::string_view` does not own anything — it is a pointer and a length:

```
void log(std::string_view text);   // accepts string, const char*, literal

log(name);             // no copy
log("literal");        // no copy, no strlen at runtime
```

Strings

`std::string` owns its characters, grows on demand, and frees itself.

```
1 std::string name = "alice";
2 name += " smith";      // no malloc, no strcat, no overflow
```

`std::string_view` does not own anything — it is a pointer and a length:

```
3 void log(std::string_view text);   // accepts string, const char*, literal

5 log(name);             // no copy
6 log("literal");        // no copy, no strlen at runtime
```

Pitfall

A view does not keep anything alive

The characters must outlive the view. Returning a `string_view` to a local `string` is the modern dangling pointer.

String literals are not strings

A literal is an *array of characters* — it is not a `std::string`:

```
"alice"              // const char[6] -- five letters plus a NUL
auto a = "alice";    // const char*   -- the array decayed to a pointer
```

String literals are not strings

A literal is an *array of characters* — it is not a `std::string`:

```
"alice"              // const char[6] -- five letters plus a NUL
auto a = "alice";    // const char*   -- the array decayed to a pointer
```

So C's habits still bite:

```
auto greeting = "hello, " + name;  // error: no operator+ for arrays
a == "alice";                      // compares addresses, not characters
strlen(a);                         // walks the whole string, every time
```


C-style

String literals are not strings

A literal is an *array of characters* — it is not a `std::string`:

```
"alice"              // const char[6] -- five letters plus a NUL
auto a = "alice";    // const char*   -- the array decayed to a pointer
```

So C's habits still bite:

```
auto greeting = "hello, " + name;  // error: no operator+ for arrays
a == "alice";                      // compares addresses, not characters
strlen(a);                         // walks the whole string, every time
```


C-style

The `s` suffix builds a `std::string` on the spot:

```
using namespace std::string_literals;

auto b = "alice"s;                  // std::string -- owns it, knows its size
auto greeting = "hello, "s + name;  // fine
```

Vectors

`std::vector` is the default container: a resizable array that owns its storage.

```
1 std::vector<int> values;          // empty
2 values.push_back(10);
3 values.push_back(20);
4
5 std::println("{} values: {}", values.size(), values);
6 // 2 values: [10, 20]
```

Vectors

`std::vector` is the default container: a resizable array that owns its storage.

```cpp
std::vector<int> values;          // empty
values.push_back(10);
values.push_back(20);

std::println("{} values: {}", values.size(), values);
// 2 values: [10, 20]
```

```cpp
std::vector<std::string> names{"alice", "bob"};   // brace-initialised
names[0];         // no bounds check, like C
names.at(0);      // bounds-checked
```

Vectors

`std::vector` is the default container: a resizable array that owns its storage.

```
std::vector<int> values;          // empty
values.push_back(10);
values.push_back(20);

std::println("{} values: {}", values.size(), values);
// 2 values: [10, 20]
```

```
std::vector<std::string> names{"alice", "bob"};   // brace-initialised
names[0];         // no bounds check, like C
names.at(0);      // bounds-checked
```

It frees its storage when it goes out of scope — there is no `free(values)` to forget.

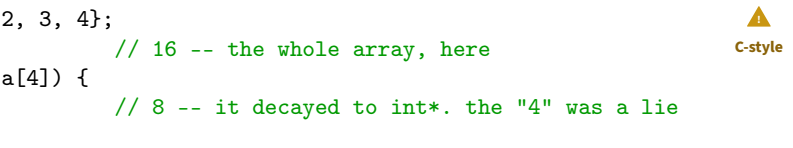
C arrays and `std::vector`

A C array is fixed at compile time, and forgets its size the moment you pass it anywhere:


```
int c[4] = {1, 2, 3, 4};
sizeof(c);              // 16 -- the whole array, here
void takes(int a[4]) {
    sizeof(a);          // 8 -- it decayed to int*. the "4" was a lie
}
```

C arrays and `std::vector`

A C array is fixed at compile time, and forgets its size the moment you pass it anywhere:




```cpp
int c[4] = {1, 2, 3, 4};
sizeof(c);              // 16 -- the whole array, here
void takes(int a[4]) {
    sizeof(a);          // 8 -- it decayed to int*. the "4" was a lie
}
```

	`int c[4]`	`std::vector<int>`
size	fixed at compile time	grows at run time
knows its size	only where declared	always, `.size()`
passed to a function	decays to `int*`	stays a vector
copy, assign, return	✗	✓

if/else

The familiar form, unchanged from C:

```
1 if (count > 0) { /* ... */ } else { /* ... */ }
```

if/else

The familiar form, unchanged from C:

```
1 if (count > 0) { /* ... */ } else { /* ... */ }
```

C++17 lets you declare the thing you are testing inside the `if`:

```
2 if (auto entry = parse(line); entry) {
3     use(*entry);
4 }
5 // entry is out of scope here -- it cannot leak into the rest of the function
```

if/else

The familiar form, unchanged from C:

```
1 if (count > 0) { /* ... */ } else { /* ... */ }
```

C++17 lets you declare the thing you are testing inside the `if`:

```
2 if (auto entry = parse(line); entry) {
3     use(*entry);
4 }
5 // entry is out of scope here -- it cannot leak into the rest of the function
```

Same shape as the `for` loop you already know: initialiser, then condition.

if/else

The familiar form, unchanged from C:

```
1 if (count > 0) { /* ... */ } else { /* ... */ }
```

C++17 lets you declare the thing you are testing inside the `if`:

```
2 if (auto entry = parse(line); entry) {
3     use(*entry);
4 }
5 // entry is out of scope here -- it cannot leak into the rest of the function
```

Same shape as the `for` loop you already know: initialiser, then condition.

Best practice

Keep variable scopes as small as possible.

switch

```
1 switch (level) {
2 case Level::Debug: return "DEBUG";
3 case Level::Info:  return "INFO";
4 case Level::Warn:  return "WARN";
5 case Level::Error: return "ERROR";
6 }
```

switch

```
switch (level) {
case Level::Debug: return "DEBUG";
case Level::Info:  return "INFO";
case Level::Warn:  return "WARN";
case Level::Error: return "ERROR";
}
```

Over an `enum class` with no `default`, the compiler warns if you forget a case — worth more than the `default` would be.

switch

```
switch (level) {
case Level::Debug: return "DEBUG";
case Level::Info:  return "INFO";
case Level::Warn:  return "WARN";
case Level::Error: return "ERROR";
}
```

Over an `enum class` with no `default`, the compiler warns if you forget a case — worth more than the `default` would be.
Fall-through is still the default, and still usually a bug. Say when you mean it:

```
case Level::Warn:
    ++warnings;
    [[fallthrough]];     // deliberate: silences the warning
case Level::Error:
    ++problems;  break;
```

Loops

`while` and the C-style `for` are unchanged:

```
1 for (std::size_t i = 0; i < values.size(); ++i)
2     std::println("{}", values[i]);
```


prefer range-for

Loops

`while` and the C-style `for` are unchanged:

```
1 for (std::size_t i = 0; i < values.size(); ++i)
2     std::println("{}", values[i]);
```


prefer range-for

...and you will hardly ever write one again. The range-based `for` says what you meant:

```
3 for (auto const& value : values)
4     std::println("{}", value);
```

Loops

`while` and the C-style `for` are unchanged:

```
1 for (std::size_t i = 0; i < values.size(); ++i)
2     std::println("{}", values[i]);
```


prefer range-for

...and you will hardly ever write one again. The range-based `for` says what you meant:

```
3 for (auto const& value : values)
4     std::println("{}", value);
```

- No index to get wrong, no `size()` to call, no < vs <=.
- Works on anything iterable: vector, string, map, array, your own types.

Choosing the loop variable

```
1 for (auto const& entry : entries)   // read only, no copy   <- the default
2 for (auto& entry : entries)         // modify in place
3 for (auto entry : entries)          // an explicit copy of each element
```

Choosing the loop variable

```
1 for (auto const& entry : entries)   // read only, no copy   <- the default
2 for (auto& entry : entries)         // modify in place
3 for (auto entry : entries)          // an explicit copy of each element
```

The third one is a silent performance bug when the element is expensive to copy:

```
4 for (auto entry : entries)   // copies every Entry, strings and all
5     std::println("{}", entry.message);
```

Choosing the loop variable

```
for (auto const& entry : entries)   // read only, no copy   <- the default
for (auto& entry : entries)         // modify in place
for (auto entry : entries)          // an explicit copy of each element
```

The third one is a silent performance bug when the element is expensive to copy:

```
for (auto entry : entries)   // copies every Entry, strings and all
    std::println("{}", entry.message);
```

Best practice

Write `auto const&` until you have a reason not to.

Exercise 1 — `split`

Implement `split` in `log.cc`:

```
1 std::vector<std::string_view> split(std::string_view text, char delim);
```

It splits `text` at every `delim` and *drops empty fields*:

```
2 split("a,b,,c", ',')                  ->  { "a", "b", "c" }
3 split("115 ERROR disk giving up", ' ') ->  { "115", "ERROR", "disk", ... }
```

Exercise 1 — `split`

Implement `split` in `log.cc`:

```
1 std::vector<std::string_view> split(std::string_view text, char delim);
```

It splits `text` at every `delim` and *drops empty fields*:

```
2 split("a,b,,c", ',')                  ->  { "a", "b", "c" }
3 split("115 ERROR disk giving up", ' ') ->  { "115", "ERROR", "disk", ... }
```

Hints:

- `text | std::views::split(delim)` hands you the fields one by one.
- A field becomes a view with `std::string_view{field}`.
- Or do it by hand with `find` and `substr` — both are fine.

```
$ make test-split && ./test-split
```

BLOCK 2 / 4

Functions, data and formatting

- Functions
- Lambdas
- Compound types
- Formatting

Exercises 2 and 3

Declaring functions

The C form still works:

```
1 int increase(int value, int step);
```

Declaring functions

The C form still works:

```
1 int increase(int value, int step);
```

C++11 added a trailing return type:

```
2 auto increase(int value, int step) -> int;
```

Declaring functions

The C form still works:

```
1 int increase(int value, int step);
```

C++11 added a trailing return type:

```
2 auto increase(int value, int step) -> int;
```

...which looks like noise here, but pays off once the return type uses the parameters:

```
3 template <typename A, typename B>
4 auto multiply(A a, B b) -> decltype(a * b);   // a, b only exist from here on
```

Written in front, `decltype(a * b)` fails: `a` is not declared yet. Templates themselves come later. Block 4

Declaring functions

The C form still works:

```
1 int increase(int value, int step);
```

C++11 added a trailing return type:

```
2 auto increase(int value, int step) -> int;
```

...which looks like noise here, but pays off once the return type uses the parameters:

```
3 template <typename A, typename B>
4 auto multiply(A a, B b) -> decltype(a * b);   // a, b only exist from here on
```

Written in front, `decltype(a * b)` fails: `a` is not declared yet. Templates themselves come later. Block 4

Best practice

Pick one version per codebase and stick to it.

Deduced return types

Leave the return type out entirely and the compiler works it out:

```
1 auto square(int x) { return x * x; }      // returns int
```

Deduced return types

Leave the return type out entirely and the compiler works it out:

```
1 auto square(int x) { return x * x; }      // returns int
```

All `return` statements must agree:

```
2 auto confused(bool flag)
3 {
4     if (flag) return 1;      // int
5     else      return 2.5;    // double -- compile error
6 }
```

Deduced return types

Leave the return type out entirely and the compiler works it out:

```
1 auto square(int x) { return x * x; }      // returns int
```

All `return` statements must agree:

```
2 auto confused(bool flag)
3 {
4     if (flag) return 1;      // int
5     else      return 2.5;    // double -- compile error
6 }
```

Pitfall

The caller has to read the body

A deduced return type is convenient in a template and unhelpful in a public header. Spell it out in interfaces.

Default arguments and overloading

```
1 std::vector<std::string_view> split(std::string_view text, char delim = ',');
2
3 split(line);        // uses ','
4 split(line, ' ');   // uses ' '
```

Default arguments and overloading

```
1 std::vector<std::string_view> split(std::string_view text, char delim = ',');
2
3 split(line);        // uses ','
4 split(line, ' ');   // uses ' '
```

Several functions may share a name if the compiler can tell the calls apart:

```
5 void log(std::string_view message);
6 void log(Level level, std::string_view message);
7 void log(Entry const& entry);
```

Default arguments and overloading

```
1 std::vector<std::string_view> split(std::string_view text, char delim = ',');
2
3 split(line);        // uses ','
4 split(line, ' ');   // uses ' '
```

Several functions may share a name if the compiler can tell the calls apart:

```
5 void log(std::string_view message);
6 void log(Level level, std::string_view message);
7 void log(Entry const& entry);
```

Overloads are resolved on argument *types* and count — never on the return type:

```
8 int  get();
9 auto get() -> double;   // error: cannot overload on return type alone
```

When overloading goes wrong

```
1 void print(int value);
2 void print(double value);
3
4 print(42);      // int
5 print(3.5);     // double
6 print('a');     // int      -- char promotes
7 print(42u);     // ambiguous? no: unsigned -> int wins
```

When overloading goes wrong

```
1 void print(int value);
2 void print(double value);
3
4 print(42);      // int
5 print(3.5);     // double
6 print('a');     // int      -- char promotes
7 print(42u);     // ambiguous? no: unsigned -> int wins
```

Overload resolution has a long and precise rulebook. You do not need to memorise it; you need to recognise when you are relying on it.

When overloading goes wrong

```
1 void print(int value);
2 void print(double value);
3
4 print(42);      // int
5 print(3.5);     // double
6 print('a');     // int      -- char promotes
7 print(42u);     // ambiguous? no: unsigned -> int wins
```

Overload resolution has a long and precise rulebook. You do not need to memorise it; you need to recognise when you are relying on it.

