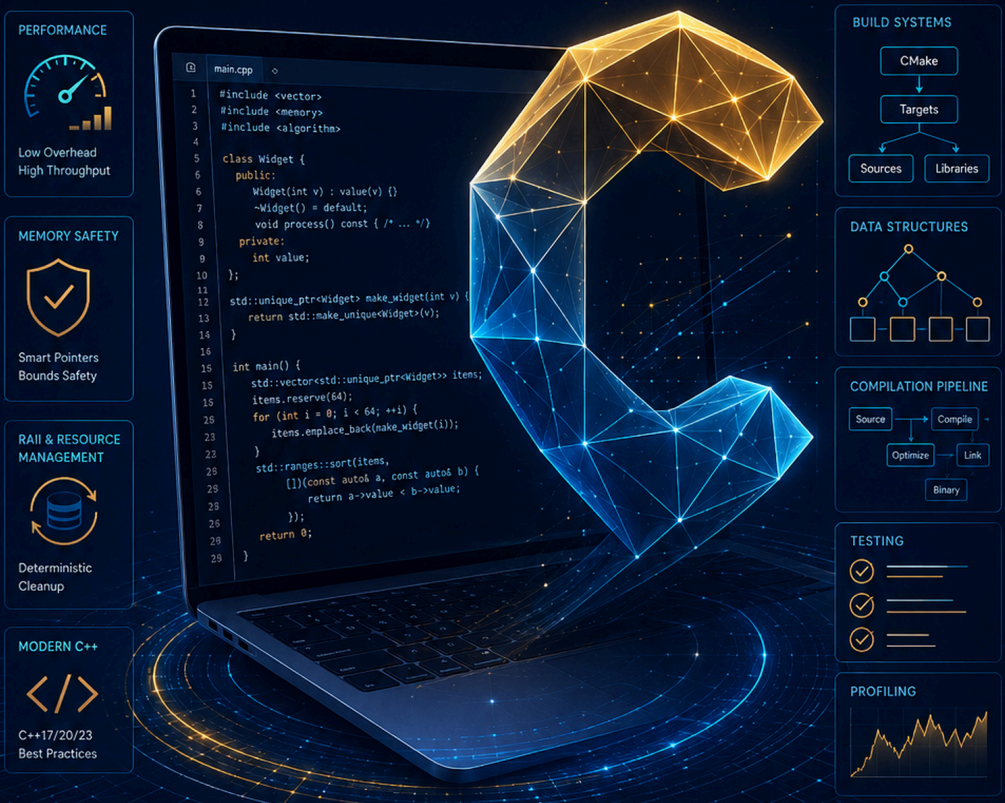
BEYOND ZERO SERIES

C++ Beyond Zero



AN INTERMEDIATE GUIDE TO

MODERN C++ DEVELOPMENT

LANCASTER SOLUTIONS LLC

C++

BEYOND ZERO

Intermediate Modern C++ Engineering, Safety, Performance, and Delivery

Expanded Professional Edition

Architecture • maintainability • testing • security • performance • deployment • guided labs • production capstone

Curtis Wayne Lancaster II

Published by Lancaster Solutions LLC

Beyond Zero Series • Practical technology education for real-world builders

About This Expanded Edition

> **AUDIENCE:** Readers who completed C++ From Zero or understand classes, RAII, containers, templates, and basic CMake.

This revised edition develops the lifetime, ownership, build, diagnostic, concurrency, performance, and API judgment required to turn small programs into reliable C++ systems.

Technical baseline

Portable C++20 practice with C++23 features identified explicitly; current published ISO standard C++23.

What “intermediate” means in this series

You can already write small working programs or pages. This book focuses on what happens when the work must be changed safely, tested repeatedly, deployed predictably, and understood by someone other than its original author.

The objective is not to memorize every feature. It is to build judgment: choosing boundaries, stating contracts, designing failure behavior, gathering evidence, and balancing complexity against the real needs of a project.

Examples are intentionally compact. The labs, projects, and capstone ask you to supply missing production details such as validation, logging, authorization, migration, accessibility, or rollback. Those omissions are part of the learning design rather than permission to ignore them.

How to use the book

- Read the chapter thesis first and write down what you expect the technique to improve.
- Run or reproduce the example in a small, disposable project before transplanting it into a larger codebase.
- Complete the guided lab with tests and notes. Then attempt the independent challenge without copying the lab structure mechanically.
- Keep an engineering notebook containing assumptions, decisions, failed experiments, measurements, and follow-up work.
- Treat the capstone as a portfolio-quality delivery: source code is only one deliverable alongside evidence, documentation, and an operational plan.

