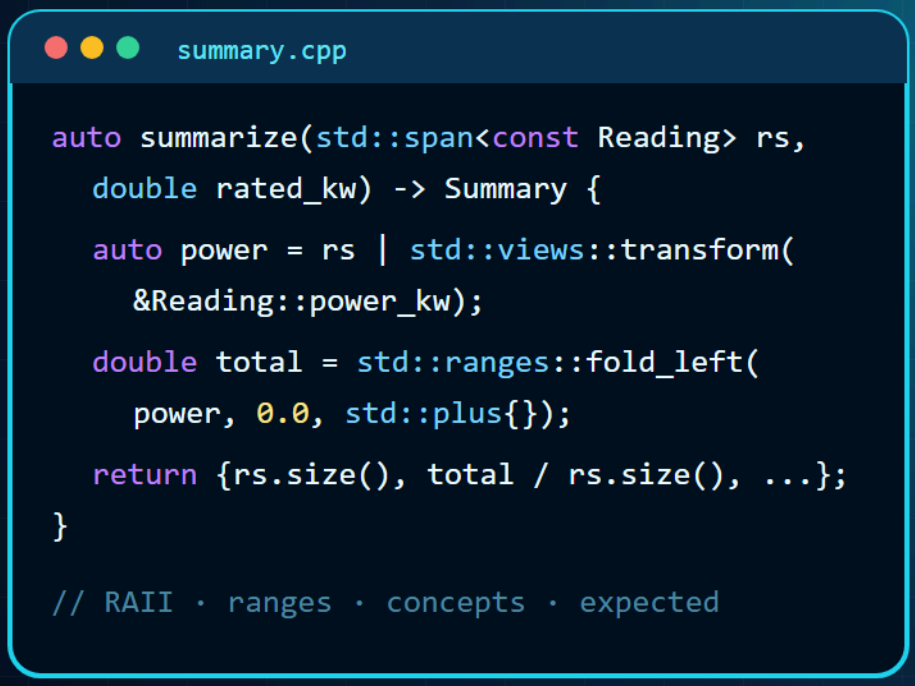


Gateway · Asio

CLI · ferro

ferrowind_core

Tests · GoogleTest

```
$ cmake --build --preset release
$ ctest --preset asan 100% tests passed
$ ./ferro summary data/readings.csv
```

A COMPLETE, PRACTICAL GUIDE

Modern C++ from Zero to Production

Learn modern C++ and build, test, and ship a real networked service with CMake

C++20 · C++23 · RAII · Ranges · Concepts · Coroutines
CMake · GoogleTest · Sanitizers · vcpkg · Asio

LANGUAGE · LIBRARY · CONCURRENCY · PRODUCTION

THENEXBOOKS

Modern C++ from Zero to Production

A Complete, Practical Guide to C++20 and C++23
From your first program to a tested, sanitized, networked service — built with CMake

First Edition

THENEXBOOKS

Modern C++ from Zero to Production
A Complete, Practical Guide to C++20 and C++23

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein. The C++ standard, compilers, standard libraries, and third-party libraries discussed in this book evolve; always consult the official documentation for the versions you use.

Microsoft, Visual Studio, Visual Studio Code, and Windows are trademarks of the Microsoft group of companies. Apple, macOS, and Xcode are trademarks of Apple Inc. Linux is a registered trademark of Linus Torvalds. CMake is a trademark of Kitware, Inc. JetBrains and CLion are trademarks of JetBrains s.r.o. GitHub is a trademark of GitHub, Inc. Docker is a trademark of Docker, Inc. Other product names are trademarks of their respective owners. Their use in this book is for identification only and does not imply endorsement. This book is independent and is not affiliated with, sponsored by, or endorsed by the ISO C++ committee, the Standard C++ Foundation, or any compiler or library vendor.

Ferrowind, the wind-farm operator used as the running example in this book, is fictional, as are its sites, turbines, readings, and data.

Preface

C++ runs a surprising share of the world. It sits inside web browsers and databases, game engines and trading systems, medical devices and car dashboards, compilers and operating systems. When software has to be fast, predictable, close to the hardware, and maintained for decades, C++ is very often the language underneath.

