

C++23

From your first variable to **deducing this**, **std::expected** and ranges: one booklet you can code from.

```
main.cpp - g++ -std=c++23

#include <print>
#include <ranges>
#include <expected>

auto half(int n) {
    if (n % 2) return std::unexpected("odd");
    return n / 2;
}

int main() {
    for (auto [i, n] : std::views::enumerate({8, 3}))
        std::println("{}: {}", i, half(n).value_or(-1));
}
```

Contents

0	Before You Start	5
0.1	How to use this booklet	5
0.2	Your toolchain in one page	5
1	The Basics	7
1.1	Anatomy of a program	7
1.2	Variables and types	7
1.3	Initializing, <code>const</code> and <code>auto</code>	9
1.4	Operators and conversions	10
1.5	Scope and lifetime	11
1.6	Text: <code>std::string</code> and <code>string_view</code>	12
1.7	Input and output	12
1.8	Arrays and vectors	14
2	Decisions and Loops	15
2.1	Making decisions with <code>if</code>	15
2.2	<code>switch</code>	16
2.3	Loops	16
2.4	Choosing at compile time	18
2.5	Ranges and views	18
2.6	Structured bindings	21
3	Functions	22
3.1	Anatomy of a function	22
3.2	Passing parameters	23
3.3	Returning values	24
3.4	Lambdas	24
3.5	Functions evaluated at compile time	25
3.6	Recursion	26
3.7	A first look at templates and concepts	26
3.8	C++23 upgrades to callables	27
4	Pointers, Step by Step	29
4.1	Start here: memory is a long row of numbered boxes	29
4.2	A pointer is a variable that holds an address	30
4.3	Watching a pointer work, one line at a time	31
4.4	Why bother? Functions that change their caller's data	32
4.5	Pointers and arrays: walking through memory	33
4.6	Pointers to structs: the arrow	34
4.7	<code>nullptr</code> , and the pointer that points nowhere good	35
4.8	<code>const</code> and pointers: read the declaration right to left	36
4.9	Pointers on the heap: where <code>new</code> comes in	37
4.10	Pointers to functions and pointers to pointers	38
4.11	Choosing the right tool	38

5	Memory & Ownership	40
5.1	Stack and heap	40
5.2	Pointers and references	40
5.3	Smart pointers	41
5.4	RAII: cleanup you cannot forget	42
5.5	Move semantics	43
6	Structs, Classes & OOP	45
6.1	Structs	45
6.2	Classes	45
6.3	Inheritance and polymorphism	47
6.4	Operator overloading	49
6.5	Enumerations	50
6.6	Vocabulary types: <code>optional</code> and <code>variant</code>	51
6.7	How a struct sits in memory	51
7	Templates & Concepts	53
7.1	Function and class templates	53
7.2	Concepts	54
7.3	Variadic templates and fold expressions	55
8	The Standard Library	56
8.1	Containers at a glance	56
8.2	Iterators	57
8.3	Algorithms	58
8.4	Views without owning: <code>span</code>	59
8.5	C++23 New library pieces	60
9	Errors & Robustness	63
9.1	Exceptions	63
9.2	C++23 <code>std::expected</code> : errors as values	64
9.3	<code>optional</code> chaining	65
9.4	Checking assumptions	65
9.5	Undefined behavior	66
10	C++23 Extras at a Glance	67
10.1	Small but handy	67
10.2	Callable: <code>static</code> operators and <code>move_only_function</code>	68
10.3	Modules and <code>import std;</code>	69
10.4	Everything C++23 added, in one table	70
11	Concurrency	71
11.1	Why concurrency, and why it is hard	71
11.2	Starting threads: <code>jthread</code>	71
11.3	The data race, made visible	71
11.4	Mutexes: one thread at a time	72
11.5	Atomics and memory ordering	73
11.6	Futures and <code>std::async</code> : ask for a result	74
11.7	Waiting for work: condition variables and a queue	75
11.8	Stopping threads cleanly	76

12	Building Real Projects	78
12.1	From source text to a running program	78
12.2	Headers and source files	79
12.3	CMake: describe the project, let it generate the build	80
12.4	Compiler flags worth knowing	82
12.5	On a Mac	82
12.6	Dependencies and libraries	82
13	Performance & Layout	84
13.1	Measure before you change anything	84
13.2	The memory hierarchy: why the cache decides	85
13.3	Struct of arrays: put the data you loop over together	86
13.4	Avoid asking for memory in hot code	87
13.5	False sharing: when threads fight over a cache line	88
13.6	Other habits that matter	89
14	Engine-Style C++	90
14.1	Why engines avoid the general-purpose allocator	90
14.2	Handles: references that cannot dangle	92
14.3	Entities and components	94
14.4	Layout, alignment and matching the GPU	95
14.5	SIMD: one instruction, four numbers	96
14.6	Where ARM64 helps: inline assembly and NEON	97
14.7	Putting the engine pieces together	101
15	Reference Pages	102
15.1	Operator precedence	102
15.2	Header cheat sheet	103
15.3	Compiler flags, sanitizers and a minimal build	103
15.4	Pitfall checklist	104
15.5	Glossary	104
15.6	"I want to..." finder	105
15.7	Where to go next	105

Before You Start

How this booklet works, which tools you need, and what “C++23” actually adds. Skim this once, then dive in.

0.1 How to use this booklet

This is a tutorial and a reference in one. Read chapters 1–7 in order once (they build on each other; chapter 4 takes pointers slowly, so take your time there). Chapters 8–14 can be read in any order; after that, use the contents page or the **look-up table on the last pages** to jump straight to what you need. Every page tries to show an idea three ways: a short explanation, a picture, and code you can paste into a file and run.

Remember • A tip box holds a rule of thumb you will use constantly.

Pitfall • A pitfall box warns about a classic beginner trap.

New in C++23 • This color marks features that arrived in C++23, always with a **C++23** badge.

Reading a code box

✓ compiled & run

```
// comments are gray italics
int keyword = 1;
std::vector<int> types;
[[nodiscard]] // attributes
```

OUTPUT

this dark strip is real program output

✓ **compiled & run** I compiled the example with `-std=c++23 -Wall -Wextra -pedantic` (g++ 13, or clang++ 18 where noted) with zero warnings, ran it, and pasted the genuine output.

◇ **newer toolchain**

Valid C++23, but it needs a library newer than the one I could test with (GCC 14/15 or a recent libc++). The *expected* output was derived by hand from the language rules, not run here.

0.2 Your toolchain in one page

You need a compiler. On a Mac, `xcode-select --install` gives you Apple Clang; for the very newest C++23 library features also run `brew install gcc llvm`. On Linux, install g++ or clang++ from your package manager.

Compile & run

```
g++      -std=c++23 -Wall -Wextra -pedantic -O2 main.cpp -o main  # GCC
clang++  -std=c++23 -Wall -Wextra -pedantic -O2 main.cpp -o main  # Clang
./main                                     # run it
```

Flag	What it does
<code>-std=c++23</code>	Turn on the C++23 language rules (older compilers: <code>-std=c++2b</code>).
<code>-Wall -Wextra</code>	Ask for helpful warnings. Treat them as bugs waiting to happen.
<code>-O0 / -O2</code>	No optimization (fast build, easy debugging) / normal release speed.
<code>-g</code>	Keep debug info so a debugger can show your source lines.
<code>-fsanitize=address,undefined</code>	Catch memory errors and undefined behavior while the program runs.
<code>-DNDEBUG</code>	Switch off <code>assert(...)</code> for release builds.

Which C++23 features can I use today?

Library features depend on your *standard library* (libstdc++ for GCC, libc++ for Clang, MSVC STL) as much as on the compiler. Version numbers below are approximate; the authoritative table is the “Compiler support” page on [cppreference.com](https://en.cppreference.com).

Feature	GCC	Clang	Tested here
if constexpr	12	14	✓ g++ 13
Multi-index operator[]	12	15	✓ g++ 13
auto(x), [[assume]], uz literals	12–13	13–18	✓ g++ 13
static operator() / static lambdas	13	16	✓ g++ 13
Deducing this	14	18	✓ clang++ 18
std::expected	12	16*	✓ g++ 13
views::zip, enumerate, chunk, slide, stride	13	19+*	✓ g++ 13
ranges::fold_left, contains	13	17+*	✓ g++ 13
std::move_only_function	12	newer*	✓ g++ 13
std::stacktrace	12	—	✓ g++ 13 (syntax)
std::print / println	14	18*	✗ needs newer
ranges::to, std::generator	14	17+* / —	✗ needs newer
std::mdspan	15	18*	✗ needs newer
std::flat_map, flat_set	15	20*	✗ needs newer
import std;	15	18+*	✗ needs newer

* depends on libc++ version; check your platform.

The Basics

Programs, variables, types, text and arrays: everything you need before the first if.

01

1.1 Anatomy of a program

A C++ program is a text file that a *compiler* translates into machine code. Execution always begins at a function called `main`. Everything else is built from **statements** (each ends with `;`), grouped into **blocks** with `{ }`.

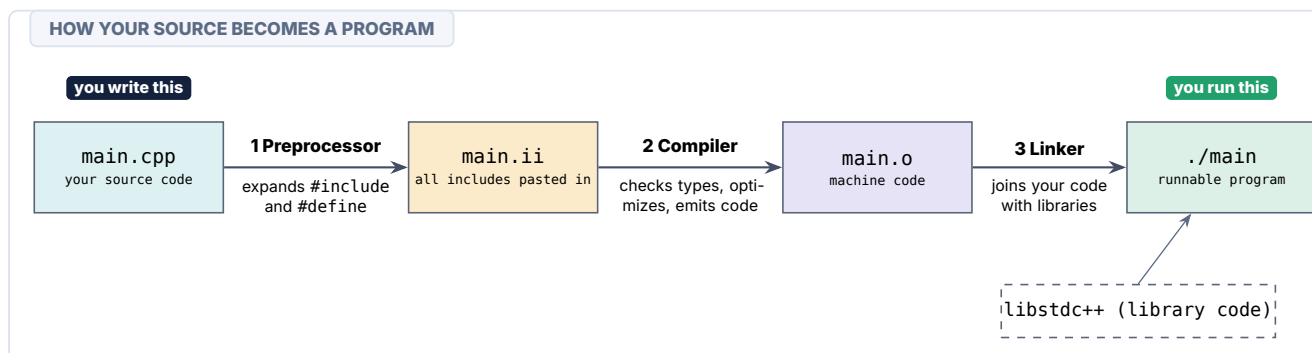
```
hello.cpp ✓ compiled & run · g++ 13

#include <iostream>

int main() {
    std::cout << "Hello, C++23!\n";
}

OUTPUT
Hello, C++23!
```

- `#include <iostream>` pastes in the declarations for input/output.
- `std::cout` is the console output stream; `<<` sends things into it. `std::` is the standard library's *namespace* (a surname that avoids name clashes).
- `\n` is a newline. Returning from `main` without a `return` means success (0).



1.2 Variables and types

A **variable** is a named box in memory. Its **type** decides how many bytes the box has and how those bytes are interpreted. You must declare a type before you use a name; C++ checks the types for you *before* the program ever runs.

```
Declaring variables of every basic type ✓ compiled & run · g++ 13

#include <cstdint>
#include <iostream>

int main() {
    bool    flag    = true;           // true / false
}
```

```

char        letter = 'A';           // one character (really a tiny int)
short       small  = 32000;
int         count  = 42;             // the default whole number
long long   big    = 9000000000LL;  // needs the LL suffix
unsigned    u      = 3000000000u;    // no negatives, double the top end
float       f      = 2.5f;
double     pi      = 3.14159265;    // the default decimal
std::int32_t exact = -7;             // exactly 32 bits, everywhere
std::uint8_t byte  = 200;           // 0..255

std::cout << "bytes: bool=" << sizeof(flag) << " char=" << sizeof(letter)
           << " short=" << sizeof(small) << " int=" << sizeof(count)
           << " long long=" << sizeof(big) << " float=" << sizeof(f)
           << " double=" << sizeof(pi) << '\n';
std::cout << "letter as number: " << static_cast<int>(letter)
           << ", byte: " << +byte << '\n';    // unary + prints it as a number
std::cout << u << ' ' << exact << '\n';
}

```

OUTPUT

```

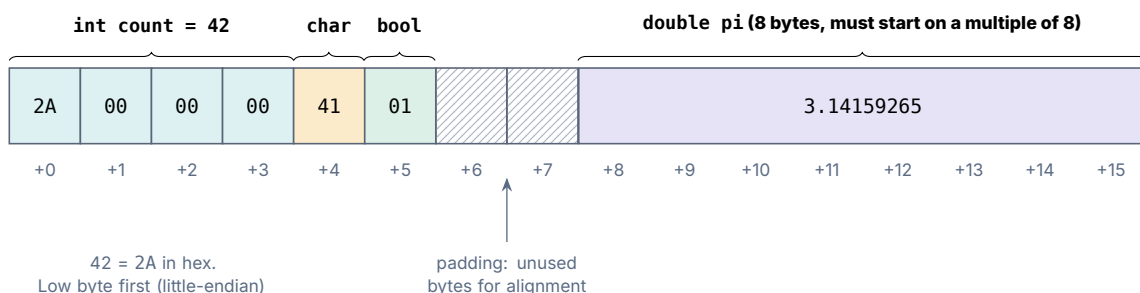
bytes: bool=1 char=1 short=2 int=4 long long=8 float=4 double=8
letter as number: 65, byte: 200
3000000000 -7

```

Type	Bytes	Range / precision	Use it for
bool	1	true or false	yes/no flags
char	1	usually -128 ... 127	a single character
short	2	-32,768 ... 32,767	rarely; saving memory
int	4	±2.1 billion	default whole number
long long	8	±9.2 × 10 ¹⁸	big counts, IDs, timestamps
unsigned	4	0 ... 4.29 billion	bit patterns, never-negative
std::size_t	8	0 ... 1.8 × 10 ¹⁹	sizes and array indices
float	4	±3.4 × 10 ³⁸ , ≈7 digits	graphics, big arrays
double	8	±1.8 × 10 ³⁰⁸ , ≈16 digits	default decimal number

Sizes shown are typical for 64-bit Linux, macOS and Windows (except long, which is 4 or 8). Check the real numbers with `sizeof`. Fixed-width types such as `std::int32_t` and `std::uint8_t` (header `<cstdint>`) are exact everywhere.

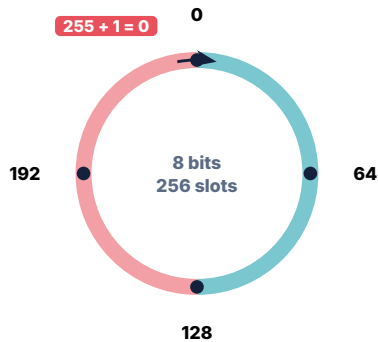
WHERE THE BYTES LIVE: THE VARIABLES FROM THE EXAMPLE ABOVE, STORED SIDE BY SIDE



Numbers wrap around

Whole numbers have a fixed number of bits, so they behave like a clock: past the top value they roll over to the bottom. For unsigned types that is well-defined; for *signed* types it is **undefined behavior**, so never rely on it.

AN 8-BIT INTEGER IS A CLOCK WITH 256 POSITIONS



uint8_t (unsigned) counts 0 → 255 around the whole ring.

int8_t (signed) uses the same ring but the bottom half means -128 ... -1.

255 + 1 wraps to 0 (and 0 - 1 wraps to 255) for unsigned types: well-defined.

127 + 1 on a signed type is **undefined behavior**: the compiler may assume it never happens.

1.3 Initializing, const and auto

Always give a variable a value when you create it; reading an uninitialized variable is undefined behavior. Prefer **brace initialization**: it refuses lossy conversions (narrowing) that other styles silently allow.

Initialization styles, auto, const, constexpr

✓ compiled & run · g++ 13

```
#include <iostream>

int main() {
    int a = 5;           // copy-initialization
    int b(5);           // direct-initialization
    int c{5};           // brace-initialization <-- preferred
    int d{};            // empty braces = zero
    // int bad{3.7};     // ERROR: narrowing conversion is caught

    auto x = 5;          // deduced: int
    auto y = 5.0;        // deduced: double
    auto z = 5uz;        // deduced: std::size_t (C++23 suffix)
    const int limit = 10; // can never change
    constexpr int area = limit * limit; // computed at compile time

    std::cout << a << b << c << d << ' ' << x << ' ' << y << ' '
               << z << ' ' << area << '\n';
}
```

OUTPUT

5550 5 5 5 100

Keyword	Meaning
const	Cannot be modified after initialization (checked by the compiler).
constexpr	Value is known <i>at compile time</i> ; usable for array sizes, <code>static_assert</code> , etc.
constexpr	(functions) Must be evaluated at compile time. See chapter 3.
constinit	A static variable that must be initialized at compile time but may change later.
auto	Let the compiler deduce the type from the initializer. Great for long types; avoid when the type itself is the point.

Remember • Default to **const** on everything, and remove it only when you really must change the value. Code you cannot accidentally modify is code you do not have to re-read.

New in C++23 • `5uz` is a literal of type `std::size_t`, handy in loops like `for (auto i = 0uz; i < v.size(); ++i)` so the index type matches `v.size()`. **C++23**

1.4 Operators and conversions

Arithmetic, logic, bitwise and the spaceship operator

✓ compiled & run · g++ 13

```
#include <compare>
#include <iostream>

int main() {
    int a = 10, b = 3;
    a += 5;           // a = a + 5      -> 15
    a %= b;           // a = a % 3      -> 0
    ++b;              // pre-increment -> b is 4
    int c = b++;       // post: c gets 4, THEN b becomes 5
    std::cout << a << ' ' << b << ' ' << c << '\n';

    bool in_range = (b > 2) && (b <= 10); // && (and) || (or) ! (not)
    std::cout << in_range << ' ' << (b == 5) << ' ' << (b != 5) << '\n';

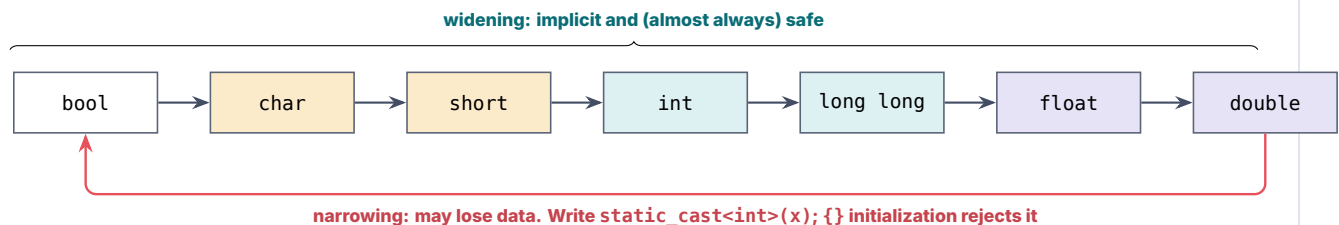
    unsigned flags = 0b1010;             // binary literal
    std::cout << (flags & 0b0110) << ' ' << (flags | 1) << ' '
               << (flags ^ 0b1111) << ' ' << (flags << 2) << '\n';

    auto order = 2 <=> 5;                  // three-way comparison
    std::cout << (order < 0) << '\n';      // 1: 2 is less than 5
}
```

OUTPUT

```
0 5 4
1 1 0
2 11 5 40
1
```

TYPE CONVERSIONS: WIDENING IS AUTOMATIC, NARROWING IS YOUR DECISION



Integer division, casts and wrap-around

✓ compiled & run · g++ 13

```
#include <iostream>

int main() {
    int i = 7, j = 2;
    std::cout << i / j << '\n';           // 3    integer division!
    std::cout << i % j << '\n';           // 1    remainder
    std::cout << static_cast<double>(i) / j << '\n'; // 3.5  convert first

    double d = 3.99;
    int cut = static_cast<int>(d);         // 3    truncates, never rounds
    std::cout << cut << '\n';

    unsigned char wrap = 255;
    ++wrap;                                // wraps around to 0
    std::cout << +wrap << '\n';
}
```

OUTPUT

```
3
1
3.5
3
0
```

Pitfall • $7 / 2$ is **3**, not 3.5: dividing two integers gives an integer. Convert one side first, e.g. $7 / 2.0$ or `static_cast<double>(i) / j`. Also never compare floating-point numbers with `==`; compare with a small tolerance instead.

1.5 Scope and lifetime

A name exists from its declaration to the closing brace of the block it was declared in. When a local variable goes out of scope it is destroyed; this rule is the foundation of everything in chapter 4.

Block scope, shadowing and static locals

✓ compiled & run · g++ 13

```
#include <iostream>

int global = 1;                // lives for the whole program

int counter() {
    static int n = 0;          // created once, remembers its value
    return ++n;
}

int main() {
    int outer = 2;
    {                          // a new scope begins
        int inner = 3;
        int outer = 99;        // shadows the outer 'outer' (avoid!)
        std::cout << inner << ' ' << outer << ' ' << global << '\n';
    }                          // 'inner' dies here
    std::cout << outer << '\n';  // the original 'outer' is back: 2
    counter();
    std::cout << counter() << '\n'; // 2
}
```

OUTPUT

```
3 99 1
2
2
```

SCOPES NEST LIKE BOXES; INNER NAMES HIDE OUTER ONES

FILE SCOPE `int global = 1;` lives for the whole program

`main() int outer = 2;`

Sees `outer = 2` and `global`.
After the inner block ends, `inner` is destroyed and
`outer` is 2 again.

{ } inner block

```
int inner = 3;
int outer = 99; ← shadows outer
```

} inner dies

1.6 Text: `std::string` and `string_view`

Use `std::string` for text you own and may change. Use `std::string_view` for read-only function parameters: it is a cheap (pointer, length) window and never copies the characters.

Strings and string views

✓ compiled & run · g++ 13

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

int main() {
    std::string s = "hello";
    s += " world";           // append
    std::cout << s.size() << ' ' << s.substr(0, 5) << '\n';
    std::cout << s.find("wor") << '\n';      // index (or std::string::npos)
    std::cout << s.starts_with("he") << s.contains("lo w") << '\n'; // contains: C++23

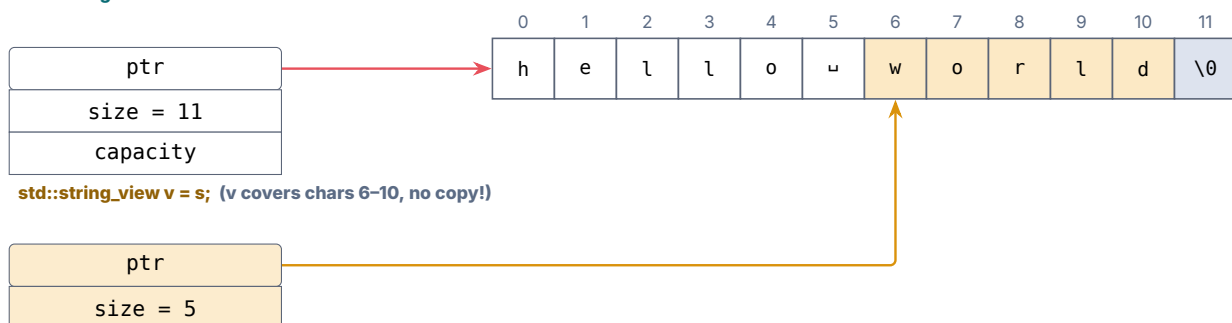
    std::string_view v = s;           // cheap read-only window, no copy
    std::cout << v.substr(6) << '\n';
}
```

OUTPUT

```
11 hello
6
11
world
```

ONE BUFFER, TWO WAYS TO LOOK AT IT

`std::string s = "hello world"`



Pitfall • A `string_view` does not own the characters. Never return one that points into a local `std::string`, and never store one longer than the string lives.

1.7 Input and output

`std::format`: pretty output without printf pitfalls

✓ compiled & run · g++ 13

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

int main() {
    std::cout << std::format("{: >6}\n", 42);           // right-align in 6
    std::cout << std::format("{: <6}\n", 42);           // left-align
    std::cout << std::format("{: ^6}\n", 42);           // center
    std::cout << std::format("{: *^9}\n", "hi");         // center, fill with *
    std::cout << std::format("{:.3f} {:08.2f}\n", 3.14159, 3.14159);
    std::cout << std::format("{:#x} {:#b} {:o}\n", 255, 5, 8);
    std::cout << std::format("{1} before {0}\n", "B", "A"); // positional
}
```

OUTPUT

```
[ 42]
[42 ]
[ 42 ]
[***hi***]
3.142 00003.14
0xff 0b101 10
A before B
```

std::print and std::println[newer toolchain](#)

```
#include <print>

int main() {
    std::println("Hello, {}!", "C++23");           // print + newline
    int n = 3;
    std::println("{} squared is {}", n, n * n);
    std::print("no newline, ");                    // print without newline
    std::print(stderr, "to stderr: {:>5}\n", n);    // any FILE*
}
```

EXPECTED OUTPUT (not run here: needs GCC 14+ / newer libc++)

```
Hello, C++23!
3 squared is 9
no newline,
```

New in C++23 • `std::print` / `std::println` (header `<print>`) combine the type-safe format strings of `std::format` with direct printing, and are usually faster than `std::cout`. Prefer them once your toolchain has them (GCC 14+, recent Clang/MSVC). [C++23](#)

Spec	Example	Meaning
{}	{}	next argument, default format
{1} {0}	{1} {0}	pick arguments by position
{:>8} {:<8} {:^8}	{:^9}	right / left / center in a field of that width
{:*^9}	{:*^9}	same, padded with *
{:.3f}	{:08.2f}	3 decimals; zero-pad to width 8 with 2 decimals
{:#x} {:#b} {:#o}	{:#x}	hex with 0x / binary with 0b / octal

Reading input with cin and getline[compile-checked](#) · [g++ 13](#)

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

int main() {
    std::string name;
    int age{};
    std::cout << "Name? ";
    std::getline(std::cin, name);           // a whole line, spaces allowed
    std::cout << "Age? ";
    std::cin >> age;                        // one whitespace-separated value
    if (!std::cin) {                        // the text was not a number
        std::cerr << "not a number\n";
        return 1;
    }
    std::cout << name << " is " << age << '\n';
}
```

1.8 Arrays and vectors

C arrays, `std::array` and `std::vector`

✓ compiled & run · g++13

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

int main() {
    int raw[3] = {1, 2, 3};           // C-style: no .size(), decays to pointer
    std::array<int, 3> fixed{1, 2, 3}; // size is part of the type
    std::vector<int> v{1, 2, 3};      // grows on demand <-- your default

    v.push_back(4);
    v.insert(v.begin(), 0);           // now {0,1,2,3,4}
    std::cout << v.size() << ' ' << v.front() << ' ' << v.back() << '\n';
    std::cout << v.at(2) << ' ' << fixed.size() << ' ' << raw[1] << '\n';

    for (int x : v) std::cout << x << ' ';    // range-based for
    std::cout << '\n';
}
```

OUTPUT

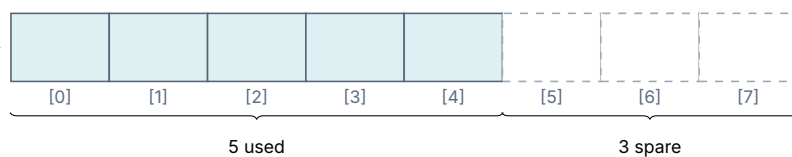
```
5 0 4
2 3 2
0 1 2 3 4
```

A `std::vector` IS A SMALL HANDLE (ON THE STACK) POINTING AT A HEAP BLOCK

`std::vector<int> v (stack)`

data
size = 5
capacity = 8

heap block: one contiguous run of ints



`push_back` into a full vector allocates a bigger block, moves the elements over and frees the old one.

Remember • Default container: `std::vector`. Use `std::array` when the size is a compile-time constant. Prefer `.at(i)` while learning (it throws on a bad index); `v[i]` does no checking at all.

Chapter 1 at a glance

```
int n{5}; auto d = 2.5;
const / constexpr
static_cast<double>(i) / j
std::string, std::string_view
std::vector<int> v{1, 2, 3};
std::format("{:>6.2f}", x)
unsigned wraps; signed overflow is UB
```

brace-init, deduced type
 can't change / known at compile time
 convert *before* dividing
 owning text, read-only window
 growable array: your default container
 formatted text; `std::println` prints it **C++23**
 never rely on signed overflow

Decisions and Loops

Controlling the order things happen in: branches, loops, compile-time choices and the modern ranges pipeline.