Best practice

If you cannot tell at a glance which overload runs, neither can the next reader. Give them different names.

`constexpr` functions

A `constexpr` function *may* run at compile time:

```
1  constexpr int square(int x) { return x * x; }
2
3  int constexpr table_size = square(4);   // computed while compiling
4  int n = read_number();
5  int runtime = square(n);                // same function, at runtime
```

`constexpr` functions

A `constexpr` function *may* run at compile time:

```
1  constexpr int square(int x) { return x * x; }
2
3  int constexpr table_size = square(4);   // computed while compiling
4  int n = read_number();
5  int runtime = square(n);                // same function, at runtime
```

`consteval` says it *must*:

```
6  consteval int checked_square(int x) { return x * x; }
7
8  int constexpr ok  = checked_square(4);   // fine
9  int bad = checked_square(n);             // compile error -- n is runtime
```

`constexpr` functions

A `constexpr` function *may* run at compile time:

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

int constexpr table_size = square(4);   // computed while compiling
int n = read_number();
int runtime = square(n);                // same function, at runtime
```

`consteval` says it *must*:

```
consteval int checked_square(int x) { return x * x; }

int constexpr ok  = checked_square(4);   // fine
int bad = checked_square(n);             // compile error -- n is runtime
```

Use `consteval` when running at runtime would be a bug — a lookup table you never want built on the fly, a compile-time validity check.

Computing at compile time

```
1  consteval auto make_table()
2  {
3      std::array<int, 16> table{};
4      for (int i = 0; i < 16; ++i)
5          table[i] = i * i;
6      return table;
7  }
8
9  auto constexpr squares = make_table();   // the loop runs in the compiler
10
11 static_assert(squares[10] == 100);       // checked, costs nothing at runtime
```

Computing at compile time

```
1  consteval auto make_table()
2  {
3      std::array<int, 16> table{};
4      for (int i = 0; i < 16; ++i)
5          table[i] = i * i;
6      return table;
7  }
8
9  auto constexpr squares = make_table();   // the loop runs in the compiler
10
11 static_assert(squares[10] == 100);       // checked, costs nothing at runtime
```

The generated program contains the finished table and none of the code that built it.

Computing at compile time

```
consteval auto make_table()
{
    std::array<int, 16> table{};
    for (int i = 0; i < 16; ++i)
        table[i] = i * i;
    return table;
}

auto constexpr squares = make_table();   // the loop runs in the compiler

static_assert(squares[10] == 100);       // checked, costs nothing at runtime
```

The generated program contains the finished table and none of the code that built it.

C++23 also has `if consteval`, to branch on whether you run at compile time.

[[nodiscard]]

```
1 [[nodiscard]] std::optional<Entry> parse(std::string_view line);
2
3 parse(line);      // warning: ignoring return value of 'parse'
```

[[nodiscard]]

```
1 [[nodiscard]] std::optional<Entry> parse(std::string_view line);
2
3 parse(line);      // warning: ignoring return value of 'parse'
```

Worth adding whenever throwing the result away is almost certainly a mistake — which is to say, on most functions that return something.

[[nodiscard]]

```
1 [[nodiscard]] std::optional<Entry> parse(std::string_view line);
2
3 parse(line);      // warning: ignoring return value of 'parse'
```

Worth adding whenever throwing the result away is almost certainly a mistake — which is to say, on most functions that return something.

Other attributes you will meet:

- `[[maybe_unused]]` — "yes, I know it is unused". The exercise stubs use it.
- `[[fallthrough]]` — "yes, I meant to fall through".
- `[[deprecated]]` — warns at every call site.

A function without a name

```
1 auto is_error = [](Entry const& entry) { return entry.level == Level::Error; };
2
3 is_error(entry);     // call it like any other function
```

A function without a name

```
1 auto is_error = [](Entry const& entry) { return entry.level == Level::Error; };
2
3 is_error(entry);     // call it like any other function
```

The anatomy:

```
4 [capture](parameters) -> return_type { body }
```

- the return type is almost always deduced — leave it out,
- the parameter list may be empty,
- the capture list may not be omitted. That `[]` is what makes it a lambda.

A function without a name

```
1 auto is_error = [](Entry const& entry) { return entry.level == Level::Error; };
2
3 is_error(entry);     // call it like any other function
```

The anatomy:

```
4 [capture](parameters) -> return_type { body }
```

- the return type is almost always deduced — leave it out,
- the parameter list may be empty,
- the capture list may not be omitted. That `[]` is what makes it a lambda.

Each lambda has its own unnameable type, which is why you store one in `auto`.

Captures

A lambda can use variables from the enclosing scope — but you must say how.

```
1  int threshold = 10;
2
3  [threshold](int x) { return x > threshold; }   // by value: a copy
4  [&threshold](int x) { return x > threshold; }  // by reference: the original
```

Captures

A lambda can use variables from the enclosing scope — but you must say how.

```
1 int threshold = 10;
2
3 [threshold](int x) { return x > threshold; }   // by value: a copy
4 [&threshold](int x) { return x > threshold; }  // by reference: the original
```

The catch-all forms:

```
5 [=]   // capture everything used, by value
6 [&]   // capture everything used, by reference
```

Captures

A lambda can use variables from the enclosing scope — but you must say how.

```
int threshold = 10;

[threshold](int x) { return x > threshold; }   // by value: a copy
[&threshold](int x) { return x > threshold; }  // by reference: the original
```

The catch-all forms:

```
[=]   // capture everything used, by value
[&]   // capture everything used, by reference
```

Pitfall

`[&]` outliving its scope is a dangling reference

Fine for a lambda you hand straight to an algorithm. Dangerous for one you store and call later. Prefer naming what you capture.

Generic lambdas

Write `auto` for a parameter and the lambda works for any type that fits:

```
auto print_twice = [](auto const& value) {
    std::println("{} {}", value, value);
};

print_twice(42);        // 42 42
print_twice("hello");   // hello hello
```

Generic lambdas

Write `auto` for a parameter and the lambda works for any type that fits:

```
auto print_twice = [](auto const& value) {
    std::println("{} {}", value, value);
};

print_twice(42);        // 42 42
print_twice("hello");   // hello hello
```

This is a template in disguise — the compiler generates one version per type you call it with. We will meet the general machinery later. → Block 4

Generic lambdas

Write `auto` for a parameter and the lambda works for any type that fits:

```
auto print_twice = [](auto const& value) {
    std::println("{} {}", value, value);
};

print_twice(42);         // 42 42
print_twice("hello");    // hello hello
```

This is a template in disguise — the compiler generates one version per type you call it with. We will meet the general machinery later. Block 4

C++23 also lets a lambda be `constexpr`, `consteval`, `static`, and carry explicit template parameters. You will rarely need any of it.

Why lambdas exist

Because algorithms need to be told *what* to do:

```
1 #include <algorithm>
2
3 std::ranges::sort(entries);                    // sort by the default ordering
```

Why lambdas exist

Because algorithms need to be told *what* to do:

```
#include <algorithm>

std::ranges::sort(entries);                    // sort by the default ordering

auto errors = std::ranges::count_if(entries,
    [](Entry const& entry) { return entry.level == Level::Error; });
```

Why lambdas exist

Because algorithms need to be told *what* to do:

```
#include <algorithm>

std::ranges::sort(entries);                    // sort by the default ordering

auto errors = std::ranges::count_if(entries,
    [](Entry const& entry) { return entry.level == Level::Error; });

std::ranges::sort(entries,
    [](Entry const& a, Entry const& b) { return a.timestamp < b.timestamp; });
```

Why lambdas exist

Because algorithms need to be told *what* to do:

```
#include <algorithm>

std::ranges::sort(entries);                    // sort by the default ordering

auto errors = std::ranges::count_if(entries,
    [](Entry const& entry) { return entry.level == Level::Error; });

std::ranges::sort(entries,
    [](Entry const& a, Entry const& b) { return a.timestamp < b.timestamp; });
```

Before lambdas you wrote a named function, or a “function object” struct with `operator()`, ten lines away from where it was used.

Projections beat comparison lambdas

The ranges algorithms take an optional *projection*: what to look at before comparing.

```
// with a lambda
std::ranges::sort(entries,
    [](Entry const& a, Entry const& b) { return a.timestamp < b.timestamp; });

// with a projection
std::ranges::sort(entries, std::less{}, &Entry::timestamp);
```

Projections beat comparison lambdas

The ranges algorithms take an optional *projection*: what to look at before comparing.

```
1 // with a lambda
2 std::ranges::sort(entries,
3     [](Entry const& a, Entry const& b) { return a.timestamp < b.timestamp; });
4
5 // with a projection
6 std::ranges::sort(entries, std::less{}, &Entry::timestamp);
```

`&Entry::timestamp` is a pointer-to-member: “the `timestamp` field of whichever entry you are looking at”.

Projections beat comparison lambdas

The ranges algorithms take an optional *projection*: what to look at before comparing.

```
1  // with a lambda
2  std::ranges::sort(entries,
3      [](Entry const& a, Entry const& b) { return a.timestamp < b.timestamp; });
4
5  // with a projection
6  std::ranges::sort(entries, std::less{}, &Entry::timestamp);
```

`&Entry::timestamp` is a pointer-to-member: “the `timestamp` field of whichever entry you are looking at”.
Reverse the order by changing the comparator, not the lambda:

```
7  std::ranges::sort(entries, std::greater{}, &Entry::timestamp);   // newest first
```

Projections beat comparison lambdas

The ranges algorithms take an optional *projection*: what to look at before comparing.

```
// with a lambda
std::ranges::sort(entries,
    [](Entry const& a, Entry const& b) { return a.timestamp < b.timestamp; });

// with a projection
std::ranges::sort(entries, std::less{}, &Entry::timestamp);
```

`&Entry::timestamp` is a pointer-to-member: "the `timestamp` field of whichever entry you are looking at".
Reverse the order by changing the comparator, not the lambda:

```
std::ranges::sort(entries, std::greater{}, &Entry::timestamp);   // newest first
```

In the exercise

You will use exactly this in Exercise 5.

A few algorithms worth knowing

`ranges::sort`	order a range
`ranges::find`	first matching element
`ranges::count_if`	how many satisfy a predicate
`ranges::any_of`	is there at least one
`ranges::all_of`	do they all
`ranges::min/max`	extremes, with a projection
`ranges::unique`	collapse adjacent duplicates
`ranges::fold_left`	reduce to a single value (C++23)

A few algorithms worth knowing

`ranges::sort`	order a range
`ranges::find`	first matching element
`ranges::count_if`	how many satisfy a predicate
`ranges::any_of`	is there at least one
`ranges::all_of`	do they all
`ranges::min/max`	extremes, with a projection
`ranges::unique`	collapse adjacent duplicates
`ranges::fold_left`	reduce to a single value (C++23)

Best practice

A named algorithm says what you meant. A hand-written loop makes the reader work it out.

Aggregates

A `struct` with no private members, no constructors and no base classes is an *aggregate*, and you can initialise it field by field:

```
1  struct Entry {
2      int         timestamp{};
3      Level       level{Level::Info};
4      std::string source;
5      std::string message;
6  };

8  Entry entry{115, Level::Error, "disk", "giving up"};
```

Aggregates

A `struct` with no private members, no constructors and no base classes is an *aggregate*, and you can initialise it field by field:

```
struct Entry {
    int         timestamp{};
    Level       level{Level::Info};
    std::string source;
    std::string message;
};

Entry entry{115, Level::Error, "disk", "giving up"};
```

The `{}` after each member is a *default member initialiser*. Without it, `Entry e;` would leave `timestamp` holding whatever was on the stack — exactly as in C.

Aggregates

A `struct` with no private members, no constructors and no base classes is an *aggregate*, and you can initialise it field by field:

```
1  struct Entry {
2      int         timestamp{};
3      Level       level{Level::Info};
4      std::string source;
5      std::string message;
6  };

8  Entry entry{115, Level::Error, "disk", "giving up"};
```

The `{}` after each member is a *default member initialiser*. Without it, `Entry e;` would leave `timestamp` holding whatever was on the stack — exactly as in C.

`std::string` members need no initialiser: they start empty by construction.

Designated initialisers

Positional initialisation stops being readable at about three fields:

```
1 Entry entry{115, Level::Error, "disk", "giving up"};   // which is which?
```

positional

Designated initialisers

Positional initialisation stops being readable at about three fields:

```
1 Entry entry{115, Level::Error, "disk", "giving up"};   // which is which?
```

positional

So name them — C99 had this, and C++20 got it back:

```
2 Entry entry{
3     .timestamp = 115,
4     .level     = Level::Error,
5     .source    = "disk",
6     .message   = "giving up",
7 };
```

Designated initialisers

Positional initialisation stops being readable at about three fields:

```
1 Entry entry{115, Level::Error, "disk", "giving up"};   // which is which?
```

positional

So name them — C99 had this, and C++20 got it back:

```
2 Entry entry{
3     .timestamp = 115,
4     .level     = Level::Error,
5     .source    = "disk",
6     .message   = "giving up",
7 };
```

- Fields must appear in declaration order (stricter than C).
- Omitted fields fall back to their default member initialiser.

Designated initialisers

Positional initialisation stops being readable at about three fields:

```
1 Entry entry{115, Level::Error, "disk", "giving up"};   // which is which?
```

positional

So name them — C99 had this, and C++20 got it back:

```
2 Entry entry{
3     .timestamp = 115,
4     .level     = Level::Error,
5     .source    = "disk",
6     .message   = "giving up",
7 };
```

- Fields must appear in declaration order (stricter than C).
- Omitted fields fall back to their default member initialiser.

In the exercise

You will build the result of `parse` exactly like this.

Structured bindings

Take a struct apart in one line:

```
1 auto const [timestamp, level, source, message] = entry;
2
3 std::println("{} from {}", message, source);
```

Structured bindings

Take a struct apart in one line:

```
1 auto const [timestamp, level, source, message] = entry;
2
3 std::println("{} from {}", message, source);
```

The names are yours to choose; the order is the declaration order.

Structured bindings

Take a struct apart in one line:

```
1 auto const [timestamp, level, source, message] = entry;
2
3 std::println("{} from {}", message, source);
```

The names are yours to choose; the order is the declaration order.

Where it really earns its keep is iterating a map:

```
4 std::map<Level, int> counts;
5
6 for (auto const& [level, count] : counts)
7     std::println("{}: {}", level, count);
```

Structured bindings

Take a struct apart in one line:

```
1 auto const [timestamp, level, source, message] = entry;
2
3 std::println("{} from {}", message, source);
```

The names are yours to choose; the order is the declaration order.

Where it really earns its keep is iterating a map:

```
4 std::map<Level, int> counts;
5
6 for (auto const& [level, count] : counts)
7     std::println("{}: {}", level, count);
```

The alternative is `pair.first` and `pair.second`, which tell you nothing.

Fixed-size data

`std::array` is a C array that knows its own size and behaves like a value:

```
1 std::array<int, 4> values{1, 1, 2, 3};
2
3 values.size();              // 4 -- no sizeof arithmetic
4 for (auto v : values) { }   // iterable
5 auto copy = values;         // actually copies, unlike a C array
```

Fixed-size data

`std::array` is a C array that knows its own size and behaves like a value:

```
1 std::array<int, 4> values{1, 1, 2, 3};
2
3 values.size();              // 4 -- no sizeof arithmetic
4 for (auto v : values) { }   // iterable
5 auto copy = values;         // actually copies, unlike a C array
```

`std::pair` and `std::tuple` bundle values with no names at all:

```
6 std::pair<int, std::string> result{404, "not found"};
7 auto const [code, text] = result;
```

Fixed-size data

`std::array` is a C array that knows its own size and behaves like a value:

```
1 std::array<int, 4> values{1, 1, 2, 3};
2
3 values.size();              // 4 -- no sizeof arithmetic
4 for (auto v : values) { }   // iterable
5 auto copy = values;         // actually copies, unlike a C array
```

`std::pair` and `std::tuple` bundle values with no names at all:

```
6 std::pair<int, std::string> result{404, "not found"};
7 auto const [code, text] = result;
```

Best practice

If the members mean something, give them names — write a `struct`.

`std::optional` — a value, or nothing

How does a function say “there is no answer”?

- In C: a sentinel return, an out-parameter, or `NULL`.
- All three rely on the caller remembering to check.

`std::optional` — a value, or nothing

How does a function say “there is no answer”?

- In C: a sentinel return, an out-parameter, or `NULL`.
- All three rely on the caller remembering to check.

```
1 std::optional<Entry> parse(std::string_view line);
```

`std::optional` — a value, or nothing

How does a function say “there is no answer”?

- In C: a sentinel return, an out-parameter, or `NULL`.
- All three rely on the caller remembering to check.

```
std::optional<Entry> parse(std::string_view line);
```

```
if (auto entry = parse(line)) {        // did it work?
    use(*entry);                       // yes -- unwrap it
} else {
    ++skipped;
}
```

`std::optional` — a value, or nothing

How does a function say “there is no answer”?

- In C: a sentinel return, an out-parameter, or `NULL`.
- All three rely on the caller remembering to check.

```
std::optional<Entry> parse(std::string_view line);

if (auto entry = parse(line)) {        // did it work?
    use(*entry);                       // yes -- unwrap it
} else {
    ++skipped;
}
```

The type now carries the “might not be there” in the signature, where the compiler and the reader can both see it.

Working with `optional`

```
1  std::optional<Level> level = level_from_string(text);
2
3  level.has_value();      // bool -- or just: if (level)
4  *level;                 // the value. UB if empty -- check first
5  level.value_or(Level::Info);   // the value, or a fallback
```

Working with `optional`

```cpp
std::optional<Level> level = level_from_string(text);

level.has_value();      // bool -- or just: if (level)
*level;                 // the value. UB if empty -- check first
level.value_or(Level::Info);   // the value, or a fallback
```

Return “nothing” with `std::nullopt`:

```cpp
std::optional<Level> level_from_string(std::string_view name)
{
    if (name == "WARN") return Level::Warn;
    /* ... */
    return std::nullopt;
}
```

Working with `optional`

```cpp
std::optional<Level> level = level_from_string(text);

level.has_value();     // bool -- or just: if (level)
*level;                // the value. UB if empty -- check first
level.value_or(Level::Info);   // the value, or a fallback
```

Return “nothing” with `std::nullopt`:

```cpp
std::optional<Level> level_from_string(std::string_view name)
{
    if (name == "WARN") return Level::Warn;
    /* ... */
    return std::nullopt;
}
```

An `optional` holds the value inline — no allocation, no indirection. It is one `bool` bigger than the value itself.

`std::span` — a view of contiguous values

The C way to pass an array is a pointer and a length, related only by convention:

```
1 void emit(Entry const* entries, size_t count);   // two things, one idea
```

C-style

`std::span` — a view of contiguous values

The C way to pass an array is a pointer and a length, related only by convention:

```
1 void emit(Entry const* entries, size_t count);   // two things, one idea
2 void emit(std::span<Entry const> entries);       // one thing
```

C-style

`std::span` — a view of contiguous values

The C way to pass an array is a pointer and a length, related only by convention:

```
1 void emit(Entry const* entries, size_t count);   // two things, one idea
2 void emit(std::span<Entry const> entries);       // one thing
```

C-style

It accepts anything contiguous, without copying:

```
3 std::vector<Entry> entries = /* ... */;
4 std::array<Entry, 4> fixed = /* ... */;
5
6 emit(entries);        // fine
7 emit(fixed);          // fine
8 emit({ptr, count});   // still fine
```

`std::span` — a view of contiguous values

The C way to pass an array is a pointer and a length, related only by convention:

```
1 void emit(Entry const* entries, size_t count);   // two things, one idea
2 void emit(std::span<Entry const> entries);       // one thing
```

C-style

It accepts anything contiguous, without copying:

```
3 std::vector<Entry> entries = /* ... */;
4 std::array<Entry, 4> fixed = /* ... */;

6 emit(entries);         // fine
7 emit(fixed);           // fine
8 emit({ptr, count});    // still fine
```

Like `string_view`, it owns nothing: the data must outlive the span.

Owning versus viewing

Owns	Views	Holds
`std::string`	`std::string_view`	characters
`std::vector`	`std::span`	anything contiguous

Owning versus viewing

Owns	Views	Holds
`std::string`	`std::string_view`	characters
`std::vector`	`std::span`	anything contiguous

- An owner allocates, frees, copies and moves its data.
- A view is a pointer and a length. Copying one costs nothing.

Owning versus viewing

Owns	Views	Holds
`std::string`	`std::string_view`	characters
`std::vector`	`std::span`	anything contiguous

- An owner allocates, frees, copies and moves its data.
- A view is a pointer and a length. Copying one costs nothing.
- Take a *view* as a parameter when you only read.
- Return an *owner* when the caller needs to keep it.

Owning versus viewing

Owns	Views	Holds
`std::string`	`std::string_view`	characters
`std::vector`	`std::span`	anything contiguous

- An owner allocates, frees, copies and moves its data.
- A view is a pointer and a length. Copying one costs nothing.
- Take a *view* as a parameter when you only read.
- Return an *owner* when the caller needs to keep it.

Pitfall

A view outlives what it views at your peril

A view into something destroyed is a dangling pointer with better syntax.

Exercise 2 — `parse`

In `log.cc`, implement:

```
std::optional<Entry> parse(std::string_view line);
```

A line looks like this — timestamp, level, source, then the message:

```
115 ERROR disk giving up after 3 retries
```

Exercise 2 — `parse`

In `log.cc`, implement:

```
std::optional<Entry> parse(std::string_view line);
```

A line looks like this — timestamp, level, source, then the message:

```
115 ERROR disk giving up after 3 retries
```

Anything that does not fit that shape gives `std::nullopt`.

Exercise 2 — `parse`

In `log.cc`, implement:

```
1 std::optional<Entry> parse(std::string_view line);
```

A line looks like this — timestamp, level, source, then the message:

```
2 115 ERROR disk giving up after 3 retries
```

Anything that does not fit that shape gives `std::nullopt`.
Hints:

- `split(line, ' ')` — you wrote that already.
- `std::from_chars` turns the first field into an `int`.
- `level_from_string` is given, and returns an `optional<Level>`.
- Everything from the fourth field on is the message.
- Build the result with designated initialisers.

```
$ make test-parse && ./test-parse
```

std::format

`std::format` builds a string; `std::print` writes one; `std::println` adds a newline.

```
1 std::string s = std::format("{} items", n);
2 std::print("{} items\n", n);
3 std::println("{} items", n);
4 std::println(stderr, "cannot open {}", path);
```

std::format

`std::format` builds a string; `std::print` writes one; `std::println` adds a newline.

```
std::string s = std::format("{} items", n);
std::print("{} items\n", n);
std::println("{} items", n);
std::println(stderr, "cannot open {}", path);
```

Arguments fill the `{}` in order, or by index:

```
std::println("{0} then {1}, and {0} again", "a", "b");
// a then b, and a again
```

std::format

`std::format` builds a string; `std::print` writes one; `std::println` adds a newline.

```
std::string s = std::format("{} items", n);
std::print("{} items\n", n);
std::println("{} items", n);
std::println(stderr, "cannot open {}", path);
```

Arguments fill the `{}` in order, or by index:

```
std::println("{0} then {1}, and {0} again", "a", "b");
// a then b, and a again
```

A literal brace is doubled:

```
std::println("{{}}");      // prints {}
```

The format spec

After a colon comes the spec: `{:spec}`.

```
1 std::println("[{:5}]",    42);      // [   42]   width 5
2 std::println("[{:<5}]",   42);      // [42   ]   left
3 std::println("[{:>5}]",   42);      // [   42]   right
4 std::println("[{:^5}]",   42);      // [ 42  ]   centred
5 std::println("[{:*^7}]",  42);      // [**42***]  fill with *
```

The format spec

After a colon comes the spec: `{:spec}`.

```
std::println("[{:5}]",    42);      // [   42]   width 5
std::println("[{:<5}]",   42);      // [42   ]   left
std::println("[{:>5}]",   42);      // [   42]   right
std::println("[{:^5}]",   42);      // [ 42  ]   centred
std::println("[{:*^7}]",  42);      // [**42***]  fill with *

std::println("{:.2f}",  3.14159);   // 3.14
std::println("{:08.3f}", 3.14159);  // 0003.142
std::println("{:x} {:b} {:o}", 255, 5, 64);   // ff 101 100
```

The format spec

After a colon comes the spec: `{:spec}`.

```
1 std::println("[{:5}]",    42);      // [   42]   width 5
2 std::println("[{:<5}]",   42);      // [42   ]   left
3 std::println("[{:>5}]",   42);      // [   42]   right
4 std::println("[{:^5}]",   42);      // [ 42  ]   centred
5 std::println("[{:*^7}]",  42);      // [**42***]  fill with *

6 std::println("{:.2f}",  3.14159);   // 3.14
7 std::println("{:08.3f}", 3.14159);  // 0003.142
8 std::println("{:x} {:b} {:o}", 255, 5, 64);   // ff 101 100
```

Containers format themselves, which is a gift while debugging:

```
9  std::println("{}", std::vector{1, 2, 3});   // [1, 2, 3]
10 std::println("{}", counts);                 // {"disk": 2, "net": 3}
```

Teaching `format` about your type

`std::format` knows nothing about `Level`:

```
1 std::println("{}", Level::Warn);    // compile error, and rightly so
```

Teaching `format` about your type

`std::format` knows nothing about `Level`:

```
std::println("{}", Level::Warn);    // compile error, and rightly so
```

You teach it by specialising `std::formatter`:

```
template <>
struct std::formatter<Level> {
    constexpr auto parse(std::format_parse_context& ctx);
    auto format(Level level, auto& ctx) const;
};
```

Teaching `format` about your type

`std::format` knows nothing about `Level`:

```
1 std::println("{}", Level::Warn);    // compile error, and rightly so
```

You teach it by specialising `std::formatter`:

```
2 template <>
3 struct std::formatter<Level> {
4     constexpr auto parse(std::format_parse_context& ctx);
5     auto format(Level level, auto& ctx) const;
6 };
```

- `parse` reads the spec — the characters between `:` and `}`.
- `format` writes the output.

Teaching `format` about your type

`std::format` knows nothing about `Level`:

```
1 std::println("{}", Level::Warn);    // compile error, and rightly so
```

You teach it by specialising `std::formatter`:

```
2 template <>
3 struct std::formatter<Level> {
4     constexpr auto parse(std::format_parse_context& ctx);
5     auto format(Level level, auto& ctx) const;
6 };
```

- `parse` reads the spec — the characters between `:` and `}`.
- `format` writes the output.

Being a template, it lives in a header; templates come later. Block 4

The lazy way — inherit a formatter

Usually you want the spec grammar `std::string_view` already has, so inherit it:

```
template <>
struct std::formatter<Level> : std::formatter<std::string_view> {
    auto format(Level level, auto& ctx) const
    {
        return std::formatter<std::string_view>::format(to_string(level), ctx);
    }
};
```

The lazy way — inherit a formatter

Usually you want the spec grammar `std::string_view` already has, so inherit it:

```
template <>
struct std::formatter<Level> : std::formatter<std::string_view> {
    auto format(Level level, auto& ctx) const
    {
        return std::formatter<std::string_view>::format(to_string(level), ctx);
    }
};
```

We inherited `parse`, and with it alignment, width and fill — for free:

```
std::println("[{:<5}]", Level::Warn);    // [WARN ]
std::println("[{:>7}]", Level::Error);   // [  ERROR]
```

The full way — write `parse` yourself for `Entry`

When you want a spec of your own invention:

```
template <>
struct std::formatter<Entry> {
    bool longform = true;              // what parse() learned
    constexpr auto parse(std::format_parse_context& ctx)
    {
        auto it = ctx.begin();
        if (it != ctx.end() and (*it == 'l' or *it == 's')) {
            longform = (*it == 'l');
            ++it;
        }
        return it;                     // must point at the '}'
    }
    /* format() overleaf */
};
```

The full way — write `parse` yourself for `Entry`

When you want a spec of your own invention:

```
template <>
struct std::formatter<Entry> {
    bool longform = true;              // what parse() learned
    constexpr auto parse(std::format_parse_context& ctx)
    {
        auto it = ctx.begin();
        if (it != ctx.end() and (*it == 'l' or *it == 's')) {
            longform = (*it == 'l');
            ++it;
        }
        return it;                     // must point at the '}'
    }
    /* format() overleaf */
};
```

`parse` is `constexpr` — with a literal format string it runs while compiling.

...and then format

```
auto format(Entry const& entry, auto& ctx) const
{
    if (not longform)
        return std::format_to(ctx.out(), "{}", entry.message);

    return std::format_to(ctx.out(), "{:>6} [{:<5}] {}: {}",
                          entry.timestamp, entry.level,
                          entry.source, entry.message);
}
```

...and then `format`

```
1       auto format(Entry const& entry, auto& ctx) const
2       {
3           if (not longform)
4               return std::format_to(ctx.out(), "{}", entry.message);
5
6           return std::format_to(ctx.out(), "{:>6} [{:<5}] {}: {}",
7                                 entry.timestamp, entry.level,
8                                 entry.source, entry.message);
9       }
```

`std::format_to` writes into the output iterator `ctx.out()`. Note that `{:<5}` on `entry.level` uses the formatter from the previous slide — they compose.

...and then `format`

```
1       auto format(Entry const& entry, auto& ctx) const
2       {
3           if (not longform)
4               return std::format_to(ctx.out(), "{}", entry.message);
5
6           return std::format_to(ctx.out(), "{:>6} [{:<5}] {}: {}",
7                                 entry.timestamp, entry.level,
8                                 entry.source, entry.message);
9       }
```

`std::format_to` writes into the output iterator `ctx.out()`. Note that `{:<5}` on `entry.level` uses the formatter from the previous slide — they compose.

```
10  std::println("{:l}", entry);   //    115 [ERROR] disk: giving up
11  std::println("{:s}", entry);   // giving up
```

Exercise 3 — formatters

In `format.h`, make all three work:

```
std::println("[{:<5}]", Level::Warn);   // [WARN ]
std::println("{:l}", entry);            //    115 [ERROR] disk: giving up
std::println("{:s}", entry);            // giving up
```

Exercise 3 — formatters

In `format.h`, make all three work:

```
std::println("[{:<5}]", Level::Warn);   // [WARN ]
std::println("{:1}", entry);            //    115 [ERROR] disk: giving up
std::println("{:s}", entry);            // giving up
```

Hints:

- Inherit `formatter<std::string_view>` to get width and alignment for free; hand it `to_string(level)`.
- For `Entry`, write `parse` yourself: recognise `l` and `s`, remember which in a member, and read it back in `format`.
- `parse` must return an iterator to the `}`.
- Long form: `"{:>6} [{:<5}] {}: {}"` on timestamp, level, source, message.

```
$ make test-format && ./test-format
```

BLOCK 3 / 4

Ownership and classes

- Ownership and value semantics
- Classes

Exercise 4

The problem, in C

```c
Entry* make_entry(void) {
    Entry* e = malloc(sizeof(Entry));
    e->source = strdup("disk");
    return e;                    // who frees this? when?
}
```

The problem, in C

```c
Entry* make_entry(void) {
    Entry* e = malloc(sizeof(Entry));
    e->source = strdup("disk");
    return e;                    // who frees this? when?
}
```


Every allocation creates an obligation the compiler does not track: forget to free → leak; free twice → corruption; free and keep using → use-after-free.

The problem, in C

```c
Entry* make_entry(void) {
    Entry* e = malloc(sizeof(Entry));
    e->source = strdup("disk");
    return e;                    // who frees this? when?
}
```


Every allocation creates an obligation the compiler does not track: forget to free → leak; free twice → corruption; free and keep using → use-after-free.

Key idea

C++ does not make these mistakes harder to make — it makes them unnecessary.

RAII

Resource **A**cquisition **I**s **I**nitialisation: tie a resource's lifetime to an object's.

```
{
    std::vector<Entry> entries;        // allocates as it grows
    entries.push_back(entry);
    /* ... */
}                                      // destructor runs here. always.
```

RAII

Resource **A**cquisition **I**s **I**nitialisation: tie a resource's lifetime to an object's.

```
1 {
2     std::vector<Entry> entries;       // allocates as it grows
3     entries.push_back(entry);
4     /* ... */
5 }                                     // destructor runs here. always.
```

The destructor runs when the scope ends — whether you fall off the end, `return` early, `break`, or an exception unwinds through.

RAII

Resource **A**cquisition **I**s **I**nitialisation: tie a resource's lifetime to an object's.

```
{
    std::vector<Entry> entries;        // allocates as it grows
    entries.push_back(entry);
    /* ... */
}                                      // destructor runs here. always.
```

The destructor runs when the scope ends — whether you fall off the end, `return` early, `break`, or an exception unwinds through.

Key idea

This is the idea the rest of the language is built on. Files, sockets, locks and memory are all the same problem, and RAII is the answer to all of them.

Value semantics

A C++ object behaves like a value: copying it copies what it owns.

```
1 std::vector<int> a{1, 2, 3};
2 std::vector<int> b = a;       // b gets its own copy of the elements
3 b.push_back(4);               // a is still {1, 2, 3}
```

Value semantics

A C++ object behaves like a value: copying it copies what it owns.

```
1 std::vector<int> a{1, 2, 3};
2 std::vector<int> b = a;       // b gets its own copy of the elements
3 b.push_back(4);               // a is still {1, 2, 3}
```

Compare with Python, where `b = a` gives you a second name for one list, and with C, where copying a struct copies the pointers: two owners, one allocation.

Value semantics

A C++ object behaves like a value: copying it copies what it owns.

```
1 std::vector<int> a{1, 2, 3};
2 std::vector<int> b = a;       // b gets its own copy of the elements
3 b.push_back(4);               // a is still {1, 2, 3}
```

Compare with Python, where `b = a` gives you a second name for one list, and with C, where copying a struct copies the pointers: two owners, one allocation.

Copies are correct by default. The remaining question is whether they are *cheap* — which is what the next few slides are about.

Moving instead of copying

Sometimes the source is about to die anyway, and the copy is waste:

```
1 std::string message = build_message();      // why copy the buffer?
2 entries.push_back(std::move(message));      // steal it instead
```

Moving instead of copying

Sometimes the source is about to die anyway, and the copy is waste:

```
1 std::string message = build_message();      // why copy the buffer?
2 entries.push_back(std::move(message));      // steal it instead
```

A *move* transfers the internals — pointer, size, capacity — and leaves the source empty: a few assignments instead of an allocation and a copy.

Moving instead of copying

Sometimes the source is about to die anyway, and the copy is waste:

```
1 std::string message = build_message();      // why copy the buffer?
2 entries.push_back(std::move(message));      // steal it instead
```

A *move* transfers the internals — pointer, size, capacity — and leaves the source empty: a few assignments instead of an allocation and a copy.

`std::move` does not move anything. It is a cast that says “I am done with this, feel free to take it apart”.

Moving instead of copying

Sometimes the source is about to die anyway, and the copy is waste:

```
1 std::string message = build_message();      // why copy the buffer?
2 entries.push_back(std::move(message));      // steal it instead
```

A *move* transfers the internals — pointer, size, capacity — and leaves the source empty: a few assignments instead of an allocation and a copy.

`std::move` does not move anything. It is a cast that says “I am done with this, feel free to take it apart”.

Pitfall

A moved-from object is still alive

It is valid but unspecified — you may assign to it or destroy it, but do not read it expecting the old value. Unlike Rust, the compiler will not stop you.

`T&&` — the movable type

A cast to *what*, though? To `T&&`, an *rvalue reference*.

`T*`	a pointer	may be null, may be re-pointed
`T&`	a named object	bound once, never null
`T&&`	a temporary, or `std::move(x)`	yours to take apart

`T&&` — the movable type

A cast to *what*, though? To `T&&`, an *rvalue reference*.

`T*`	a pointer	may be null, may be re-pointed
`T&`	a named object	bound once, never null
`T&&`	a temporary, or `std::move(x)`	yours to take apart

All three are an address at runtime; what differs is what the compiler lets you do:

```
1 void take(Big const& b);   // binds to anything       -- must copy
2 void take(Big&& b);        // binds to movable values -- may steal
```

`T&&` — the movable type

A cast to *what*, though? To `T&&`, an *rvalue reference*.

`T*`	a pointer	may be null, may be re-pointed
`T&`	a named object	bound once, never null
`T&&`	a temporary, or `std::move(x)`	yours to take apart

All three are an address at runtime; what differs is what the compiler lets you do:

```
1 void take(Big const& b);   // binds to anything       -- must copy
2 void take(Big&& b);        // binds to movable values -- may steal
```

Choosing between those two overloads is the whole mechanism behind move semantics — and why a move constructor is spelled `Big(Big&& other)`.

The rule of zero

A class that owns nothing directly needs none of this written out:

```
struct Entry {
    int         timestamp{};
    Level       level{Level::Info};
    std::string source;
    std::string message;
};
```

The rule of zero

A class that owns nothing directly needs none of this written out:

```
struct Entry {
    int         timestamp{};
    Level       level{Level::Info};
    std::string source;
    std::string message;
};
```

`Entry` is copyable, movable, and destructible — correctly — and we wrote no constructor, no destructor, no assignment operator. Its members already know how to do all of it.

The rule of zero

A class that owns nothing directly needs none of this written out:

```
struct Entry {
    int         timestamp{};
    Level       level{Level::Info};
    std::string source;
    std::string message;
};
```

`Entry` is copyable, movable, and destructible — correctly — and we wrote no constructor, no destructor, no assignment operator. Its members already know how to do all of it.

Best practice

The rule of zero: write none of the five. Let members own their resources.

The rule of five

If you *do* write one of these, you almost certainly need all five:

destructor	`~Big();`
copy constructor	`Big(Big const& other);`
copy assignment	`Big& operator=(Big const& other);`
move constructor	`Big(Big&& other) noexcept;`
move assignment	`Big& operator=(Big&& other) noexcept;`

The rule of five

If you *do* write one of these, you almost certainly need all five:

destructor	`~Big();`
copy constructor	`Big(Big const& other);`
copy assignment	`Big& operator=(Big const& other);`
move constructor	`Big(Big&& other) noexcept;`
move assignment	`Big& operator=(Big&& other) noexcept;`

A custom destructor means you own something. Owning something means copying needs thought. Needing thought about copying means needing thought about moving.

The rule of five

If you *do* write one of these, you almost certainly need all five:

destructor	`~Big();`
copy constructor	`Big(Big const& other);`
copy assignment	`Big& operator=(Big const& other);`
move constructor	`Big(Big&& other) noexcept;`
move assignment	`Big& operator=(Big&& other) noexcept;`

A custom destructor means you own something. Owning something means copying needs thought. Needing thought about copying means needing thought about moving.

You can ask for the default, or forbid it outright:

```
1 Big(Big const&) = delete;      // non-copyable
2 Big(Big&&) noexcept = default; // but movable
```

unique_ptr

When you genuinely need a heap object, do not hold it by raw pointer:

```
#include <memory>

auto report = std::make_unique<Histogram>();
report->emit(entries);
// deleted automatically at the end of the scope
```

unique_ptr

When you genuinely need a heap object, do not hold it by raw pointer:

```
#include <memory>

auto report = std::make_unique<Histogram>();
report->emit(entries);
// deleted automatically at the end of the scope
```

`unique_ptr` is the single owner. It cannot be copied — that would mean two owners — but it can be moved:

```
std::vector<std::unique_ptr<Report>> reports;
reports.push_back(std::move(report));      // ownership handed over
```

unique_ptr

When you genuinely need a heap object, do not hold it by raw pointer:

```
#include <memory>

auto report = std::make_unique<Histogram>();
report->emit(entries);
// deleted automatically at the end of the scope
```

`unique_ptr` is the single owner. It cannot be copied — that would mean two owners — but it can be moved:

```
std::vector<std::unique_ptr<Report>> reports;
reports.push_back(std::move(report));      // ownership handed over
```

It costs exactly as much as a raw pointer. There is no reason to prefer `new` and `delete`.

Which pointer?

`T` (a value)	the default — no indirection at all
`std::unique_ptr<T>`	one owner, polymorphic or large
`std::shared_ptr<T>`	genuinely shared ownership
`T*` / `T&`	observing something you do not own

Which pointer?

`T` (a value)	the default — no indirection at all
`std::unique_ptr<T>`	one owner, polymorphic or large
`std::shared_ptr<T>`	genuinely shared ownership
`T*` / `T&`	observing something you do not own

`shared_ptr` keeps a reference count, which costs memory and atomics. Reach for it when you can prove you need it — most of the time “shared” turns out to mean “I had not decided who owns this”.

Which pointer?

`T` (a value)	the default — no indirection at all
`std::unique_ptr<T>`	one owner, polymorphic or large
`std::shared_ptr<T>`	genuinely shared ownership
`T*` / `T&`	observing something you do not own

`shared_ptr` keeps a reference count, which costs memory and atomics. Reach for it when you can prove you need it — most of the time “shared” turns out to mean “I had not decided who owns this”.

A raw pointer is still fine, as long as it only *observes*. A raw pointer that owns is the thing we are getting rid of.

Copy elision

Returning by value looks expensive and usually is not:

```
std::vector<Entry> load(std::string_view path) {
    std::vector<Entry> entries;
    /* fill it */
    return entries;      // no copy, no move -- built in place
}
```

Copy elision

Returning by value looks expensive and usually is not:

```
std::vector<Entry> load(std::string_view path) {
    std::vector<Entry> entries;
    /* fill it */
    return entries;      // no copy, no move -- built in place
}
```

The compiler builds `entries` directly in the caller's storage — since C++17 guaranteed, not merely permitted.

Copy elision

Returning by value looks expensive and usually is not:

```
std::vector<Entry> load(std::string_view path) {
    std::vector<Entry> entries;
    /* fill it */
    return entries;      // no copy, no move -- built in place
}
```

The compiler builds `entries` directly in the caller's storage — since C++17 guaranteed, not merely permitted.

Pitfall

Do not "help"

`return std::move(entries);` is slower — it forces a move where elision would have done nothing at all.

Dangling views

The one thing RAII does not protect you from:

```
1 std::string_view first_word(std::string_view line)
2 {
3     std::string copy{line};              // a local string
4     return split(copy, ' ').front();     // view into it
5 }                                        // copy dies here
```


dangling view

Dangling views

The one thing RAII does not protect you from:

```
1  std::string_view first_word(std::string_view line)
2  {
3      std::string copy{line};              // a local string
4      return split(copy, ' ').front();     // view into it
5  }                                        // copy dies here
```


dangling view

The returned view points at freed memory. This compiles without a murmur.

Dangling views

The one thing RAII does not protect you from:

```
std::string_view first_word(std::string_view line)
{
    std::string copy{line};              // a local string
    return split(copy, ' ').front();     // view into it
}                                        // copy dies here
```


dangling view

The returned view points at freed memory. This compiles without a murmur.

The rule: a view must not outlive what it views. In practice

- views are excellent *parameters*,
- views are dangerous *return values* and *members*,
- when in doubt, return an owning `std::string`.

Dangling views

The one thing RAII does not protect you from:

```
1 std::string_view first_word(std::string_view line)
2 {
3     std::string copy{line};              // a local string
4     return split(copy, ' ').front();     // view into it
5 }                                        // copy dies here
```


dangling view

The returned view points at freed memory. This compiles without a murmur.

The rule: a view must not outlive what it views. In practice

- views are excellent *parameters*,
- views are dangerous *return values* and *members*,
- when in doubt, return an owning `std::string`.

Build with `-fsanitize=address` and it will catch you.

`struct` and `class`

They are the same thing, with one difference:

- `struct` members are `public` by default,
- `class` members are `private` by default.

`struct` and `class`

They are the same thing, with one difference:

- `struct` members are `public` by default,
- `class` members are `private` by default.

```
1  class Counter {
2      int count = 0;              // private
3
4  public:
5      void bump() { ++count; }
6      int value() const { return count; }
7  };
```

`struct` and `class`

They are the same thing, with one difference:

- `struct` members are `public` by default,
- `class` members are `private` by default.

```
class Counter {
    int count = 0;              // private

public:
    void bump() { ++count; }
    int value() const { return count; }
};
```

Convention, not a rule: `struct` for plain data with no invariant to protect, `class` when the type has to keep something true about itself.

Constructors

```
1  class Entry {
2      int         timestamp;
3      std::string source;
4
5  public:
6      Entry(int ts, std::string src)
7          : timestamp{ts}, source{std::move(src)}    // member init list
8      { }
9  };
```

Constructors

```
1  class Entry {
2      int         timestamp;
3      std::string source;
4
5  public:
6      Entry(int ts, std::string src)
7          : timestamp{ts}, source{std::move(src)}    // member init list
8      { }
9  };
```

The list after the : *initialises* the members. Assigning in the body instead would default-construct them first and then overwrite them.

Constructors

```
1  class Entry {
2      int         timestamp;
3      std::string source;
4
5  public:
6      Entry(int ts, std::string src)
7          : timestamp{ts}, source{std::move(src)}    // member init list
8      { }
9  };
```

The list after the `:` *initialises* the members. Assigning in the body instead would default-construct them first and then overwrite them.

Members are initialised in *declaration* order, whatever order you write them in the list. `-Wall` warns when the two disagree.

explicit

A one-argument constructor doubles as an implicit conversion:

```
1 class Timeout { public: Timeout(int seconds); };
2 void wait(Timeout t);
3 wait(30);            // compiles. 30 seconds? milliseconds? a count?
```

explicit

A one-argument constructor doubles as an implicit conversion:

```
1 class Timeout { public: Timeout(int seconds); };
2 void wait(Timeout t);
3 wait(30);            // compiles. 30 seconds? milliseconds? a count?

4 class Timeout { public: explicit Timeout(int seconds); };
5 wait(30);            // now a compile error
6 wait(Timeout{30});   // say what you mean
```


implicit

explicit

A one-argument constructor doubles as an implicit conversion:

```
1 class Timeout { public: Timeout(int seconds); };
2 void wait(Timeout t);
3 wait(30);             // compiles. 30 seconds? milliseconds? a count?

4 class Timeout { public: explicit Timeout(int seconds); };
5 wait(30);             // now a compile error
6 wait(Timeout{30});    // say what you mean
```


Best practice

Mark single-argument constructors `explicit` unless the conversion is genuinely something you want.

`const` member functions

```
1  class Counter {
2      int count = 0;
3
4  public:
5      int  value() const { return count; }   // promises not to modify
6      void bump()        { ++count; }        // may modify
7  };
```

`const` member functions

```
1  class Counter {
2      int count = 0;
3
4  public:
5      int  value() const { return count; }   // promises not to modify
6      void bump()        { ++count; }        // may modify
7  };
```

The `const` after the parameter list applies to the object itself. Only `const` member functions may be called on a `const` object:

```
8  void show(Counter const& c) {
9      std::println("{}", c.value());   // fine
10     c.bump();                        // compile error
11 }
```

`const` member functions

```
1  class Counter {
2      int count = 0;
3
4  public:
5      int  value() const { return count; }   // promises not to modify
6      void bump()        { ++count; }        // may modify
7  };
```

The `const` after the parameter list applies to the object itself. Only `const` member functions may be called on a `const` object:

```
8  void show(Counter const& c) {
9      std::println("{}", c.value());   // fine
10     c.bump();                        // compile error
11 }
```

Mark everything `const` that can be. It is what makes `const&` parameters useful.

Operator overloading

Operators are functions with punctuation for names:

```
1  struct Vec2 {
2      double x{}, y{};
3
4      Vec2& operator+=(Vec2 const& o) { x += o.x; y += o.y; return *this; }
5  };
6
7  Vec2 operator+(Vec2 a, Vec2 const& b) { return a += b; }
```

Operator overloading

Operators are functions with punctuation for names:

```
1  struct Vec2 {
2      double x{}, y{};
3
4      Vec2& operator+=(Vec2 const& o) { x += o.x; y += o.y; return *this; }
5  };
6
7  Vec2 operator+(Vec2 a, Vec2 const& b) { return a += b; }
```

Pitfall

Only when the meaning is obvious

+ on a vector is clear. + on an `Entry` is a puzzle. If a reader has to look up what your operator does, it should have been a named function.

Comparison, for free

Ask for the defaults and the compiler writes member-by-member comparisons:

```
struct Entry {
    int         timestamp{};
    Level       level{};
    std::string source;

    bool operator==(Entry const&) const = default;
    auto operator<=>(Entry const&) const = default;
};
```

Comparison, for free

Ask for the defaults and the compiler writes member-by-member comparisons:

```
1  struct Entry {
2      int         timestamp{};
3      Level       level{};
4      std::string source;
5
6      bool operator==(Entry const&) const = default;
7      auto operator<=>(Entry const&) const = default;
8  };
```

<=> — the three-way or “spaceship” operator — yields less, equal or greater in one comparison. Declaring it gives you <, <=, > and >= all at once.

Comparison, for free

Ask for the defaults and the compiler writes member-by-member comparisons:

```
1  struct Entry {
2      int         timestamp{};
3      Level       level{};
4      std::string source;
5
6      bool operator==(Entry const&) const = default;
7      auto operator<=>(Entry const&) const = default;
8  };
```

<=> — the three-way or “spaceship” operator — yields less, equal or greater in one comparison. Declaring it gives you <, <=, > and >= all at once.

== is separate, because equality is often cheaper than ordering.

Inheritance

```
1 class Detailed : public Report {
2     /* ... */
3 };
```

A `Detailed` *is a* `Report`: it has everything `Report` has, and can be used wherever a `Report` is expected.

Inheritance

```
class Detailed : public Report {
    /* ... */
};
```

A `Detailed` *is a* `Report`: it has everything `Report` has, and can be used wherever a `Report` is expected.

The access specifier before the base restricts what the outside world sees of it. In practice you will write `public` every time; `private` inheritance means “implemented in terms of”, which composition says more clearly.

Inheritance

```
class Detailed : public Report {
    /* ... */
};
```

A `Detailed` *is a* `Report`: it has everything `Report` has, and can be used wherever a `Report` is expected.

The access specifier before the base restricts what the outside world sees of it. In practice you will write `public` every time; `private` inheritance means “implemented in terms of”, which composition says more clearly.

Best practice

Inheritance is not for code reuse; it says one type is substitutable for another.

Virtual functions

Without `virtual`, the compiler picks the function from the *static* type — what the variable is declared as:

```
1 struct Base    { void greet() { std::println("base"); } };
2 struct Derived : Base { void greet() { std::println("derived"); } };
3 Derived d;  Base& b = d;
4 b.greet();          // "base"  -- probably not what you wanted
```


non-virtual

Virtual functions

Without `virtual`, the compiler picks the function from the *static* type — what the variable is declared as:

```
1  struct Base    { void greet() { std::println("base"); } };
2  struct Derived : Base { void greet() { std::println("derived"); } };
3  Derived d;  Base& b = d;
4  b.greet();          // "base"  -- probably not what you wanted
```

non-virtual

```
5  struct Base    { virtual void greet() { std::println("base"); } };
6  struct Derived : Base { void greet() override { std::println("derived"); } };

8  b.greet();          // "derived"
```

Virtual functions

Without `virtual`, the compiler picks the function from the *static* type — what the variable is declared as:

```
1  struct Base    { void greet() { std::println("base"); } };
2  struct Derived : Base { void greet() { std::println("derived"); } };
3  Derived d;  Base& b = d;
4  b.greet();          // "base"  -- probably not what you wanted

5  struct Base    { virtual void greet() { std::println("base"); } };
6  struct Derived : Base { void greet() override { std::println("derived"); } };

8  b.greet();          // "derived"
```

non-virtual

`virtual` means “look up the real type at run time”. It costs one indirection through a table of function pointers.

Virtual dispatch needs indirection

`virtual` only takes effect when you reach the object *through* a reference or a pointer:

```
1 Derived d;
2 Base& ref =  d;   ref.greet();    // "derived" -- dynamic dispatch
3 Base* ptr = &d;   ptr->greet();   // "derived" -- dynamic dispatch
4 Base  val =  d;   val.greet();    // "base"    -- static dispatch
```

Virtual dispatch needs indirection

`virtual` only takes effect when you reach the object *through* a reference or a pointer:

```
1 Derived d;
2 Base& ref =  d;   ref.greet();    // "derived" -- dynamic dispatch
3 Base* ptr = &d;   ptr->greet();   // "derived" -- dynamic dispatch
4 Base  val =  d;   val.greet();    // "base"    -- static dispatch
```

A reference or pointer holds only an address — the object keeps its real type, and the lookup happens at run time.

Virtual dispatch needs indirection

`virtual` only takes effect when you reach the object *through* a reference or a pointer:

```
1 Derived d;
2 Base& ref =  d;   ref.greet();    // "derived" -- dynamic dispatch
3 Base* ptr = &d;   ptr->greet();   // "derived" -- dynamic dispatch
4 Base  val =  d;   val.greet();    // "base"    -- static dispatch
```

A reference or pointer holds only an address — the object keeps its real type, and the lookup happens at run time.

A `Base` variable *is* a `Base`: `sizeof(Base)` bytes, no room for what `Derived` added, nothing left to dispatch on.

Virtual dispatch needs indirection

`virtual` only takes effect when you reach the object *through* a reference or a pointer:

```
1 Derived d;
2 Base& ref =  d;   ref.greet();    // "derived" -- dynamic dispatch
3 Base* ptr = &d;   ptr->greet();   // "derived" -- dynamic dispatch
4 Base  val =  d;   val.greet();    // "base"    -- static dispatch
```

A reference or pointer holds only an address — the object keeps its real type, and the lookup happens at run time.
A `Base` variable *is* a `Base`: `sizeof(Base)` bytes, no room for what `Derived` added, nothing left to dispatch on.

Best practice

Take and store polymorphic objects by reference or `unique_ptr`, never by value.

override and final

```
1  struct Report {
2      virtual void emit(std::span<Entry const> entries) const;
3  };
4
5  struct Detailed : Report {
6      void emit(std::span<Entry const> entries) override;
7  };
```

`override` and `final`

```
1  struct Report {
2      virtual void emit(std::span<Entry const> entries) const;
3  };
4
5  struct Detailed : Report {
6      void emit(std::span<Entry const> entries) override;
7  };
```

`override` is optional and you should always write it. Get the signature subtly wrong — a missing `const`, the wrong parameter type — and without it you have silently defined a *new* function that never gets called:

```
8  void emit(std::span<Entry const> entries);          // not const! no error
9  void emit(std::span<Entry const> entries) override; // error, as it should be
```

`override` and `final`

```
1  struct Report {
2      virtual void emit(std::span<Entry const> entries) const;
3  };

5  struct Detailed : Report {
6      void emit(std::span<Entry const> entries) override;
7  };
```

`override` is optional and you should always write it. Get the signature subtly wrong — a missing `const`, the wrong parameter type — and without it you have silently defined a *new* function that never gets called:

```
8  void emit(std::span<Entry const> entries);          // not const! no error
9  void emit(std::span<Entry const> entries) override; // error, as it should be
```

`final` stops further overriding, on a function or a whole class.

Abstract classes

A pure virtual function has no body, and makes the class abstract:

```
1  class Report {
2  public:
3      virtual ~Report() = default;
4
5      virtual std::string_view name() const = 0;
6      virtual void emit(std::span<Entry const> entries) const = 0;
7  };
8
9  Report r;      // error: cannot instantiate an abstract class
```

Abstract classes

A pure virtual function has no body, and makes the class abstract:

```
class Report {
public:
    virtual ~Report() = default;

    virtual std::string_view name() const = 0;
    virtual void emit(std::span<Entry const> entries) const = 0;
};

Report r;      // error: cannot instantiate an abstract class
```

This is an interface: it says what every report must be able to do and nothing about how. Derived classes must supply every pure virtual function before they can be created.

The virtual destructor

```cpp
struct Report  { ~Report() = default; };          // NOT virtual
struct Detailed : Report { std::string buffer; };

std::unique_ptr<Report> r = std::make_unique<Detailed>();
// when r goes out of scope: ~Report() runs, ~Detailed() does not.
// buffer leaks.
```


non-virtual dtor

The virtual destructor

```
1  struct Report  { ~Report() = default; };          // NOT virtual
2  struct Detailed : Report { std::string buffer; };

4  std::unique_ptr<Report> r = std::make_unique<Detailed>();
5  // when r goes out of scope: ~Report() runs, ~Detailed() does not.
6  // buffer leaks.
```


non-virtual dtor

Deleting a derived object through a base pointer with a non-virtual destructor is undefined behaviour. In practice: the derived part is never destroyed.

The virtual destructor

```cpp
struct Report  { ~Report() = default; };          // NOT virtual
struct Detailed : Report { std::string buffer; };

std::unique_ptr<Report> r = std::make_unique<Detailed>();
// when r goes out of scope: ~Report() runs, ~Detailed() does not.
// buffer leaks.
```


non-virtual dtor

Deleting a derived object through a base pointer with a non-virtual destructor is undefined behaviour. In practice: the derived part is never destroyed.

Best practice

If a class has any virtual function, give it a `virtual` destructor.
`virtual ~Report() = default;` is enough.

Slicing

Copying a derived object into a base *value* has a name, and discards the derived part:

```
1 Detailed detailed;
2 Report copy = detailed;    // only the Report part is copied. The rest is
3                            // sliced off. copy.emit() calls Report::emit.
```

Slicing

Copying a derived object into a base *value* has a name, and discards the derived part:

```
Detailed detailed;
Report copy = detailed;    // only the Report part is copied. The rest is
                           // sliced off. copy.emit() calls Report::emit.
```

Which is why polymorphic objects are held as:

```
Report&                    reference;   // borrowed
std::unique_ptr<Report>    owned;       // owned
std::vector<std::unique_ptr<Report>> many;
```

Slicing

Copying a derived object into a base *value* has a name, and discards the derived part:

```
Detailed detailed;
Report copy = detailed;    // only the Report part is copied. The rest is
                           // sliced off. copy.emit() calls Report::emit.
```

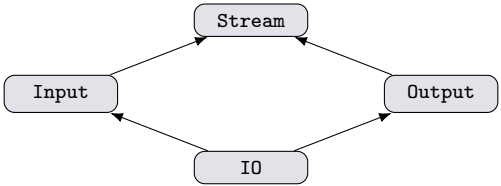
Which is why polymorphic objects are held as:

```
Report&                    reference;   // borrowed
std::unique_ptr<Report>    owned;       // owned
std::vector<std::unique_ptr<Report>> many;
```

...and never as `std::vector<Report>`, which cannot hold a `Detailed` at all.

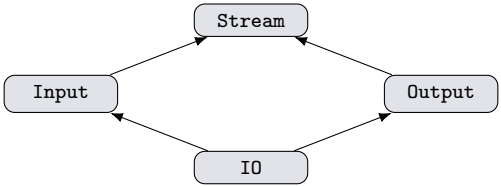
The diamond

Inherit from two classes that share a base, and you get two copies of it:



The diamond

Inherit from two classes that share a base, and you get two copies of it:




```
1 struct Stream { int position; };
2 struct Input  : Stream {};
3 struct Output : Stream {};
4 struct IO : Input, Output {};
5
6 IO io;
7 io.position;          // error: ambiguous -- there are two of them
```

Virtual base classes

`virtual` on the base says “share it”:

```
1 struct Input  : virtual Stream {};
2 struct Output : virtual Stream {};
3 struct IO : Input, Output {};
4
5 IO io;
6 io.position;          // fine -- there is exactly one Stream now
```

Virtual base classes

`virtual` on the base says “share it”:

```
1 struct Input  : virtual Stream {};
2 struct Output : virtual Stream {};
3 struct IO : Input, Output {};
4
5 IO io;
6 io.position;          // fine -- there is exactly one Stream now
```

The cost: the most-derived class becomes responsible for constructing the shared base, object layout needs a runtime offset, and casts get more expensive.

Virtual base classes

`virtual` on the base says “share it”:

```
1 struct Input  : virtual Stream {};
2 struct Output : virtual Stream {};
3 struct IO : Input, Output {};
4
5 IO io;
6 io.position;          // fine -- there is exactly one Stream now
```

The cost: the most-derived class becomes responsible for constructing the shared base, object layout needs a runtime offset, and casts get more expensive.

Best practice

Worth recognising when you meet it. Not worth designing towards — in new code, this is a sign the hierarchy wanted to be composition.

Prefer composition

Inheritance is the most rigid relationship the language offers: chosen at compile time, permanent, and it exposes the base's interface to everyone.

Prefer composition

Inheritance is the most rigid relationship the language offers: chosen at compile time, permanent, and it exposes the base's interface to everyone.

Before reaching for it, ask:

- Is this genuinely "is a", or just "has a"? $\rightarrow$ a member

Prefer composition

Inheritance is the most rigid relationship the language offers: chosen at compile time, permanent, and it exposes the base's interface to everyone.

Before reaching for it, ask:

- Is this genuinely "is a", or just "has a"? → a member
- Do I need runtime dispatch, or would a template do? Block 4

Prefer composition

Inheritance is the most rigid relationship the language offers: chosen at compile time, permanent, and it exposes the base's interface to everyone.

Before reaching for it, ask:

- Is this genuinely "is a", or just "has a"? → a member
- Do I need runtime dispatch, or would a template do? Block 4
- Is the set of alternatives closed and known? → `std::variant`

Prefer composition

Inheritance is the most rigid relationship the language offers: chosen at compile time, permanent, and it exposes the base's interface to everyone.

Before reaching for it, ask:

- Is this genuinely "is a", or just "has a"? → a member
- Do I need runtime dispatch, or would a template do? Block 4
- Is the set of alternatives closed and known? → `std::variant`
- Do I just need to store a callable? → a lambda

Prefer composition

Inheritance is the most rigid relationship the language offers: chosen at compile time, permanent, and it exposes the base's interface to everyone.

Before reaching for it, ask:

- Is this genuinely "is a", or just "has a"? → a member
- Do I need runtime dispatch, or would a template do? Block 4
- Is the set of alternatives closed and known? → `std::variant`
- Do I just need to store a callable? → a lambda

In the exercise

For our reports runtime dispatch is genuinely right: the set of report types is open, and which one runs is decided by a command-line flag.

Exercise 4 — reports

In `report.cc`, implement the two reports declared in `report.h`:

```
class Detailed final : public Report {  // one line per entry, using {:1}
class Histogram final : public Report { // count and a bar per level
```

Exercise 4 — reports

In `report.cc`, implement the two reports declared in `report.h`:

```
class Detailed final : public Report {  // one line per entry, using {:l}
class Histogram final : public Report { // count and a bar per level
```

Expected output for the sample entries:

```
-- detailed                -- histogram
   100 [INFO ] net: ...     INFO    2 ##
   105 [ERROR] disk: ...    WARN    1 #
   ...                      ERROR   2 ##
```

Exercise 4 — reports

In `report.cc`, implement the two reports declared in `report.h`:

```
class Detailed final : public Report {  // one line per entry, using {:l}
class Histogram final : public Report { // count and a bar per level
```

Expected output for the sample entries:

```
-- detailed                -- histogram
   100 [INFO ] net: ...     INFO    2 ##
   105 [ERROR] disk: ...    WARN    1 #
   ...                      ERROR   2 ##
```

Hints:

- `std::map<Level, int>` counts and sorts in one step.
- `for (auto const& [level, count] : counts)` reads well.
- `std::string(count, '#')` makes the bar; the line is `"{:<5} {:>3} {}"`.

```
$ make test-report && ./test-report
```

BLOCK 4 / 4

Generics and the library

- Templates and concepts
- Ranges and views
- Wrapping up

04

Exercises 5 and 6

The problem

The same function, three times:

```
1 int    max(int a, int b)       { return a > b ? a : b; }
2 double max(double a, double b) { return a > b ? a : b; }
3 std::string max(std::string a, std::string b) { /* ... */ }
```

repetitive

The problem

The same function, three times:

```
1 int    max(int a, int b)       { return a > b ? a : b; }
2 double max(double a, double b) { return a > b ? a : b; }
3 std::string max(std::string a, std::string b) { /* ... */ }
```

repetitive

In C you would reach for a macro, and lose type checking:

```
4 #define MAX(a, b) ((a) > (b) ? (a) : (b))
5 MAX(i++, j)        // evaluates i++ twice
```

C macro

The problem

The same function, three times:

```
1 int    max(int a, int b)       { return a > b ? a : b; }
2 double max(double a, double b) { return a > b ? a : b; }
3 std::string max(std::string a, std::string b) { /* ... */ }
```

repetitive

In C you would reach for a macro, and lose type checking:

```
4 #define MAX(a, b) ((a) > (b) ? (a) : (b))
5 MAX(i++, j)        // evaluates i++ twice
```

C macro

```
6 template <typename T>
7 T max(T a, T b) { return a > b ? a : b; }
```

The problem

The same function, three times:

```
1 int    max(int a, int b)       { return a > b ? a : b; }
2 double max(double a, double b) { return a > b ? a : b; }
3 std::string max(std::string a, std::string b) { /* ... */ }
```

repetitive

In C you would reach for a macro, and lose type checking:

```
4 #define MAX(a, b) ((a) > (b) ? (a) : (b))
5 MAX(i++, j)        // evaluates i++ twice
```

C macro

```
6 template <typename T>
7 T max(T a, T b) { return a > b ? a : b; }
```

One definition; one real, type-checked function per type you call it with.

Using a template

```
1 max(3, 7);              // T = int, deduced
2 max(3.5, 7.5);          // T = double, deduced
3 max<double>(3, 7.5);    // T said explicitly
4 max(3, 7.5);            // error: is T int or double?
```

Using a template

```
max(3, 7);              // T = int, deduced
max(3.5, 7.5);          // T = double, deduced
max<double>(3, 7.5);    // T said explicitly
max(3, 7.5);            // error: is T int or double?
```

Class templates work the same way:

```
template <typename T>
class Stack {
    std::vector<T> items;
public:
    void push(T item) { items.push_back(std::move(item)); }
};

Stack<int> numbers;
Stack      deduced{};    // C++17 can often deduce this too
```

Using a template

```
max(3, 7);              // T = int, deduced
max(3.5, 7.5);          // T = double, deduced
max<double>(3, 7.5);    // T said explicitly
max(3, 7.5);            // error: is T int or double?
```

Class templates work the same way:

```
template <typename T>
class Stack {
    std::vector<T> items;
public:
    void push(T item) { items.push_back(std::move(item)); }
};

Stack<int> numbers;
Stack      deduced{};    // C++17 can often deduce this too
```

You have used class templates all day: `vector<T>`, `optional<T>`, `span<T>`.

Why templates live in headers

The compiler cannot generate code for `max<int>` until it sees both the template *and* a call with `T = int`.

```
// stats.h
template <typename R>
Stats summarise(R&& entries) { /* the body must be here */ }
```

Why templates live in headers

The compiler cannot generate code for `max<int>` until it sees both the template *and* a call with `T = int`.

```
// stats.h
template <typename R>
Stats summarise(R&& entries) { /* the body must be here */ }
```

Put the body in a `.cc` file and every other translation unit gets a declaration it cannot instantiate — a link error, not a compile error, which is why it is confusing the first time.

Why templates live in headers

The compiler cannot generate code for `max<int>` until it sees both the template *and* a call with `T = int`.