It also has a reputation: huge, old, full of sharp edges. Part of that reputation is deserved — C++ will let you write code that crashes in mysterious ways. But the language has changed enormously. Modern C++ gives you automatic resource management, smart pointers, a rich standard library, ranges, concepts, safe formatting, and tools that catch whole classes of bugs before they reach users. Written the modern way, C++ is expressive, safe by default in most everyday code, and pleasant to work with.

This book teaches C++ the modern way from the very first chapter. We never start with the old habits and then ask you to unlearn them. Instead, we build one real application step by step, introducing each idea at the moment it solves a problem we actually have, and explaining *why* it works the way it does.

Who This Book Is For

- **Programmers from other languages** such as Python, JavaScript, Java, C#, or Go who want to learn C++ properly.
- **Beginners with a little programming experience** who are ready to practice and want a language that teaches them how computers really work.
- **Developers who learned "old" C++** and want to catch up with C++11 through C++23: move semantics, smart pointers, ranges, concepts, `std::expected`, and modern tooling.
- **Students and career changers** aiming at systems, embedded, games, finance, or performance-critical work, who want a portfolio-quality project built the way professional teams build software.

What You Will Learn

- How C++ code becomes a program: compilers, the standard library, linking, and building with CMake.
- The language from the ground up: types, strings, formatting, control flow, functions, and classes.
- Memory and resources done right: object lifetime, references and pointers, RAII, smart pointers, and move semantics.
- The standard library: containers, algorithms, lambdas, ranges and views, error handling with exceptions, `std::optional`, and `std::expected`, files, and time.
- Generic and modern C++: templates, concepts, `constexpr`, code organization, modules, and testing with GoogleTest.

- Concurrency and systems programming: threads, mutexes, atomics, thread pools, coroutines, networking with Asio, and performance work.
- Production C++: undefined behavior, sanitizers, static analysis, dependency management with vcpkg and Conan, CMake presets, logging, CI, and shipping.

How This Book Is Organized

Part I — Getting Started (Chapters 1–5) sets up your toolchain and covers the language fundamentals. **Part II — Objects and Resources** (Chapters 6–10) covers classes, memory, RAII, smart pointers, value semantics, and polymorphism. **Part III — The Standard Library** (Chapters 11–15) covers containers, algorithms, ranges, error handling, files, and time. **Part IV — Generic and Modern C++** (Chapters 16–20) covers templates, concepts, compile-time programming, code organization, and testing. **Part V — Concurrency and Systems Programming** (Chapters 21–24) covers threads, asynchronous work, networking, and performance. **Part VI — Production C++** (Chapters 25–28) covers undefined behavior and tooling, dependencies and builds, logging and CI, and a complete case study. The appendices provide a cheat sheet, templates, a production checklist, and a glossary.

Each chapter ends with exercises and a **Key Takeaways** box for quick review.

The Running Example: Ferrowind

Throughout the book we build the software for **Ferrowind**, a fictional wind-farm operator with 24 turbines across three sites — North Shoal (offshore), Highland Ridge, and Brackwater. Every ten minutes each turbine reports its wind speed, power output, rotor speed, and nacelle temperature. It starts with a simple need: read those measurements from a CSV file and summarize them. Over the chapters, a small command-line tool called `ferro` grows to parse and validate data, raise alerts, write a compact binary format, and produce site reports. In Part V it becomes a multithreaded network gateway that receives live readings over TCP and serves summaries over HTTP, and in Part VI we harden, test, package, and ship it.

Versions and Conventions

Examples use **C++20** as the baseline and **C++23** features where today's compilers support them — we always say which standard a feature needs. We build with **CMake** and test with recent versions of **GCC**, **Clang**, and **MSVC** in mind. Compiler and standard-library support for the newest features still varies, so where details may change between releases, a Note points you to your toolchain's documentation rather than guessing.

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version differences, or a useful aside.

> **WARNING**
>
> A common mistake or something that can cause bugs, crashes, undefined behavior, or security problems.