02

2.1 Making decisions with `if`

An `if` runs a block only when its condition is true. Chain with `else if`; the first true branch wins and the rest are skipped. Since C++17 you can declare a variable right in the `if` (the *init-statement*); it is scoped to the whole `if/else`.

if, else if, init-statements and the ternary operator

✓ compiled & run · g++ 13

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

int main() {
    int score = 72;
    char grade;
    if (score >= 90) grade = 'A';
    else if (score >= 80) grade = 'B';
    else if (score >= 70) grade = 'C';
    else grade = 'F';
    std::cout << grade << '\n';

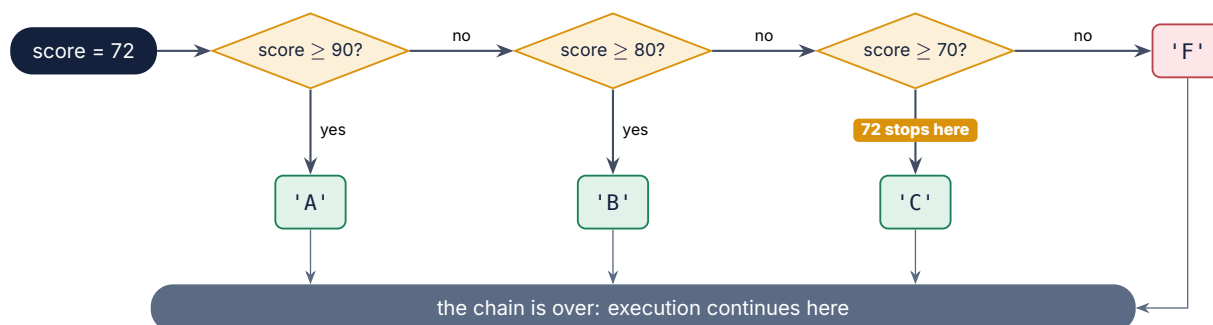
    std::map<std::string, int> ages{{"ann", 31}, {"bo", 25}};
    if (auto it = ages.find("bo"); it != ages.end()) { // init; condition
        std::cout << it->first << " is " << it->second << '\n';
    } // 'it' ends here

    std::string verdict = (score >= 50) ? "pass" : "fail"; // ternary
    std::cout << verdict << '\n';
}
```

OUTPUT

```
C
bo is 25
pass
```

FLOW OF AN `if / else if / else` CHAIN (THE GRADE EXAMPLE)



Remember • Always use braces { } even around a single statement; skipping them is how the famous “goto fail” security bug happened. Compare with == (two equals!); a single = assigns.

2.2 switch

Use **switch** to choose between many fixed values of an integer or enum. Each **case** *falls through* into the next one unless you **break** or **return**; mark intentional fall-through with `[[fallthrough]]`;

switch on an enum class, fall-through and std::unreachable

✓ compiled & run · g++ 13

```
#include <iostream>
#include <utility>

enum class Color { Red, Green, Blue };

const char* name(Color c) {
    switch (c) {
        // compiler warns if a case is missing
        case Color::Red:    return "red";
        case Color::Green:  return "green";
        case Color::Blue:   return "blue";
    }
    std::unreachable(); // C++23: "control can never get here"
}

int main() {
    std::cout << name(Color::Green) << '\n';

    int day = 6;
    switch (day) {
        case 6:
            std::cout << "weekend ";
            [[fallthrough]]; // deliberate: run into the next case
        case 7:
            std::cout << "no work\n";
            break; // without break, execution keeps falling
        default:
            std::cout << "work\n";
    }
}
```

OUTPUT
green
weekend no work

New in C++23 • `std::unreachable()` (header `<utility>`) tells the compiler a code path can never be taken, so it can delete the check. If it is reached the behavior is undefined, so use it only when you have already handled every case. **C++23**

2.3 Loops

for, while, do-while, range-for, break and continue

✓ compiled & run · g++ 13

```
#include <iostream>
#include <vector>

int main() {
    for (int i = 0; i < 3; ++i) std::cout << i << ' '; // counting loop
    std::cout << '\n';

    int n = 10, steps = 0;
    while (n != 1) { // test first
        n = (n % 2 == 0) ? n / 2 : 3 * n + 1;
    }
}
```

```

    ++steps;
}
std::cout << steps << '\n';           // Collatz steps for 10

int tries = 0;
do { ++tries; } while (tries < 3);     // body runs at least once
std::cout << tries << '\n';

std::vector<int> v{5, 6, 7, 8, 9};
for (int x : v) {                     // range-based for
    if (x == 6) continue;             // skip this one
    if (x == 9) break;               // leave the loop
    std::cout << x << ' ';
}
std::cout << '\n';
}

```

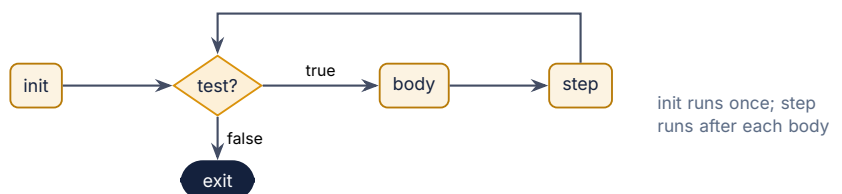
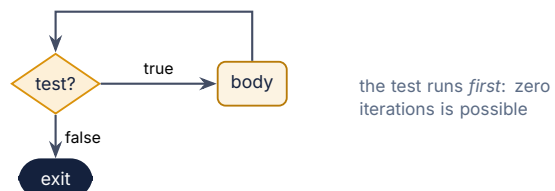
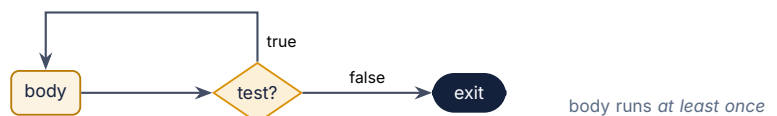
OUTPUT

```

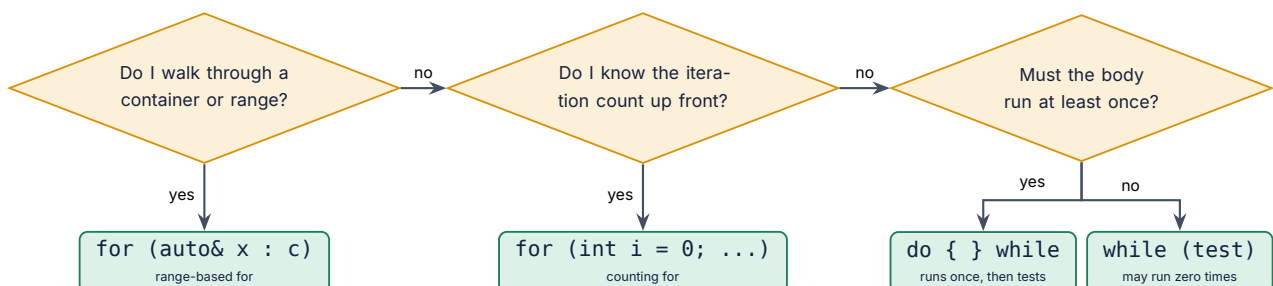
0 1 2
6
3
5 7 8

```

HOW EACH LOOP FLOWS

for (init; test; step)**while (test)****do { } while (test);**

WHICH LOOP SHOULD I WRITE?



Pitfall • Comparing a signed `int` index with the unsigned `v.size()` is a classic bug source (the signed value converts to a huge unsigned one). Prefer range-based `for`, or write

```
for (std::size_t i = 0; i < v.size(); ++i).
```

2.4 Choosing at compile time

Some decisions can be made while compiling, producing faster code with no run-time test at all. `if constexpr` discards the branch that does not apply to a template's type, so unused branches need not even compile. `if constexpr` **C++23** asks: "am I being evaluated by the compiler right now?"

if constexpr and if constexpr

✓ compiled & run · g++13

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

template <class T>
std::string kind(T) {
    if constexpr (std::is_integral_v<T>)      return "integer";
    else if constexpr (std::is_floating_point_v<T>) return "float";
    else                                       return "something else";
}

constexpr const char* where() {
    if constexpr { return "compile time"; }    // C++23
    else       { return "run time"; }
}

int main() {
    std::cout << kind(1) << ", " << kind(2.5) << ", " << kind("hi") << '\n';
    constexpr auto a = where();               // forced at compile time
    auto b = where();                          // ordinary call
    std::cout << a << " / " << b << '\n';
}
```

OUTPUT

```
integer, float, something else
compile time / run time
```

2.5 Ranges and views

A **range** is anything you can iterate (a vector, a string, ...). A **view** is a lazy, cheap-to-copy lens over a range: it *computes nothing until you loop over it*, and never copies the elements. Chain views with the pipe operator `|`, reading left to right like a Unix pipeline.

A lazy pipeline: filter, transform, take

✓ compiled & run · g++13

```
#include <iostream>
#include <ranges>
#include <vector>

int main() {
    namespace rv = std::views;
    std::vector<int> nums{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

    auto result = nums
        | rv::filter([](int x) { return x % 2 == 0; }) // keep the evens
        | rv::transform([](int x) { return x * x; })   // square them
        | rv::take(3);                                 // first three only

    for (int x : result) std::cout << x << ' ';      // 4 16 36
    std::cout << '\n';
}
```

```

for (int x : rv::iota(1, 6) | rv::reverse) std::cout << x << ' ';
std::cout << '\n';
}

```

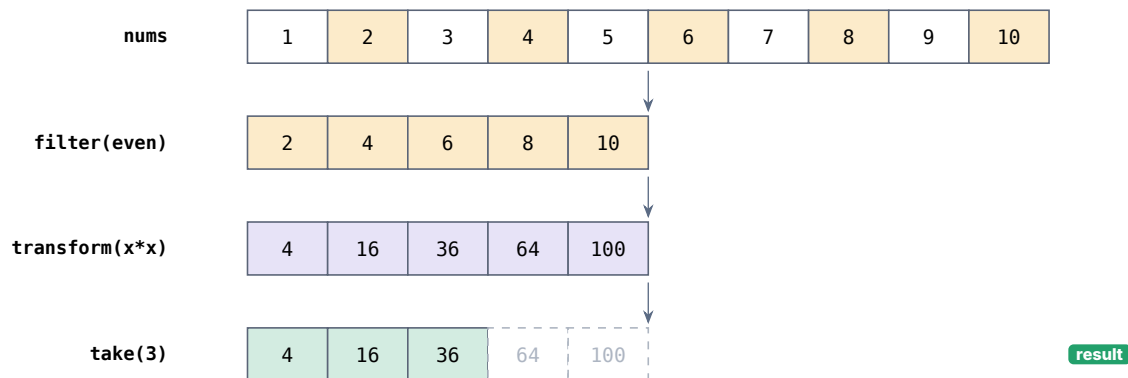
OUTPUT

```

4 16 36
5 4 3 2 1

```

THE PIPELINE, ONE STAGE AT A TIME (NOTHING RUNS UNTIL THE FOR LOOP ASKS)



More views arrive in C++23

enumerate, zip, chunk, slide, stride, cartesian_product

✓ compiled & run · g++ 13

```

#include <iostream>
#include <ranges>
#include <string>
#include <vector>

int main() {
    namespace rv = std::views;
    std::vector<std::string> names{"ann", "bo", "cy"};
    std::vector<int> ages{31, 25, 40};

    for (auto [i, n] : rv::enumerate(names)) // index + element
        std::cout << i << ":" << n << ' ';
    std::cout << '\n';

    for (auto [n, a] : rv::zip(names, ages)) // walk two ranges together
        std::cout << n << "=" << a << ' ';
    std::cout << '\n';

    std::vector<int> v{1, 2, 3, 4, 5, 6, 7};
    for (auto chunk : v | rv::chunk(3)) { // blocks of 3
        for (int x : chunk) std::cout << x;
        std::cout << '|';
    }
    std::cout << '\n';

    for (auto w : v | rv::slide(3)) // sliding window of 3
        std::cout << w[0] + w[1] + w[2] << ' ';
    std::cout << '\n';

    for (int x : rv::iota(0, 10) | rv::stride(3)) // every 3rd element
        std::cout << x << ' ';
    std::cout << '\n';

    for (auto [a, b] : rv::cartesian_product(rv::iota(0, 2), rv::iota(0, 2)))
        std::cout << "(" << a << ", " << b << ")"; // all pairs
}

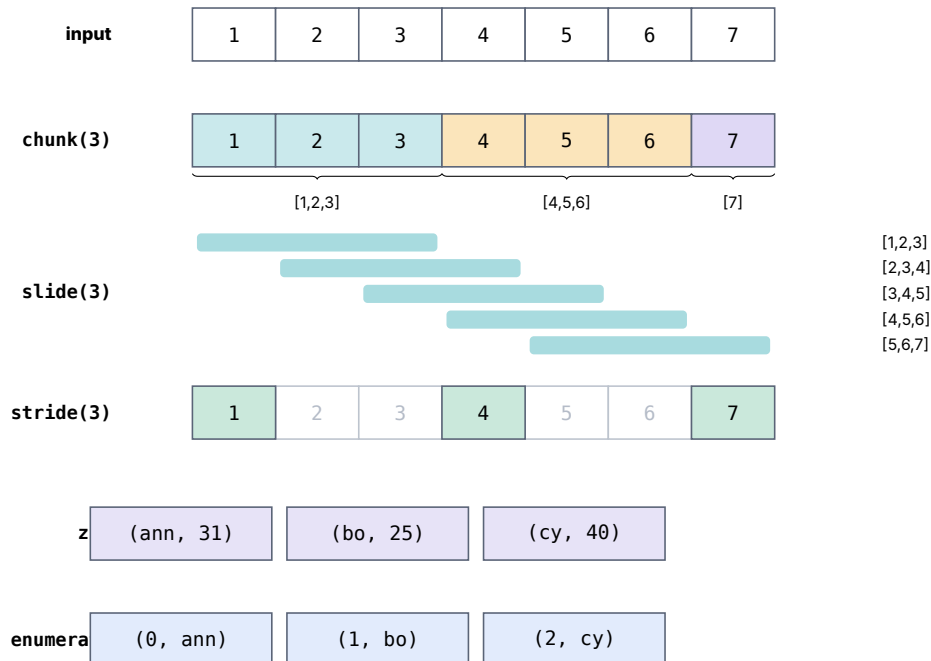
```

```
std::cout << '\n';
}
```

OUTPUT

```
0:ann 1:bo 2:cy
ann=31 bo=25 cy=40
123|456|7|
6 9 12 15 18
0 3 6 9
(0,0) (0,1) (1,0) (1,1)
```

WHAT THE NEW VIEWS DO TO {1,2,3,4,5,6,7}



View

What it gives you

filter(pred), transform(f)
 take(n), drop(n), take_while(p)
 reverse, iota(a, b), keys,
 values
 join, split(delim),
 join_with(sep) **C++23**
 zip, enumerate **C++23**
 chunk(n), slide(n), stride(n)
C++23
 adjacent<N>, pairwise **C++23**
 cartesian_product **C++23**
 chunk_by(p), repeat(x) **C++23**

keep matching elements / map each element
 first n, all but first n, prefix while true
 backwards, counting sequence, map keys / values
 flatten nested ranges, split, join with a separator
 walk several ranges together / pair each element with its index
 blocks of n / overlapping windows of n / every nth element
 neighboring tuples, e.g. (x[i], x[i+1])
 every combination of several ranges (a nested loop in one line)
 chunk while predicate holds between neighbors / endless repeats

2.6 Structured bindings

Unpacking structs, map entries and tuples

✓ compiled & run · g++ 13

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

struct Point { int x; int y; };

int main() {
    Point p{3, 4};
    auto [x, y] = p; // unpack a struct
    std::cout << x << ', ' << y << '\n';

    std::map<std::string, int> stock{{"apple", 5}, {"pear", 2}};
    for (const auto& [fruit, qty] : stock) // unpack each key/value pair
        std::cout << fruit << ':' << qty << ' ';
    std::cout << '\n';

    auto [lo, hi, label] = std::tuple{1, 9, "range"};
    std::cout << label << ' ' << lo << '-' << hi << '\n';
}
```

OUTPUT

```
3,4
apple:5 pear:2
range 1-9
```

Chapter 2 at a glance

<code>if (auto it = m.find(k); it != m.end())</code>	declare a variable inside the <code>if</code>
<code>switch + [[fallthrough]] + break</code>	choose among fixed values
<code>for (auto& x : v)</code>	your default loop over a container
<code>if constexpr (cond)</code>	compile-time branch inside templates
<code>v views::filter(p) views::transform(f)</code>	lazy pipeline; nothing runs until iterated
<code>views::enumerate / zip / chunk / slide</code>	C++23 index, pairing and windowing views
<code>auto [a, b] = pair_or_struct;</code>	unpack several values at once

Functions

Naming a piece of work so you can reuse it: parameters, return values, lambdas, compile-time functions and the new C++23 callable features.

03

3.1 Anatomy of a function

A **function** bundles statements under a name. It has a *signature* (name and parameter types), a *return type* and a *body*. A **declaration** (prototype) announces the signature so other code can call it; the **definition** supplies the body. **Overloads** share a name but differ in parameter types, and the compiler picks the best match.

Declaration, definition, default arguments and overloading

✓ compiled & run · g++ 13

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

int square(int x);           // declaration (prototype)
int square(int x) { return x * x; } // definition

double area(double w, double h = 1.0) { // default argument
    return w * h;
}

std::string describe(int n)    { return "int " + std::to_string(n); } // overloads:
std::string describe(double d) { return "double " + std::to_string(d); } // same name,
std::string describe(bool b)   { return b ? "bool true" : "bool false"; } // other types

int main() {
    std::cout << square(5) << ' ' << area(3) << ' ' << area(3, 2) << '\n';
    std::cout << describe(7) << " | " << describe(2.5) << " | " << describe(true) << '\n';
}
```

OUTPUT

```
25 3 6
int 7 | double 2.500000 | bool true
```

THE PARTS OF A FUNCTION DEFINITION

```
double area(double w, double h = 1.0) { return w * h; }
```

return type

name

parameter list

default argument

body

Call it with `area(3)` (uses `h = 1.0`) or `area(3, 2)`. Default arguments must be the *last* parameters.

3.2 Passing parameters

How a parameter is declared decides whether the function works on a *copy*, on the *original*, or merely *looks* at the original.

By value, by reference, by pointer and by const reference

✓ compiled & run · g++ 13

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

void by_value(int n) { n += 1; }           // changes a COPY
void by_ref(int& n) { n += 1; }           // changes the original
void by_ptr(int* n) { if (n) *n += 1; }   // original, via its address
int length(const std::string& s) {       // read-only, and no copy made
    return static_cast<int>(s.size());
}

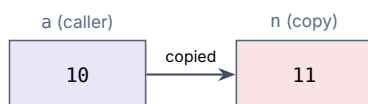
int main() {
    int a = 10;
    by_value(a); std::cout << a << ' '; // 10
    by_ref(a);   std::cout << a << ' '; // 11
    by_ptr(&a);  std::cout << a << '\n'; // 12
    std::cout << length("hello there") << '\n'; // 11
}
```

OUTPUT

```
10 11 12
11
```

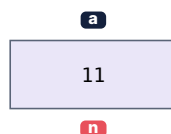
THREE WAYS TO HAND a TO A FUNCTION

void f(int n) by value



Changing `n` leaves `a` untouched. Fine for `int`, `double`, pointers.

void f(int& n) by reference



One object with two names. Changing `n` changes `a`. Use `const int&` to look without touching.

void f(int* n) by pointer



Pass `&a`; use `*n`. The pointer may be null: check it first.

Situation	Parameter	Why
Small, cheap type (<code>int</code> , <code>double</code> , <code>bool</code> , <code>enums</code>)	<code>T x</code>	copying 8 bytes is as cheap as passing a reference
Large object, only read it	<code>const T& x</code>	no copy, no modification
Must modify the caller's object	<code>T& x</code>	shows up as an alias of the original
Function will keep / own the argument	<code>T x + std::move(x)</code>	caller can move or copy; function takes ownership
Text you only read	<code>std::string_view s</code>	accepts literals, strings, substrings without copying
A sequence you only read	<code>std::span<const T> xs</code>	accepts arrays, vectors and <code>std::array</code>
Argument may be absent	<code>std::optional<T> / T*</code>	make "nothing" explicit

3.3 Returning values

Return by value, even for large objects: the compiler elides the copy or moves it. Return several values with a small **struct** (best: it names the fields) or a `std::pair`/`std::tuple`, and unpack with structured bindings.

Returning structs and pairs; `[[nodiscard]]`, `noexcept`, deduced return type

✓ compiled & run · g++ 13

```
#include <algorithm>
#include <iostream>
#include <string>
#include <utility>

struct Stats { double mean; double max; };

Stats analyze(double a, double b, double c) {           // several results at once
    return {(a + b + c) / 3, std::max({a, b, c})};
}

[[nodiscard]] int risky() noexcept { return 42; }        // caller must use the result

auto numbered(int n) { return std::pair{n, std::to_string(n)}; } // deduced type

int main() {
    auto [mean, mx] = analyze(1, 5, 9);
    std::cout << mean << ' ' << mx << '\n';
    int v = risky();                                     // ignoring risky() would trigger a warning
    auto [n, s] = numbered(v);
    std::cout << n << ' ' << s << '\n';
}
```

OUTPUT

```
5 9
42 42
```

Pitfall • Never return a reference or pointer to a *local* variable: the variable dies at the closing brace and you are left holding a dangling reference.

3.4 Lambdas

A **lambda** is an unnamed function you write in place: `[captures](params) { body }`. The *capture list* decides which outside variables it can see, and whether it copies or references them.

Captures, generic lambdas, `std::function` and sorting

✓ compiled & run · g++ 13

```
#include <algorithm>
#include <functional>
#include <iostream>
#include <vector>

int main() {
    int factor = 3;
    auto times = [factor](int x) { return x * factor; }; // capture by COPY
    auto bump = [&factor]() { ++factor; };              // capture by REFERENCE
    bump();                                              // factor is now 4
    std::cout << times(5) << ' ' << factor << '\n';      // 15 4 (copy kept 3)

    auto add = [](auto a, auto b) { return a + b; };    // generic lambda
    std::cout << add(1, 2) << ' ' << add(1.5, 2.5) << '\n';

    std::function<int(int)> f = times;                  // store any callable
    std::cout << f(10) << '\n';

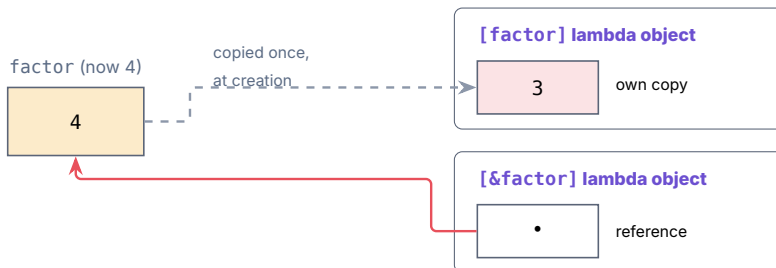
    std::vector<int> v{5, 2, 9, 1};
    std::ranges::sort(v, [](int a, int b) { return a > b; }); // descending
}
```

```
for (int x : v) std::cout << x << ' ';
std::cout << '\n';
}
```

OUTPUT

```
15 4
3 4
30
9 5 2 1
```

CAPTURE BY COPY FREEZES THE VALUE; CAPTURE BY REFERENCE FOLLOWS THE ORIGINAL



Using a reference capture after the original variable has been destroyed is undefined behavior.

Capture	Meaning
<code>[]</code>	capture nothing
<code>[x] / [&x]</code>	copy x / refer to x
<code>[=] / [&]</code>	copy / refer to everything the lambda uses (prefer explicit lists)
<code>[p = std::move(ptr)]</code>	<i>init-capture</i> : move or build a new member inside the lambda
<code>[this] / [*this]</code>	the current object by pointer / by copy

3.5 Functions evaluated at compile time

constexpr, consteval and static_assert

✓ compiled & run · g++ 13

```
#include <iostream>

constexpr int factorial(int n) { // may run at compile time OR run time
    return n <= 1 ? 1 : n * factorial(n - 1);
}

consteval int cube(int n) { return n * n * n; } // MUST run at compile time

static_assert(factorial(5) == 120); // the compiler checks this
constexpr int table[] = {factorial(1), factorial(2), factorial(3)};

int main() {
    int runtime = 6;
    std::cout << factorial(runtime) << ' ' << cube(3) << ' ' << table[2] << '\n';
    // cube(runtime); // ERROR: not a constant expression
}
```

OUTPUT

```
720 27 6
```

Keyword	On a variable	On a function
<code>const</code>	value never changes (may be computed at run time)	—
<code>constexpr</code>	computable at compile time	may run at compile time or run time
<code>constexpr</code>	—	must run at compile time (an “immediate function”)
C++23		
<code>constinit</code>	initialized at compile time, still mutable	—

3.6 Recursion

A function that calls itself needs a **base case** that stops the calls, and each call must move closer to it. Every call gets its own *stack frame* holding its parameters and locals.

Factorial and Fibonacci

✓ compiled & run · g++ 13

```
#include <iostream>

unsigned long long fact(unsigned n) {
    if (n <= 1) return 1;           // base case: stops the recursion
    return n * fact(n - 1);        // recursive case: a smaller problem
}

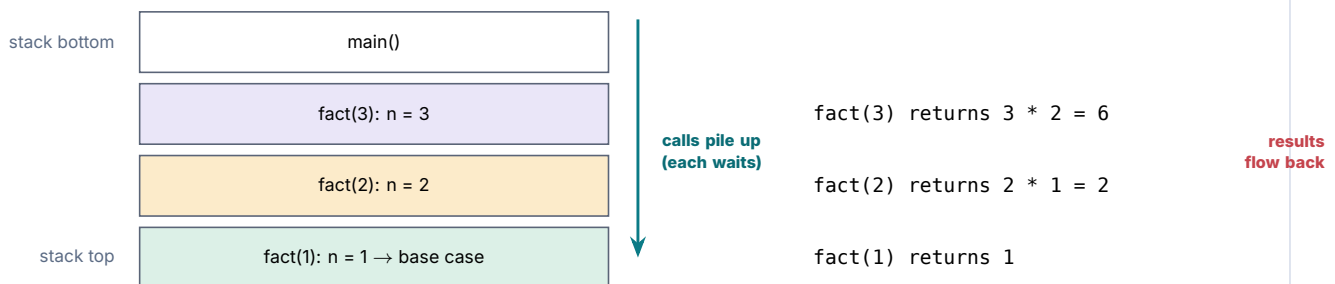
int fib(int n) { return n < 2 ? n : fib(n - 1) + fib(n - 2); } // clear but slow

int main() {
    std::cout << fact(3) << ' ' << fact(10) << ' ' << fib(10) << '\n';
}
```

OUTPUT

6 3628800 55

THE CALL STACK WHILE EVALUATING fact(3)



Pitfall • Without a base case (or with one that is never reached) the recursion runs until the stack is exhausted: a *stack overflow*, which crashes the program. Deep recursion over big inputs is usually better written as a loop.

3.7 A first look at templates and concepts

A **template** is a function (or class) written once for *any* type. A **concept** names a requirement a type must meet, giving clear error messages when it does not. Chapter 7 goes deeper.

Function templates, a constrained template and a fold expression

✓ compiled & run · g++ 13

```
#include <concepts>
#include <iostream>
```

```
#include <string>

template <typename T>                // works for ANY T that has operator>
T biggest(T a, T b) { return a > b ? a : b; }

template <std::integral T>           // constrained: whole numbers only
T half(T x) { return x / 2; }

auto sum_all(auto... xs) { return (xs + ...); } // variadic + fold expression

int main() {
    std::cout << biggest(3, 7) << ' ' << biggest(2.5, 1.5) << ' '
               << biggest(std::string("pear"), std::string("apple")) << '\n';
    std::cout << half(9) << '\n';                // 4
    // half(9.0);                               // ERROR: double is not integral
    std::cout << sum_all(1, 2, 3, 4) << '\n';    // 10
}
```

OUTPUT

```
7 2.5 pear
4
10
```

3.8 C++23 upgrades to callables

Static call operators and static lambdas C++23

A function object used to need a (pointless) object pointer for every call. Mark `operator()` or a lambda `static` when it uses no member data; the call becomes as cheap as a plain function, and the lambda converts to a plain function pointer more efficiently.

static operator() and static lambdas ✓ compiled & run · g++ 13

```
#include <iostream>

struct Square {
    static int operator()(int x) { return x * x; } // C++23: no object needed
};

int main() {
    auto cube = [](int x) static { return x * x * x; }; // C++23: static lambda
    std::cout << Square{}(4) << ' ' << Square::operator()(5) << ' ' << cube(3) << '\n';
}
```

OUTPUT

```
16 25 27
```

Deducing this C++23

An *explicit object parameter* makes the hidden `this` an ordinary parameter named `self`. One template then replaces the four classic overloads, and lambdas can finally call themselves.

Explicit object parameter: one getter, and a recursive lambda ✓ compiled & run · clang++ 18

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

struct Box {
    std::string text{"hi"};

    // ONE function replaces the const / non-const / lvalue / rvalue overloads
    template <class Self>
```

```

    auto&& get(this Self&& self) { return std::forward<Self>(self).text; }
};

int main() {
    Box b;
    const Box cb;
    b.get() += "!"; // non-const object: may modify
    std::cout << b.get() << ' ' << cb.get() << '\n';

    // A lambda that can call ITSELF, with no std::function needed:
    auto fib = [](this auto self, int n) -> int {
        return n < 2 ? n : self(n - 1) + self(n - 2);
    };
    std::cout << fib(10) << '\n';
}

```

OUTPUT

```

hi! hi
55

```

Before C++23, you wrote all four Called when the object is...

T& get() &;	a non-const lvalue
const T& get() const&;	a const lvalue
T&& get() &&;	a temporary / moved object
const T&& get() const&&;	a const temporary (rare)

std::move_only_function C++23

std::function requires its callable to be copyable. std::move_only_function lifts that restriction, so a task can own a unique_ptr or a file handle.