Publication and educational use

Contents and Learning Roadmap

The roadmap is organized into four stages. Each stage introduces decisions that the next stage assumes you can explain and verify.

Stage	Focus	Coverage	Core topics
Part 1	Structure, Models, and Contracts	Chapters 1–4	Engineering a Multi-Target C++ Project, Value Categories, Moves, and Lifetime Reasoning, RAII, Ownership, and Smart Pointers, Interfaces, Composition, and Dependency Inversion
Part 2	Boundaries, Data, and Collaboration	Chapters 5–8	Templates, Concepts, and Generic Design, Ranges, Algorithms, and Data Transformation, Error Models: Exceptions, expected, and Status Values, Containers, Allocators, and Data-Oriented Choices
Part 3	Reliability, Scale, and Verification	Chapters 9–11	Concurrency, Threads, Atomics, and Work Coordination, Persistence, Serialization, and Schema Evolution, Testing, Sanitizers, Static Analysis, and Fuzzing
Part 4	Delivery, Operations, and Evolution	Chapters 12–14	Profiling, Benchmarking, and Mechanical Sympathy, API Design, ABI, and Library Boundaries, Dependencies, Packaging, CI, and Deployment

PART I — Structure, Models, and Contracts

1. Engineering a Multi-Target C++ Project
2. Value Categories, Moves, and Lifetime Reasoning
3. RAII, Ownership, and Smart Pointers
4. Interfaces, Composition, and Dependency Inversion

PART II — Boundaries, Data, and Collaboration

5. Templates, Concepts, and Generic Design
6. Ranges, Algorithms, and Data Transformation
7. Error Models: Exceptions, expected, and Status Values
8. Containers, Allocators, and Data-Oriented Choices

PART III — Reliability, Scale, and Verification

9. Concurrency, Threads, Atomics, and Work Coordination
10. Persistence, Serialization, and Schema Evolution
11. Testing, Sanitizers, Static Analysis, and Fuzzing

PART IV — Delivery, Operations, and Evolution

12. Profiling, Benchmarking, and Mechanical Sympathy
13. API Design, ABI, and Library Boundaries

14. Dependencies, Packaging, CI, and Deployment

Projects, capstone, and references

Project 1 — Versioned Telemetry Importer
Project 2 — Concurrent Job Processor
Capstone — Offline Asset Inventory Engine
Engineering Review Checklists
Twelve-Week Practice Plan
Quick Reference
Glossary
Official Sources and Further Study

PART I

Structure, Models, and Contracts

Chapters 1–4

1. Engineering a Multi-Target C++ Project
2. Value Categories, Moves, and Lifetime Reasoning
3. RAII, Ownership, and Smart Pointers
4. Interfaces, Composition, and Dependency Inversion

CHAPTER 1
Engineering a Multi-Target C++ Project

> **CHAPTER THESIS:** Intermediate C++ starts with a reproducible build graph, explicit compiler requirements, and targets whose dependencies communicate architecture.

Learning outcomes

- Explain how targets over global flags shapes boundaries, testability, and failure behavior in a maintainable implementation.
- Apply translation units in a focused implementation and identify the contract it creates for callers.
- Compare an implementation that uses build variants with a simpler alternative, then justify the added complexity.
- Evaluate the operational and maintenance consequences of dependency boundaries under realistic change and failure.
- Evaluate the cmake Target Graph example, identify assumptions about targets over global flags and translation units, and justify one viable alternative.
- For “Engineering a Multi-Target C++ Project”, produce evidence using compiler diagnostics, sanitizer output, focused tests, benchmark records, and reproducible CMake presets; state what the evidence proves and what remains uncertain.

The intermediate shift

The intermediate challenge in “Engineering a Multi-Target C++ Project” is not learning one more API; it is deciding where the behavior belongs and what must remain true when the surrounding system changes.

Targets over global flags, Translation units, Build variants, and Dependency boundaries reinforce one another, but they optimize different risks. The chapter asks you to decide which concern owns each rule, where translation occurs, and how the design behaves when one assumption changes.

Credible evidence for “Engineering a Multi-Target C++ Project” includes compiler diagnostics, sanitizer output, focused tests, benchmark records, and reproducible CMake presets. Capture the setup and expected result so another reader can reproduce the claim instead of trusting a successful demonstration.

Targets over global flags

Modern CMake associates include paths, compile features, definitions, and libraries with targets. Directory-wide state makes unrelated code inherit accidental settings.

Implementation guidance. Create library and executable targets, declare PUBLIC, PRIVATE, and INTERFACE usage requirements, and keep warnings on project code rather than external dependencies.

For Targets over global flags, the design payoff is controlled change: callers depend on the contract while the implementation can evolve behind it.