Code appears in `monospace`. Long lines are wrapped to fit the page. File names above code blocks, such as `include/ferrowind/reading.hpp`, tell you where the code lives. Terminal commands work the same on Windows, macOS, and Linux unless noted.

Contents

Preface

PART I — GETTING STARTED

1 Why C++? The Language, Compilers, and Your Setup

2 Your First Programs: Compiling, Linking, and Building with CMake

3 Types, Variables, and Operators

4 Strings, Formatting, and Console Input and Output

5 Control Flow and Functions

PART II — OBJECTS AND RESOURCES

6 Classes, Constructors, and Encapsulation

7 Memory, Pointers, References, and Object Lifetime

8 RAII and Smart Pointers

9 Copy, Move, and Value Semantics

10 Inheritance, Polymorphism, and Its Alternatives

PART III — THE STANDARD LIBRARY

11 Containers: vector, array, map, unordered_map, and span

12 Algorithms, Iterators, and Lambdas

13 Ranges and Views

14 Error Handling: Exceptions, std::optional, and std::expected

15 Files, Streams, the Filesystem Library, and Time with chrono

PART IV — GENERIC AND MODERN C++

16 Templates and Generic Programming

17 Concepts and Constraints

18 constexpr and Compile-Time Programming

19 Code Organization: Headers, Namespaces, Libraries, and Modules

20 Testing C++ with GoogleTest

PART V — CONCURRENCY AND SYSTEMS PROGRAMMING

21 Threads, Mutexes, Atomics, and the Memory Model

22 Asynchronous Work: Futures, jthread, Thread Pools, and Coroutines

23 Networking: Building the Ferrowind Gateway with Asio and HTTP

24 Performance: Measuring, Cache-Friendly Data, and Allocations

PART VI — PRODUCTION C++

25 Undefined Behavior, Sanitizers, and Static Analysis

26 Dependencies and Builds: vcpkg, Conan, CMake Presets, and Cross-Platform Projects

27 Logging, Configuration, CI, and Shipping

28 Case Study: Ferrowind from Prototype to Production

APPENDICES

A Modern C++ Cheat Sheet

B Templates and Reference

C Production Checklist

D Glossary

PART I

Getting Started

What C++ is good at and how to set up a modern toolchain, how source code becomes a program, building with CMake, and the fundamentals every C++ program is built from: types, operators, strings, formatting, console input and output, control flow, and functions.

1 · Why C++?

2 · Your First Programs

3 · Types, Variables, and Operators

4 · Strings, Formatting, and Console I/O

5 · Control Flow and Functions

CHAPTER 1 · PART I — GETTING STARTED

Why C++? The Language, Compilers, and Your Setup

C++ is one of the few languages that lets you write high-level, expressive code and still decide exactly how every byte is laid out and every resource is released. In this chapter we look at what C++ is good at, how the language evolves, how compilers and standard libraries fit together, and how to install a working toolchain on Windows, macOS, and Linux. We finish by meeting Ferrowind, the wind-farm operator whose software we will build, test, and ship over the course of this book.

What C++ Is, in One Page

C++ is a compiled, statically typed, general-purpose programming language. *Compiled* means that a program called a compiler translates your source code into machine code for a specific processor and operating system before the program ever runs. *Statically typed* means that every variable has a type that is known at compile time, so many mistakes are caught before you run anything. *General-purpose* means that the same language is used for operating system components, games, trading engines, scientific simulations, embedded firmware, desktop applications, and network services.

The language was created by Bjarne Stroustrup in the early 1980s as "C with Classes": an extension of C that added the ability to define your own types with their own behavior. Since then it has grown into a very different language. Modern C++, the style this book teaches, rarely looks like C at all. It favors values over raw pointers, containers over hand-managed arrays, and objects that clean up after themselves over manual cleanup code.