A callable that owns a unique_ptr

✓ compiled & run · g++ 13

```

#include <functional>
#include <iostream>
#include <memory>

int main() {
    // std::function needs copyable callables; this one may own a unique_ptr
    std::move_only_function<int()> job =
        [p = std::make_unique<int>(42)] { return *p; };
    std::cout << job() << '\n';
}

```

OUTPUT

```

42

```

Chapter 3 at a glance

double f(double x, double h = 1.0)	signature with a default argument
const T& / T& / T	read big objects / modify the original / cheap or sink
[=] / [&] / [x = std::move(p)]	capture styles
constexpr / constexpr	may / must run at compile time
static operator() C++23	call operator with no object
auto f(this auto&& self) C++23	explicit object parameter (deducing this)
std::move_only_function<R(A)> C++23	type-erased callable that may own resources

Pointers, Step by Step

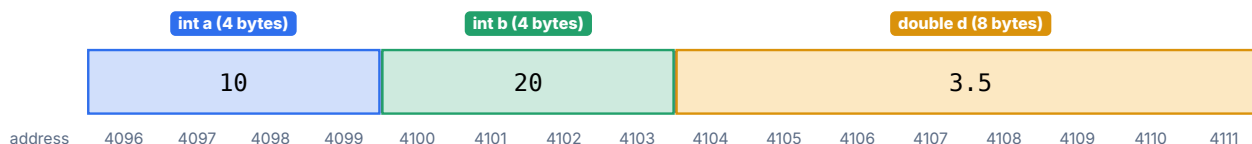
Pointers are the idea beginners find hardest, and almost every difficulty comes from one missing picture: where things live in memory. This chapter builds that picture slowly, one small step at a time, and every step has a diagram and a program you can run.

04

4.1 Start here: memory is a long row of numbered boxes

Forget C++ for a moment. Your computer's memory is a gigantic row of tiny boxes called **bytes**. Every box has a number, its **address**, exactly like the numbers on houses along a street. The boxes hold nothing but numbers; what those numbers *mean* (a whole number, a letter, a decimal) depends on how your program reads them.

MEMORY: BYTES WITH ADDRESSES. A VARIABLE IS A NAME FOR A STRETCH OF BYTES



The number under each box is its address. *a* is the *name* you use in your code for the four bytes starting at address 4096; an `int` takes 4 bytes, a `double` takes 8. (The exact addresses and layout are chosen by the compiler and the system, so yours will differ.)

The program below prints how far apart some variables sit. The numbers are different on every machine, but the idea is always the same: variables are stretches of bytes, each starting at its own address, and each type has a size.

Variables live at addresses; types have sizes

✓ compiled & run

```
#include <cstdint>
#include <iostream>

// Every variable lives at an address. We print addresses RELATIVE to the first variable,
// because the real numbers change from run to run (and look like 0x7ffd5a3c9e14).
int main() {
    int a = 10;
    int b = 20;
    double d = 3.5;
    char c = 'x';

    auto base = reinterpret_cast<std::uintptr_t>(&a);
    auto offset = [&](const auto& v) { return static_cast<long long>(reinterpret_cast<std::uintptr_t>(&v)) - static_cast<long long>(base); };

    std::cout << "sizeof(int)=" << sizeof(int) << " sizeof(double)=" << sizeof(double)
              << " sizeof(char)=" << sizeof(char) << "\n";
    std::cout << "address of a is the reference point (offset 0)\n";
    std::cout << "offset of b: " << offset(b) << " bytes\n";
    std::cout << "offset of d: " << offset(d) << " bytes\n";
    std::cout << "offset of c: " << offset(c) << " bytes\n";
    std::cout << "(the compiler chooses the layout, so these numbers can differ on your machine)\n";
}
```

OUTPUT

```
sizeof(int)=4 sizeof(double)=8 sizeof(char)=1
address of a is the reference point (offset 0)
offset of b: 4 bytes
offset of d: 8 bytes
offset of c: -1 bytes
(the compiler chooses the layout, so these numbers can differ on your machine)
```

The one-sentence idea • Every object you create sits somewhere in that row of boxes, so every object has an address, and **a pointer is simply a variable whose value is one of those addresses**. Everything else in this chapter follows from that sentence.

4.2 A pointer is a variable that holds an address

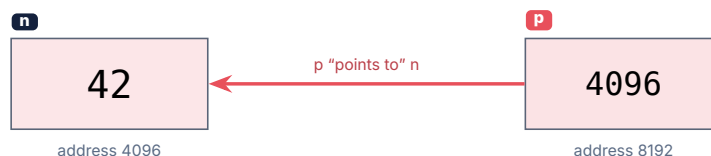
Here are two declarations:

```
int n = 42;      int* p = &n;
```

The first creates an ordinary `int` named `n`. The second creates a different variable, `p`, whose type is “pointer to `int`”, and puts into it the address of `n` (that is what `&n` means: “the address of `n`”).

TWO VARIABLES: `n` HOLDS A NUMBER, `p` HOLDS THE ADDRESS WHERE THAT NUMBER LIVES

```
int n = 42;
int* p = &n;
```



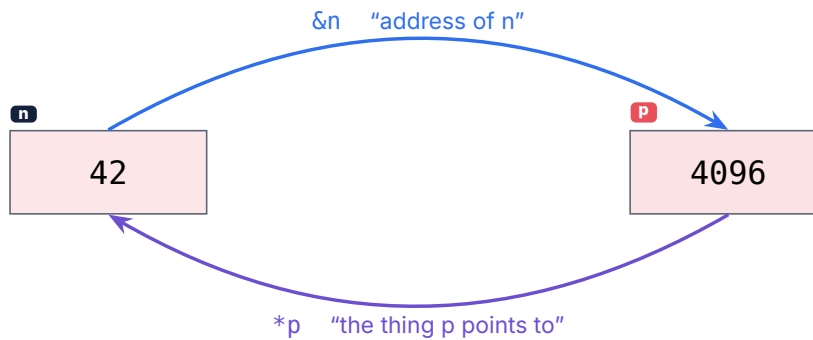
`p` is a normal variable with its own address (8192). Its *value* happens to be 4096, the address of `n`. We draw that as an arrow, but inside the machine the arrow is just that number.

How big is a pointer? • On a 64-bit machine (every Mac you can buy today), a pointer is 8 bytes, *whatever it points to*. A pointer to a `char` and a pointer to a giant `struct` are the same size: both are just an address.

Two symbols, two directions

Pointers come with two operators that are opposites of each other:

& TAKES YOU FROM A VARIABLE TO ITS ADDRESS; * FOLLOWS AN ADDRESS BACK TO THE VARIABLE



&n produces the number 4096 (the address). *p looks at the address stored in p, travels there, and gives you the variable that lives at that address, which is n itself: you can read it, and you can write to it.

The star means two different things • In a *declaration* (`int* p;`) the star is part of the type and means “p is a pointer”. In an *expression* (`*p = 6;`) the star is the operator “follow the pointer”. Same symbol, different jobs; the context tells you which. Likewise, & in a declaration (`int& r = n;`) makes a reference, while in an expression (`&n`) it takes an address.

4.3 Watching a pointer work, one line at a time

Read this program line by line and follow the pictures beneath it. After every statement, ask: “what is in each box now?”

Following and re-aiming a pointer, line by line

✓ compiled & run

```
#include <iostream>

int main() {
    int n = 5;
    int* p = &n;           // 1. p holds the address of n

    std::cout << "n = " << n << ", *p = " << *p << "\n";
    std::cout << "p == &n ? " << std::boolalpha << (p == &n) << "\n";

    *p = 6;                 // 2. follow the arrow and write: n changes
    std::cout << "after *p = 6:  n = " << n << "\n";

    int m = *p + 100;       // 3. follow the arrow and read; m is a separate copy
    std::cout << "m = " << m << " (a copy: changing it does not touch n)\n";

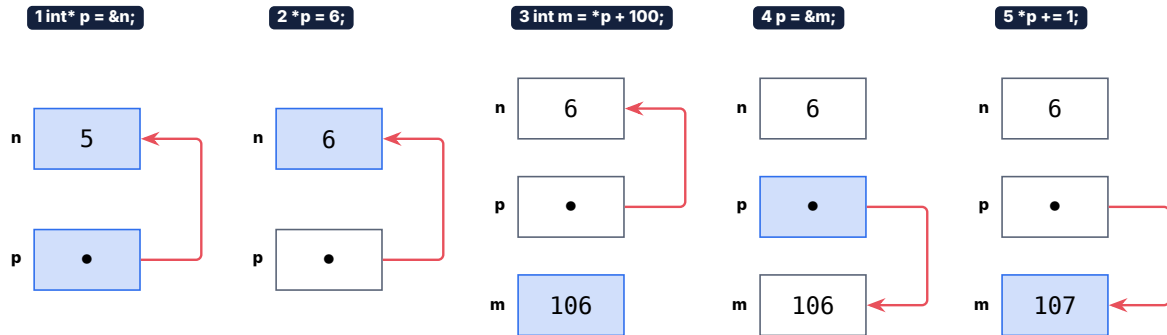
    p = &m;                 // 4. re-aim the pointer; n is left alone
    *p += 1;
    std::cout << "after p = &m; *p += 1:  n = " << n << ", m = " << m << "\n";

    int** pp = &p;         // 5. a pointer to a pointer (rarely needed)
    **pp = 0;
    std::cout << "after **pp = 0:  m = " << m << "\n";
}
```

OUTPUT

```
n = 5, *p = 5
p == &n ? true
after *p = 6:  n = 6
m = 106 (a copy: changing it does not touch n)
after p = &m; *p += 1:  n = 6, m = 107
after **pp = 0:  m = 0
```

THE SAME PROGRAM, FRAME BY FRAME (BLUE = JUST CHANGED)



(Before frame 1: `int n = 5;`.) Frame 2 changes `n` through the pointer. Frame 4 changes the pointer itself (re-aiming it) and leaves `n` alone. Frame 5 now writes to `m`. The difference between `p = ...` (change where it points) and `*p = ...` (change what it points at) is the key distinction to hold onto.

Say it aloud • Read `int* p = &n;` as “`p` is a pointer to `int`, set to the address of `n`”. Read `*p = 6;` as “put 6 where `p` points”. Read `p = &m;` as “make `p` point at `m` instead”. Translating code into sentences is the fastest cure for pointer confusion.

4.4 Why bother? Functions that change their caller's data

When you pass an argument to a function, C++ normally hands over a *copy*. The function works on its own copy and the original is untouched, which is why a “swap” that takes plain `ints` does nothing. Passing the *address* (or a reference, a pointer in disguise) lets the function reach the caller's variable.

Swapping: copies versus pointers versus references ✓ compiled & run

```
#include <iostream>

void swapByValue(int a, int b) { int t = a; a = b; b = t; } // swaps COPIES: no effect
void swapByPointer(int* a, int* b) { int t = *a; *a = *b; *b = t; }
void swapByReference(int& a, int& b) { int t = a; a = b; b = t; }

void addOne(int* p) {
    if (p == nullptr) return; // a pointer may be null, so check
    ++*p;
}

int main() {
    int x = 1, y = 2;
    swapByValue(x, y);
    std::cout << "by value:    x=" << x << " y=" << y << " (unchanged)\n";
    swapByPointer(&x, &y);
    std::cout << "by pointer:  x=" << x << " y=" << y << "\n";
    swapByReference(x, y);
    std::cout << "by reference: x=" << x << " y=" << y << "\n";

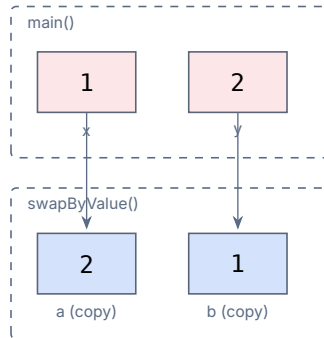
    addOne(&x);
    addOne(nullptr); // safe: the function checks
    std::cout << "after addOne: x=" << x << "\n";
}
```

OUTPUT

```
by value:    x=1 y=2 (unchanged)
by pointer:  x=2 y=1
by reference: x=1 y=2
after addOne: x=2
```

BY VALUE: THE FUNCTION GETS ITS OWN COPIES. BY POINTER: IT GETS THE ADDRESSES

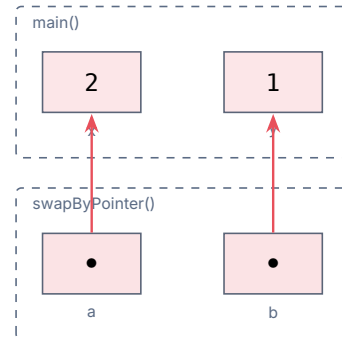
swapByValue(x, y)



The copies were swapped. When the function returns they vanish; x and y never changed.

The function receives addresses. *a and *b are x and y, so swapping them swaps the caller's variables.

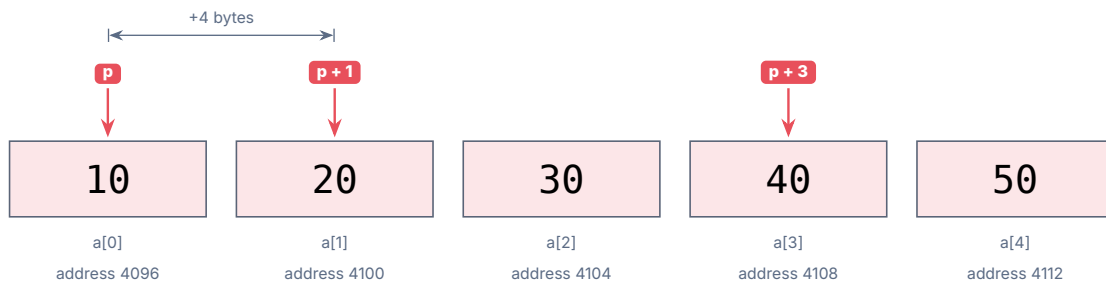
swapByPointer(&x, &y)



Pointer or reference? • For “this must refer to a real object” use a **reference** (`int&`, or `const T&` for read-only access to something big). It is a pointer that cannot be null and needs no *. Use a **pointer** when “no object” is a valid answer (`nullptr`), or when you need to re-aim it later.

4.5 Pointers and arrays: walking through memory

An array is a block of equal-size elements stored one after the other, with no gaps. If you know the address of the first element, you can find any other by arithmetic. That is exactly what pointer arithmetic does, with one friendly twist: adding 1 to a pointer moves it by one *element*, not one byte. The compiler multiplies by the element size for you.

INT ARRAY: `p + 1` MOVES 4 BYTES BECAUSE EACH `int` IS 4 BYTES

`a[i]` is shorthand for `*(a + i)`. That is the whole connection between arrays and pointers: the name of an array, used in an expression, turns into a pointer to its first element (“decays”), and indexing is pointer arithmetic.

Pointer arithmetic, past-the-end pointers, and `sizeof`

✓ compiled & run

```
#include <cstdint>
#include <iostream>

int main() {
    int a[5] = {10, 20, 30, 40, 50};
    int* p = a;                                // the array name becomes &a[0]

    std::cout << "p          = " << *p << "\n";
    std::cout << "*(p + 1) = " << *(p + 1) << "    (p+1 is 4 bytes further: sizeof(int))\n";
    std::cout << "p[3]      = " << p[3] << "    (p[3] is exactly *(p + 3))\n";

    auto bytes = [](const void* from, const void* to) {
        return static_cast<const char*>(to) - static_cast<const char*>(from);
    };
    std::cout << "bytes from &a[0] to &a[1]: " << bytes(&a[0], &a[1]) << "\n";
    std::cout << "bytes from &a[0] to &a[4]: " << bytes(&a[0], &a[4]) << "\n";
}
```

```

int* end = a + 5;                                // one PAST the last element: legal to form, never
dereference
std::cout << "elements between p and end: " << (end - p) << " (a pointer difference counts
elements)\n";

int sum = 0;
for (int* q = p; q != end; ++q) sum += *q;        // walk the array with a pointer
std::cout << "sum = " << sum << "\n";

std::size_t n = sizeof a / sizeof a[0];          // sizeof knows the array length; a pointer does not
int* fresh = a;
std::cout << "sizeof(a) = " << sizeof a << ", sizeof(fresh) = " << sizeof fresh
           << ", elements in a = " << n << "\n";
}

```

OUTPUT

```

*p           = 10
*(p + 1)     = 20  (p+1 is 4 bytes further: sizeof(int))
p[3]        = 40  (p[3] is exactly *(p + 3))
bytes from &a[0] to &a[1]: 4
bytes from &a[0] to &a[4]: 16
elements between p and end: 5 (a pointer difference counts elements)
sum = 150
sizeof(a) = 20, sizeof(fresh) = 8, elements in a = 5

```

Look at the last line of the output. `sizeof(a)` is 20 because the array knows it holds five ints, but `sizeof(fresh)` is 8: once you have a plain pointer, *the length is gone*. A pointer carries no information about how much memory follows it. That lost length is the source of countless bugs.

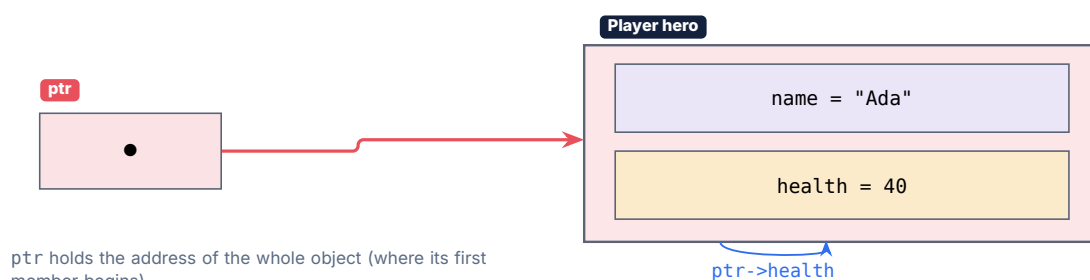
Out-of-bounds is not an error message, it is a gamble • Forming `a + 5` (one past the end) is legal as an end marker, but dereferencing it, or going to `a + 6`, `a[-1]`, or `a[100]` is *undefined behavior*: the program may crash, print garbage, or appear to work today and fail next month. The compiler will not stop you. Prefer `std::vector` and `std::array` (chapter 1), range-based `for`, and `std::span` (below), which remember their length.

`std::span`: a pointer that remembers its length • `std::span<int>` (C++20) is a pointer plus a length packaged together. Pass `std::span<const int>` to a function instead of `(const int* p, int n)` and you cannot mix up the two, and range-`for` works on it. Chapter 8 covers it with the rest of the standard library.

4.6 Pointers to structs: the arrow

Pointers to `struct` and `class` objects are everywhere, so C++ has a shorthand. If `ptr` points to a `Player`, then `(*ptr).health` means “follow the pointer, then take the `health` member”. Writing that with the parentheses gets old fast, so `ptr->health` means exactly the same thing.

THE ARROW OPERATOR: `ptr->health` IS `(*ptr).health`



Reaching members through a pointer with →

✓ compiled & run

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

struct Player {
    std::string name;
    int health = 100;
};

void heal(Player* p, int amount) {
    if (!p) return;
    p->health += amount;           // p->health is shorthand for (*p).health
}

int main() {
    Player hero{"Ada", 40};
    Player* ptr = &hero;

    std::cout << hero.name << " has " << hero.health << " (via the object)\n";
    std::cout << ptr->name << " has " << ptr->health << " (via the pointer)\n";
    std::cout << "(*ptr).health = " << (*ptr).health << " (same thing, wordier)\n";

    heal(ptr, 25);
    std::cout << "after heal: " << hero.health << "\n";
}
```

OUTPUT

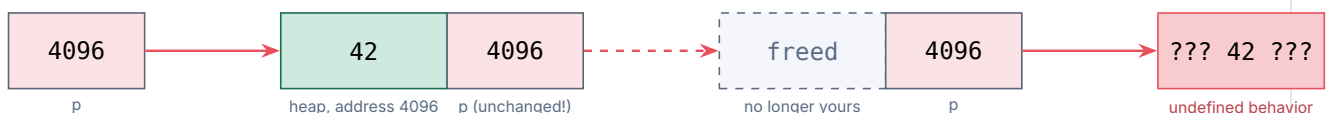
```
Ada has 40 (via the object)
Ada has 40 (via the pointer)
(*ptr).health = 40 (same thing, wordier)
after heal: 65
```

4.7 nullptr, and the pointer that points nowhere good

A pointer needs a legal value too. The special value `nullptr` means “points to nothing”. It is how you say “no object here”, and it lets a function signal “not found”. Always initialize pointers: either to a real address or to `nullptr`. An uninitialized pointer holds a random leftover number, and dereferencing it is a trip to a random address.

There are three classic ways a pointer goes wrong, and all of them are the same disease: **the pointer holds an address, but nothing valid lives there any more.**

THE DANGLING POINTER: THE MEMORY IS GONE BUT THE ADDRESS IS STILL IN THE POINTER

1 `int* p = new int(42);`2 `delete p;`3 `std::cout << *p;`

`delete` hands the memory back but does *not* change `p`. The pointer still holds 4096, so it looks perfectly healthy. Reading through it may print the old 42, print garbage, crash, or work for months and then fail. The danger is that it *sometimes works*.

The other two members of the family look the same: a pointer to a **local variable** whose function has returned (the stack space was reused), and a pointer into a vector that has since grown and moved its storage. Rather than argue about it, let a tool catch it. Build with `-fsanitize=address` and run the faulty program from this chapter's first section:

A use-after-free, caught by AddressSanitizer✓ compiled & run · with `-fsanitize=address`

```
// Deliberate bug: use after free. Compile with: g++ -std=c++23 -g -fsanitize=address
#include <iostream>

int main() {
    int* p = new int{42};
    delete p;           // the memory goes back to the allocator...
    std::cout << *p << "\n"; // ...but p still holds the old address: a DANGLING pointer
}
```

OUTPUT

```
=====
==2668==ERROR: AddressSanitizer: heap-use-after-free on address 0x... at
READ of size 4 at 0x... thread T0
#0 0x... in main snip/c04_asan.cpp:7

0x... is located 0 bytes inside of 4-byte region [0x502000000010,0x502000000014)
freed by thread T0 here:
#0 0x... in operator delete(void*, unsigned long) ../../../../src/libsanitizer/asan/
asan_new_delete.cpp:164
#1 0x... in main snip/c04_asan.cpp:6

previously allocated by thread T0 here:
#0 0x... in operator new(unsigned long) ../../../../src/libsanitizer/asan/asan_new_delete.cpp:95
#1 0x... in main snip/c04_asan.cpp:5
```

The report names the exact line that read freed memory, the line that freed it, and the line that allocated it. Make a habit of building your debug configuration with `-fsanitize=address,undefined`; it turns invisible memory bugs into clear messages.

How to avoid the three classic pointer bugs

Bug	The cure
Uninitialized pointer	Always write <code>int* p = nullptr</code> ; (or a real address) at the declaration
Dangling pointer (freed, expired local, moved vector storage)	Do not keep raw pointers longer than the thing they point at; use <code>unique_ptr</code> for owners, references for borrowers; do not return the address of a local
Out of bounds	Use vector, array, span, range-for; avoid pointer arithmetic in new code
Dereferencing null	Check (<code>if (p)</code>) when null is legal; use references when it is not
Double delete	Do not write <code>delete</code> ; use smart pointers

4.8 const and pointers: read the declaration right to left

Pointer declarations look scary when `const` joins in, but there is a simple, reliable trick: **read from the variable name outward to the left**, translating each symbol: `*` is “pointer to”, `const` is “constant”.

THREE DECLARATIONS, READ RIGHT TO LEFT

<code>const int* p</code>	p is a pointer to a const int	re-aim p? ✓	change *p? ✗
<code>int* const p</code>	p is a const pointer to an int	re-aim p? ✗	change *p? ✓
<code>const int* const p</code>	p is a const pointer to a const int	re-aim p? ✗	change *p? ✗

The three const pointer forms (the errors are commented out)

✓ compiled & run

```
#include <iostream>

int main() {
    int a = 1, b = 2;

    const int* p1 = &a;           // pointer to const int: cannot change *p1, CAN re-aim p1
    // *p1 = 9;                   // error: assignment of read-only location
    p1 = &b;                       // fine

    int* const p2 = &a;           // const pointer to int: CAN change *p2, cannot re-aim p2
    *p2 = 9;                       // fine
    // p2 = &b;                   // error: assignment of read-only variable

    const int* const p3 = &a;      // both locked
    // *p3 = 0;  p3 = &b;          // both errors

    std::cout << *p1 << ' ' << *p2 << ' ' << *p3 << "\n";    // 2 9 9
}
```

OUTPUT
2 9 9

The form you will use constantly is `const T*` (or the reference `const T&`): “I will look at this object but promise not to change it”. It lets you pass large objects to functions without copying them *and* without allowing changes.

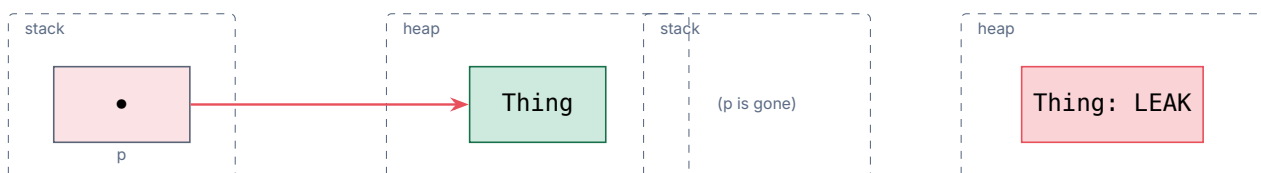
4.9 Pointers on the heap: where new comes in

So far every object lived on the **stack**: variables are created when their block starts and destroyed when it ends (chapter 5 has the full picture). Sometimes you need an object to outlive the function that made it, or its size is only known at run time. For that there is the **heap**: you ask for memory with `new`, get back its address, and keep that address in a pointer. The memory stays yours until you give it back with `delete`.

FORGET delete AND THE MEMORY LEAKS: THE POINTER DIES WITH ITS BLOCK, THE OBJECT DOES NOT

inside the function

after the function returns (no delete)



Nothing points to the Thing any more, so nobody can ever delete it: the memory is lost until the program ends. Do this in a loop and the program slowly eats all the memory. Modern C++ avoids the whole problem: a `std::unique_ptr` deletes its object automatically when it goes out of scope.

new/delete versus unique_ptr: who does the cleanup

✓ compiled & run

```
#include <iostream>
#include <memory>

struct Thing {
    int id;
    explicit Thing(int i) : id(i) { std::cout << " Thing " << id << " created\n"; }
    ~Thing() { std::cout << " Thing " << id << " destroyed\n"; }
};

int main() {
    std::cout << "raw new / delete (you must remember the delete):\n";
    Thing* a = new Thing(1);
    delete a;                       // forget this line and the memory leaks
}
```

```

std::cout << "unique_ptr (cleanup is automatic):\n";
{
    auto b = std::make_unique<Thing>(2);
    std::cout << "    using Thing " << b->id << "\n";
}
// b goes out of scope here: Thing 2 is destroyed
std::cout << "end of main\n";
}

```

OUTPUT