```
// stats.h
template <typename R>
Stats summarise(R&& entries) { /* the body must be here */ }
```

Put the body in a `.cc` file and every other translation unit gets a declaration it cannot instantiate — a link error, not a compile error, which is why it is confusing the first time.

In the exercise

Templates go in headers. This is why `stats.h` has no `stats.cc`.

Abbreviated templates

`auto` as a parameter type quietly makes a template:

```
1 auto max(auto a, auto b) { return a > b ? a : b; }
```

...which is exactly

```
2 template <typename T, typename U>
3 auto max(T a, U b) { return a > b ? a : b; }
```

Abbreviated templates

`auto` as a parameter type quietly makes a template:

```
1 auto max(auto a, auto b) { return a > b ? a : b; }
```

...which is exactly

```
2 template <typename T, typename U>
3 auto max(T a, U b) { return a > b ? a : b; }
```

This is the same rule that makes a lambda with `auto` parameters generic.

Abbreviated templates

`auto` as a parameter type quietly makes a template:

```
1 auto max(auto a, auto b) { return a > b ? a : b; }
```

...which is exactly

```
2 template <typename T, typename U>
3 auto max(T a, U b) { return a > b ? a : b; }
```

This is the same rule that makes a lambda with `auto` parameters generic.

Short, and completely unconstrained: it accepts anything, and fails somewhere deep inside the body when the type does not fit. Which brings us to the point of this section.

The trouble with unconstrained templates

```
1 template <typename T>
2 T max(T a, T b) { return a > b ? a : b; }
3 struct Entry { /* no operator> */ };
4 max(e1, e2);
```


unconstrained

The trouble with unconstrained templates

```
1 template <typename T>
2 T max(T a, T b) { return a > b ? a : b; }
3 struct Entry { /* no operator> */ };
4 max(e1, e2);
```


unconstrained

Compiler error

```
main.cc:4:4:   required from 'T max(T, T) [with T = Entry]'
main.cc:2:28: error: no match for 'operator>'
                     (operand types are 'Entry' and 'Entry')
```

The trouble with unconstrained templates

```
1 template <typename T>
2 T max(T a, T b) { return a > b ? a : b; }
3 struct Entry { /* no operator> */ };
4 max(e1, e2);
```


unconstrained

Compiler error

```
main.cc:4:4:   required from 'T max(T, T) [with T = Entry]'
main.cc:2:28: error: no match for 'operator>'
                     (operand types are 'Entry' and 'Entry')
```

The `error:` lands on line 2, inside the template — not on line 4, where you made the mistake. Via `std::sort` the same slip gives 1373 lines.

The trouble with unconstrained templates

```
1 template <typename T>
2 T max(T a, T b) { return a > b ? a : b; }
3 struct Entry { /* no operator> */ };
4 max(e1, e2);
```


Compiler error

```
main.cc:4:4:   required from 'T max(T, T) [with T = Entry]'
main.cc:2:28: error: no match for 'operator>'
                     (operand types are 'Entry' and 'Entry')
```

The `error:` lands on line 2, inside the template — not on line 4, where you made the mistake. Via `std::sort` the same slip gives 1373 lines.

Key idea

A concept is that requirement, written down.

Concepts

```
1 template <typename T>
2 concept Ordered = requires(T const& a, T const& b) {
3     { a > b } -> std::convertible_to<bool>;
4 };
5 template <Ordered T>
6 T max(T a, T b) { return a > b ? a : b; }
```

Concepts

```
1 template <typename T>
2 concept Ordered = requires(T const& a, T const& b) {
3     { a > b } -> std::convertible_to<bool>;
4 };
5 template <Ordered T>
6 T max(T a, T b) { return a > b ? a : b; }
```

Now the error arrives at the call site and says what is wrong:

Compiler error

```
main.cc:9:31: error: no matching function for call to 'max(Entry&, Entry&)'
              constraints not satisfied
              the required expression '(a > b)' is invalid
```

Concepts

```
1  template <typename T>
2  concept Ordered = requires(T const& a, T const& b) {
3      { a > b } -> std::convertible_to<bool>;
4  };
5  template <Ordered T>
6  T max(T a, T b) { return a > b ? a : b; }
```

Now the error arrives at the call site and says what is wrong:

Compiler error

```
main.cc:9:31: error: no matching function for call to 'max(Entry&, Entry&)'
              constraints not satisfied
              the required expression '(a > b)' is invalid
```

A concept is a compile-time predicate on types — and, usefully, just a `bool`:

```
10 static_assert(Ordered<int>);   static_assert(not Ordered<Entry>);
```

Four ways to spell the same constraint

```
1  template <Ordered T>
2  T max(T a, T b);
3
4  template <typename T> requires Ordered<T>
5  T max(T a, T b);
6
7  template <typename T>
8  T max(T a, T b) requires Ordered<T>;
9
10 Ordered auto max(Ordered auto a, Ordered auto b);   // abbreviated
```

Four ways to spell the same constraint

```
1  template <Ordered T>
2  T max(T a, T b);
3
4  template <typename T> requires Ordered<T>
5  T max(T a, T b);
6
7  template <typename T>
8  T max(T a, T b) requires Ordered<T>;
9
10 Ordered auto max(Ordered auto a, Ordered auto b);   // abbreviated
```

All equivalent. The first and the last are the ones you will read most.

Four ways to spell the same constraint

```
1  template <Ordered T>
2  T max(T a, T b);

4  template <typename T> requires Ordered<T>
5  T max(T a, T b);

7  template <typename T>
8  T max(T a, T b) requires Ordered<T>;

10 Ordered auto max(Ordered auto a, Ordered auto b);   // abbreviated
```

All equivalent. The first and the last are the ones you will read most.

The abbreviated form is what `summarise` uses:

```
11 Stats summarise(EntryRange auto&& entries);
```

The standard concepts

Most of what you need already exists, in `<concepts>` and `<ranges>`:

`std::integral`, `std::floating_point`	number kinds
`std::same_as<T, U>`	exactly this type
`std::convertible_to<T, U>`	converts to
`std::derived_from<T, B>`	inherits from
`std::invocable<F, Args...>`	callable with
`std::ranges::input_range<R>`	can be iterated
`std::ranges::range_value_t<R>`	what it yields

The standard concepts

Most of what you need already exists, in `<concepts>` and `<ranges>`:

`std::integral`, `std::floating_point`	number kinds
`std::same_as<T, U>`	exactly this type
`std::convertible_to<T, U>`	converts to
`std::derived_from<T, B>`	inherits from
`std::invocable<F, Args...>`	callable with
`std::ranges::input_range<R>`	can be iterated
`std::ranges::range_value_t<R>`	what it yields

Compose them with `and`, `or` and `not`:

```
1 template <typename T>
2 concept Numeric = std::integral<T> or std::floating_point<T>;
```

Constraining on a range

The concept behind Exercise 6:

```
template <typename R>
concept EntryRange =
    std::ranges::input_range<R> and
    std::same_as<std::remove_cvref_t<std::ranges::range_value_t<R>>, Entry>;
```

Constraining on a range

The concept behind Exercise 6:

```
template <typename R>
concept EntryRange =
    std::ranges::input_range<R> and
    std::same_as<std::remove_cvref_t<std::ranges::range_value_t<R>>, Entry>;
```

`remove_cvref_t` strips `const` and the reference. Without it, iterating a `std::vector<Entry> const&` yields `Entry const&`, which is not `same_as<Entry>`, and the concept rejects a perfectly good argument.

Constraining on a range

The concept behind Exercise 6:

```
template <typename R>
concept EntryRange =
    std::ranges::input_range<R> and
    std::same_as<std::remove_cvref_t<std::ranges::range_value_t<R>>, Entry>;
```

`remove_cvref_t` strips `const` and the reference. Without it, iterating a `std::vector<Entry> const&` yields `Entry const&`, which is not `same_as<Entry>`, and the concept rejects a perfectly good argument.

Constraining on the *range* rather than on `std::vector<Entry>` is the whole point:

```
summarise(entries);                                  // a vector
summarise(entries | std::views::filter(is_error));   // a lazy view, no copy
```

Ranges

A *range* is anything with a beginning and an end. Every container is one.

```
1 std::sort(entries.begin(), entries.end());   // the old way
2 std::ranges::sort(entries);                  // the same thing
```


pre-C++20

Ranges

A *range* is anything with a beginning and an end. Every container is one.

```
1 std::sort(entries.begin(), entries.end());   // the old way
2 std::ranges::sort(entries);                  // the same thing
```

pre-C++20

Beyond saving typing, this removes a real class of bug:

```
3 std::sort(a.begin(), b.end());    // compiles. undefined behaviour.
```

mismatched iterators

Ranges

A *range* is anything with a beginning and an end. Every container is one.

```
1 std::sort(entries.begin(), entries.end());   // the old way
2 std::ranges::sort(entries);                  // the same thing
```

pre-C++20

Beyond saving typing, this removes a real class of bug:

```
3 std::sort(a.begin(), b.end());    // compiles. undefined behaviour.
```

mismatched iterators

Best practice

Every algorithm you know has a `std::ranges::` version that takes the range itself, plus an optional projection. Prefer them.

Views

A *view* is a range that computes its elements on demand, and owns nothing:

```
1 auto errors = entries | std::views::filter(is_error);
```

Views

A *view* is a range that computes its elements on demand, and owns nothing:

```
1 auto errors = entries | std::views::filter(is_error);
```

Nothing has happened yet. No memory has been allocated, no entry has been examined. The work happens when you iterate:

```
2 for (auto const& entry : errors)     // filtering happens here, one at a time
3     std::println("{:l}", entry);
```

Views

A *view* is a range that computes its elements on demand, and owns nothing:

```
auto errors = entries | std::views::filter(is_error);
```

Nothing has happened yet. No memory has been allocated, no entry has been examined. The work happens when you iterate:

```
for (auto const& entry : errors)      // filtering happens here, one at a time
    std::println("{:l}", entry);
```

Key idea

This is the same idea as a Python generator — and, like a generator, it produces each element exactly once, as it is asked for.

Composing views

Views chain with |, left to right:

```
auto recent_errors = entries
                   | std::views::filter(is_error)
                   | std::views::take(5)
                   | std::views::transform(&Entry::source);
```

Composing views

Views chain with `|`, left to right:

```
auto recent_errors = entries
                   | std::views::filter(is_error)
                   | std::views::take(5)
                   | std::views::transform(&Entry::source);
```

Read it as a pipeline: take the entries, keep the errors, keep the first five of those, then look at each one's source.

Composing views

Views chain with `|`, left to right:

```
auto recent_errors = entries
                   | std::views::filter(is_error)
                   | std::views::take(5)
                   | std::views::transform(&Entry::source);
```

Read it as a pipeline: take the entries, keep the errors, keep the first five of those, then look at each one's source.

Still nothing has run. And when it does, `filter` stops as soon as `take` has its five — the rest of the entries are never examined.

From a view back to a container

A view is not a container. When you need one, ask:

```
auto sources = entries
             | std::views::transform(&Entry::source)
             | std::ranges::to<std::vector>();      // C++23
```

From a view back to a container

A view is not a container. When you need one, ask:

```
auto sources = entries
             | std::views::transform(&Entry::source)
             | std::ranges::to<std::vector>();      // C++23
```

This is where the work happens and the memory is allocated — once, at the end, instead of once per pipeline stage.

From a view back to a container

A view is not a container. When you need one, ask:

```
auto sources = entries
             | std::views::transform(&Entry::source)
             | std::ranges::to<std::vector>();      // C++23
```

This is where the work happens and the memory is allocated — once, at the end, instead of once per pipeline stage.

Pitfall

A `const` view is often not a range

`filter_view` caches where its first element is, so `begin()` is not `const`. `auto const v = entries | filter(...)` will not compile — drop the `const` on the view itself.

The views worth knowing

`views::filter`	keep what matches a predicate	
`views::transform`	apply a function to each element	
`views::take(n)`	the first `n`	
`views::drop(n)`	all but the first `n`	
`views::reverse`	backwards	
`views::split(d)`	cut a range at a delimiter	
`views::join`	flatten a range of ranges	
`views::enumerate`	pairs of (index, element)	C++23
`views::zip`	walk several ranges in step	C++23

The views worth knowing

`views::filter`	keep what matches a predicate	
`views::transform`	apply a function to each element	
`views::take(n)`	the first `n`	
`views::drop(n)`	all but the first `n`	
`views::reverse`	backwards	
`views::split(d)`	cut a range at a delimiter	
`views::join`	flatten a range of ranges	
`views::enumerate`	pairs of (index, element)	C++23
`views::zip`	walk several ranges in step	C++23

```
1 for (auto const& [i, entry] : std::views::enumerate(entries))
2     std::println("{:3}: {:1}", i, entry);
```

`std::string`, briefly

```cpp
std::string line = "115 ERROR disk giving up";

line.size();  line.empty();
line.starts_with("115");        // C++20
line.contains("ERROR");         // C++23
line.find(' ');                 // index, or std::string::npos
line.substr(4, 5);              // "ERROR"
```

`std::string`, briefly

```
1  std::string line = "115 ERROR disk giving up";
2
3  line.size();  line.empty();
4  line.starts_with("115");          // C++20
5  line.contains("ERROR");           // C++23
6  line.find(' ');                   // index, or std::string::npos
7  line.substr(4, 5);                // "ERROR"
```

`std::string_view` has almost the same interface, minus anything that would modify or allocate. Write your helpers against `string_view` and they work on both.

`std::string`, briefly

```cpp
std::string line = "115 ERROR disk giving up";

line.size();  line.empty();
line.starts_with("115");        // C++20
line.contains("ERROR");         // C++23
line.find(' ');                 // index, or std::string::npos
line.substr(4, 5);              // "ERROR"
```

`std::string_view` has almost the same interface, minus anything that would modify or allocate. Write your helpers against `string_view` and they work on both.

Converting a number:

```cpp
std::from_chars(first, last, value);   // fast, no allocation, no locale
std::stoi(text);                       // convenient, throws on failure
std::format("{}", value);              // the other direction
```

`std::variant` — one of several types

An `enum class` says which alternative. A `variant` carries the data too:

```
1  using Value = std::variant<int, double, std::string>;
2
3  Value v = 42;
4  v = "now a string";          // legal: the alternative changed
```

`std::variant` — one of several types

An `enum class` says which alternative. A `variant` carries the data too:

```
using Value = std::variant<int, double, std::string>;