If you come from Python, JavaScript, Java, or C#, three things will feel different at first:

- **There is no garbage collector.** Objects are destroyed at well-defined points, usually when they go out of scope. This sounds like extra work, but modern C++ turns it into a strength: files close, locks release, and memory returns to the system exactly when an object's lifetime ends.
- **There is no virtual machine.** Your program becomes a native executable. Nothing interprets or just-in-time compiles it at run time, and nothing has to be installed on the user's machine besides your program and its libraries.
- **The compiler trusts you.** C++ has rules that, when broken, produce *undefined behavior*: the language makes no promises about what happens. We will point out every such rule as we meet it, and Chapter 25 is devoted to the tools that find these mistakes automatically.

What C++ Is Good At

No language is best at everything. C++ earns its place when a few specific qualities matter more than anything else.

Performance and predictability

C++ compiles to native code and gives the optimizer a great deal of freedom. Just as important as raw speed is *predictability*. Because there is no garbage collector, there are no collection pauses at unpredictable moments. When a program must react within a fixed time budget, such as rendering a frame, filling an order, or controlling a motor, predictable timing can matter more than average speed.

Control over memory and hardware

You decide whether an object lives on the stack, inside another object, in a contiguous array, or in dynamically allocated memory. You can choose the exact layout of a data structure, which matters for cache efficiency, for talking to hardware, and for reading binary file formats. We will use this control in Chapter 15 to write a compact binary format for wind-turbine readings and in Chapter 24 to reorganize data for faster processing.

Zero-cost abstractions

The guiding design principle of C++ is often summarized as "you don't pay for what you don't use, and what you do use, you couldn't hand-code any better." A `std::vector<double>` is a convenient, safe-to-use dynamic array, yet the compiler typically turns a loop over it into the same machine code you would get from a raw C array. Templates, inline functions, and `constexpr` computations let you build expressive abstractions that disappear at compile time. This is a goal, not a guarantee, and in Chapter 24 we will learn to measure rather than assume.

Long-lived systems

C++ takes backward compatibility very seriously. Code written decades ago usually still compiles with a modern compiler, sometimes with a few warnings. That stability is why so many large systems with long lifetimes are written in C++: their owners can adopt new language features gradually without rewriting everything. The flip side is that the language carries historical baggage, and part of learning modern C++ is learning which old features to leave alone.

Where C++ is used

You will find C++ in many places, often underneath the software you use every day:

- **Games and game engines**, where frame budgets are tight and memory layout matters.
- **Finance**, especially trading systems where latency and predictable timing are critical.
- **Embedded and industrial systems**, from automotive controllers to industrial equipment, where resources are limited and hardware access is direct.
- **Web browsers**, whose rendering engines and JavaScript runtimes are large C++ code bases.

- **Databases and storage engines**, where careful control of memory, files, and threads pays off.
- **Compilers, language runtimes, and developer tools**, including the toolchains you will install in this chapter.
- **Scientific and engineering computing**, including simulation, signal processing, and machine-learning infrastructure.

C++ is not the right choice for everything; a quick file-renaming script is better written in Python. C++ shines when performance, resource control, or integration with existing native systems is part of the requirements.

Old C++ Versus Modern C++

Many people have an impression of C++ formed by code written before 2011: raw pointers everywhere, manual `new` and `delete`, and memory leaks waiting to happen. Here is a small taste of the difference. Both programs compute the average of some wind-speed values. First, the old style, which we show only so you can recognize it:

old_style.cpp (do not write code like this)

```
#include <iostream>

int main() {
    int count = 4;
    double* speeds = new double[count];   // manual allocation
    speeds[0] = 11.4; speeds[1] = 12.1; speeds[2] = 7.9; speeds[3] = 9.2;

    double total = 0;
    for (int i = 0; i < count; i++)
        total += speeds[i];

    std::cout << "Average: " << total / count << "\n";
    delete[] speeds;                      // forget this and you leak memory
    return 0;
}
```