```

raw new / delete (you must remember the delete):
  Thing 1 created
  Thing 1 destroyed
unique_ptr (cleanup is automatic):
  Thing 2 created
  using Thing 2
  Thing 2 destroyed
end of main

```

The modern rule • In new code you should almost never write new or delete. Use `std::make_unique` (one owner), `std::make_shared` (shared ownership), or a `std::vector` (a growable array that manages its own heap memory). Raw pointers are for *borrowing*: “look at this object for a moment”. Chapter 5 covers smart pointers and RAI in depth.

4.10 Pointers to functions and pointers to pointers

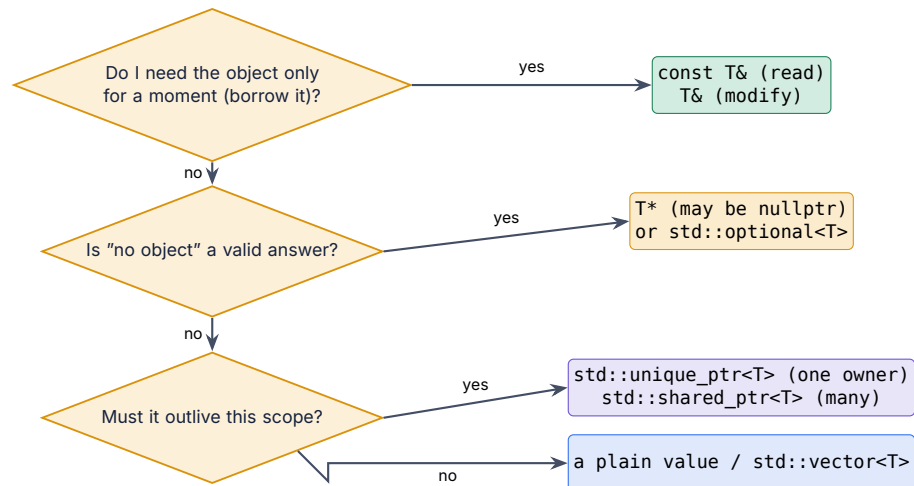
Two more forms appear in real code, and neither is as frightening as it looks.

- **A pointer to a pointer** (`int** pp`) holds the address of a pointer variable. It appears when a function must change *which object a pointer points at* (the function gets the address of your pointer, just like `swapByPointer` got the address of your ints), and in older C-style APIs. The one-more-star rule is the whole idea: `**pp` follows two arrows. In modern C++ a reference to a pointer (`int*&`) or a return value does the job more readably.
- **A function pointer** holds the address of the first instruction of a function, so you can store “what to call” in a variable and call it later. Its syntax is ugly (`int (*fn)(int, int)`), so modern C++ uses `std::function` or a template parameter, or a lambda (chapter 3). C APIs and callbacks to operating-system functions still use raw function pointers.
- **`void*`** is a pointer to “something, type unknown”: you must cast it back to the real type before using it. It shows up in C libraries; avoid it in your own C++ code.

4.11 Choosing the right tool

Most of the time you do not want a raw pointer at all. This diagram collects the decisions from this chapter and the next:

HOW DO I REFER TO THIS THING?



(Hold an array's address and length together with `std::span<T>`.)

Chapter 4 in a nutshell

- Memory is numbered bytes. Every object has an address. A **pointer is a variable that holds an address**; a pointer is 8 bytes on 64-bit systems.
- `&x` gives the address of `x`; `*p` follows the pointer to the object; `p->m` is `(*p).m`. `p = &m` re-aims; `*p = v` writes through.
- Passing an address lets a function change the caller's variable. A reference is a pointer that cannot be null and needs no star.
- `p + 1` moves one *element*; `a[i]` is `*(a + i)`; a pointer does not know how much memory follows it.
- Dangling, uninitialized, out-of-bounds and null pointers are undefined behavior. Initialize, prefer vector/span, and test with `-fsanitize=address`.
- Read declarations right to left (`const int* const p`). Modern code owns heap memory with smart pointers and borrows with references; raw `new/delete` is rare.

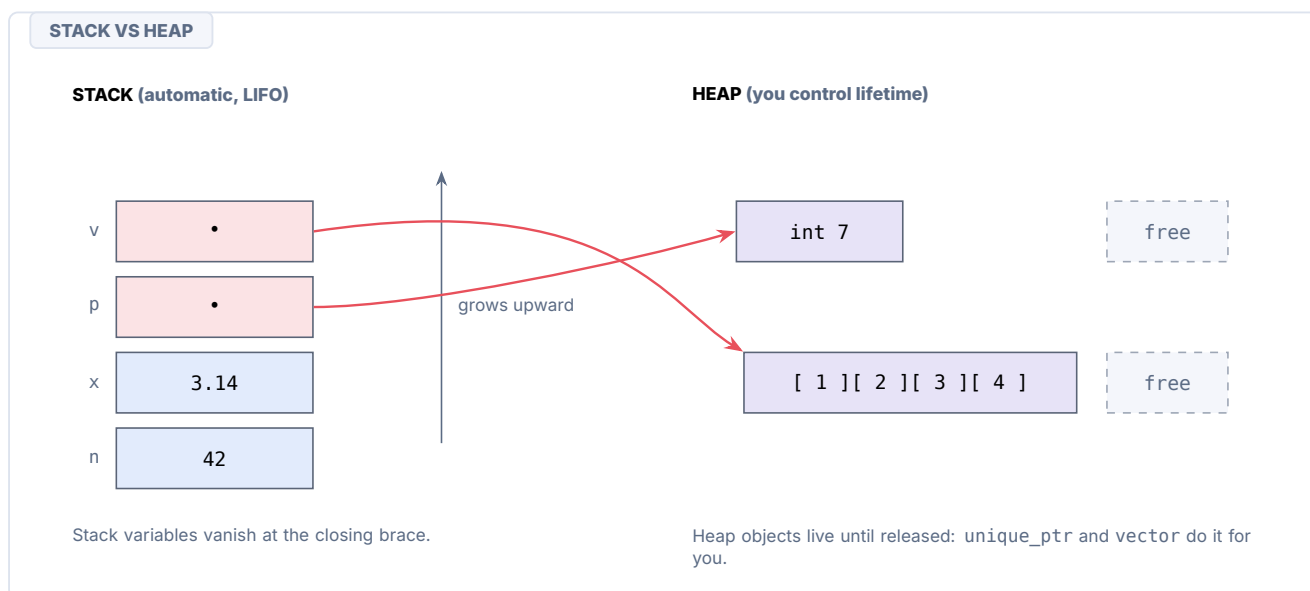
Memory & Ownership

Where your data lives, how to refer to it without copying it, and how modern C++ cleans up after itself automatically.

05

5.1 Stack and heap

Every running program has two main places to keep objects. The **stack** holds local variables: allocation is instant and the memory is released automatically when the enclosing block ends. The **heap** (free store) holds objects whose lifetime you choose; it is larger and more flexible but slower, and somebody must remember to release it. In modern C++ that somebody is a *smart pointer* or a container, never you calling `delete` by hand.



5.2 Pointers and references

Chapter 4 explained pointers slowly; here is the recap. A **pointer** stores the *address* of another object. `&x` takes the address of `x`; `*p` follows a pointer to the object (dereferencing). A **reference** is simply a second *name* for an existing object: it must be initialized, can never be re-seated and cannot be null.

Pointers, references, arrays and `nullptr`

✓ compiled & run · g++ 13

```
#include <iostream>

int main() {
    int n = 5;
    int* p = &n;           // p holds the ADDRESS of n
    *p = 6;                 // dereference: follow the address, write 6 into n
    std::cout << n << ' ' << *p << '\n';

    int& r = n;             // reference: another NAME for n (never null)
    r = 7;
    std::cout << n << '\n';
}
```

```

int arr[4] = {10, 20, 30, 40};
int* q = arr;           // an array decays to a pointer to element 0
std::cout << *(q + 2) << ' ' << q[3] << '\n';    // pointer arithmetic

int* nothing = nullptr; // "points at nothing"; never dereference it!
if (!nothing) std::cout << "null\n";
}

```

OUTPUT

```

6 6
7
30 40
null

```

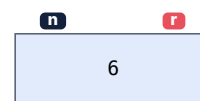
POINTER VS REFERENCE

`int* p = &n;`



Its own object holding an address. Can be reassigned or `nullptr`.
Use `*p` to reach `n`.

`int& r = n;`



No separate storage in your model: `r` is `n`. Always valid; use it like a plain variable.

Pitfall • Never dereference `nullptr`, a pointer to a destroyed object (*dangling*) or one past the end of an array: that is undefined behavior. Prefer references for “must exist” parameters and `std::optional` or a pointer for “may be absent”. Do not return references or pointers to local variables.

5.3 Smart pointers

Header `<memory>`. A smart pointer is an object that owns a heap allocation and frees it in its destructor. **Default to `std::unique_ptr`**; choose `shared_ptr` only when ownership is truly shared, and `weak_ptr` to observe without owning (also to break reference cycles).

unique_ptr, shared_ptr and weak_ptr ✓ compiled & run · g++ 13

```

#include <iostream>
#include <memory>

struct Gadget {
    int id;
    explicit Gadget(int i) : id(i) { std::cout << "make " << id << '\n'; }
    ~Gadget() { std::cout << "free " << id << '\n'; }
};

int main() {
    auto u = std::make_unique<Gadget>(1);           // sole owner
    {
        auto s1 = std::make_shared<Gadget>(2);      // shared owner
        auto s2 = s1;                               // count is now 2
        std::cout << "owners: " << s1.use_count() << '\n';
    }                                                 // both gone -> Gadget 2 freed
    std::weak_ptr<Gadget> w;                         // observes, does not own
    {
        auto s = std::make_shared<Gadget>(3);
        w = s;
        std::cout << "alive: " << !w.expired() << '\n';
    }
}

```

```

    }
    std::cout << "alive: " << !w.expired() << '\n';
} // u freed here

```

OUTPUT

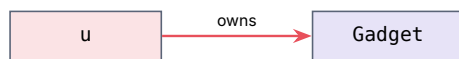
```

make 1
make 2
owners: 2
free 2
make 3
alive: 1
free 3
alive: 0
free 1

```

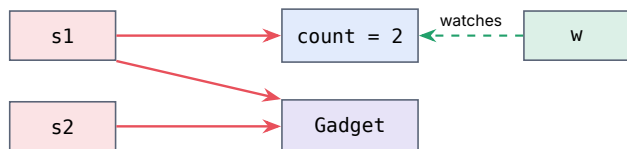
WHO OWNS WHAT

unique_ptr



Exactly one owner. Cannot be copied, only moved. Zero overhead.

shared_ptr + weak_ptr



Last owner out frees the object. `w.lock()` gives a temporary `shared_ptr`, or empty if it is gone.

Remember • Create with `std::make_unique<T>(args...)` and `std::make_shared<T>(args...)`. They are exception-safe and (for shared) do one allocation instead of two. Avoid raw `new` and `delete` in modern code.

5.4 RAI: cleanup you cannot forget

Resource Acquisition Is Initialization: acquire a resource (memory, file, lock, socket) in a constructor and release it in the destructor. C++ runs destructors automatically when a scope ends, in *reverse order* of construction, even if an exception is thrown. `std::vector`, `std::string`, `std::ifstream`, `std::lock_guard` and the smart pointers are all RAI types.

A scope-bound resource

✓ compiled & run · g++ 13

```

#include <iostream>
#include <string>
#include <utility>

class Log {
    std::string name_;
public:
    explicit Log(std::string n) : name_(std::move(n)) {
        std::cout << "open " << name_ << '\n';
    }
    ~Log() { std::cout << "close " << name_ << '\n'; }
    Log(const Log&) = delete; // a Log cannot be copied
    Log& operator=(const Log&) = delete;
};

void work() {
    Log a("A");
    Log b("B");
    std::cout << "working\n";
}

```

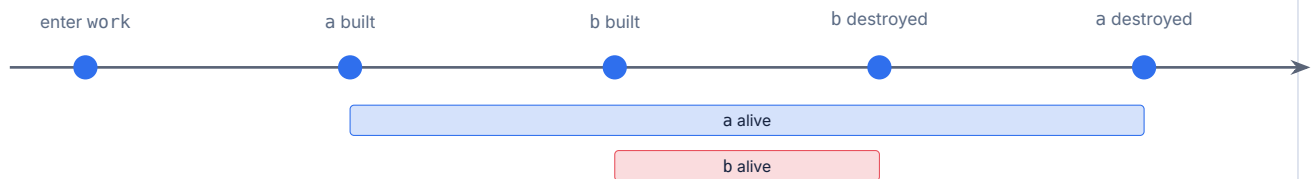
```
} // destroyed in REVERSE order: B, A
```

```
int main() { work(); }
```

OUTPUT

```
open A
open B
working
close B
close A
```

LIFETIME TIMELINE FOR work()



Last in, first out: b is closed before a.

5.5 Move semantics

Copying duplicates a resource; *moving* transfers it, leaving the source valid but unspecified (treat it as empty). Moving is how returning big objects by value stays cheap. `std::move(x)` does not move anything itself: it casts `x` to an *rvalue reference* (T&&) so the move constructor or move assignment is selected.

Copy versus move

✓ compiled & run · g++ 13

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

struct Tracker {
    Tracker() { std::cout << "ctor "; }
    Tracker(const Tracker&) { std::cout << "COPY "; }
    Tracker(Tracker&& noexcept) { std::cout << "move "; }
};

int main() {
    Tracker a;
    Tracker b = a; // copy: duplicates the resource
    Tracker c = std::move(a); // move: steals it (a is left "empty")
    std::cout << '\n';

    std::string big(1000, 'x');
    std::string stolen = std::move(big); // pointer hand-over, no 1000-char copy
    std::cout << stolen.size() << '\n';

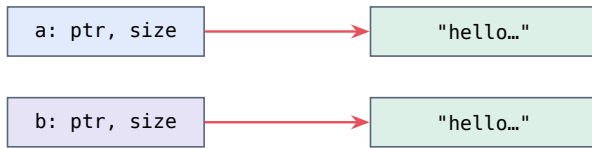
    std::vector<Tracker> v;
    v.reserve(2);
    v.push_back(Tracker{}); // a temporary is moved in
    std::cout << '\n';
    (void)b; (void)c;
}
```

OUTPUT

```
ctor COPY move
1000
ctor move
```

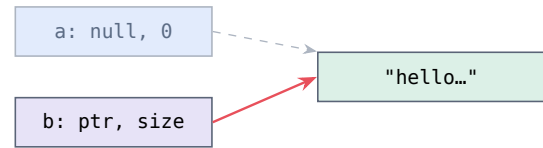
COPY VS MOVE OF A `std::string`

Copy



Two buffers; all characters duplicated. Cost grows with length.

Move



Only the pointer changes hands. Constant time, whatever the length.

The rule of 0, 3 and 5

If your class owns raw resources and you write a destructor, copy operations or move operations yourself, you probably need to write *all five* special members. Better: own resources through RAII members (vector, `unique_ptr`...) and write *none*. That is the **rule of zero**, and it is the goal.

Chapter 5 in a nutshell

- Locals live on the **stack**; heap objects are owned by **smart pointers** or containers.
- A **reference** is another name; a **pointer** holds an address and may be `nullptr`.
- Default to `unique_ptr`; use `shared_ptr` only for genuine sharing; `weak_ptr` observes.
- **RAII**: constructors acquire, destructors release, in reverse order.
- `std::move` enables a cheap transfer; the moved-from object must not be relied upon.
- Aim for the **rule of zero**.

Structs, Classes & OOP

Bundling data with the functions that use it: constructors, access control, inheritance, polymorphism, operators and enums.

06

6.1 Structs

A **struct** groups named *members* into one type. Members default to `public`. A struct with no constructors is an **aggregate**: you can fill it with braces, in order, or by name using *designated initializers* (C++20; names must appear in declaration order).

A struct with defaults, a method and designated initializers

✓ compiled & run · g++ 13

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

struct Player {
    std::string name;
    int hp = 100;           // default member initializers
    int level = 1;

    void heal(int amount) { hp += amount; }
    bool alive() const { return hp > 0; } // const: promises not to modify
};

int main() {
    Player a{"Ana"};           // aggregate initialization
    Player b{.name = "Bo", .level = 7}; // designated initializers (C++20)
    a.heal(25);
    std::cout << a.name << ' ' << a.hp << ' ' << a.level << '\n';
    std::cout << b.name << ' ' << b.hp << ' ' << b.level << ' ' << b.alive() << '\n';
}
```

OUTPUT

```
Ana 125 1
Bo 100 7 1
```

Remember • A trailing `const` on a member function (`bool alive() const`) promises that it will not modify the object. It lets you call it on `const` objects and references, so make every read-only method `const`.

6.2 Classes

A **class** is a struct whose members default to `private`. Convention: use `struct` for plain data bundles, `class` when you have an *invariant* to protect (a rule that must always hold, such as “balance is never negative”). Keep data `private`, expose a small public interface.

Constructor, member initializer list, access control and a static member

✓ compiled & run · g++ 13

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

```

class Account {
public:
    Account(std::string owner, double opening)    // constructor
        : owner_(std::move(owner)), balance_(opening) // member initializer list
    { ++count_; }

    void deposit(double amt) {
        if (amt > 0) balance_ += amt;           // guard the invariant
    }
    double balance() const { return balance_; } // getter
    const std::string& owner() const { return owner_; }
    static int count() { return count_; }        // belongs to the class

private:
    std::string owner_;
    double balance_;
    static inline int count_ = 0;                // one counter, shared
};

int main() {
    Account a("Ana", 50.0), b("Bo", 10.0);
    a.deposit(25);
    a.deposit(-5);
    std::cout << a.owner() << ' ' << a.balance() << ' ' << b.balance() << ' '
                << Account::count() << '\n';
}

```

OUTPUT

Ana 75 10 2

ANATOMY OF AN OBJECT

class Account

public (the interface)

deposit() balance() owner()

private (hidden state)

owner_ balance_

(only member functions touch these)

static (one per class)

count_

shared by all Accounts

Outside code calls a.deposit(10); a.balance_ will not compile.

Constructors and the special members

Constructors run when an object is created. Prefer the **member initializer list** (`: a_(x), b_(y)`) over assigning in the body: it initializes directly, and it is the only way to set `const` and reference members. Members are initialized in *declaration order*, regardless of how you list them.

Delegating, explicit and defaulted special members

✓ compiled & run · g++ 13

```

#include <iostream>
#include <string>
#include <utility>

class Widget {
public:
    Widget() : Widget("unnamed", 0) {} // delegating constructor
    Widget(std::string n, int v) : name_(std::move(n)), v_(v) {}
    explicit Widget(int v) : Widget("num", v) {} // explicit: no silent conversion
}

```

```

Widget(const Widget&)           = default;           // copy
Widget& operator=(const Widget&) = default;
Widget(Widget&&) noexcept       = default;           // move
Widget& operator=(Widget&&) noexcept = default;
~Widget()                      = default;

void show() const { std::cout << name_ << '/' << v_ << '\n'; }
private:
    std::string name_;
    int v_;
};

int main() {
    Widget a;           a.show();
    Widget b("box", 3); b.show();
    Widget c{7};        c.show();
    // Widget d = 7;    // ERROR: the constructor is explicit
}

```

```

OUTPUT
unnamed/0
box/3
num/7

```

THE SIX SPECIAL MEMBER FUNCTIONS

Member	Signature	Runs when...
Default constructor	T()	T t;
Destructor	~T()	the object's lifetime ends
Copy constructor	T(const T&)	T b = a; or pass/return by value
Copy assignment	T& operator=(const T&)	b = a;
Move constructor	T(T&&) noexcept	T b = std::move(a); or from a temporary
Move assignment	T& operator=(T&&) noexcept	b = std::move(a);

Use = default / = delete • Write `= default` to ask for the compiler's version and `= delete` to forbid an operation (for instance, to make a type non-copyable). Mark single-argument constructors `explicit` so `Widget d = 7;` cannot happen by accident.

6.3 Inheritance and polymorphism

Inheritance lets a *derived* class reuse and specialize a *base* class (an *is-a* relationship). A virtual function is looked up at *run time* from the object's real type, so code written against `Shape&` works with every present and future shape. A function declared `= 0` is **pure virtual**; a class with one is **abstract** and cannot be instantiated.

Abstract base class, override, final and runtime dispatch ✓ compiled & run · g++ 13

```

#include <iostream>
#include <memory>
#include <vector>

class Shape {
public:
    virtual ~Shape() = default;           // ALWAYS virtual in a base class
    virtual double area() const = 0;      // pure virtual: Shape is abstract
    virtual const char* name() const { return "shape"; }
};

class Circle : public Shape {

```

```

    double r_;
public:
    explicit Circle(double r) : r_(r) {}
    double area() const override { return 3.14159 * r_ * r_; }
    const char* name() const override { return "circle"; }
};

class Rect final : public Shape {           // final: nobody may derive from Rect
    double w_, h_;
public:
    Rect(double w, double h) : w_(w), h_(h) {}
    double area() const override { return w_ * h_; }
};

int main() {
    std::vector<std::unique_ptr<Shape>> shapes;
    shapes.push_back(std::make_unique<Circle>(1.0));
    shapes.push_back(std::make_unique<Rect>(2.0, 3.0));
    for (const auto& s : shapes)           // one loop, many behaviors
        std::cout << s->name() << ' ' << s->area() << '\n';
}

```

OUTPUT

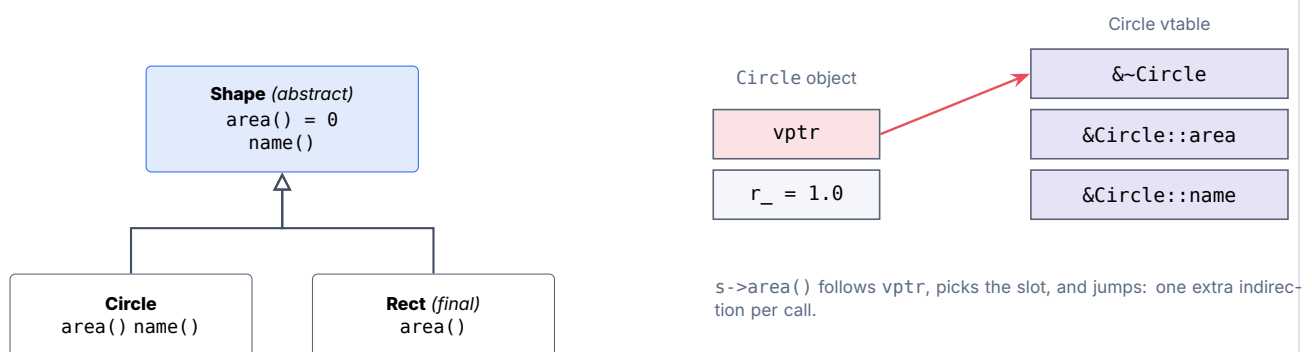
```

circle 3.14159
shape 6

```

CLASS HIERARCHY AND THE VIRTUAL TABLE

Each polymorphic object holds a hidden pointer to its class's table:



Pitfall • (1) Give every base class a virtual destructor, or deleting a derived object through a base pointer is undefined behavior. (2) Always write `override`: the compiler then catches signature typos. (3) Keep polymorphic objects behind pointers or references: passing a `Circle` to a `Shape` *by value* slices off the derived part.

Prefer composition • Ask *is-a* or *has-a*? A Car *has an* Engine: store one as a member. Inherit only for genuine substitutability, and keep hierarchies shallow.

6.4 Operator overloading

You can give your types the familiar operators. Keep the meaning natural (+ adds, == compares) and mirror the built-ins. C++20 made comparisons easy: declare `operator<=>` as `=` default and you get `<`, `>`, `<=`, `>=`; add a defaulted `==` for equality.

Arithmetic, defaulted comparisons and stream output

✓ compiled & run · g++ 13

```
#include <compare>
#include <iostream>

struct Vec2 {
    double x = 0, y = 0;

    Vec2 operator+(const Vec2& o) const { return {x + o.x, y + o.y}; }
    Vec2& operator*=(double k)          { x *= k; y *= k; return *this; }
    bool operator==(const Vec2& const) = default;           // member-wise ==
    auto operator<=>(const Vec2& const) = default;           // gives < > <= >= for free
};

std::ostream& operator<<(std::ostream& os, const Vec2& v) {
    return os << '(' << v.x << ", " << v.y << ')';
}

int main() {
    Vec2 a{1, 2}, b{3, 4};
    Vec2 c = a + b;
    c *= 2;
    std::cout << c << ' ' << (a == b) << ' ' << (a < b) << '\n';
}
```

OUTPUT

(8, 12) 0 1

C++23 Multidimensional operator[]

Before C++23, `operator[]` took exactly one argument, forcing `grid(r, c)` or `grid[r][c]` with proxy objects. Now it may take any number, so `g[1, 2]` works.

operator[] with two indices

✓ compiled & run · g++ 13

```
#include <iostream>
#include <vector>

class Grid {
    int w_, h_;
    std::vector<int> cells_;
public:
    Grid(int w, int h) : w_(w), h_(h), cells_(w * h, 0) {}
    int& operator[](int row, int col) { // C++23: several indices
        return cells_[row * w_ + col];
    }
    int height() const { return h_; }
};

int main() {
    Grid g(3, 2);
    g[1, 2] = 9;
    g[0, 0] = 4;
    std::cout << g[1, 2] << ' ' << g[0, 0] << ' ' << g.height() << '\n';
}
```

OUTPUT

9 4 2

C++23 **Deducing this**

An *explicit object parameter* (`this Self&& self`) lets one template replace the usual `const`/`non-const`/`lvalue`/`rvalue` overload set, and lets lambdas call themselves recursively.

One getter instead of four, and a recursive lambda

✓ compiled & run · clang++ 18

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

struct Box {
    std::string text{"hi"};

    // ONE function replaces the const / non-const / lvalue / rvalue overloads
    template <class Self>
    auto&& get(this Self&& self) { return std::forward<Self>(self).text; }
};

int main() {
    Box b;
    const Box cb;
    b.get() += "!"; // non-const object: may modify
    std::cout << b.get() << ' ' << cb.get() << '\n';

    // A lambda that can call ITSELF, with no std::function needed:
    auto fib = [] (this auto self, int n) -> int {
        return n < 2 ? n : self(n - 1) + self(n - 2);
    };
    std::cout << fib(10) << '\n';
}
```

OUTPUT

```
hi! hi
55
```

6.5 Enumerations

Use **enum class**: its names are scoped (`Level::Mid`) and it never converts to an integer by accident. You may choose the underlying type. **C++23** `std::to_underlying` converts to that type safely.

Scoped enums and std::to_underlying

✓ compiled & run · g++ 13

```
#include <iostream>
#include <utility>

enum class Level : unsigned char { Low = 1, Mid = 5, High = 10 };
enum Old { A, B }; // plain enum: leaks names, converts to int silently (avoid)

int main() {
    Level l = Level::Mid;
    // int bad = l; // ERROR: no implicit conversion
    int n = static_cast<int>(l); // the old way
    auto m = std::to_underlying(l); // C++23: shorter, can't pick wrong type
    std::cout << n << ' ' << +m << ' ' << (l == Level::Mid) << ' ' << A << B << '\n';
}
```

OUTPUT

```
5 5 1 01
```

6.6 Vocabulary types: optional and variant

Header `<optional>`: `std::optional<T>` holds a `T` or *nothing*, an honest replacement for magic values like `-1`. Header `<variant>`: `std::variant<A, B>` holds *exactly one* of several types (a type-safe union); `std::visit` applies a function to whichever is active.

optional and variant

✓ compiled & run · g++ 13

```
#include <iostream>
#include <optional>
#include <string>
#include <variant>

std::optional<int> parse(const std::string& s) {    // "an int, or nothing"
    try { return std::stoi(s); }
    catch (...) { return std::nullopt; }
}

int main() {
    std::cout << parse("42").value_or(-1) << ' ' << parse("abc").value_or(-1) << '\n';

    std::variant<int, std::string> v = 7;          // holds exactly ONE of these
    v = std::string("seven");
    std::visit([](const auto& x) { std::cout << x << '\n'; }, v);
    std::cout << std::holds_alternative<std::string>(v) << '\n';
}
```

OUTPUT
42 -1
seven
1

6.7 How a struct sits in memory

The compiler aligns each member to a multiple of its own size, inserting invisible *padding*. Ordering members from largest to smallest often shrinks the object. Use `sizeof` and `alignof` to check.

Padding and member order

✓ compiled & run · g++ 13

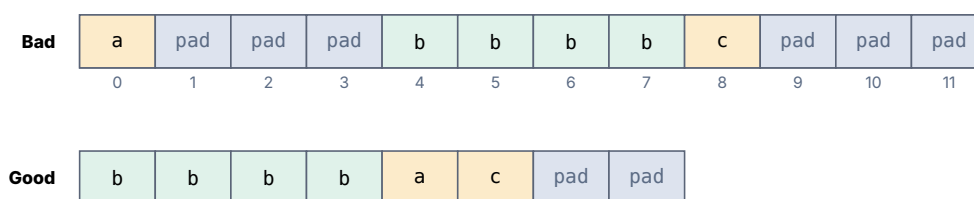
```
#include <iostream>

struct Bad { char a; int b; char c; };    // padding wastes space
struct Good { int b; char a; char c; };   // big members first

int main() {
    std::cout << sizeof(Bad) << ' ' << sizeof(Good) << ' ' << alignof(int) << '\n';
}
```

OUTPUT
12 8 4

BYTE LAYOUT OF Bad (12 BYTES) VS Good (8 BYTES)



An `int` must start on a 4-byte boundary, so the compiler pads after the first `char` and after the last, so arrays of the struct stay aligned.

Chapter 6 in a nutshell

- `struct` = public data bundle; `class` = private state plus an interface that guards invariants.
- Initialize members in the **initializer list**; mark one-argument constructors `explicit`.
- Mark read-only methods `const`; aim for the **rule of zero**.
- Polymorphism needs `virtual` (and a virtual destructor), `override`, and pointers or references.
- Default `operator<=>` and `==` give all comparisons; C++23 adds `operator[]` with many indices and *deducing this*.
- Prefer `enum class`, `optional` and `variant` over magic numbers and raw unions.

Templates & Concepts

Writing code once for many types, and telling the compiler exactly which types are allowed.

07

7.1 Function and class templates

A **template** is a blueprint with type placeholders (template <typename T>). The compiler stamps out a real function or class for each type you use: this is *instantiation*, and it happens at compile time, so there is no run-time cost. Function template arguments are usually deduced (max(3, 4)); for class templates you write them (Stack<int>), though C++17 deduction often lets you omit them.

A class template: Stack<T>

✓ compiled & run · g++ 13

```
#include <iostream>
#include <stdexcept>
#include <string>
#include <utility>
#include <vector>

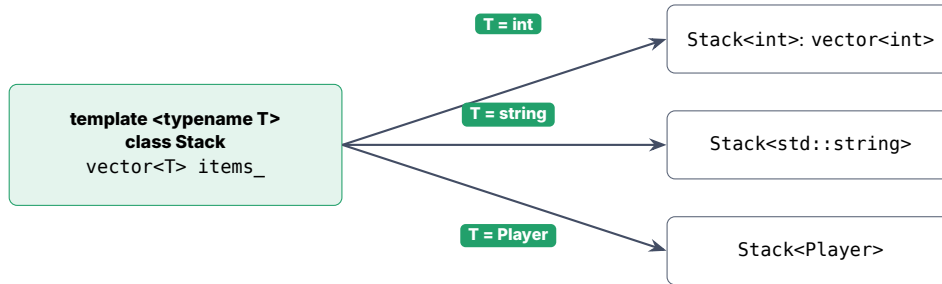
template <typename T>
class Stack {
    std::vector<T> items_;
public:
    void push(T v) { items_.push_back(std::move(v)); }
    T pop() {
        if (items_.empty()) throw std::out_of_range("empty stack");
        T top = std::move(items_.back());
        items_.pop_back();
        return top;
    }
    bool empty() const { return items_.empty(); }
    std::size_t size() const { return items_.size(); }
};

int main() {
    Stack<int> s;
    s.push(1); s.push(2); s.push(3);
    std::cout << s.pop() << s.pop() << ' ' << s.size() << '\n';
    Stack<std::string> names; // same code, different T
    names.push("ann");
    std::cout << names.pop() << '\n';
}
```

OUTPUT

```
32 1
ann
```

ONE TEMPLATE, MANY CLASSES



The compiler writes each version for you at compile time. Template code must be visible to the compiler, so it normally lives entirely in headers.