Value v = 42;
v = "now a string";         // legal: the alternative changed
```

Unlike a `union`, it remembers which alternative is active and destroys it properly. Reading the wrong one is an error, not corruption:

```
std::get<int>(v);           // throws if v is not currently an int
std::holds_alternative<int>(v);
```

`std::variant` — one of several types

An `enum class` says which alternative. A `variant` carries the data too:

```
1 using Value = std::variant<int, double, std::string>;
2
3 Value v = 42;
4 v = "now a string";          // legal: the alternative changed
```

Unlike a `union`, it remembers which alternative is active and destroys it properly. Reading the wrong one is an error, not corruption:

```
5 std::get<int>(v);            // throws if v is not currently an int
6 std::holds_alternative<int>(v);
```

Use it for a *closed* set of alternatives — one you control and do not expect to grow. For an open set, use virtual functions.

std::visit

Handle every alternative, checked at compile time:

```
template <typename... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

std::visit(overloaded{
    [](int i)                { std::println("int {}", i); },
    [](double d)             { std::println("double {}", d); },
    [](std::string const& s) { std::println("string {}", s); },
}, value);
```

std::visit

Handle every alternative, checked at compile time:

```
template <typename... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

std::visit(overloaded{
    [](int i)                { std::println("int {}", i); },
    [](double d)             { std::println("double {}", d); },
    [](std::string const& s) { std::println("string {}", s); },
}, value);
```

Miss an alternative and it does not compile — the same guarantee a `switch` over an `enum class` gives you, for types.

std::visit

Handle every alternative, checked at compile time:

```
template <typename... Ts>
struct overloaded : Ts... { using Ts::operator()...; };

std::visit(overloaded{
    [](int i)                { std::println("int {}", i); },
    [](double d)             { std::println("double {}", d); },
    [](std::string const& s) { std::println("string {}", s); },
}, value);
```

Miss an alternative and it does not compile — the same guarantee a `switch` over an `enum class` gives you, for types.

That `overloaded` helper is three lines of pure template machinery that everyone copies and nobody writes twice.

Exercise 5 — queries

In `query.cc`:

```
1 std::vector<Entry> at_least(std::span<Entry const> entries, Level min);
2 std::vector<Entry> most_recent(std::span<Entry const> entries, std::size_t n);
3 std::vector<std::string> sources(std::span<Entry const> entries);
```

Exercise 5 — queries

In `query.cc`:

```
std::vector<Entry> at_least(std::span<Entry const> entries, Level min);
std::vector<Entry> most_recent(std::span<Entry const> entries, std::size_t n);
std::vector<std::string> sources(std::span<Entry const> entries);
```

- `at_least` — entries at or above `min`, original order preserved.
- `most_recent` — the `n` newest, newest first. Fewer if there are not `n`.
- `sources` — each distinct source once, alphabetically.

Exercise 5 — queries

In `query.cc`:

```
std::vector<Entry> at_least(std::span<Entry const> entries, Level min);
std::vector<Entry> most_recent(std::span<Entry const> entries, std::size_t n);
std::vector<std::string> sources(std::span<Entry const> entries);
```

- `at_least` — entries at or above `min`, original order preserved.
- `most_recent` — the `n` newest, newest first. Fewer if there are not `n`.
- `sources` — each distinct source once, alphabetically.

Hints:

- `views::filter`, then `ranges::to<std::vector>()`.
- `ranges::sort` with `std::greater{}` and the projection `&Entry::timestamp`.
- `ranges::unique` plus `erase` to drop the duplicates.

```
$ make test-query && ./test-query
```

Exercise 6 — statistics

In `stats.h`:

```
1 template <typename R>
2 concept EntryRange = true;    // TODO -- accepts anything right now
3 Stats summarise(EntryRange auto&& entries);
```

Exercise 6 — statistics

In `stats.h`:

```
template <typename R>
concept EntryRange = true;    // TODO -- accepts anything right now
Stats summarise(EntryRange auto&& entries);
```

- `EntryRange` — tighten it so it accepts only ranges of `Entry`.
- `summarise` — count the entries and the errors, and give the error rate as a percentage (0 for an empty range).

Exercise 6 — statistics

In `stats.h`:

```
template <typename R>
concept EntryRange = true;    // TODO -- accepts anything right now
Stats summarise(EntryRange auto&& entries);
```

- `EntryRange` — tighten it so it accepts only ranges of `Entry`.
- `summarise` — count the entries and the errors, and give the error rate as a percentage (0 for an empty range).

The test checks that your concept *rejects* `std::vector<std::string>` and `int`, and accepts both `std::vector<Entry>` and a lazy view.

```
$ make test-stats && ./test-stats
```

Exercise 6 — statistics

In `stats.h`:

```
template <typename R>
concept EntryRange = true;    // TODO -- accepts anything right now
Stats summarise(EntryRange auto&& entries);
```

- `EntryRange` — tighten it so it accepts only ranges of `Entry`.
- `summarise` — count the entries and the errors, and give the error rate as a percentage (0 for an empty range).

The test checks that your concept *rejects* `std::vector<std::string>` and `int`, and accepts both `std::vector<Entry>` and a lazy view.

```
$ make test-stats && ./test-stats
```

Then run the finished tool:

```
$ make analyse && ./analyse data/sample.log
```

The short version

- Let objects own their resources. Write no `delete`.

The short version

- Let objects own their resources. Write no `delete`.
- Default to `const`, to `auto const&` parameters, and to values.

The short version

- Let objects own their resources. Write no `delete`.
- Default to `const`, to `auto const&` parameters, and to values.
- Say what you mean in the type: `optional` for maybe, `enum class` for a choice, `span` and `string_view` for borrowed data.

The short version

- Let objects own their resources. Write no `delete`.
- Default to `const`, to `auto const&` parameters, and to values.
- Say what you mean in the type: `optional` for maybe, `enum class` for a choice, `span` and `string_view` for borrowed data.
- Name the operation. An algorithm beats a loop; a projection beats a comparison lambda.

The short version

- Let objects own their resources. Write no `delete`.
- Default to `const`, to `auto const&` parameters, and to values.
- Say what you mean in the type: `optional` for maybe, `enum class` for a choice, `span` and `string_view` for borrowed data.
- Name the operation. An algorithm beats a loop; a projection beats a comparison lambda.
- Constrain your templates. A concept turns a page of errors into one line.

The short version

- Let objects own their resources. Write no `delete`.
- Default to `const`, to `auto const&` parameters, and to values.
- Say what you mean in the type: `optional` for maybe, `enum class` for a choice, `span` and `string_view` for borrowed data.
- Name the operation. An algorithm beats a loop; a projection beats a comparison lambda.
- Constrain your templates. A concept turns a page of errors into one line.
- Inheritance last, and only for genuine substitutability.

The short version

- Let objects own their resources. Write no `delete`.
- Default to `const`, to `auto const&` parameters, and to values.
- Say what you mean in the type: `optional` for maybe, `enum class` for a choice, `span` and `string_view` for borrowed data.
- Name the operation. An algorithm beats a loop; a projection beats a comparison lambda.
- Constrain your templates. A concept turns a page of errors into one line.
- Inheritance last, and only for genuine substitutability.

Key idea

Much of modern C++ is making the compiler check what used to be your job.

What we skipped

- **Concurrency** — `std::thread`, `std::atomic`, mutexes, `std::async`, coroutines.
  → the concurrency lecture

What we skipped

- **Concurrency** — `std::thread`, `std::atomic`, mutexes, `std::async`, coroutines.
  → the concurrency lecture
- **Exceptions** — `throw`, `try`/`catch`, `noexcept`, exception safety. RAII is what makes them work.

What we skipped

- **Concurrency** — `std::thread`, `std::atomic`, mutexes, `std::async`, coroutines.
  → the concurrency lecture
- **Exceptions** — `throw`, `try`/`catch`, `noexcept`, exception safety. RAII is what makes them work.
- `std::expected` — like `optional`, but it carries a reason for the failure. C++23, and worth an afternoon.

What we skipped

- **Concurrency** — `std::thread`, `std::atomic`, mutexes, `std::async`, coroutines.
  → the concurrency lecture
- **Exceptions** — `throw`, `try`/`catch`, `noexcept`, exception safety. RAII is what makes them work.
- `std::expected` — like `optional`, but it carries a reason for the failure. C++23, and worth an afternoon.
- **Template metaprogramming** — variadic templates, `if constexpr`, type traits, CRTP.

What we skipped

- **Concurrency** — `std::thread`, `std::atomic`, mutexes, `std::async`, coroutines.
  → the concurrency lecture
- **Exceptions** — `throw`, `try`/`catch`, `noexcept`, exception safety. RAII is what makes them work.
- `std::expected` — like `optional`, but it carries a reason for the failure. C++23, and worth an afternoon.
- **Template metaprogramming** — variadic templates, `if constexpr`, type traits, CRTP.
- **Modules** at scale, and the build systems that support them.

What we skipped

- **Concurrency** — `std::thread`, `std::atomic`, mutexes, `std::async`, coroutines.
  → the concurrency lecture
- **Exceptions** — `throw`, `try`/`catch`, `noexcept`, exception safety. RAII is what makes them work.
- `std::expected` — like `optional`, but it carries a reason for the failure. C++23, and worth an afternoon.
- **Template metaprogramming** — variadic templates, `if constexpr`, type traits, CRTP.
- **Modules** at scale, and the build systems that support them.
- Most of the standard library. `<chrono>`, `<filesystem>` and `<random>` alone would fill another day.

Where to look next

- **cppreference.com** — the reference. Not a tutorial, but once you can read it you will not need much else.
- **A Tour of C++**, Bjarne Stroustrup — the whole language in a couple of hundred pages.
- **C++ Core Guidelines** — “what should I do here”, with reasons.
- **Compiler Explorer** (godbolt.org) — try something, see the assembly, across every compiler at once.

Where to look next

- **cppreference.com** — the reference. Not a tutorial, but once you can read it you will not need much else.
- **A Tour of C++**, Bjarne Stroustrup — the whole language in a couple of hundred pages.
- **C++ Core Guidelines** — “what should I do here”, with reasons.
- **Compiler Explorer** (godbolt.org) — try something, see the assembly, across every compiler at once.

And when the compiler shouts at you: read the *first* error, fix that one, recompile. The rest are usually its children.

Questions?

`make check` should be all green by now.

Useful compiler flags

```
-std=c++23              the standard. never omit it
-Wall -Wextra           the warnings worth having
-Wpedantic              complain about non-standard extensions
-g                      debug information
-O2                     optimise -- and enable more warnings
-fsanitize=address      catch use-after-free, overflow, leaks
-fsanitize=undefined    catch UB the compiler can see at runtime
```

Useful compiler flags

```
-std=c++23              the standard. never omit it
-Wall -Wextra           the warnings worth having
-Wpedantic              complain about non-standard extensions
-g                      debug information
-O2                     optimise -- and enable more warnings
-fsanitize=address      catch use-after-free, overflow, leaks
-fsanitize=undefined    catch UB the compiler can see at runtime
```

The sanitizers slow the program down by a few times and are worth running over every test you have. Most of the bugs in this lecture's "don't do that" slides are caught instantly by `-fsanitize=address`.

Reading a template error

1. Read the *first* error only.
2. Find the line in *your* code — ignore the standard library frames.
3. Look for “required from here” and “constraints not satisfied”.
4. With GCC, add `-fconcepts-diagnostics-depth=2` for the real reason.

Reading a template error

1. Read the *first* error only.
2. Find the line in *your* code — ignore the standard library frames.
3. Look for “required from here” and “constraints not satisfied”.
4. With GCC, add `-fconcepts-diagnostics-depth=2` for the real reason.

```
error: no matching function for call to 'summarise(vector<string>&)'
  note: candidate: 'Stats summarise(auto:1&&) requires EntryRange<auto:1>'
  note: constraints not satisfied
  note:   'std::same_as<std::string, Entry>' evaluated to false
```

Reading a template error

1. Read the *first* error only.
2. Find the line in *your* code — ignore the standard library frames.
3. Look for “required from here” and “constraints not satisfied”.
4. With GCC, add `-fconcepts-diagnostics-depth=2` for the real reason.

```
error: no matching function for call to 'summarise(vector<string>&)'
  note: candidate: 'Stats summarise(auto:1&&) requires EntryRange<auto:1>'
  note: constraints not satisfied
  note:   'std::same_as<std::string, Entry>' evaluated to false
```

The last line is the answer: it wanted `Entry`, it got `std::string`.

`std::expected`, in one slide

`optional` tells you that it failed. `expected` tells you why:

```
enum class ParseError { TooFewFields, BadTimestamp, BadLevel };

std::expected<Entry, ParseError> parse(std::string_view line) {
    if (parts.size() < 4)
        return std::unexpected(ParseError::TooFewFields);
    /* ... */
}
```

`std::expected`, in one slide

`optional` tells you that it failed. `expected` tells you why:

```
enum class ParseError { TooFewFields, BadTimestamp, BadLevel };

std::expected<Entry, ParseError> parse(std::string_view line) {
    if (parts.size() < 4)
        return std::unexpected(ParseError::TooFewFields);
    /* ... */
}
```

```
if (auto entry = parse(line))
    entries.push_back(*std::move(entry));
else
    std::println(stderr, "line {}: {}", n, entry.error());
```

`std::expected`, in one slide

`optional` tells you that it failed. `expected` tells you why:

```
enum class ParseError { TooFewFields, BadTimestamp, BadLevel };

std::expected<Entry, ParseError> parse(std::string_view line) {
    if (parts.size() < 4)
        return std::unexpected(ParseError::TooFewFields);
    /* ... */
}
```

```
if (auto entry = parse(line))
    entries.push_back(*std::move(entry));
else
    std::println(stderr, "line {}: {}", n, entry.error());
```

In the exercise

The natural next step for the exercise, if you want one.