And the modern version, the style you will learn in this book:

modern_style.cpp

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

int main() {
    const std::vector<double> speeds{11.4, 12.1, 7.9, 9.2};

    const double total = std::accumulate(speeds.begin(), speeds.end(), 0.0);
    std::cout << "Average: " << total / static_cast<double>(speeds.size()) << "\n";
}
```

The modern version has no manual memory management at all. The `std::vector` owns its elements and releases them automatically when it goes out of scope. The size is stored inside the vector, so it cannot get out of sync with the data. The values are `const`, so we cannot change them by accident. Do not worry about the details yet; every piece of this program is explained in the next few chapters. The point is that modern C++ is a safer and more pleasant language than its reputation suggests.

How C++ Evolves: The ISO Standard

C++ is not owned by a company. It is defined by an international standard published by ISO, the International Organization for Standardization, and developed by a committee known as WG21. Compiler vendors, companies, universities, and individuals participate. The standard document describes what a conforming C++ implementation must do; it does not provide a compiler itself.

Since C++11, the committee has shipped a new standard on a regular three-year cycle. Each version is named after its year:

Standard	Highlights (selection)
C++98 / C++03	The first standard: templates, the standard library containers and algorithms, exceptions.
C++11	The start of "modern C++": move semantics, `auto`, lambdas, smart pointers, range-based `for`, threads, `constexpr`.
C++14	Refinements: generic lambdas, `std::make_unique`, relaxed `constexpr`.
C++17	`std::optional`, `std::variant`, `std::string_view`, `std::filesystem`, structured bindings, `if constexpr`.
C++20	Concepts, ranges, coroutines, modules, `std::span`, `std::format`, three-way comparison, `std::jthread`.
C++23	`std::expected`, `std::print`, `std::ranges::to`, deducing `this`, more views, `std::mdspan`.

The next standard, C++26, is upcoming. We will mention it occasionally in general terms, but every example in this book uses features that are already standardized.

A new standard is not instantly available everywhere. Compiler and library vendors implement features over months or years, and each feature arrives on its own schedule. That is why this book sets a clear baseline:

- **C++20 is our baseline.** Every example compiles as C++20 unless we say otherwise.
- **C++23 features are used where recent compilers support them**, and we always label them, for example "`std::expected` (C++23)". Where support is still patchy, we show a C++20 fallback.

> **NOTE**
>
> Each compiler publishes a table showing which standard features it supports, and the community-maintained page "C++ compiler support" on cppreference.com collects these tables in one place. When an example in this book does not compile for you, checking that table is the fastest way to find out whether your compiler is simply too old.

Compilers and Standard Libraries

When people say "the compiler," they usually mean two separate pieces that ship together. The **compiler** translates C++ code into machine code. The **standard library** provides `std::vector`, `std::string`, `std::format`, threads, file I/O, and everything else in the `std` namespace. Language features such as concepts depend on the compiler; library features such as `std::format` depend on the standard library. A compiler can support a language feature long before its library supports a related library feature, and vice versa.

There are three major compilers and three major standard library implementations:

Compiler	Typical standard library	Where you meet it
GCC (`g++`)	libstdc++	Linux distributions, MSYS2 on Windows
Clang (`clang++`)	libc++ or libstdc++	macOS (Apple Clang), Linux, Windows
MSVC (`cl`)	Microsoft STL	Visual Studio on Windows

Clang is flexible: on macOS it uses libc++, while on most Linux distributions it uses the system's libstdc++ by default. On Windows, Clang can also act as a drop-in replacement for MSVC through `clang-cl`, using the Microsoft STL. When a feature such as `std::format` works with one combination but not another, the library is usually the reason.

The book targets recent versions of all three toolchains: GCC 13 or 14, Clang 17 or 18 or newer, and Visual Studio 2022 with a current update. Newer versions are always fine. Older versions will compile most of the C++20 material but may lack some library features. We will remind you whenever a feature needs a particularly recent toolchain.

> **TIP**
>
> Install two compilers if you can, for example GCC and Clang on Linux, or MSVC and Clang on Windows. Different compilers produce different warnings and catch different mistakes. Code that compiles cleanly on two compilers is much more likely to be correct, portable C++ rather than something that happens to work on one machine.

The Tools You Need

A working C++ environment has four parts:

1. **A compiler and standard library**: GCC, Clang, or MSVC.
2. **A build system generator**: we use **CMake**, version 3.28 or newer. CMake reads a description of your project and generates the actual build instructions.
3. **A build tool**: we use **Ninja**, a small and fast tool that executes the build instructions CMake generates. On Windows, Visual Studio's own build system also works.
4. **An editor or IDE with a debugger**: VS Code, CLion, Visual Studio, or any editor you like, plus GDB, LLDB, or the Visual Studio debugger.

You may wonder why we need CMake at all when we could call the compiler directly. For a single file, calling the compiler directly is fine, and Chapter 2 does exactly that. Real projects have dozens of files, libraries, tests, and third-party dependencies, and must build on several operating systems. CMake is the most widely used way to describe such projects portably, and every major IDE understands it. Learning it from the start saves you from rewriting your build later.

Pick the section below for your operating system. You do not need to follow the others.

Installing a Toolchain on Windows

Option 1: Visual Studio 2022 (recommended for most Windows users)

Visual Studio Community edition is free for individuals, students, and open-source work; check the license terms for your situation. Download the Visual Studio Installer from Microsoft's website, and in the **Workloads** tab select **Desktop development with C++**. This workload includes the MSVC compiler, the Microsoft STL, the Windows SDK, the debugger, and also CMake and Ninja integrated with the IDE. In the optional components you can additionally tick **C++ Clang tools for Windows** to get Clang as a second compiler.

The MSVC compiler is not on your normal command-line path. To use it from a terminal, open **Developer PowerShell for VS 2022** or **x64 Native Tools Command Prompt for VS 2022** from the Start menu. These shortcuts set up the environment variables that `cl` needs.

Developer PowerShell for VS 2022