Pitfall • Template errors used to be walls of text pointing deep inside the library. Concepts (next) fix that. Also remember that members of a template are only checked when instantiated, so an unused broken member may compile silently.

7.2 Concepts

C++20 A **concept** is a named, compile-time *requirement* on a template argument. Put it where `typename` would be (template <Number T>) or after the parameter list with `requires`. If the type does not satisfy it, you get a short, readable error at the call site. The header <concepts> supplies ready-made ones: `std::integral`, `std::floating_point`, `std::same_as`, `std::convertible_to`, `std::invocable`, `std::copyable` and more.

Defining and using concepts

✓ compiled & run · g++ 13

```

#include <concepts>
#include <iostream>
#include <string>

template <typename T>
concept Printable = requires(const T& x, std::ostream& os) {
    { os << x } -> std::same_as<std::ostream&>; // "x can be streamed"
};

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

template <Printable T>
void show(const T& x) { std::cout << "<" << x << ">\n"; }

template <Number T>
T average(T a, T b) { return (a + b) / 2; }

int main() {
    show(42);
    show(std::string("hi"));
    std::cout << average(3, 8) << ' ' << average(3.0, 8.0) << '\n';
    // average(std::string("a"), std::string("b")); // ERROR: readable message
}
  
```

OUTPUT

```

<42>
<hi>
5 5.5
  
```

FOUR EQUIVALENT WAYS TO CONSTRAIN A TEMPLATE

Style	Example
Constrained parameter	<code>template <Number T> T f(T a);</code>
requires clause	<code>template <typename T> requires Number<T> T f(T a);</code>
Trailing requires	<code>template <typename T> T f(T a) requires Number<T>;</code>
Abbreviated (auto)	<code>auto f(Number auto a);</code>

7.3 Variadic templates and fold expressions

A **parameter pack** (`typename... Ts`, `Ts... xs`) accepts any number of arguments. A **fold expression** applies an operator across the pack: `(xs + ...)` sums them, `(std::cout << ... << xs)` streams them, `(xs, ...)` runs something for each.

Parameter packs and folds

✓ compiled & run · g++ 13

```
#include <iostream>

template <typename... Ts>
void print_all(const Ts&... xs) {
    ((std::cout << xs << ' '), ...); // fold over the comma operator
    std::cout << '\n';
}

template <typename... Ts>
auto total(Ts... xs) { return (xs + ... + 0); } // fold with a starting value

int main() {
    print_all(1, "two", 3.5, 'c');
    std::cout << total(1, 2, 3) << ' ' << total() << '\n';
}
```

OUTPUT

```
1 two 3.5 c
6 0
```

HOW A FOLD EXPANDS FOR `total(1, 2, 3)`

`(xs + ... + 0)` $\longrightarrow$ `1 + (2 + (3 + 0))` $\longrightarrow$ 6

With an empty pack, only the starting value remains, so `total()` returns 0.

Rule of thumb • Reach for a template when the *same algorithm* must work on many types. For different *behavior* per type chosen at run time, use virtual functions instead (Chapter 5). To branch on a type at compile time inside a template, use `if constexpr` (Chapter 2).

Chapter 7 in a nutshell

- `template <typename T>` writes code once; the compiler instantiates it per type.
- Constrain templates with **concepts** to get clear errors and self-documenting interfaces.
- Parameter packs plus **fold expressions** replace recursive variadic tricks.
- Templates usually live in headers; prefer standard concepts from `<concepts>`.

The Standard Library

Containers, iterators, algorithms and views: the ready-made toolbox you should reach for before writing your own.

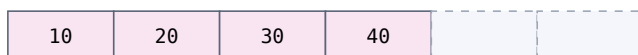
08

8.1 Containers at a glance

A **container** stores a collection of objects and cleans up after itself (RAII). Pick by how you will *use* the data. When unsure, start with `std::vector`: it is contiguous in memory, which modern CPUs love, and it beats fancier structures surprisingly often.

HOW THE MAIN CONTAINERS ARE LAID OUT IN MEMORY

vector: one contiguous block (grows by reallocating)



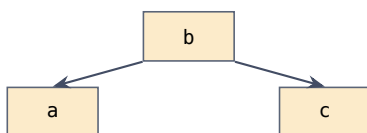
size 4, capacity 6: spare room at the end

List: separate nodes linked by pointers

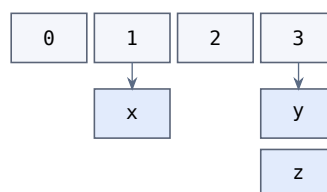


nodes scattered: slow to walk

map / set: balanced tree, kept sorted



unordered_map: hash table (buckets)



map, unordered_map, set, stack, queue, priority_queue

✓ compiled & run · g++ 13

```
#include <iostream>
#include <map>
#include <queue>
#include <set>
#include <stack>
#include <string>
#include <unordered_map>

int main() {
    std::map<std::string, int> sorted{{"b", 2}, {"a", 1}}; // ordered by key
    sorted["c"] = 3;
    for (auto& [k, v] : sorted) std::cout << k << v << ' ';
    std::cout << '\n';

    std::unordered_map<std::string, int> fast{{"x", 10}}; // hash table, unordered
    fast["y"] = 20;
    std::cout << fast.at("y") << ' ' << fast.contains("z") << '\n';
}
```

```

std::set<int> uniq{3, 1, 3, 2, 1}; // sorted, no duplicates
std::cout << uniq.size() << ' ' << *uniq.begin() << '\n';

std::stack<int> st;          st.push(1); st.push(2); // LIFO: last in, first out
std::queue<int> q;          q.push(1); q.push(2);   // FIFO: first in, first out
std::priority_queue<int> pq; pq.push(5); pq.push(9); pq.push(1); // biggest first
std::cout << st.top() << q.front() << pq.top() << '\n';
}

```

OUTPUT

```

a1 b2 c3
20 0
3 1
219

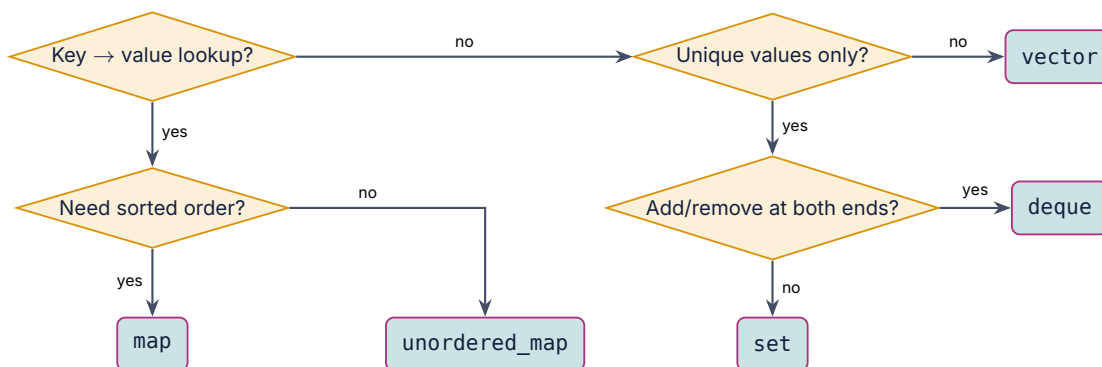
```

Speed cheat sheet

Container	Index	Push back	Insert middle	Find	Keeps order?
array<T,N>	O(1)	—	—	O(n)	as inserted
vector	O(1)	O(1)*	O(n)	O(n)	as inserted
deque	O(1)	O(1)	O(n)	O(n)	as inserted
list	—	O(1)	O(1) [†]	O(n)	as inserted
set / map	—	—	O(log n)	O(log n)	sorted
unordered_*	—	—	O(1)*	O(1)*	none
flat_map C++23	—	—	O(n)	O(log n)	sorted

* amortized or average case. † once you hold an iterator to the spot.

WHICH CONTAINER SHOULD I USE?



(the set branch is for unique, sorted items; otherwise default to vector)

Remember • Subscripting a map with a missing key *inserts* a default value. Use `.at(key)` (throws), `.find(key)` or `.contains(key)` (C++20) when you only want to look.

8.2 Iterators

An **iterator** is a pointer-like handle to an element. `begin()` points at the first element and `end()` one *past* the last (never dereference it). Every container provides iterators, which is what lets one algorithm work on all of them. vector iterators can jump (`it + 2`); list iterators step one at a time (`std::advance`).

Walking, jumping and reversing

✓ compiled & run · g++ 13

```

#include <iostream>
#include <iterator>
#include <list>
#include <vector>

int main() {
    std::vector<int> v{10, 20, 30, 40};
    for (auto it = v.begin(); it != v.end(); ++it)    // end() = one PAST the last
        std::cout << *it << ' ';
    std::cout << '\n';

    auto mid = v.begin() + 2;                        // vector: jump anywhere
    std::cout << *mid << ' ' << std::distance(v.begin(), mid) << '\n';

    std::list<int> l{1, 2, 3};
    auto it = l.begin();
    std::advance(it, 2);                             // list: step by step
    l.insert(it, 99);                                // insert before it: cheap
    for (int x : l) std::cout << x << ' ';
    std::cout << '\n';

    for (auto r = v.rbegin(); r != v.rend(); ++r)    // reverse iterators
        std::cout << *r << ' ';
    std::cout << '\n';
}

```

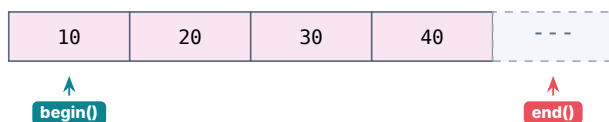
OUTPUT

```

10 20 30 40
30 2
1 2 99 3
40 30 20 10

```

THE HALF-OPEN RANGE [begin, end)



end() marks the stop; an empty range has begin() == end().

8.3 Algorithms

Header `<algorithm>` (plus `<numeric>`). Algorithms name the *intent* (sort, find, count, transform) and are tested and optimized. Since C++20, the `std::ranges::` versions take the whole container (`std::ranges::sort(v)`) instead of a begin/end pair. **C++23** adds `ranges::contains`, `fold_left`, `fold_right`, `starts_with`, `ends_with`, `find_last` and more.

Sort, find, count, accumulate, erase, and two C++23 additions

✓ compiled & run · g++ 13

```

#include <algorithm>
#include <functional>
#include <iostream>
#include <numeric>
#include <vector>

int main() {
    std::vector<int> v{5, 3, 8, 1, 9, 2};

    std::ranges::sort(v);                                // 1 2 3 5 8 9
    auto it = std::ranges::find(v, 8);                  // iterator to the 8
    std::cout << (it - v.begin()) << ' ';                // its index: 4
    std::cout << std::ranges::count_if(v, [](int x) { return x > 3; }) << ' ';
    std::cout << std::accumulate(v.begin(), v.end(), 0) << ' '; // sum: 28
}

```

```

std::cout << std::ranges::max(v) << '\n';

std::erase_if(v, [](int x) { return x % 2 == 0; });           // remove the evens
for (int x : v) std::cout << x << ' ';
std::cout << '\n';

std::cout << std::ranges::contains(v, 9) << ' '              // C++23
          << std::ranges::fold_left(v, 0, std::plus<>{}) << '\n'; // C++23
}

```

OUTPUT

```

4 3 28 9
1 3 5 9
1 18

```

ALGORITHM CHEAT SHEET

I want to...	Use...
Sort / partially sort	<code>ranges::sort(v)</code> , <code>ranges::partial_sort</code> , <code>ranges::stable_sort</code>
Find an element / condition	<code>ranges::find(v, x)</code> , <code>ranges::find_if(v, pred)</code>
Test membership	<code>ranges::contains(v, x)</code> C++23
Count matches	<code>ranges::count(v, x)</code> , <code>ranges::count_if(v, pred)</code>
Any / all / none satisfy	<code>ranges::any_of</code> , <code>all_of</code> , <code>none_of</code>
Transform each element	<code>ranges::transform(in, out, f)</code>
Remove items	<code>std::erase(v, x)</code> , <code>std::erase_if(v, pred)</code>
Sum / reduce	<code>std::accumulate</code> , <code>ranges::fold_left</code> C++23
Min, max, clamp	<code>ranges::min</code> , <code>ranges::max</code> , <code>std::clamp(x, lo, hi)</code>
Reverse, rotate, unique	<code>ranges::reverse</code> , <code>ranges::rotate</code> , <code>ranges::unique</code>
Binary search (sorted input)	<code>ranges::binary_search</code> , <code>lower_bound</code>

Pitfall • `binary_search` and `lower_bound` require *sorted* data. Modifying a container (push, erase) can invalidate iterators and references into it: do not keep using an old iterator after growing a vector.

8.4 Views without owning: span

`std::span<T>` is a non-owning window onto a contiguous run of elements (pointer plus length). One function can accept a vector, array, or C array without copying. Use `span<const T>` for read-only access. It is the modern replacement for “pointer and size” parameter pairs.

One function, any contiguous sequence

✓ compiled & run · g++ 13

```

#include <iostream>
#include <span>
#include <vector>

int sum(std::span<const int> xs) {           // accepts vector, array, C array...
    int s = 0;
    for (int x : xs) s += x;
    return s;
}

int main() {
    std::vector<int> v{1, 2, 3, 4, 5};
    int raw[3] = {10, 20, 30};
    std::cout << sum(v) << ' ' << sum(raw) << ' '
              << sum(std::span{v}.subspan(1, 3)) << '\n'; // just {2,3,4}
}

```

OUTPUT

15 60 9

8.5 **C++23** **New library pieces**

These need a recent compiler and library (see Chapter 0). Their outputs below are worked out by hand and marked as not run here.

std::flat_map and std::flat_set

Same interface as map and set, but stored as sorted contiguous arrays: much faster to iterate and more cache-friendly, with slower insertion in the middle. Good for small or read-mostly collections.

std::flat_map

◇ newer toolchain

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

int main() {
    // Same interface as std::map, but stored as two sorted vectors:
    // faster to iterate, smaller in memory, slower to insert in the middle.
    std::flat_map<std::string, int> m{{"b", 2}, {"a", 1}};
    m["c"] = 3;
    for (const auto& [k, v] : m) std::cout << k << v << ' '; // a1 b2 c3
}
```

EXPECTED OUTPUT (not run here: needs GCC 14+ / newer libc++)

a1 b2 c3

std::mdspan

A multi-dimensional, non-owning view over flat memory. Together with C++23's multi-index operator[] it gives you `m[row, col]` over a plain vector.

std::mdspan

◇ newer toolchain

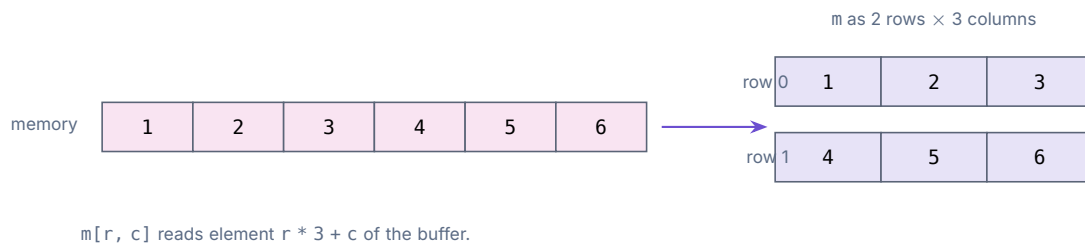
```
#include <iostream>
#include <mdspan>
#include <vector>

int main() {
    std::vector<int> data{1, 2, 3, 4, 5, 6};
    std::mdspan m(data.data(), 2, 3); // view flat memory as 2 rows x 3 cols
    for (std::size_t r = 0; r < m.extent(0); ++r) {
        for (std::size_t c = 0; c < m.extent(1); ++c)
            std::cout << m[r, c] << ' '; // multi-index operator[]
        std::cout << '\n';
    }
}
```

EXPECTED OUTPUT (not run here: needs GCC 14+ / newer libc++)

1 2 3
4 5 6

mdspan: ONE FLAT BUFFER, TWO DIMENSIONS

**std::generator (lazy sequences)**

A coroutine that produces values one at a time with `co_yield`. It computes the next value only when asked, so infinite sequences are fine; combine with `views::take`.

A lazy, infinite Fibonacci generator

◇ newer toolchain

```
#include <generator>
#include <iostream>
#include <ranges>

std::generator<int> fib() {                // a coroutine: pauses at each co_yield
    int a = 0, b = 1;
    while (true) {
        co_yield a;
        auto next = a + b;
        a = b;
        b = next;
    }
}

int main() {
    for (int x : fib() | std::views::take(8)) std::cout << x << ' ';
}
```

EXPECTED OUTPUT (not run here: needs GCC 14+ / newer libc++)

0 1 1 2 3 5 8 13

std::ranges::to

Finishes a lazy pipeline by collecting it into a real container.

From view pipeline to vector

◇ newer toolchain

```
#include <iostream>
#include <ranges>
#include <vector>

int main() {
    namespace rv = std::views;
    auto squares = rv::iota(1, 6)
        | rv::transform([](int x) { return x * x; })
        | std::ranges::to<std::vector>();    // C++23: pipeline -> container
    for (int x : squares) std::cout << x << ' ';    // 1 4 9 16 25
}
```

EXPECTED OUTPUT (not run here: needs GCC 14+ / newer libc++)

1 4 9 16 25

Chapter 8 in a nutshell

- Default to `std::vector`; use `map/set` for sorted keys, `unordered_*` for fast lookup.
- Iterators connect containers to algorithms; `end()` is one past the last element.
- Prefer `std::ranges::` algorithms and `erase_if` over hand-written loops.
- `span` borrows a contiguous range; `string_view` borrows text.
- C++23 adds `flat_map`, `mdspan`, `generator`, `ranges::to`, `fold_left` and `contains`.

Errors & Robustness

Things go wrong: files vanish, input is junk, indexes overflow. Choose the right tool to report, handle and prevent problems.

09

9.1 Exceptions

An **exception** reports an error that the current function cannot handle. `throw` unwinds the stack, running destructors (RAII!) on the way, until a matching catch is found. Catch *by const reference*, most specific type first. Standard exceptions derive from `std::exception`, which offers `what()`.

throw, try and catch

✓ compiled & run · g++ 13

```
#include <iostream>
#include <stdexcept>

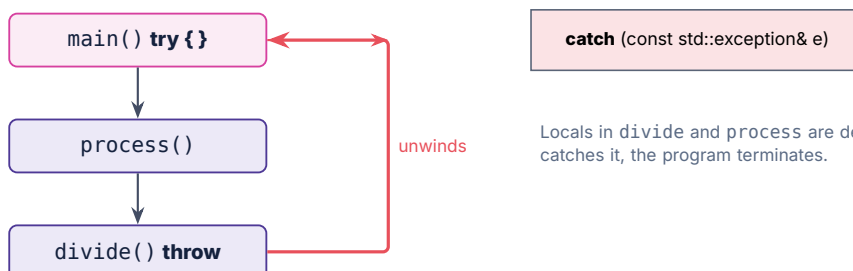
double divide(double a, double b) {
    if (b == 0) throw std::invalid_argument("divide by zero");
    return a / b;
}

int main() {
    try {
        std::cout << divide(6, 3) << '\n';
        std::cout << divide(1, 0) << '\n';           // throws: the rest is skipped
        std::cout << "never printed\n";
    } catch (const std::invalid_argument& e) {        // catch by const reference
        std::cout << "caught: " << e.what() << '\n';
    } catch (const std::exception& e) {              // any other standard exception
        std::cout << "other: " << e.what() << '\n';
    }
    std::cout << "done\n";
}
```

OUTPUT

```
2
caught: divide by zero
done
```

HOW A THROW TRAVELS UP THE CALL STACK



Locals in `divide` and `process` are destroyed on the way up. If nobody catches it, the program terminates.

Pitfall • Never `throw` from a destructor. Do not use exceptions for ordinary control flow ("key not found" in a loop). Mark functions that cannot throw `noexcept`; move operations should be `noexcept` so

containers use them.

9.2 C++23 `std::expected`: errors as values

`std::expected<T, E>` holds *either* a value of type `T` or an error of type `E`. The error is part of the function's signature, so callers cannot ignore it by accident, and there is no stack unwinding cost. Return `std::unexpected(err)` to fail. Chain with `.transform()` (map the value), `.and_then()` (next step that may fail) and `.or_else()` (recover).

Returning a value or an error

✓ compiled & run · g++ 13

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

enum class ParseError { Empty, NotANumber };

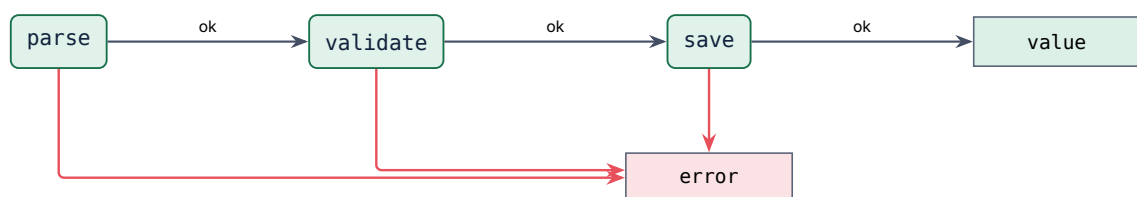
std::expected<int, ParseError> parse(const std::string& s) {
    if (s.empty()) return std::unexpected(ParseError::Empty);
    for (char c : s)
        if (c < '0' || c > '9') return std::unexpected(ParseError::NotANumber);
    return std::stoi(s); // success value
}

int main() {
    for (const char* text : {"42", "", "4x"}) {
        auto r = parse(text);
        if (r) std::cout << "ok " << *r << '\n';
        else std::cout << "error " << static_cast<int>(r.error()) << '\n';
    }
    auto doubled = parse("21").transform([](int n) { return n * 2; }); // chaining
    std::cout << doubled.value_or(-1) << '\n';
}
```

OUTPUT

```
ok 42
error 0
error 1
42
```

expected CHAINING: THE FIRST ERROR SHORT-CIRCUITS



Any step that fails jumps straight to the error; later steps are skipped.

Exceptions or expected? • Use **exceptions** for rare, unexpected failures (out of memory, broken invariant) or where there is no sensible local recovery. Use **expected** (or optional) for *anticipated* failures such as bad user input or missing files that the caller will routinely handle.

9.3 optional chaining

C++23 `std::optional` gained monadic operations: `.transform(f)` applies `f` if a value is present, `.and_then(f)` chains a function that itself returns an `optional`, and `.or_else(f)` supplies a fallback. This replaces nested `if (x.has_value())` ladders.

transform and or_else

✓ compiled & run · g++ 13

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

std::optional<std::string> find_user(int id) {
    if (id == 1) return "ann";
    return std::nullopt;
}

int main() {
    for (int id : {1, 2}) {
        auto len = find_user(id)
            .transform([](const std::string& s) { return s.size(); }) // C++23
            .or_else([] { return std::optional<std::size_t>{0}; }); // C++23
        std::cout << *len << ' ';
    }
    std::cout << '\n';
}
```

OUTPUT

3 0

9.4 Checking assumptions

- **`static_assert(cond, "msg")`**: checked at compile time. Use it for sizes, type traits and template requirements.
- **`assert(cond)`** (`<cassert>`): checked at run time in debug builds, removed by `-DNDEBUG`. For bugs in *your* code, never for user input.
- **`[[nodiscard]]`**: the compiler warns if the caller ignores the return value (put it on functions returning errors).

Compile-time and run-time assertions

✓ compiled & run · g++ 13

```
#include <cassert>
#include <iostream>

int main() {
    static_assert(sizeof(int) >= 4, "need 32-bit ints"); // checked at compile time
    int x = 5;
    assert(x > 0 && "x must be positive"); // checked at run time
    std::cout << "ok\n"; // (off with -DNDEBUG)
}
```

OUTPUT

ok

C++23 `std::stacktrace`

`std::stacktrace::current()` captures the chain of calls that led to the current line, and it can be printed or stored in a log. Handy when reporting an unexpected error. (Syntax-checked here only; it may need an extra link flag such as `-lstdc++exp` or `-lstdc++_libbacktrace`.)

Printing the call chain

✓ compile-checked · g++ 13

```
#include <iostream>
#include <stacktrace>

void inner() { std::cout << std::stacktrace::current() << '\n'; }
void outer() { inner(); }

int main() { outer(); }
```

9.5 Undefined behavior

Undefined behavior (UB) means the language places *no* requirements on what happens: the program might crash, return garbage, or appear to work until you change compilers. The optimizer assumes UB never happens, so it can quietly delete your checks.

COMMON SOURCES OF UNDEFINED BEHAVIOR

Mistake	Safer habit
Reading an uninitialized variable	Always initialize: <code>int n{}</code> ;
Out-of-bounds <code>v[i]</code>	Use <code>.at(i)</code> while debugging, or iterate with <code>range-for</code>
Dereferencing null or dangling pointers	Smart pointers, references, <code>optional</code>
Signed integer overflow	Check ranges; use unsigned or wider types deliberately
Using an iterator after the container changed	Re-obtain iterators; prefer <code>erase_if</code>
Data race (two threads, one writes)	<code>std::mutex</code> , <code>std::atomic</code> , or avoid sharing
Deleting a derived object via a non-virtual base	Make base destructors <code>virtual</code>

Catch them early · Compile with `-fsanitize=address,undefined -g` during development: the sanitizers turn many silent UB bugs into clear runtime reports (see the Compiler Flags page in Chapter 10).

Chapter 9 in a nutshell

- `throw / catch` by `const&` for exceptional failures; RAII keeps cleanup safe.
- `expected<T,E>` for failures callers should handle; `optional<T>` for “maybe a value”.
- `static_assert` at compile time, `assert` at run time, `[[nodiscard]]` for ignorable-by-mistake results.
- Avoid UB by initializing everything, bounds-checking and using RAII types; test with sanitizers.

C++23 Extras at a Glance

The smaller additions that make daily code shorter, plus how modules and `import std`; change the way you include things.

10

10.1 Small but handy

This chapter collects the C++23 features not covered elsewhere. Items that appeared in earlier chapters (`operator[]` with many indices, *deducing this*, `expected`, `flat_map`, `mdspan`, `generator`, `ranges::to`, `to_underlying`, `print`) are listed in the table at the end.

uz literals, `auto(x)`, `byteswap`, `[[assume]]`, named escapes

✓ compiled & run · g++ 13

```
#include <bit>
#include <cstdint>
#include <iostream>
#include <vector>

void count(std::vector<int> v) { std::cout << v.size() << ' '; }

int main() {
    auto n = 5uz; // uz / zu: a std::size_t literal
    std::vector<int> v(n, 1);
    count(auto(v)); // auto(x): an explicit copy of x
    std::cout << std::hex << std::byteswap(std::uint16_t{0x1234}) << '\n'; // 3412

    int x = 3;
    [[assume(x > 0)]]; // a promise to the optimizer
    char c = '\N{LATIN SMALL LETTER A}'; // named character escape
    std::cout << std::dec << c << ' ' << x << '\n';
}
```

OUTPUT

```
5 3412
a 3
```

WHAT EACH LINE OF THE SNIPPET DOES

Feature	Meaning
<code>5uz, 5zu</code>	Literal of type <code>std::size_t</code> : perfect for loop indices that match <code>.size()</code>
<code>auto(v), auto{v}</code>	Make a prvalue copy of <code>v</code> right here (decay-copy), e.g. to pass a copy explicitly
<code>std::byteswap(x)</code>	Reverse the bytes of an integer (endianness conversion)
<code>[[assume(expr)]]</code>	Promise the optimizer that <code>expr</code> is true. Wrong promises are UB!
<code>'\N{NAME}'</code>	Character by its Unicode name
<code>std::unreachable()</code>	Mark a code path as impossible (UB if reached)
<code>std::to_underlying(e)</code>	Enum to its integer type
<code>if constexpr { }</code>	Branch on whether the function runs at compile time

if constexpr

Inside a constexpr function, `if constexpr` lets you pick a different implementation depending on whether the call is being evaluated by the compiler or at run time.

Compile-time versus run-time branch

✓ compiled & run · g++ 13

```
#include <iostream>

constexpr int pick(int x) {
    if constexpr { // C++23: true only during compile-time evaluation
        return x * 2;
    } else {
        return x * 3; // ordinary run-time call
    }
}

int main() {
    constexpr int a = pick(5); // evaluated by the compiler
    int n = 5;
    std::cout << a << ' ' << pick(n) << '\n'; // second call happens at run time
}
```

OUTPUT

10 15

10.2 Callables: static operators and move_only_function

static operator() and static lambdas

A function object that holds no data can declare its call operator static: no hidden `this`, so it is as cheap as a plain function and can be called as `Type::operator()(args)`. Capture-less lambdas can be marked static too. (C++23 also allows a static operator[.])

static operator() and static lambdas

✓ compiled & run · g++ 13

```
#include <iostream>

struct Square {
    static int operator()(int x) { return x * x; } // C++23: no object needed
};

int main() {
    auto cube = [](int x) static { return x * x * x; }; // C++23: static lambda
    std::cout << Square{}(4) << ' ' << Square::operator()(5) << ' ' << cube(3) << '\n';
}
```

OUTPUT

16 25 27

std::move_only_function

`std::function` requires its target to be copyable. `std::move_only_function` accepts callables that own unique resources (like a `unique_ptr` capture) and supports `const`, `noexcept` and reference qualifiers in its signature. Ideal for task queues and callbacks.

A callback that owns a unique_ptr

✓ compiled & run · g++ 13

```
#include <functional>
#include <iostream>
#include <memory>

int main() {
    // std::function needs copyable callables; this one may own a unique_ptr
}
```

```
std::move_only_function<int()> job =
    [p = std::make_unique<int>(42)] { return *p; };
std::cout << job() << '\n';
}
```

OUTPUT
42

function VS move_only_function

	std::function	std::move_only_function
Holds copy-only callables	✓	✓
Holds move-only callables	✗	✓
Copyable itself	✓	✗
Honors const/noexcept in signature	✗	✓

10.3 Modules and `import std;`

Headers are textually pasted into every file that includes them, which is slow and leaks macros. A **module** is compiled once and *imported*. C++23 provides the standard library as modules: `import std;` replaces nearly all `#include <...>` lines, and `import std.compat;` also brings the C names (`::printf, ...`).

The entire standard library with one line ◇ newer toolchain

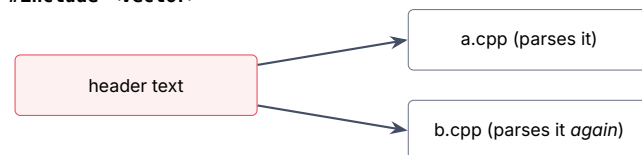
```
import std;                                     // C++23: the whole standard library

int main() {
    std::println("Hello from a module!");
}
```

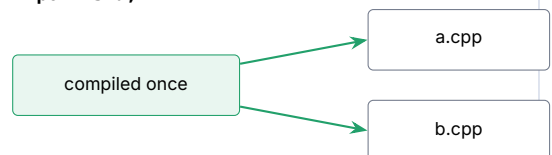
EXPECTED OUTPUT (not run here: needs GCC 14+ / newer libc++)
Hello from a module!

INCLUDE VS IMPORT

`#include <vector>`



`import std;`



Toolchain support • `import std;` needs a very recent compiler, a build system that supports modules (CMake 3.30+ with Ninja, or MSVC), and extra setup. If your toolchain does not support it yet, keep using `#include`. Both styles can coexist in one project.

10.4 Everything C++23 added, in one table

Feature	What it gives you	Chapter
<code>std::print/println</code>	Type-safe formatted output, faster than <code>cout</code>	1
<code>if constexpr & constexpr</code>	Different code at compile time vs run time	3
Deducing <code>this</code>	One member function for all value categories; recursive lambdas	5
Multi-index operator[]	<code>grid[r, c]</code>	5
<code>static operator()</code> / lambdas	No hidden <code>this</code>	9
<code>std::to_underlying</code>	Enum to integer safely	5
<code>std::expected</code>	Value-or-error return type	8
optional monadic ops	<code>transform</code> , <code>and_then</code> , <code>or_else</code> : chain without <code>if</code> ladders	8
<code>std::generator</code>	Lazy coroutine sequences	7
<code>flat_map</code> / <code>flat_set</code>	Sorted-vector associative containers	7
<code>std::mdspan</code>	Multi-dimensional views	7
Range additions	<code>ranges::to</code> , <code>fold_left</code> , <code>contains</code> , <code>zip</code> , <code>enumerate</code> , <code>chunk</code> , <code>slide</code> , <code>stride</code>	2, 7
<code>std::stacktrace</code>	Capture the call chain	8
<code>std::move_only_function</code>	Callable wrapper for move-only targets	9
<code>import std;</code>	The standard library as a module	9
<code>uz</code> , <code>auto(x)</code> , <code>[[assume]]</code> , <code>unreachable</code> , <code>byteswap</code>	Small utilities	9

Chapter 10 in a nutshell

- C++23 is mostly *polish*: fewer boilerplate overloads, better error handling, richer ranges.
- `static call` operators and `move_only_function` improve callbacks.
- Modules (`import std;`) speed up builds but need modern tooling: check support first.

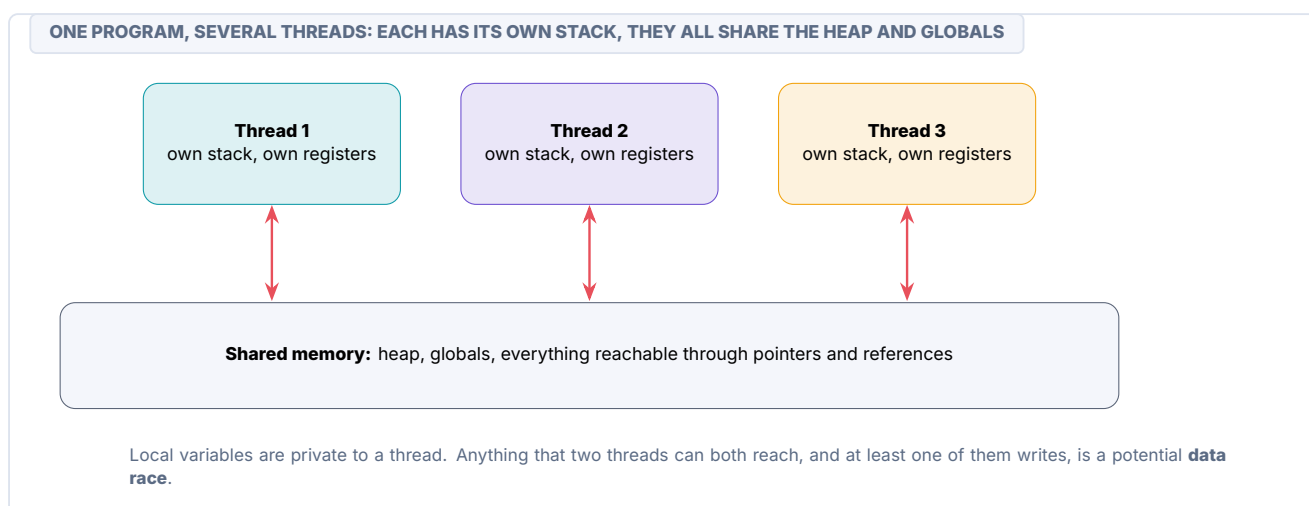
Concurrency

Doing several things at once: threads, locks, atomics, futures and a blocking queue. The goal is not to make everything parallel, it is to understand the few rules that keep parallel code correct.

11

11.1 Why concurrency, and why it is hard

A modern CPU has several **cores**, each able to run its own instruction stream. A **thread** is one such stream inside your program. Running independent work on several threads can make a program finish sooner (a physics step split across cores) or stay responsive (loading a file without freezing the window). The difficulty is that all threads of a program *share the same memory*, and the order in which they touch it is not under your control.



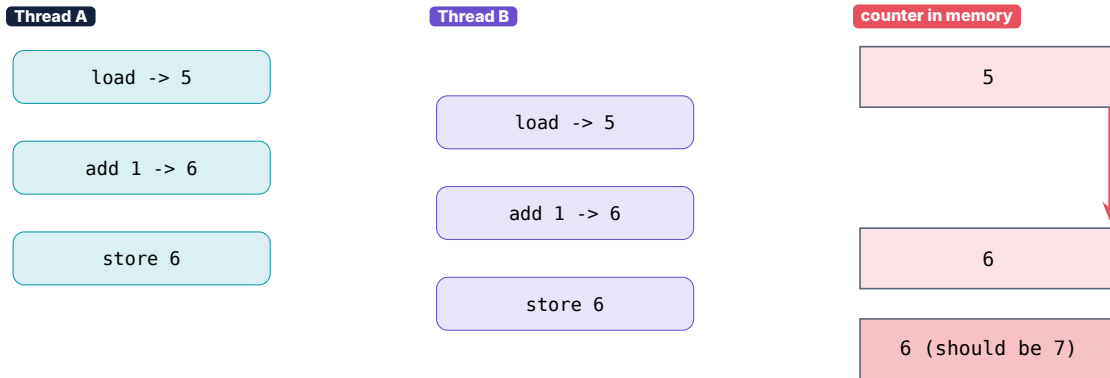
11.2 Starting threads: `jthread`

`std::thread` starts a new thread running a function or lambda. You *must* join it (wait for it to finish) before it is destroyed, or the program terminates. C++20's **`std::jthread`** fixes this footgun: it joins automatically in its destructor, and it also supports cancellation (below). Prefer it.

11.3 The data race, made visible

Consider two threads each incrementing a shared `int` a thousand times. `++counter` looks like one step but is three: load the value, add one, store it back. If both threads load the same old value, both store the same new value, and one increment is lost.

A LOST UPDATE: TWO THREADS READ 5, BOTH WRITE 6, ONE INCREMENT DISAPPEARS



The scary thing is that the program may print the right answer on a quiet day. Do not trust your eyes; use a tool. **ThreadSanitizer** (`-fsanitize=thread`) watches memory accesses and reports races even when the final answer happened to look right:

A data race, found by ThreadSanitizer

✓ compiled & run · with `-fsanitize=thread`

```
// Deliberate data race. Compile with: g++ -std=c++23 -g -fsanitize=thread
#include <iostream>
#include <thread>

int main() {
    int counter = 0; // shared, unprotected
    std::jthread a([&] { for (int i = 0; i < 1000; ++i) ++counter; });
    std::jthread b([&] { for (int i = 0; i < 1000; ++i) ++counter; });
    a.join();
    b.join();
    std::cout << counter << "\n"; // should be 2000; often is not
}
```

OUTPUT

```
WARNING: ThreadSanitizer: data race
Read of size 4 at 0x7ffcd52f6b24 by thread T2:
#0 operator() snip/c11_race.cpp:8
Previous write of size 4 at 0x7ffcd52f6b24 by thread T1:
#0 operator() snip/c11_race.cpp:7
SUMMARY: ThreadSanitizer: data race snip/c11_race.cpp:8 in operator()
2000
```

11.4 Mutexes: one thread at a time

A **mutex** (mutual exclusion lock) lets one thread at a time into a protected region. Always lock with an RAII guard (chapter 5): `std::lock_guard` for the simple case, `std::scoped_lock` to take several mutexes safely without deadlock. Never call `lock()` and `unlock()` by hand: an early return or an exception between them leaves the mutex locked forever.

Four jthreads, one mutex-protected counter

✓ compiled & run

```
#include <iostream>
#include <mutex>
#include <thread>
#include <vector>

int main() {
    constexpr int kThreads = 4, kPerThread = 100'000;

    int counter = 0;
    std::mutex m;
```

```

{
    std::vector<std::jthread> workers;                // jthread joins automatically in its
    destructor
    for (int t = 0; t < kThreads; ++t)
        workers.emplace_back([&] {
            for (int i = 0; i < kPerThread; ++i) {
                std::lock_guard lock(m);                // only one thread at a time past this line
                ++counter;
            }
        });
}
std::cout << "expected " << kThreads * kPerThread << ", got " << counter << "\n";
// all four are joined here
}

```

OUTPUT

```
expected 400000, got 400000
```

Pitfalls with locks • Deadlock: thread A holds lock 1 and waits for lock 2 while B holds lock 2 and waits for lock 1. Take multiple locks together with `std::scoped_lock(m1, m2)`, or always in the same order. **Holding a lock too long** (while doing I/O or calling unknown code) serializes your program and invites deadlock. **Forgetting which data a mutex protects:** keep the mutex and the data it guards in the same class and make the data private.

11.5 Atomics and memory ordering

For a single counter or flag, a lock is overkill. A `std::atomic<T>` makes each read, write or `fetch_add` one indivisible step. Atomics also control *visibility*: the CPU and compiler may reorder ordinary reads and writes, so one thread's writes may appear to another in a different order. **Memory orders** say how much ordering you need:

Memory orders you will actually use

Order	Guarantee
<code>memory_order_relaxed</code>	The operation itself is atomic, with no ordering of surrounding reads and writes. Fine for counters nobody synchronizes on
<code>memory_order_release</code> (store)	Everything this thread wrote <i>before</i> the store is visible to a thread that sees the stored value with acquire
<code>memory_order_acquire</code> (load)	After seeing the value, this thread sees everything the releasing thread wrote before it
<code>memory_order_seq_cst</code> (default)	One global order for all <code>seq_cst</code> operations. Slowest, easiest to reason about

If you are unsure, use the default (`seq_cst`) or a mutex. Relaxed and acquire/release are optimizations to reach for after measuring.

RELEASE / ACQUIRE: THE FLAG PUBLISHES THE DATA

writer thread

payload = 42;

ready.store(true, release);

reader thread

while(!ready.load(acquire)) {}

print(payload); // 42

synchronizes-with

Everything before the release store is visible after the acquire load that sees it. So the reader is guaranteed to print 42, never the stale 0.

An atomic counter, and publishing data with release/acquire

✓ compiled & run

```
#include <atomic>
#include <iostream>
#include <thread>
#include <vector>

int main() {
    std::atomic<int> hits{0};
    {
        std::vector<std::jthread> workers;
        for (int t = 0; t < 4; ++t)
            workers.emplace_back([&] { for (int i = 0; i < 100'000; ++i) hits.fetch_add(1, std::memory_order_relaxed); });
    }
    std::cout << "atomic counter: " << hits << "\n";

    // Publishing data to another thread: release on the writer, acquire on the reader.
    int payload = 0;
    std::atomic<bool> ready{false};
    std::jthread writer([&] {
        payload = 42; // 1. write the data
        ready.store(true, std::memory_order_release); // 2. then publish the flag
    });
    std::jthread reader([&] {
        while (!ready.load(std::memory_order_acquire)) {} // wait for the flag
        std::cout << "reader sees payload = " << payload << "\n"; // guaranteed to see 42
    });
}
```

OUTPUT

```
atomic counter: 400000
reader sees payload = 42
```

11.6 Futures and `std::async`: ask for a result

When you only want a *value* computed elsewhere, you do not need to manage threads or locks. `std::async` runs a function and returns a `std::future`; calling `get()` waits for the result (and rethrows any exception). The listing splits a sum in two with `std::span` (chapter 8): one half on another thread, one half here.

Splitting work with `std::async` and `std::span`

✓ compiled & run

```
#include <future>
#include <iostream>
#include <numeric>
#include <span>
#include <vector>

long long sumRange(std::span<const int> s) { return std::accumulate(s.begin(), s.end(), 0LL); }
```

```

int main() {
    std::vector<int> data(1'000'000);
    std::iota(data.begin(), data.end(), 1);           // 1, 2, 3, ... 1,000,000

    const std::size_t half = data.size() / 2;
    std::span<const int> all(data);
    auto left = std::async(std::launch::async, sumRange, all.first(half)); // runs on another
    thread                                           // runs here,
    long long right = sumRange(all.subspan(half));   // get() waits for the
    meanwhile
    long long total = left.get() + right;           // result

    std::cout << "sum = " << total << " (formula says " << 1'000'000LL * 1'000'001LL / 2 << ")\\n";
}

```

OUTPUT

```
sum = 500000500000 (formula says 500000500000)
```

11.7 Waiting for work: condition variables and a queue

A thread that has nothing to do should *sleep*, not spin in a loop eating a core. A **condition variable** lets a thread sleep until another thread signals that something changed. The standard pattern is a **blocking queue**: producers push work, a consumer pops and sleeps when it is empty. Notice the predicate passed to `wait`: threads can be woken spuriously, so always re-check the condition.

A blocking queue with condition_variable

✓ compiled & run

```

#include <condition_variable>
#include <iostream>
#include <mutex>
#include <optional>
#include <queue>
#include <thread>

// A tiny blocking queue: pop() sleeps until there is work or the queue is closed.
template <class T>
class BlockingQueue {
    std::queue<T> items_;
    std::mutex m_;
    std::condition_variable cv_;
    bool closed_ = false;

public:
    void push(T v) {
        { std::lock_guard lock(m_); items_.push(std::move(v)); }
        cv_.notify_one();
    }
    void close() {
        { std::lock_guard lock(m_); closed_ = true; }
        cv_.notify_all();
    }
    std::optional<T> pop() {
        std::unique_lock lock(m_);
        cv_.wait(lock, [&] { return !items_.empty() || closed_; }); // re-checks the condition on
        wake-up
        if (items_.empty()) return std::nullopt;                    // closed and drained
        T v = std::move(items_.front());
        items_.pop();
        return v;
    }
};

int main() {

```

```

BlockingQueue<int> q;
long long total = 0;

std::jthread consumer([&] { while (auto v = q.pop()) total += *v; });

for (int i = 1; i <= 100; ++i) q.push(i);           // producer: this thread
q.close();
consumer.join();
std::cout << "consumer summed " << total << " (expected 5050)\n";
}

```

OUTPUT

```
consumer summed 5050 (expected 5050)
```

11.8 Stopping threads cleanly

You cannot (and should not) kill a thread from outside. C++20 adds cooperative cancellation: `jthread` owns a `std::stop_source`; you call `request_stop()` and the thread's loop checks its `std::stop_token` and exits. The destructor requests a stop and joins, so a `jthread` cannot be forgotten.

Cooperative cancellation with `stop_token` ✓ compiled & run

```

#include <atomic>
#include <chrono>
#include <iostream>
#include <thread>

int main() {
    std::atomic<long> ticks{0};
    {
        std::jthread worker([&](std::stop_token st) {           // jthread passes a stop_token
            automatically
                while (!st.stop_requested()) {
                    ticks.fetch_add(1, std::memory_order_relaxed);
                    std::this_thread::sleep_for(std::chrono::milliseconds(1));
                }
            });
            std::this_thread::sleep_for(std::chrono::milliseconds(50));
            worker.request_stop();                               // ask politely; the loop checks and
            exits
        }                                                       // destructor joins
        std::cout << "worker stopped cleanly after some ticks: " << (ticks > 0 ? "yes" : "no") << "\n";
    }
}

```

OUTPUT

```
worker stopped cleanly after some ticks: yes
```

Which tool for which job?

Need	Use
Run something in the background, get a result	<code>std::async</code> / <code>std::future</code>
Long-lived worker with a loop	<code>std::jthread</code> + <code>stop_token</code>
Protect shared data	<code>std::mutex</code> with <code>lock_guard</code> / <code>scoped_lock</code>
A single shared counter or flag	<code>std::atomic</code>
Hand work between threads	A blocking queue (mutex + <code>condition_variable</code>)
Do the same thing to a huge array	Split it across threads, or a parallel algorithm (below)

Parallel algorithms, with a caveat • C++17 added execution policies, e.g.

`std::sort(std::execution::par, v.begin(), v.end())`. Support depends on your standard library: GCC needs Intel TBB installed, and Apple's libc++ has historically offered these only experimentally, so I have not run them here. Check your toolchain, and consider Grand Central Dispatch (`dispatch_apply`) or a job system of your own on macOS.

The best concurrency bug is the one you design away • Share as little as possible. Give each thread its own data and combine results at the end. If threads exchange immutable messages through a queue, there is no data race to find. Reach for locks and atomics only where sharing is truly needed.

Chapter 11 in a nutshell

- Threads share memory; a data race (unsynchronized access where one thread writes) is undefined behavior. Test with `-fsanitize=thread`.
- Use `std::jthread` (joins itself, supports `stop_token`). Protect shared data with a mutex held by an RAII guard; take several locks with `std::scoped_lock`.
- `std::atomic` makes single operations indivisible; release/acquire publishes data; default to `seq_cst`.
- `std::async` returns a future for one-shot results; a condition variable plus mutex makes a blocking queue; never busy-wait.
- Prefer designs that share nothing.

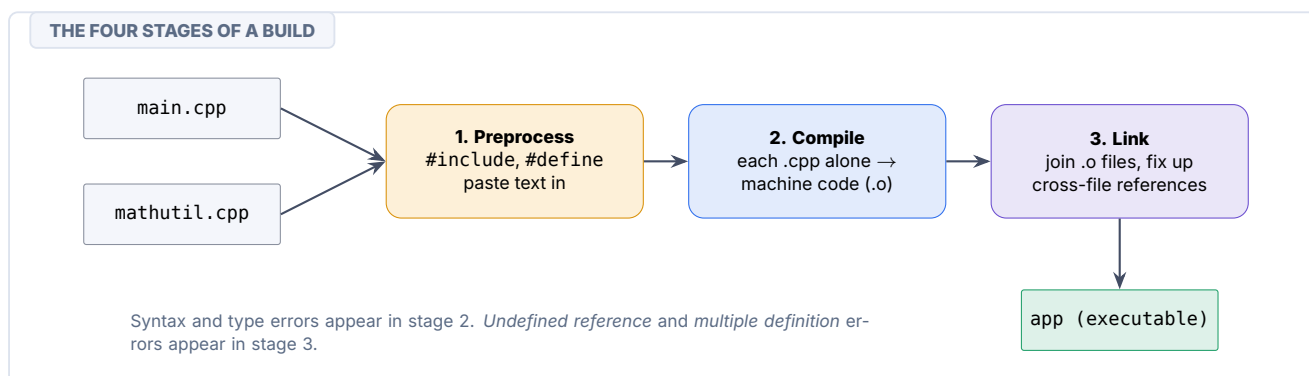
Building Real Projects

12

How source files become a program: the four stages of a build, headers versus source files, the linker errors that confuse everyone once, and CMake, the build system nearly every modern C++ project uses.

12.1 From source text to a running program

A .cpp file is only text. Turning it into something the CPU can run takes four steps, and knowing which step an error comes from tells you where to look.



The compiler never sees two files at once: it translates each .cpp (a **translation unit**) on its own. When main.cpp calls `mu::square`, the compiler only knows the *declaration* (the promise that such a function exists, found in a header). It leaves a blank to be filled in later. The **linker** then connects every blank to the matching *definition* in another object file. You can watch this happen:

The stages, one command each (commands, then what they printed) ✓ ran here with GCC 13

```
// Not a program: the four stages of building, one command each (run in cmdemo/src).
// 1. Preprocess: paste #include files in, expand #define
g++ -std=c++23 -E main.cpp           # prints the expanded source (thousands of lines)
// 2. Compile: translate ONE .cpp file to machine code in an object file (.o)
g++ -std=c++23 -c main.cpp -o main.o
// 3. Inspect what the object file needs from elsewhere
nm -C main.o                       # 'U' = undefined: promised, but defined in another file
// 4. Link: combine object files, resolve every promise, produce the program
g++ main.o mathutil.o -o app
```

OUTPUT

```
$ g++-13 -std=c++23 -E main.cpp | wc -l
44660
$ g++-13 -std=c++23 -c main.cpp -o main.o      # compile only: no link
$ nm -C main.o | grep -E " (T|U) .* (main|square|mean)"
                 U mu::mean(double const*, int)
                 U mu::square(int)
0000000000000000 T main
$ g++-13 main.o -o app                        # link WITHOUT mathutil.o
main.cpp:(.text+0x72): undefined reference to `mu::square(int)'
main.cpp:(.text+0xbe): undefined reference to `mu::mean(double const*, int)'
$ g++-13 -std=c++23 -c mathutil.cpp -o mathutil.o && g++-13 main.o mathutil.o -o app && ./app
square(7) = 49
mean      = 2.5
```

Read the output carefully. The preprocessed `main.cpp` is over 44,000 lines, because `#include <iostream>` pastes in an enormous header. The `nm` listing shows the object file has `T` (defined: `main`) and `U` (undefined: `mu::square`, `mu::mean`). Linking without `mathutil.o` fails with the famous *undefined reference*; linking with it succeeds.

12.2 Headers and source files

The convention that makes this workable: put **declarations** in a header (`.hpp`) that anyone may include, and **definitions** in a source file (`.cpp`) that is compiled once.

ONE FUNCTION, TWO FILES

mathutil.hpp

```
# pragma once
namespace mu {
    int square(int x); // declaration
}
```

mathutil.cpp

```
#include "mathutil.hpp"
namespace mu {
    int square(int x) {
        return x * x; // definition
    }
}
```

Every file that *calls* `square` includes the header. The *one* `.cpp` file holds the body. The header says *what exists*; the source says *how it works*.

Rules that prevent most linker pain

Rule	Why
Start every header with <code>#pragma once</code> (or an include guard)	Otherwise a header included twice (directly and through another) declares things twice and fails to compile
One definition per function or variable across the whole program (the <i>One Definition Rule</i>)	Define a non-inline function in a header and every file that includes it gets a copy: "multiple definition" at link time
Mark small functions defined in headers <code>inline</code> , and templates and <code>constexpr</code> functions may live in headers	These are allowed to appear in many translation units
Do not put <code>using namespace std;</code> in a header	It leaks into every file that includes yours
Include what you use	Do not rely on another header happening to include it

Reading a compiler error

Compilers produce long messages. The skill is knowing where to look: **read the first error first** (later ones are often fallout), find the file and line, and read the one-line summary after `error:`. Here is a real message from GCC 13 for two deliberate mistakes:

Two mistakes: a wrong argument type and a missing semicolon

✓ [compile attempt run here](#)

```
#include <vector>

int main() {
    std::vector<int> v = {1, 2, 3};
    int first = v.at("zero"); // wrong argument type
    return first
}
```

OUTPUT

```

snip/cl2_error.cpp: In function 'int main()':
snip/cl2_error.cpp:5:22: error: invalid conversion from 'const char*' to 'std::vector<int>::size_type'
      {aka 'long unsigned int'} [-fpermissi
  5 |         int first = v.at("zero");           // wrong argument type
      |                        ^~~~~~
      |                        |
      |                        const char*
In file included from /usr/include/c++/13/vector:66,
      from snip/cl2_error.cpp:1:
stl_vector.h:1178:20: note:   initializing argument 1 of 'constexpr std::vector<Tp, _Alloc>::
      reference std::vector<Tp, _Alloc>::at(size_ty
  1178 |         at(size_type __n)
      |         ~~~~~^~~~~
snip/cl2_error.cpp:6:17: error: expected ';' before '}' token

```

12.3 CMake: describe the project, let it generate the build

Typing compiler commands by hand stops being practical beyond a few files. A **build system** describes your targets and their dependencies, compiles only what changed, and runs compilation in parallel. **CMake** is the common choice; it does not build anything itself but *generates* build files for Ninja, Make or Xcode.

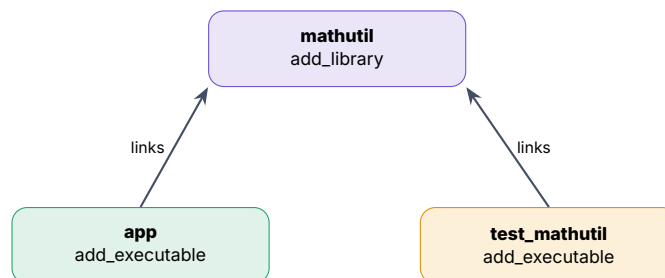
A SMALL PROJECT, ITS FILES, AND HOW CMAKE TARGETS DEPEND ON EACH OTHER

```

demo/
  CMakeLists.txt
  src/
    mathutil.hpp
    mathutil.cpp
    main.cpp
  tests/
    test_mathutil.cpp

```

A **target** is a library or executable.



CMakeLists.txt for a library, an app and a test

✓ verified here: [CMake 3.28](#), [Ninja](#), [GCC 13](#)

```

cmake_minimum_required(VERSION 3.28)
project(demo LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)           # -std=c++23, not gnu++23
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)   # writes compile_commands.json for editors

# A library that holds the reusable code...
add_library(mathutil src/mathutil.cpp)
target_include_directories(mathutil PUBLIC src)
target_compile_options(mathutil PRIVATE -Wall -Wextra -Wpedantic)

# ...an executable that uses it...
add_executable(app src/main.cpp)
target_link_libraries(app PRIVATE mathutil)

# ...and a test.
enable_testing()
add_executable(test_mathutil tests/test_mathutil.cpp)
target_link_libraries(test_mathutil PRIVATE mathutil)
add_test(NAME mathutil COMMAND test_mathutil)

```

The three commands you actually type (the first two are one-time per build folder; run them from the project root):

```
cmake -S . -B build -G Ninja -DCMAKE_BUILD_TYPE=Release
cmake --build build
ctest --test-dir build
```

OUTPUT OF THOSE THREE COMMANDS (RUN HERE: CMAKE 3.28, NINJA, GCC 13)