```
PS> cl
Microsoft (R) C/C++ Optimizing Compiler Version 19.xx.xxxxx for x64
Copyright (C) Microsoft Corporation.  All rights reserved.

usage: cl [ option... ] filename... [ /link linkoption... ]
PS> cmake --version
cmake version 3.xx.x
PS> ninja --version
1.xx.x
```

The exact version numbers on your machine will differ; that is expected. Visual Studio 2022 ships compilers numbered 19.3x and 19.4x, depending on the update you install.

Option 2: MSYS2 with GCC or Clang

If you prefer a GCC or Clang toolchain that behaves like it does on Linux, install MSYS2 from its official website. MSYS2 provides a package manager called `pacman` and several environments; the **UCRT64** environment is the usual choice for modern Windows. Open the "MSYS2 UCRT64" terminal and install the tools:

MSYS2 UCRT64 terminal

```
$ pacman -Syu
$ pacman -S mingw-w64-ucrt-x86_64-gcc mingw-w64-ucrt-x86_64-gdb \
            mingw-w64-ucrt-x86_64-cmake mingw-w64-ucrt-x86_64-ninja
$ pacman -S mingw-w64-ucrt-x86_64-clang     # optional second compiler
```

The first command updates the package database and may ask you to close and reopen the terminal; just run it again afterward. To use these tools from an ordinary PowerShell window or from VS Code, add the folder `C:\msys64\ucrt64\bin` to your `PATH` (adjust the path if you installed MSYS2 elsewhere).

Option 3: LLVM for Windows

The LLVM project publishes a Windows installer that provides `clang`, `clang++`, `clang-cl`, `clang-format`, and `clang-tidy`. On its own, Clang on Windows still needs a standard library and Windows headers, which it normally takes from a Visual Studio installation. In practice, most people combine LLVM with Visual Studio or simply use the Clang component of the Visual Studio Installer.

> **WARNING**
>
> Do not mix object files or libraries built by different Windows toolchains, for example MSVC and MSYS2 GCC, in one program. They use different standard libraries and different binary conventions, and the result either fails to link or crashes in confusing ways. Choose one toolchain per build directory and stay with it.

Installing a Toolchain on macOS

On macOS, start with Apple's Command Line Tools, which include Apple Clang, libc++, LLDB, and `make`:

Terminal (macOS)

```
$ xcode-select --install
$ clang++ --version
Apple clang version ...
Target: arm64-apple-darwin...
```

Apple Clang is a fine compiler, but Apple's release schedule does not match the LLVM project's, so it can lag behind on newer C++ features. For the most complete C++20 and C++23 support, also install the upstream LLVM toolchain with Homebrew, together with CMake and Ninja:

Terminal (macOS)