```
-- Configuring done (0.2s)
-- Generating done (0.0s)
-- Build files have been written to: /home/claude/booklet/cmdemo/build
[1/6] Building CXX object CMakeFiles/test_mathutil.dir/tests/test_mathutil.cpp.o
[2/6] Building CXX object CMakeFiles/mathutil.dir/src/mathutil.cpp.o
[3/6] Linking CXX static library libmathutil.a
[4/6] Linking CXX executable test_mathutil
[5/6] Building CXX object CMakeFiles/app.dir/src/main.cpp.o
[6/6] Linking CXX executable app
1/1 Test #1: mathutil ..... Passed    0.00 sec
100% tests passed, 0 tests failed out of 1
square(7) = 49
mean      = 2.5
```

CMake vocabulary

Command	Meaning
<code>add_library(name files...)</code>	Compile these files into a library target
<code>add_executable(name files...)</code>	Compile and link a program
<code>target_link_libraries(a PRIVATE b)</code>	a uses b. PRIVATE: only a's own code; PUBLIC: a's users need b too; INTERFACE: only users
<code>target_include_directories(t PUBLIC dir)</code>	Where to find headers (and pass that on to users)
<code>target_compile_options(t PRIVATE ...)</code>	Flags for one target (warnings, sanitizers)
<code>CMAKE_BUILD_TYPE</code>	Debug (-g, no optimization), Release (-O3 -DNDEBUG), RelWithDebInfo
<code>CMAKE_EXPORT_COMPILE_COMMANDS</code>	Writes <code>compile_commands.json</code> , which editors (clangd) use to understand your code

Always build out of source • The `build/` directory holds every generated file, so you can delete it to start clean and never pollute your source tree. Keep Debug and Release in separate build folders (`build-debug`, `build-release`).

12.4 Compiler flags worth knowing

Everyday flags (GCC and Clang)

Flag	Purpose
-std=c++23	Choose the language version (older compilers spell it c++2b)
-Wall -Wextra -Wpedantic	Turn on the warnings that catch real bugs. Read them; fix them
-Werror	Make warnings errors (good for CI, annoying locally)
-O0 / -O2 / -O3	Optimization level: none (fast compile, debuggable) / standard release / aggressive
-g	Include debug information so debuggers show your names and lines
-fsanitize=address,undefined	Catch memory errors and undefined behavior at run time (chapter 4)
-fsanitize=thread	Catch data races (chapter 11); cannot combine with address
-DNAME=value	Define a preprocessor macro from the command line

12.5 On a Mac

- The compiler that comes with Xcode (`xcrun clang++`) is Apple's build of Clang. Its C++23 feature coverage **lags** the upstream compiler and the standard library (`libc++`) differs from GCC's, so some C++23 library features arrive later. Check your version with `clang++ --version` and the compiler-support table on cp-preference.com, or install a newer LLVM with Homebrew.
- CMake can generate an Xcode project (`-G Xcode`) so you can use Xcode's debugger and GPU tools while keeping CMake as the source of truth.
- Programs that call Apple frameworks (Metal, AppKit) need those frameworks linked, and Objective-C++ source files (`.mm`) to call them directly. The *Graphics guide* avoids this by using `metal-cpp`, a header-only C++ wrapper.

Not verified here • I ran everything in this chapter on Linux with GCC 13, CMake 3.28 and Ninja. I have not run these builds on macOS with Apple Clang or the Xcode generator, and C++20 modules / `import std`; need a newer CMake (3.30 or later reports experimental support) and compiler than I had. Treat those as things to check against current CMake and Clang documentation.

12.6 Dependencies and libraries

Libraries come in two flavors. A **static library** (`.a`) is copied into your executable at link time. A **shared library** (`.so`, `.dylib`) is loaded when the program starts, so many programs can share one copy and you can update it independently. In CMake you choose with `add_library(name STATIC ...)` or `SHARED`.

To use a third-party library, CMake offers `find_package(Name)` (use one installed on the system), and `FetchContent` (download and build the source as part of your project; it needs network access, so I have not run it here). Package managers such as `vcpkg` and `Conan` automate it further. Whatever you choose, record the exact version so the build is reproducible.

Chapter 12 in a nutshell

- A build is preprocess, compile (each `.cpp` alone), link. Compile errors come from stage 2; *undefined reference* and *multiple definition* from stage 3.
- Declarations go in headers (with `#pragma once`); definitions go in one `.cpp`. Follow the One Definition Rule.
- Read the first error first. Compile with `-Wall -Wextra` and fix warnings; use sanitizers in debug builds.
- CMake describes targets (libraries, executables) and links them; build out of source with Ninja; keep Debug and Release separate.

- On macOS check Apple Clang's C++23 support rather than assuming it matches GCC.

Performance & Layout

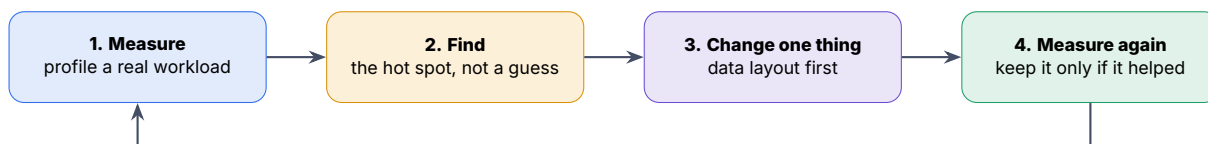
13

Fast C++ is mostly about memory: where your data sits, in what order it is touched, and how often you ask the allocator for more. Measure first, then use the handful of ideas here, each demonstrated by a program run for this book.

13.1 Measure before you change anything

Programmers are famously bad at guessing what is slow. The first rule is to **measure**, the second is to measure a **release build** (-O2 or higher; a debug build can be ten times slower and slow in different places), and the third is to change one thing at a time.

THE OPTIMIZATION LOOP



A minimal benchmark harness (use steady_clock, repeat, keep the minimum, use the result)

✓ compiled & run -O2

```

#include <chrono>
#include <cstdio>
#include <numeric>
#include <vector>

// A minimal benchmark harness: time a callable, repeat, keep the minimum (least disturbed by noise).
template <class F>
double bestOfMs(F& f, int repeats = 5) {
    using clock = std::chrono::steady_clock;    // monotonic: never goes backwards
    double best = 1e300;
    for (int i = 0; i < repeats; ++i) {
        auto t0 = clock::now();
        f();
        auto t1 = clock::now();
        best = std::min(best, std::chrono::duration<double, std::milli>(t1 - t0).count());
    }
    return best;
}

int main() {
    std::vector<int> v(10'000'000, 1);
    long long sink = 0;                                // use the result, or the optimizer deletes the work
    double ms = bestOfMs([&] { sink += std::accumulate(v.begin(), v.end(), 0LL); });
    std::printf("summing 10M ints: %.2f ms (checksum %lld)\n", ms, sink);
}
  
```

OUTPUT

summing 10M ints: 5.52 ms (checksum 50000000)

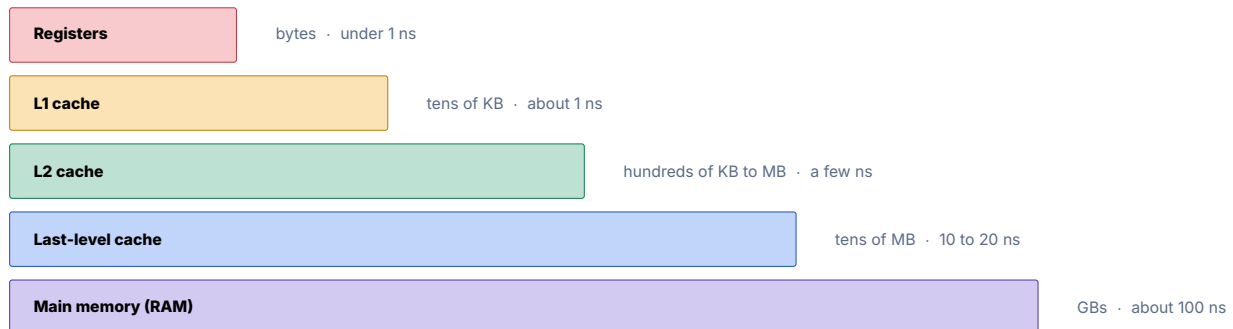
Benchmark traps • **Unused results**: if nothing uses the value you computed, the optimizer deletes the whole loop and you time nothing (hence the `sink` variable). **Cold caches and one-off runs**: repeat and take the minimum. **Debug builds**: meaningless for speed. **Different hardware**: every timing in this chapter came from my Linux x86-64 sandbox with two cores. Absolute numbers will differ on your Mac;

the *ratios and shapes* are what carry over.

13.2 The memory hierarchy: why the cache decides

A CPU can do arithmetic far faster than it can fetch data from main memory. To hide that gap, there are small, fast **caches** between the cores and RAM. Data moves between levels in fixed-size blocks called **cache lines**. Touch one byte and the whole line (commonly 64 bytes on x86; Apple silicon reports 128, which you can check with `sysctl hw.cachelinesize`) is brought in.

THE MEMORY HIERARCHY: EACH STEP DOWN IS BIGGER AND SLOWER (ORDERS OF MAGNITUDE, NOT EXACT NUMBERS)



Latency figures are typical orders of magnitude for modern CPUs, from general computer-architecture knowledge rather than measurements of your machine. The point: a miss all the way to RAM costs roughly a hundred times a hit in L1.

The consequence for you: **use every byte of every cache line you load, and touch memory in order.** The next program makes that concrete by reading the same big array with different strides. At stride 16 only one int in every cache line is used, so the program does sixteen times fewer additions; yet it still takes well over half as long as reading everything, because the cost is dominated by fetching cache lines, not by adding numbers. Only at stride 64 (whole lines skipped) does the time finally drop sharply.

Reading memory with increasing stride (timings from this sandbox)

✓ compiled & run --O2

```
#include <chrono>
#include <cstdio>
#include <vector>

int main() {
    constexpr std::size_t N = 1 << 25; // 32M ints = 128 MB
    std::vector<int> a(N, 1);
    long long sink = 0;
    std::printf("stride useful bytes per pass time\n");
    for (std::size_t stride : {1, 2, 4, 8, 16, 32, 64}) {
        auto t0 = std::chrono::steady_clock::now();
        long long s = 0;
        for (int pass = 0; pass < 4; ++pass)
            for (std::size_t i = 0; i < N; i += stride) s += a[i];
        sink += s;
        double ms = std::chrono::duration<double, std::milli>(std::chrono::steady_clock::now() - t0).
count();
        std::printf("%5zu %20zu %7.1f ms\n", stride, 4 * (N / stride), ms);
    }
    std::printf("(checksum %lld)\n", sink);
}
```

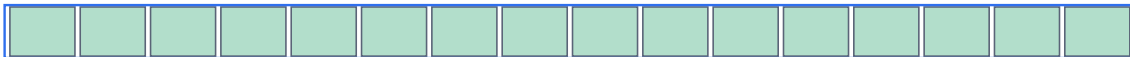
OUTPUT

stride	useful bytes per pass	time
1	134217728	82.4 ms
2	67108864	73.7 ms
4	33554432	56.0 ms
8	16777216	54.0 ms
16	8388608	50.4 ms
32	4194304	46.6 ms
64	2097152	26.8 ms

(checksum 266338304)

HOW A CACHE LINE IS USED AT STRIDE 1 AND STRIDE 16

stride 1: all 16 ints in the line are used



one 64-byte cache line = 16 ints

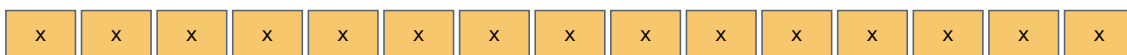
stride 16: 1 of 16 ints is used

**13.3 Struct of arrays: put the data you loop over together**

Suppose a particle has eight fields but one loop only needs *x*. In an **array of structs** (AoS) the *x* values are 64 bytes apart, so every cache line brings in one useful float and fifteen you do not need. In a **struct of arrays** (SoA) all the *x*s sit next to each other and every byte of every line is used.

ARRAY OF STRUCTS VERSUS STRUCT OF ARRAYS: SAME DATA, DIFFERENT ORDER IN MEMORY

AoS: particle 0, particle 1, ... (fields interleaved)

SoA: all *x*, then all *y*, ... (fields separated)

Loop that reads only *x*: in the AoS row the yellow cells are scattered, so most of each cache line is wasted; in the SoA row they are contiguous, which also lets the compiler use SIMD (chapter 14).

Summing one field: array of structs vs struct of arrays (timings from this sandbox)

✓ compiled & run -O2

```
#include <chrono>
#include <cstdio>
#include <vector>

struct ParticleAoS { float x, y, z, vx, vy, vz, mass, life; float pad[8]; }; // 64 bytes per particle

struct ParticlesSoA { // each field in its own array
    std::vector<float> x, y, z, vx, vy, vz, mass, life;
    explicit ParticlesSoA(std::size_t n) : x(n, 1.f), y(n, 1.f), z(n, 1.f), vx(n, 1.f), vy(n, 1.f), vz(n, 1.f), mass(n, 1.f), life(n, 1.f) {}
};

template <class F> double ms(F&& f) {
    auto t0 = std::chrono::steady_clock::now(); f();
    return std::chrono::duration<double, std::milli>(std::chrono::steady_clock::now() - t0).count();
}
```

```

}

int main() {
    constexpr std::size_t n = 4'000'000;
    std::vector<ParticleAoS> aos(n, ParticleAoS{1, 1, 1, 1, 1, 1, 1, 1, {}});
    ParticlesSoA soa(n);
    float sink = 0;

    // Task: sum only the x coordinate of every particle.
    double tAoS = 1e9, tSoA = 1e9;
    for (int r = 0; r < 5; ++r) {
        tAoS = std::min(tAoS, ms([&] { float s = 0; for (auto& p : aos) s += p.x; sink += s; }));
        tSoA = std::min(tSoA, ms([&] { float s = 0; for (float v : soa.x) s += v; sink += s; }));
    }
    std::printf("array of structs : %7.2f ms   (each cache line brings in 1 useful float of 16)\n",
        tAoS);
    std::printf("struct of arrays : %7.2f ms   (each cache line is 16 useful floats)\n", tSoA);
    std::printf("ratio: %.1fx   (checksum %.0f)\n", tAoS / tSoA, sink);
}

```

OUTPUT

```

array of structs :   26.56 ms   (each cache line brings in 1 useful float of 16)
struct of arrays :    5.17 ms   (each cache line is 16 useful floats)
ratio: 5.1x   (checksum 40000000)

```

The difference is large for this access pattern. Notice that SoA only wins when loops use *some* fields. If you always touch all fields of one particle together, AoS is just as good or better; the right layout follows the access pattern, which is why you measure.

13.4 Avoid asking for memory in hot code

Calling the allocator (`new`, `malloc`, anything that grows a vector or string) is expensive and unpredictable. A vector that grows by `push_back` allocates a bigger block, copies everything over and frees the old one, again and again. If you know roughly how many elements are coming, **reserve** once:

push_back with and without reserve: how many times does the vector reallocate?

✓ compiled & run --O2

```

#include <cstdio>
#include <vector>

// Count how many times a vector has to allocate a new block and copy everything over.
std::size_t reallocations(std::vector<int>& v, int n) {
    std::size_t count = 0, cap = v.capacity();
    for (int i = 0; i < n; ++i) {
        v.push_back(i);
        if (v.capacity() != cap) { ++count; cap = v.capacity(); }
    }
    return count;
}

int main() {
    std::vector<int> grow;
    std::vector<int> reserved;
    reserved.reserve(1'000'000); // one allocation up front
    std::printf("push_back 1,000,000 ints without reserve: %zu reallocations\n", reallocations(grow, 1'000'000));
    std::printf("push_back 1,000,000 ints with reserve : %zu reallocations\n", reallocations(reserved, 1'000'000));
}

```

OUTPUT

```

push_back 1,000,000 ints without reserve: 21 reallocations
push_back 1,000,000 ints with reserve : 0 reallocations

```

More habits of the same kind: reuse buffers across frames instead of creating new ones; pass `std::string_view`

and `std::span` instead of copying strings and arrays; use `emplace_back` to construct in place; give hot loops a pre-allocated scratch area (chapter 14's arena allocator).

13.5 False sharing: when threads fight over a cache line

Cores keep their own copy of a cache line and must tell the others when they write to it. If two threads write to *different* variables that happen to share a cache line, each write invalidates the other's copy and the line ping-pongs between cores, even though no data is actually shared. The fix is padding or alignment so each hot variable has its own line.

Two counters on one cache line vs on separate lines (timings from this sandbox)

✓ compiled & run · -O2

```
#include <atomic>
#include <chrono>
#include <cstdio>
#include <new>
#include <thread>

struct Shared { std::atomic<long> a{0}, b{0}; }; // same cache line
struct alignas(64) Padded { std::atomic<long> v{0}; }; // one cache line each

template <class F> double run(F&& f) {
    auto t0 = std::chrono::steady_clock::now(); f();
    return std::chrono::duration<double, std::milli>(std::chrono::steady_clock::now() - t0).count();
}

int main() {
    constexpr long N = 20'000'000;
    Shared s; Padded p1, p2;
    double tShared = run([&] {
        std::jthread t1([&] { for (long i = 0; i < N; ++i) s.a.fetch_add(1, std::memory_order_relaxed); });
        std::jthread t2([&] { for (long i = 0; i < N; ++i) s.b.fetch_add(1, std::memory_order_relaxed); });
    });
    double tPadded = run([&] {
        std::jthread t1([&] { for (long i = 0; i < N; ++i) p1.v.fetch_add(1, std::memory_order_relaxed); });
        std::jthread t2([&] { for (long i = 0; i < N; ++i) p2.v.fetch_add(1, std::memory_order_relaxed); });
    });
    std::printf("two counters on one cache line : %7.1f ms\n", tShared);
    std::printf("two counters on separate lines : %7.1f ms\n", tPadded);
    std::printf("(sizeof(Shared)=%zu, sizeof(Padded)=%zu)\n", sizeof(Shared), sizeof(Padded));
}
```

OUTPUT

```
two counters on one cache line : 587.8 ms
two counters on separate lines : 147.8 ms
(sizeof(Shared)=16, sizeof(Padded)=64)
```

A portable way to pick the padding · C++17 provides

`std::hardware_destructive_interference_size` in `<new>`, the recommended spacing to avoid false sharing. Not every standard library implements it yet and GCC warns that it can vary between builds, which is why the listing hard-codes 64. On Apple silicon, check the line size with `sysctl hw.cachelinesize` instead of assuming 64.

13.6 Other habits that matter

Performance checklist

Habit	Why it helps
Prefer <code>std::vector</code> (contiguous) over <code>std::list</code> / node-based containers	Neighbors share cache lines; no per-element allocation. Pointer-chasing misses cache constantly
Choose data layout for the loop you run most	Hot fields together (SoA or hot/cold struct split)
reserve and reuse buffers	Fewer allocations and copies
Pass big objects by <code>const&</code> , views by value	No needless copies; <code>string_view</code> /span are two words
Keep branches predictable; avoid virtual calls in the innermost loop	Mispredicted branches and indirect calls stall the pipeline; batch by type instead
Use <code>constexpr</code> and compile-time tables	Move work out of run time entirely
Profile with real tools (Instruments on macOS, <code>perf</code> on Linux)	They show the hot spot; your guess may be wrong
Do not optimize what you did not measure	Cleverness costs readability and rarely pays off outside hot spots

Chapter 13 in a nutshell

- Measure a release build, repeat, use the result, change one thing. Timings here are from one sandbox: trust shapes and ratios, not absolutes.
- Memory moves in cache lines. Touch memory in order and use every byte you load; strided access wastes lines.
- Struct of arrays beats array of structs when loops use a few fields (about 4× in the test); the right layout follows the access pattern.
- Avoid allocating in hot loops: reserve (21 reallocations became 0), reuse buffers, use views.
- Keep threads' hot variables on separate cache lines to avoid false sharing (about 4× in the test).

Engine-Style C++

14

The patterns game and graphics engines use to be fast and predictable: allocators you control, handles instead of raw pointers, entity-component storage, SIMD, and layouts that match the GPU. Each is a small, complete program you can run.

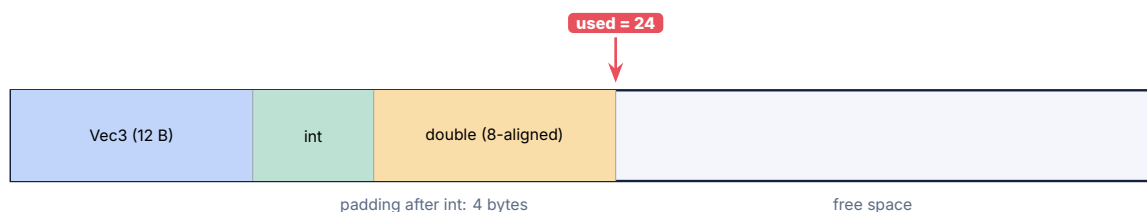
14.1 Why engines avoid the general-purpose allocator

`new` and `delete` (and `malloc`) must handle any size, any lifetime, from any thread. That generality costs time, can fragment memory, and is unpredictable, which matters when you have 16 milliseconds per frame. Engines exploit what they know about *lifetimes*: most per-frame data dies together, and many objects are all the same size.

Arena (linear) allocator: allocate fast, free everything at once

An **arena** owns one big block and a “used” offset. To allocate, round the offset up to the alignment you need and bump it. To free, *everything*, set the offset back to zero. It is perfect for scratch memory that lives for one frame or one job.

AN ARENA: ALLOCATION IS A POINTER BUMP, RESET FREES THE LOT



The int ended at byte 16 (after 12 + 4 bytes), the double needs an 8-byte boundary, and 16 is already one. After `reset()` the offset returns to zero and the same bytes are reused. There is no per-object free, no fragmentation and no bookkeeping.

A linear allocator with alignment

✓ compiled & run

```
#include <cstdlib>
#include <cstdint>
#include <cstdio>
#include <new>
#include <vector>

// A linear (arena / bump) allocator: allocation is one pointer bump, "free everything" is one
// assignment.
class Arena {
    std::byte* base_;
    std::size_t size_, used_ = 0;

public:
    explicit Arena(std::size_t bytes) : base_(static_cast<std::byte*>(::operator new(bytes))), size_(
        bytes) {}
    ~Arena() { ::operator delete(base_); }
    Arena(const Arena&) = delete;
    Arena& operator=(const Arena&) = delete;

    void* allocate(std::size_t bytes, std::size_t align = alignof(std::max_align_t)) {
```

```

    std::size_t start = (used_ + align - 1) & ~(align - 1); // round up to the alignment
    if (start + bytes > size_) return nullptr; // out of space
    used_ = start + bytes;
    return base_ + start;
}
template <class T, class... Args>
T* make(Args&&... args) { // construct in the arena (no
    // destructor call!)
    void* p = allocate(sizeof(T), alignof(T));
    return p ? new (p) T(std::forward<Args>(args)...) : nullptr;
}
void reset() { used_ = 0; } // frees everything at once
std::size_t used() const { return used_; }
};

struct Vec3 { float x, y, z; };

int main() {
    Arena frame(1024); // e.g. one frame's scratch
    // memory
    Vec3* a = frame.make<Vec3>(1.f, 2.f, 3.f);
    int* n = frame.make<int>(7);
    double* d = frame.make<double>(2.5);
    std::printf("used after 3 allocations: %zu bytes\n", frame.used());
    std::printf("d is 8-byte aligned: %s\n", reinterpret_cast<std::uintptr_t>(d) % alignof(double) ==
    0 ? "yes" : "no");
    std::printf("values: %.0f %d %.1f\n", a->y, *n, *d);
    frame.reset(); // next frame starts from
    // empty
    std::printf("used after reset: %zu bytes\n", frame.used());
}

```

OUTPUT

```

used after 3 allocations: 24 bytes
d is 8-byte aligned: yes
values: 2 7 2.5
used after reset: 0 bytes

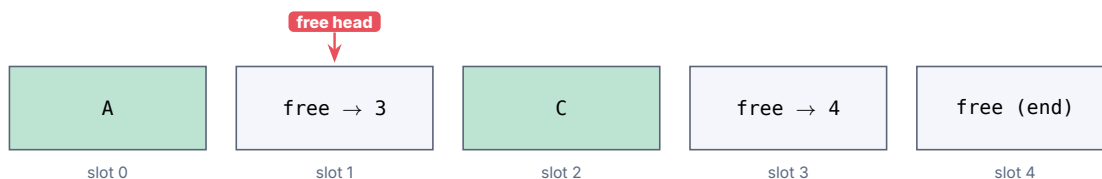
```

Arenas do not run destructors • An arena just hands out raw bytes, so objects with destructors that own resources (a `std::vector`, a file) will leak if you simply reset. Put plain data (structs of numbers) in an arena, or call destructors yourself, or use `std::pmr::monotonic_buffer_resource` from the standard library, which plugs an arena into standard containers.

Pool: many objects of one size, freed individually

Bullets, particles and nodes are created and destroyed constantly and are all the same size. A **pool** keeps an array of identical slots and a **free list** threaded through the empty ones: allocating pops the head of the list, freeing pushes the slot back. Both are one step, and memory is reused immediately.

A POOL: FREE SLOTS FORM A LINKED LIST THROUGH THE ARRAY ITSELF



`create()` takes slot 1 and moves the head to 3. `destroy(p)` makes the slot the new head. A union lets the same bytes hold either an object (when in use) or the next-free index (when free), so the list costs no extra memory.

A fixed-size object pool with an intrusive free list

✓ compiled & run

```

#include <cstdio>
#include <vector>

// A fixed-capacity object pool. Free slots form a linked list threaded through the slots themselves.
template <class T, std::size_t N>
class Pool {
    union Slot {
        alignas(T) unsigned char storage[sizeof(T)];
        std::size_t nextFree;
    };
    Slot slots_[N];
    std::size_t freeHead_ = 0;
    std::size_t live_ = 0;

public:
    Pool() { for (std::size_t i = 0; i < N; ++i) slots_[i].nextFree = i + 1; } // slot i -> slot i+1
    template <class... Args>
    T* create(Args&&... args) {
        if (freeHead_ == N) return nullptr; // full
        Slot& s = slots_[freeHead_];
        freeHead_ = s.nextFree;
        ++live_;
        return new (s.storage) T(std::forward<Args>(args)...);
    }
    void destroy(T* p) {
        p->~T();
        Slot* s = reinterpret_cast<Slot*>(p);
        s->nextFree = freeHead_; // push on the free
        list
        freeHead_ = static_cast<std::size_t>(s - slots_);
        --live_;
    }
    std::size_t live() const { return live_; }
};

struct Bullet { float x, y; };

int main() {
    Pool<Bullet, 3> pool;
    Bullet* a = pool.create(1.f, 2.f);
    Bullet* b = pool.create(3.f, 4.f);
    Bullet* c = pool.create(5.f, 6.f);
    std::printf("live=%zu, 4th create returns %s\n", pool.live(), pool.create(0.f, 0.f) ? "an object"
        : "nullptr (pool is full)");
    pool.destroy(b); // slot reused
    immediately
    Bullet* d = pool.create(7.f, 8.f);
    std::printf("slot of b reused by d: %s\n", d == b ? "yes" : "no");
    (void)a; (void)c;
}

```

OUTPUT

```

live=3, 4th create returns nullptr (pool is full)
slot of b reused by d: yes

```

14.2 Handles: references that cannot dangle

Chapter 4 showed dangling pointers. Engines often hold objects in big arrays that grow and reuse slots, so a raw pointer into one is risky. A **handle** is an *index plus a generation counter*. Removing an object bumps the slot's generation, so every old handle to it stops matching and is rejected safely instead of silently pointing at a different object.

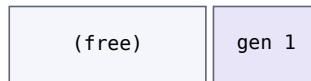
A STALE HANDLE IS DETECTED BECAUSE THE GENERATION NO LONGER MATCHES

1. add "rock"

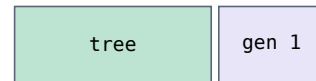


handle a = {index 0, gen 0}

2. remove it



3. add "tree"



handle b = {index 0, gen 1}

Handle a says *index 0, generation 0*. After removal the slot's generation is 1, so `get(a)` fails the check and returns `nullptr`, even though the slot has been reused for b. Handles are also easy to save to disk and send to the GPU, since they are just two integers.

A generational handle table ✓ compiled & run

```
#include <cstdint>
#include <cstdio>
#include <optional>
#include <vector>

// A handle is an index plus a generation. Using a handle after its slot was recycled is DETECTED, not
// a crash.
struct Handle { std::uint32_t index, generation; };

template <class T>
class HandleTable {
    struct Entry { T value{}; std::uint32_t generation = 0; bool alive = false; };
    std::vector<Entry> entries_;
    std::vector<std::uint32_t> freeList_;

public:
    Handle add(T v) {
        std::uint32_t i;
        if (!freeList_.empty()) { i = freeList_.back(); freeList_.pop_back(); }
        else { i = static_cast<std::uint32_t>(entries_.size()); entries_.emplace_back(); }
        entries_[i].value = std::move(v);
        entries_[i].alive = true;
        return {i, entries_[i].generation};
    }

    void remove(Handle h) {
        if (!get(h)) return;
        entries_[h.index].alive = false;
        ++entries_[h.index].generation; // invalidates every old handle to this
        slot
        freeList_.push_back(h.index);
    }

    T* get(Handle h) {
        if (h.index >= entries_.size()) return nullptr;
        Entry& e = entries_[h.index];
        return (e.alive && e.generation == h.generation) ? &e.value : nullptr;
    }
};

int main() {
    HandleTable<const char*> meshes;
    Handle a = meshes.add("rock");
    std::printf("a -> %s\n", *meshes.get(a));
    meshes.remove(a);
    Handle b = meshes.add("tree"); // reuses a's slot
    std::printf("slot reused: %s, but stale handle a is %s\n", a.index == b.index ? "yes" : "no",
        meshes.get(a) ? "STILL VALID (bug)" : "rejected (safe)");
    std::printf("b -> %s\n", *meshes.get(b));
}
```

OUTPUT