```
$ brew install llvm cmake ninja
$ "$(brew --prefix llvm)/bin/clang++" --version
Homebrew clang version ...
```

Homebrew installs LLVM "keg-only", which means it does not replace Apple's `clang++` on your `PATH`. That is deliberate. When you want to use Homebrew's Clang with CMake, tell CMake explicitly which compiler to use, as shown later in this chapter. You can also install GCC with `brew install gcc`; it is available under a versioned name such as `g++-14`, because on macOS the plain name `g++` is an alias for Apple Clang.

Installing a Toolchain on Linux

Linux distributions ship GCC and Clang through their package managers. On Ubuntu 24.04 or a similar Debian-based system:

Terminal (Ubuntu / Debian)

```
$ sudo apt update
$ sudo apt install build-essential g++-14 clang cmake ninja-build gdb
$ g++-14 --version
g++-14 (Ubuntu 14...) 14...
```

On Fedora:

Terminal (Fedora)

```
$ sudo dnf install gcc-c++ clang cmake ninja-build gdb
```

On Arch Linux the equivalent packages are `gcc`, `clang`, `cmake`, `ninja`, and `gdb`, installed with `pacman`. Package names and default versions change between distribution releases, so search your package manager if a name above is not found. If your distribution's CMake is older than 3.28, install a newer one from the official Kitware binaries or with `pip install cmake` into a user environment.

Many distributions let several GCC versions live side by side under names like `g++-13` and `g++-14`. The plain `g++` points to the distribution's default version. You do not have to change that default; you can tell CMake exactly which compiler to use.

Choosing an Editor

Any editor can edit C++ files, but good C++ support makes a big difference: code completion, jumping to definitions, inline error messages, and an integrated debugger. Common choices, in no particular order:

- **Visual Studio Code** with either Microsoft's C/C++ extension or the clangd extension, plus the CMake Tools extension. Free, cross-platform, and lightweight. The clangd extension relies on a file called `compile_commands.json`, which CMake can generate for you.
- **CLion** from JetBrains, a commercial cross-platform IDE with built-in CMake support, refactoring, and debugger integration.
- **Visual Studio 2022** on Windows, which can open a folder containing a `CMakeLists.txt` directly and has a very capable debugger.
- Terminal editors such as Vim, Neovim, or Emacs, usually combined with clangd for language support.

The book does not depend on any particular editor. All instructions use the command line, which works the same everywhere, and we point out IDE equivalents where they help.

Checking Your Setup

Let us confirm that everything works by compiling a small program that reports which C++ standard the compiler is using and exercises a C++20 library feature. Create an empty folder anywhere you like, for example `cpp-check`, and save this file inside it:

cpp-check/check.cpp

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

int main() {
    // __cplusplus holds a number such as 202002L for C++20.
    const long standard = __cplusplus;
    const std::string message =
        std::format("Hello from C++! __cplusplus = {}", standard);
    std::cout << message << "\n";

    if (standard >= 202002L) {
        std::cout << "C++20 or newer is enabled.\n";
    } else {
        std::cout << "C++20 is NOT enabled. Check your compiler flags.\n";
    }
}
```

Now compile and run it. On Linux, macOS, or MSYS2, use GCC or Clang with the `-std=c++20` flag:

Terminal

```
$ g++ -std=c++20 -Wall -Wextra check.cpp -o check
$ ./check
Hello from C++! __cplusplus = 202002
C++20 or newer is enabled.
```

Replace `g++` with `clang++` to try Clang. With MSVC, in a Developer PowerShell window:

Developer PowerShell for VS 2022

```
PS> cl /std:c++20 /EHsc /W4 /Zc:__cplusplus check.cpp
PS> .\check.exe
Hello from C++! __cplusplus = 202002
C++20 or newer is enabled.
```

Chapter 2 explains each of these flags in detail. Two MSVC flags deserve a word now. `/EHsc` enables standard C++ exception handling, which you always want. `/Zc:__cplusplus` makes MSVC report the real standard version in `__cplusplus`; without it, MSVC reports an old value for historical compatibility reasons, and our check would wrongly claim that C++20 is disabled.

If the compile step fails with an error such as "`format`: No such file or directory" or "'format' file not found", your standard library does not yet provide `std::format`. That usually means an older GCC (before 13) or an older Apple Clang. Upgrade the toolchain if you can. If you cannot, remove the `<format>` include and the `std::format` call, print `standard` directly with `std::cout`, and keep an eye out for the fallbacks we show in Chapter 4.

Finally, confirm that CMake and Ninja are installed and recent enough:

Terminal

```
$ cmake --version
cmake version 3.28.3
$ ninja --version
1.11.1
```

These numbers are illustrative; anything at or above CMake 3.28 is fine, and any recent Ninja works. If you want to use a specific compiler with CMake, pass it when you first configure a build directory, for example `-DCMAKE_CXX_COMPILER=clang++-18` on Linux or `-DCMAKE_CXX_COMPILER="$(brew --prefix llvm)/bin/clang++"` on macOS. We will configure our first CMake project in Chapter 2.

> **NOTE**
>
> If you plan to try C++23 features, replace `-std=c++20` with `-std=c++23` for GCC and Clang, or `/std:c++latest` for MSVC. Older compilers spell the GCC and Clang flag `-std=c++2b`. The reported `__cplusplus` value will then be larger than `202002`; the exact number depends on how complete the compiler considers its C++23 support.

Meet Ferrowind

Small examples are good for learning one idea at a time, but they hide the problems that real programs face: data that does not fit the happy path, code split across many files, builds that must work on three operating systems, tests, threads, and deployment. To practice those skills, this book follows one realistic project from its first line to production.

Ferrowind is a fictional company that operates wind farms. It runs 24 turbines at three sites:

Site code	Name	Turbines	Model	Rated power
NSH	North Shoal (offshore)	T-01 to T-12	FW-3600	3600 kW
HLR	Highland Ridge	T-13 to T-20	FW-2500	2500 kW
BRK	Brackwater	T-21 to T-24	FW-2500	2500 kW

Every turbine has sensors that report a *reading* at regular intervals. Each reading records which turbine it came from, a timestamp in UTC, the wind speed in meters per second, the electrical power output in kilowatts, the rotor speed in revolutions per minute, and the temperature inside the nacelle, the housing at the top of the tower that contains the generator. The readings arrive as CSV (comma-separated values) text, one reading per line, after a header line:

data/readings.csv (excerpt)

```
turbine_id,timestamp,wind_speed_ms,power_kw,rotor_rpm,nacelle_temp_c
T-07,2026-03-14T10:00:00Z,11.4,2950.0,14.2,61.5
T-07,2026-03-14T10:10:00Z,12.1,3180.0,14.8,63.0
T-03,2026-03-14T10:00:00Z,7.9,1210.0,10.6,58.4
T-15,2026-03-14T10:00:00Z,9.2,1480.0,12.9,88.1
T-21,2026-03-14T10:00:00Z,10.5,320.0,11.7,55.0
```

Look closely and you can already spot two things an operator would want to know. Turbine T-15 reports a nacelle temperature of 88.1 °C, which is above Ferrowind's alert threshold of 85 °C. Turbine T-21 produces only 320 kW in a 10.5 m/s wind, far less than a 2500 kW turbine should deliver in that wind; something may be wrong with it. Turning raw readings like these into summaries and alerts is exactly what Ferrowind's software must do.

What we will build

The software grows with your skills, part by part:

- **Parts I to IV** build `ferro`, a command-line tool. It reads a CSV file of readings, validates every line, and reports statistics and alerts. By the end of Part IV it supports commands like these:

  ```
  $ ./build/ferro summary data/readings.csv --turbine T-07
  $ ./build/ferro alerts data/readings.csv
  $ ./build/ferro report --site NSH --day 2026-03-14 data/readings.csv
  $ ./build/ferro convert data/readings.csv data/readings.bin
  ```