```
a -> rock
slot reused: yes, but stale handle a is rejected (safe)
b -> tree
```

14.3 Entities and components

Game objects are rarely a deep inheritance tree any more. An **entity-component-system (ECS)** treats an entity as just an ID, a **component** as plain data attached to it (Position, Velocity), and a **system** as a loop that processes all entities having certain components. Components of one kind live in one dense array, which is exactly the structure-of-arrays layout from chapter 13, so the loops are cache friendly.

SPARSE SET: DENSE ARRAY FOR SPEED, SPARSE TABLE FOR LOOKUP BY ENTITY ID

sparse (indexed by entity id):



Only entities 1 and 3 have a velocity, so only two entries are stored. A system loops over the dense array with no gaps and no checks.

dense (velocities, packed, in loop order):



A tiny component store and a movement system

✓ compiled & run

```
#include <cstdint>
#include <cstdio>
#include <vector>

using Entity = std::uint32_t;

// Sparse set: components are stored densely (fast to loop over); a lookup table maps entity -> dense
// slot.
template <class T>
class ComponentStore {
    std::vector<T> dense_;
    std::vector<Entity> owners_; // owners_[i] is the entity that owns dense_[i]
    std::vector<std::int32_t> sparse_; // sparse_[entity] = index into dense_, or -1

public:
    void add(Entity e, T value) {
        if (e >= sparse_.size()) sparse_.resize(e + 1, -1);
        sparse_[e] = static_cast<std::int32_t>(dense_.size());
        dense_.push_back(value);
        owners_.push_back(e);
    }
    T* find(Entity e) { return (e < sparse_.size() && sparse_[e] >= 0) ? &dense_[sparse_[e]] : nullptr; }
    std::vector<T>& all() { return dense_; }
    const std::vector<Entity>& owners() const { return owners_; }
};

struct Position { float x, y; };
struct Velocity { float dx, dy; };

int main() {
    ComponentStore<Position> positions;
    ComponentStore<Velocity> velocities;
    for (Entity e = 0; e < 5; ++e) positions.add(e, {float(e), 0}); // 5 entities have positions
    velocities.add(1, {1, 2}); // only entities 1 and 3 move
```

```

    velocities.add(3, {-1, 0.5f});

    // "System": loop over the dense velocity array; find the matching position by entity id.
    for (std::size_t i = 0; i < velocities.all().size(); ++i) {
        Entity e = velocities.owners()[i];
        Velocity& v = velocities.all()[i];
        if (Position* p = positions.find(e)) { p->x += v.dx; p->y += v.dy; }
    }
    for (Entity e = 0; e < 5; ++e) {
        Position* p = positions.find(e);
        std::printf("entity %u at (%.1f, %.1f)\n", e, p->x, p->y);
    }
}

```

OUTPUT

```

entity 0 at (0.0, 0.0)
entity 1 at (2.0, 2.0)
entity 2 at (2.0, 0.0)
entity 3 at (2.0, 0.5)
entity 4 at (4.0, 0.0)

```

Where to learn more • Production ECS libraries (EnTT is a widely used open-source one) add groups, views over several component types, and much better ergonomics. Writing the tiny version above once is the best way to understand what they are doing.

14.4 Layout, alignment and matching the GPU

Every type has a **size** and an **alignment**: an object of alignment 8 must start at an address divisible by 8. The compiler inserts **padding** to keep each member aligned, so the order of members changes the size. The same rules decide whether a C++ struct and a GPU struct agree byte for byte (the `float3` trap from the Graphics guide). Lock the layout in with compile-time checks:

Padding, alignas, and static_assert on sizes and offsets

✓ compiled & run

```

#include <cstdint>
#include <cstdio>

// Field order changes the size because of alignment padding.
struct Bad { char a; double b; char c; }; // 1 + 7 padding + 8 + 1 + 7 padding
struct Good { double b; char a; char c; }; // 8 + 1 + 1 + 6 padding

// Lock the layout in with compile-time checks: a mismatch becomes a compile error, not a mystery.
struct alignas(16) GpuLight {
    float position[3];
    float radius;
    float color[3];
    float intensity;
};
static_assert(sizeof(GpuLight) == 32, "must match the shader's struct");
static_assert(alignof(GpuLight) == 16);
static_assert(offsetof(GpuLight, radius) == 12);
static_assert(offsetof(GpuLight, color) == 16);

int main() {
    std::printf("Bad : size=%zu align=%zu  b at offset %zu\n", sizeof(Bad), alignof(Bad), offsetof(
        Bad, b));
    std::printf("Good: size=%zu align=%zu\n", sizeof(Good), alignof(Good));
    std::printf("GpuLight: size=%zu align=%zu (checked at compile time)\n", sizeof(GpuLight), alignof(
        GpuLight));
}

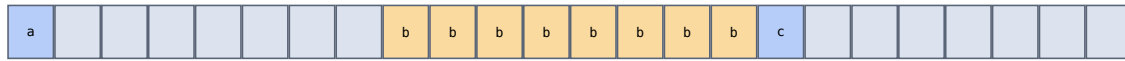
```

OUTPUT

```
Bad : size=24 align=8   b at offset 8
Good: size=16 align=8
GpuLight: size=32 align=16 (checked at compile time)
```

FIELD ORDER CHANGES THE SIZE: 24 BYTES VERSUS 16 FOR THE SAME THREE MEMBERS

```
struct Bad { char a; double b; char c; } → 24 bytes
```



```
struct Good { double b; char a; char c; } → 16 bytes
```

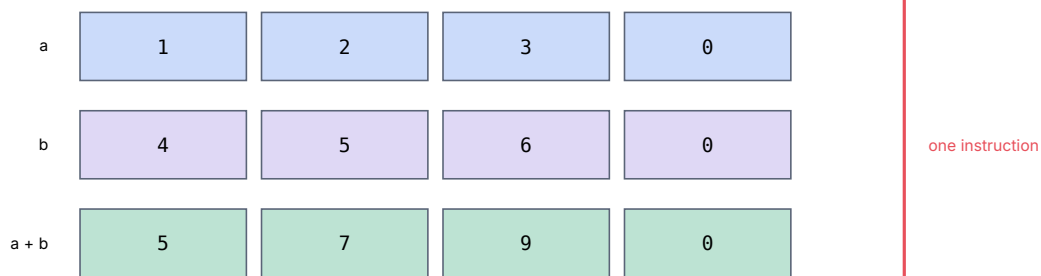


Rule of thumb: order members from largest alignment to smallest, and always `static_assert` the size and offsets of any struct shared with shaders or files.

14.5 SIMD: one instruction, four numbers

CPUs have wide registers (128 bits on Apple silicon's NEON and on x86 SSE) that hold four floats and can add, multiply or compare all four with a *single* instruction. This is **SIMD** (single instruction, multiple data), and it is how physics, skinning and image code get their speed. You have three routes: let the compiler do it automatically (**auto-vectorization**, which -O2/-O3 often achieves for simple loops over contiguous arrays); write portable code with the compiler's **vector extensions** (below); or call the architecture's **intrinsics** directly (<arm_neon.h> on Apple silicon, <immintrin.h> on Intel), which is covered in the *ARM64 Assembly guide*.

ONE VECTOR ADD: FOUR LANES, ONE INSTRUCTION



Each column is a *lane*. A scalar loop would do four separate additions; the vector unit does all four at once.

Portable 4-wide float vectors (GCC/Clang vector extensions)

✓ compiled & run

```
#include <cmath>
#include <cstdio>

// GCC and Clang vector extensions: portable SIMD. The compiler emits SSE on Intel and NEON on Apple
// silicon.
typedef float float4 __attribute__((vector_size(16)));

inline float dot4(float4 a, float4 b) {
    float4 m = a * b;                      // four multiplies in ONE instruction
    return m[0] + m[1] + m[2] + m[3];
}

int main() {
    float4 a = {1, 2, 3, 0}, b = {4, 5, 6, 0};
    float4 sum = a + b;                    // four adds in ONE instruction
```

```
float4 scaled = a * 2.0f; // scalar broadcast
std::printf("a + b      = %.0f %.0f %.0f %.0f\n", sum[0], sum[1], sum[2], sum[3]);
std::printf("a * 2      = %.0f %.0f %.0f %.0f\n", scaled[0], scaled[1], scaled[2], scaled[3]);
std::printf("dot(a, b) = %.0f (1*4 + 2*5 + 3*6)\n", dot4(a, b));
std::printf("|a|       = %.4f\n", std::sqrt(dot4(a, a)));
std::printf("sizeof(float4) = %zu bytes, alignof = %zu\n", sizeof(float4), alignof(float4));
}
```

OUTPUT

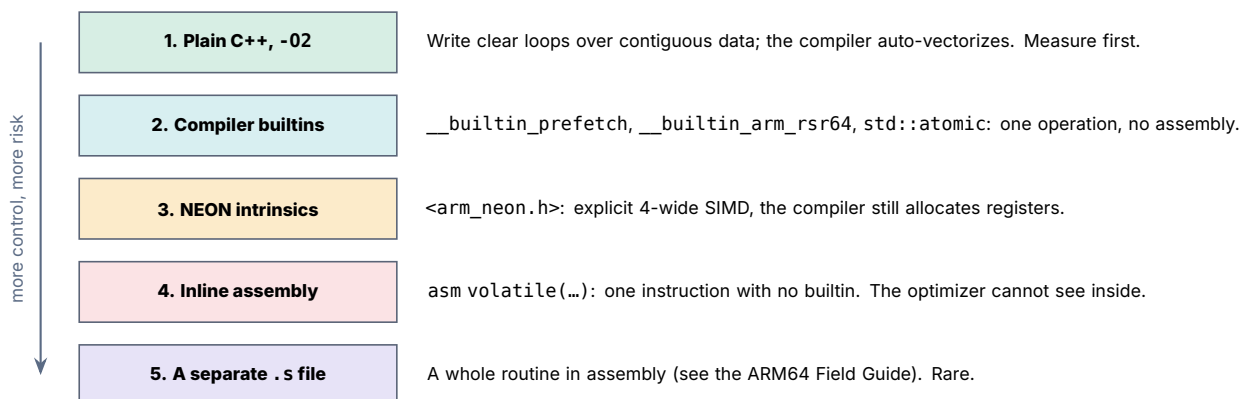
```
a + b      = 5 7 9 0
a * 2      = 2 4 6 0
dot(a, b) = 32 (1*4 + 2*5 + 3*6)
|a|       = 3.7417
sizeof(float4) = 16 bytes, alignof = 16
```

Hedge on newer APIs • A standard C++ SIMD library (`std::simd`) is slated for C++26 and an experimental version has shipped with some standard libraries, but I have not tested its availability on Apple's toolchain. The vector-extension code above compiled and ran here on x86 (SSE); on Apple silicon the same source compiles to NEON, which I could not run here.

14.6 Where ARM64 helps: inline assembly and NEON

Your Mac's CPU is an ARM64 chip, and C++ gives you three ways to reach its instruction set directly. Knowing *when* each is worth using matters more than the syntax.

CHOOSING A TOOL: START AT THE TOP, STOP AS SOON AS IT IS FAST ENOUGH



When it genuinely helps

Situation	Best tool
Hot numeric loops: dot products, filters, colour conversion, culling	Plain loops first; then NEON intrinsics (or Apple's simd and Accelerate libraries).
Timing a short piece of code	Read the counter register: <code>__builtin_arm_rsr64("cntvct_el0");</code> ticks per second come from the <code>cntfrq_el0</code> register.
Telling the CPU what you will read soon	<code>__builtin_prefetch(p)</code> (becomes <code>prfm</code>).
Ordering between threads	<code>std::atomic</code> with the right <code>memory_order</code> ; the compiler emits <code>ldar</code> , <code>stlr</code> and <code>dmb</code> as needed.
An instruction nothing else exposes, or a special system register	Inline assembly.
Reading a crash or a slow function	Reading assembly, no writing: see the ARM64 Field Guide, chapters 9–10.

Inline assembly

Clang and GCC (and so Apple Clang) accept **extended assembly**: a string holding one or more instructions, plus a list telling the compiler which C++ variables the instructions read and write. The compiler picks the registers and substitutes them for `%0`, `%1`, ... in order of appearance, outputs first.

ANATOMY OF AN EXTENDED asm STATEMENT

```
asm volatile("madd %0, %1, %2, %3" : "=r"(r) : "r"(b), "r"(c), "r"(a) : "memory");
```

template

outputs

inputs

clobbers

`=r`: write-only, any general register. `r`: read-only register. `+r`: read and written. `m`: a memory operand. `i`: a compile-time constant. Clobber `"memory"`: the asm touches memory the compiler cannot see; `"cc"`: it changes the condition flags.

Four tiny inline-assembly helpers (with a portable fallback)

✓ arm64 path compile-checked · fallback path compiled & run

```
#include <cstdint>
#include <cstdio>

#if defined(__aarch64__)
// 1. Compute a + b*c with the madd instruction. "=r" = output in any general register; "r" = inputs.
inline std::uint64_t madd(std::uint64_t a, std::uint64_t b, std::uint64_t c) {
    std::uint64_t r;
    asm("madd %0, %1, %2, %3" : "=r"(r) : "r"(b), "r"(c), "r"(a));
    return r;
}
// 2. Read the generic timer's 64-bit counter (a system register). volatile: never optimize it away.
inline std::uint64_t ticks() {
    std::uint64_t t;
    asm volatile("mrs %0, cntvct_el0" : "=r"(t));
    return t;
}
// 3. Prefetch hint: "start loading this cache line for reading". No outputs, one input.
inline void prefetch(const void* p) { asm volatile("prfm pldl1keep, [%0]" : : "r"(p)); }
// 4. Full memory barrier. The "memory" clobber also stops the COMPILER reordering memory accesses across it.
inline void barrier() { asm volatile("dmb ish" : : "memory"); }
constexpr const char* path = "arm64 inline assembly";
#else
inline std::uint64_t madd(std::uint64_t a, std::uint64_t b, std::uint64_t c) { return a + b * c; }
inline std::uint64_t ticks() { return 0; } // portable stand-ins for other CPUs
inline void prefetch(const void* p) { __builtin_prefetch(p); }
inline void barrier() { __atomic_thread_fence(__ATOMIC_SEQ_CST); }
```

```
constexpr const char* path = "portable fallback (not arm64)";
#endif

int main() {
    int data[16] = {};
    prefetch(data);
    barrier();
    std::printf("path: %s\n", path);
    std::printf("madd(10, 3, 4) = %llu\n", static_cast<unsigned long long>(madd(10, 3, 4)));
}
```

OUTPUT

```
path: portable fallback (not arm64)
madd(10, 3, 4) = 22
```

What was checked • The ARM64 branch of this file was *compile-checked* with `clang++ -target arm64-apple-macos14 -Wall -Wextra -pedantic`, but never run: the machine that built this guide is x86-64, and the standard headers were replaced by one-line stubs for that check. The output above is from the *fallback* branch, which is what a non-ARM compiler selects through `#if defined(__aarch64__)`.

What did the compiler make of them? This is real clang output for the Apple target, for two small functions that use the helpers:

Generated arm64 code (names are C++-mangled)

✓ real clang 18 output; not run

```
__Z1fyyy:
    madd    x0, x1, x2, x0
    ret
__Z5timedPKi:
    mrs     x8, CNTVCT_EL0
    prfm    pldlakeep, [x0]
    dmb     ish
    mrs     x9, CNTVCT_EL0
    sub     x0, x9, x8
    ret
```

Three things to notice. `madd` was inlined into a single instruction and the compiler chose the registers (`x0`, `x1`, `x2`): no moves at all. The two counter reads are separated by the `prefetch` and the `barrier`, and the `dmb ish` is exactly where we wrote it. And because the asm is marked `volatile`, neither counter read was merged or removed.

Prefer a builtin when one exists • `__builtin_arm_rsr64("cntvct_el0")` and `__builtin_prefetch(p)` produce the *same* `mrs` and `prfm` instructions (I compiled both) without any assembly, and the optimizer understands them. Use asm only when no builtin or intrinsic covers what you need.

Inline-assembly traps • **(1)** A wrong constraint list is not diagnosed: the code compiles and silently miscompiles. If the asm writes a register or memory you did not declare, declare it. **(2)** Without `volatile`, the compiler may delete an asm statement whose outputs it does not use, or hoist it out of a loop. **(3)** The block is opaque: no inlining across it, no constant folding through it. **(4)** It is not portable: guard it with `#if defined(__aarch64__)` and keep a fallback, as above. **(5)** MSVC does not support inline assembly for ARM64 at all.

NEON from C++: intrinsics

For SIMD, intrinsics are almost always better than assembly. `<arm_neon.h>` provides types such as `float32x4_t` (four floats) and functions that each map to roughly one instruction. Guard them with `__ARM_NEON`, which Clang defines when targeting Apple silicon.

A dot product: NEON when available, a plain loop otherwise

✓ arm64 path compile-checked · logic run on x86 via a stand-in header

```
#include <cstdint>
#if defined(__ARM_NEON)
#include <arm_neon.h>
#endif

// Dot product: NEON on Apple silicon, plain loop anywhere else.
float dot(const float* a, const float* b, std::size_t n) {
    std::size_t i = 0;
    float sum = 0.0f;
    #if defined(__ARM_NEON)
        float32x4_t acc = vdupq_n_f32(0.0f);                // four running sums
        for (; i + 4 <= n; i += 4)
            acc = vfmaq_f32(acc, vld1q_f32(a + i), vld1q_f32(b + i)); // acc += a[i..i+3] * b[i..i+3]
        sum = vaddvq_f32(acc);                                // add the four lanes together
    #endif
    for (; i < n; ++i) sum += a[i] * b[i];                    // leftover elements (and the whole job
    // without NEON)
    return sum;
}
```

OUTPUT

n = 0	dot =	0.00000	reference =	0.00000	ok
n = 1	dot =	0.46210	reference =	0.46210	ok
n = 3	dot =	0.38458	reference =	0.38458	ok
n = 4	dot =	0.32976	reference =	0.32976	ok
n = 5	dot =	-1.01964	reference =	-1.01964	ok
n = 16	dot =	-1.08422	reference =	-1.08422	ok
n = 1000	dot =	6.04510	reference =	6.04510	ok
n = 4099	dot =	7.08242	reference =	7.08243	ok

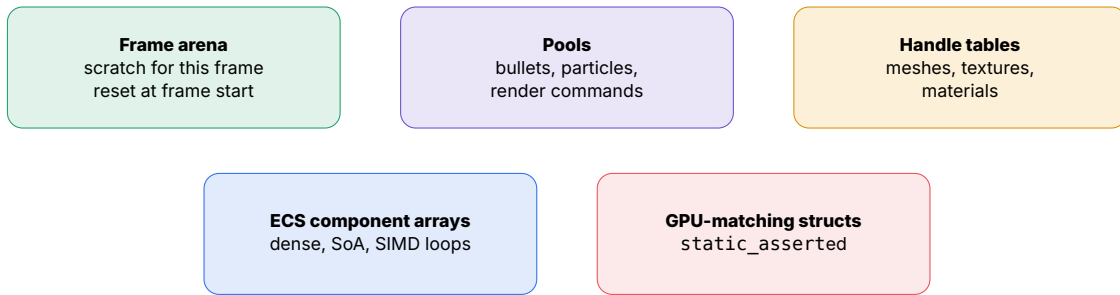
What was checked · The NEON branch was compiled for arm64-apple-macos14 (real `<arm_neon.h>` from clang 18). To test its *logic* on this machine I compiled it again against a small stand-in header that implements the four intrinsics used with ordinary C++ (including a true fused multiply-add per lane), and compared the result against a double-precision reference at eight sizes, including odd ones. Passing proves the algorithm and the tail handling; it does not prove anything about speed or about real NEON hardware.

The shape is the one from the SIMD section above: four running sums in one register (`acc`), a loop that consumes four floats per trip, a *horizontal add* at the end, and a scalar loop for the leftover 0–3 elements. The results differ from a plain sequential sum in the last digits (the additions happen in a different order), which is why the check uses a tolerance. For a deeper look at what these intrinsics become, see chapter 8 of the **ARM64 Field Guide**, where the same loop is shown as assembly.

Apple-specific · On Apple platforms also look at the `simd` module (`simd_float4x4`, `simd_mul`) and the Accelerate framework before hand-writing kernels: they are tuned per chip and already vectorized. Apple's cores run NEON at 128 bits; I do not rely on any wider vector extension here.

14.7 Putting the engine pieces together

HOW THESE PIECES FIT INTO ONE FRAME



Chapter 14 in a nutshell

- Use lifetime knowledge instead of the general allocator: an arena for per-frame scratch (bump, reset), a pool for same-size objects (free list).
- Replace raw pointers into growing arrays with handles (index plus generation); stale ones are detected.
- Store components in dense arrays (ECS) so loops are cache friendly; remember arenas do not run destructors.
- Order members large-to-small, `alignas` and `static_assert` sizes and offsets, especially for GPU data.
- SIMD does four floats per instruction: rely on auto-vectorization, vector extensions, or intrinsics. On ARM64 prefer builtins and NEON intrinsics; reach for inline asm only when nothing else exposes the instruction.

Reference Pages

Look-up tables to keep beside you while you code: operator precedence, headers, build flags, pitfalls, a glossary and an “I want to...” finder.

15

15.1 Operator precedence

Higher rows bind tighter. When in doubt, add parentheses: readers will thank you.

Level	Operators	Notes
1	::	scope
2	a++ a-- () [] . ->	postfix, call, member access
3	++a --a +a -a ! ~ * & sizeof new delete (T)	prefix/unary; right to left
4	* / %	multiplicative
5	+ -	additive
6	<< >>	shifts, stream operators
7	<=>	three-way comparison
8	< <= > >=	relational
9	== !=	equality
10	&	bitwise AND
11	^	bitwise XOR
12		bitwise OR
13	&&	logical AND (short-circuits)
14		logical OR (short-circuits)
15	?: = += -= *= throw	ternary and assignments; right to left
16	,	comma

Pitfall • Bitwise & binds *looser* than ==: write (flags & MASK) == 0, not flags & MASK == 0.

15.2 Header cheat sheet

Header	Provides
<code><iostream></code>	<code>cin, cout, cerr</code>
<code><print></code> C++23	<code>std::print, std::println</code>
<code><format></code>	<code>std::format</code>
<code><string>, <string_view></code>	<code>std::string, std::string_view, stoi, to_string</code>
<code><vector> <array> <deque> <list></code>	sequence containers
<code><map> <set> <unordered_map></code>	associative containers
<code><unordered_set></code>	
<code><flat_map> <flat_set></code> C++23	sorted-vector associative containers
<code><stack> <queue></code>	adapters (<code>priority_queue</code> is in <code><queue></code>)
<code><algorithm>, <numeric></code>	<code>sort, find, accumulate, iota...</code>
<code><ranges></code>	<code>views::, ranges::, ranges::to</code> C++23
<code><memory></code>	<code>unique_ptr, shared_ptr, make_unique</code>
<code><utility></code>	<code>move, forward, pair, to_underlying</code> C++23
<code><optional> <variant> <expected></code>	vocabulary types (<code>expected</code> C++23)
<code> <mdspan></code> C++23	non-owning views
<code><functional></code>	<code>function, move_only_function</code> C++23 , <code>plus<></code>
<code><concepts>, <type_traits></code>	concepts and compile-time type queries
<code><stdexcept>, <exception></code>	standard exception classes
<code><cmath>, <numbers>, <random></code>	math, constants (<code>numbers::pi</code>), random numbers
<code><chrono>, <thread>, <mutex>, <atomic></code>	time and concurrency
<code><fstream>, <sstream>, <filesystem></code>	files, string streams, paths
<code><generator>, <stacktrace></code> C++23	lazy coroutines, call-chain capture

15.3 Compiler flags, sanitizers and a minimal build

Flag	Purpose
<code>-std=c++23</code> (/std:c++latest on MSVC)	select the language version
<code>-Wall -Wextra -pedantic</code>	turn on helpful warnings; treat them as to-dos
<code>-Werror</code>	make warnings errors (good for CI)
<code>-g</code>	debug symbols for debuggers and sanitizers
<code>-O0 / -O2 / -O3</code>	optimization: none (debug) / normal release / aggressive
<code>-fsanitize=address</code>	detect out-of-bounds, use-after-free, leaks
<code>-fsanitize=undefined</code>	detect signed overflow, bad shifts, null derefs
<code>-fsanitize=thread</code>	detect data races (do not combine with address)
<code>-DNDEBUG</code>	disable assert

FROM SOURCE TO PROGRAM



Build commands • Single file: `g++ -std=c++23 -Wall -Wextra -pedantic -g main.cpp -o app`. For anything bigger use CMake: `set(CMAKE_CXX_STANDARD 23), add_executable(app main.cpp util.cpp)`, then `cmake -B build && cmake --build build`.

15.4 Pitfall checklist

Trap	Do this instead
<code>if (x = 5)</code> assigns	<code>==</code> to compare; keep warnings on
Uninitialized locals	initialize at declaration: <code>int n{};</code>
Narrowing in <code>{}</code> (<code>int i{3.5}</code>) is an error	that is good: convert explicitly
Mixing signed and unsigned in comparisons	use one kind; <code>ssize()</code> or <code>uz</code> literals
Integer division: <code>1 / 2 == 0</code>	make one operand floating-point
<code>auto x = a + b</code> deduces unexpected types	check the types you expect; use <code>static_assert</code>
Copying large objects into parameters	pass <code>const T&</code>
Returning a reference to a local	return by value
Missing <code>break</code> in <code>switch</code>	<code>[[fallthrough]]</code> ; when intended
Dangling <code>string_view</code> / <code>span</code>	ensure the owner outlives the view
<code>new</code> / <code>delete</code> by hand	<code>make_unique</code> , containers
Base class without virtual destructor	<code>virtual ~Base() = default;</code>
Iterator invalidation after <code>push_back</code> / <code>erase</code>	re-fetch iterators or use <code>erase_if</code>
Using a moved-from object	reassign it, or leave it alone
<code>map[key]</code> inserting by accident	<code>.find</code> , <code>.contains</code> , <code>.at</code>
using namespace <code>std</code> ; in headers	qualify names (<code>std::</code>) in headers

15.5 Glossary

Term	Meaning
Aggregate	class with public members and no user-declared constructors; initialized with braces
ABI / API	binary / source-level interface between pieces of code
Concept	named compile-time requirement on template arguments
const	promise not to modify through this name
constexpr	can be evaluated at compile time
Declaration	introduces a name and type; a definition also supplies the body or storage
Destructor	function run automatically when an object dies
Heap / stack	dynamic storage you manage via owners / automatic storage for locals
Instantiation	compiler creating a concrete function or class from a template
Invariant	condition a class guarantees is always true
Lvalue / rvalue	has a name and address / a temporary or moved-from value
Namespace	named scope that prevents name clashes (<code>std::</code>)
Overload	several functions with one name but different parameters
RAII	resource lifetime tied to object lifetime
Slicing	losing the derived part by copying a derived object into a base object
Translation unit	one source file after preprocessing
UB	undefined behavior: the program has no guaranteed meaning
Value category	<code>lvalue</code> , <code>xvalue</code> or <code>prvalue</code> : what kind of expression it is
vtable	per-class table of virtual function addresses

15.6 “I want to...” finder

I want to...	Reach for...	Ch.
Store a number or text	<code>int, double, std::string</code>	1
Repeat something	<code>range-for, while, views::iota</code>	2
Choose between cases	<code>if, switch, if constexpr</code>	2
Name a reusable action	<code>function; lambda for a one-off</code>	3
Avoid copying big arguments	<code>const T& parameters</code>	3
Own something on the heap	<code>std::make_unique</code>	4
Hand off ownership	<code>std::move, return by value</code>	4
Model a thing with rules	<code>class with private data</code>	5
Treat different shapes alike	<code>virtual functions, unique_ptr<Base></code>	5
Compare objects easily	<code>defaulted operator<=></code>	5
Write one function for many types	<code>template + concept</code>	6
Store many items	<code>std::vector (default)</code>	7
Look up by key	<code>unordered_map / map</code>	7
Sort, search, count	<code>std::ranges:: algorithms</code>	7
Take any array-like argument	<code>std::span</code>	7
Report an error	<code>exception, expected, optional</code>	8
Catch memory bugs	<code>-fsanitize=address, undefined</code>	8, 10
Make a function run at compile time	<code>constexpr / consteval</code>	3
Print nicely	<code>std::println / std::format</code>	1

15.7 Where to go next

- **Practice:** rewrite small programs you know (a to-do list, a text parser, a bank account) using the types in this guide, compiling with warnings and sanitizers on.
- **Reference:** cppreference.com documents every standard feature, including which standard and compiler version supports it. Try your snippets on godbolt.org (Compiler Explorer) with GCC 14+ or Clang 18+ to run the C++23 examples marked “newer toolchain”.
- **Style:** the C++ Core Guidelines (isocpp.github.io/CppCoreGuidelines) explain *why* the habits in this booklet are recommended.
- **Books:** *A Tour of C++* (Stroustrup) for a fast overview; *Programming: Principles and Practice Using C++* for a patient beginner path.