Translation units

Each .cpp file is compiled separately and linked. Header contents affect every translation unit that includes them, so unnecessary includes increase rebuild time and coupling.

Implementation guidance. Use include guards or #pragma once, include what you use, and prefer declarations in headers with definitions in source files when templates are not required.

Operationally, Translation units must define failure ownership, retained context, retry safety, and the signal shown to users or operators.

Build variants

Debug, release, sanitizer, and coverage builds have different objectives. A release-only workflow hides diagnostics; a debug-only workflow hides optimization-sensitive behavior.

Implementation guidance. Document supported configurations and test more than one compiler or standard-library implementation where portability matters.

Verify Build variants through its narrowest public contract, then add adversarial cases that exercise malformed input and dependency failure.

Dependency boundaries

A target should link only to what it needs. Cycles between libraries often signal unclear ownership.

Implementation guidance. Separate domain, application, infrastructure, and interface targets and let dependency direction point inward toward policy.

For maintenance, Dependency boundaries should communicate ownership through names, types, modules, and documentation rather than institutional memory.

Worked pattern

CMAKE TARGET GRAPH

```
cmake_minimum_required(VERSION 3.25)
project(study_tracker LANGUAGES CXX)

add_library(study_domain
    src/task.cpp
```

```
    src/task_service.cpp
)
target_include_directories(study_domain PUBLIC include)
target_compile_features(study_domain PUBLIC cxx_std_20)

add_executable(study_cli src/main.cpp)
target_link_libraries(study_cli PRIVATE study_domain)
```

Walkthrough

1. The library exposes headers and a language requirement to consumers.
2. The executable links to the domain target instead of compiling its files again.
3. Additional adapters can become separate targets without creating a global flag tangle.

Design decisions to notice

- Targets over global flags: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Translation units: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Build variants: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- Dependency boundaries: What responsibility is being made explicit, and which caller or layer should remain unaware of its implementation details?
- In the cmake Target Graph, which decisions express the policy described in “Engineering a Multi-Target C++ Project”, and which are replaceable mechanisms behind the boundary?
- For “Engineering a Multi-Target C++ Project”, which guarantees are supplied by the C++ type system, standard library, compiler, and build toolchain, and which still require explicit validation, cleanup, monitoring, or documentation?

Failure modes and diagnostic thinking

- Using file(GLOB) and silently missing build-system regeneration expectations.
- Putting warning flags in INTERFACE usage requirements and forcing them onto consumers.
- A header that includes an entire framework for one type declaration.
- Link cycles between modules that should communicate through an interface.

> **DIAGNOSTIC EXERCISE:** Reproduce this risk in the smallest useful example: Using file(GLOB) and silently missing build-system regeneration expectations. Identify the first observable symptom, the earliest detection boundary, one preventive control, and one operational signal that would distinguish recurrence from a different failure.

Guided lab

> **OBJECTIVE:** Split a single-executable project into domain library, persistence library, CLI, and tests.

Phase 1: Clarify the contract

Turn the objective into acceptance criteria: Split a single-executable project into domain library, persistence library, CLI, and tests. Name trusted and untrusted inputs, explicit non-goals, and the result a reviewer can observe.

Phase 2: Create a baseline

Capture a baseline for “Engineering a Multi-Target C++ Project” using compiler diagnostics, sanitizer output, focused tests, benchmark records, and reproducible CMake presets. Preserve the command, fixture, environment, or screenshot needed to repeat it.

Phase 3: Implement the smallest safe change

Implement the smallest coherent change that advances the objective. Keep targets over global flags behind a named boundary and verify each increment before adding the next.

Phase 4: Verify normal and hostile conditions

Exercise the normal case plus malformed input, boundary values, and a realistic failure involving translation units. Include cancellation, timeout, rollback, fallback, or recovery where the chapter makes it relevant.

Phase 5: Review and communicate

Review the evidence with the objective in view. Record the chosen design, the rejected alternative, remaining risk, and the signal that would justify revisiting build variants.

Independent challenge

Add GCC and Clang CI builds with debug and release configurations.

> **CONSTRAINT:** Complete the independent challenge with a different ownership model from the worked pattern. Explain how the change affects failure behavior, testing evidence, and maintenance cost.

Stretch challenge

Create install and package rules so another CMake project can consume the domain library.

Chapter review

1. What problem does targets over global flags solve, and what new complexity can it introduce when overused?
2. What problem does translation units solve, and what new complexity can it introduce when overused?
3. What problem does build variants solve, and what new complexity can it introduce when overused?

4. What problem does dependency boundaries solve, and what new complexity can it introduce when overused?
5. Which evidence would demonstrate that targets over global flags remains correct under failure, and which part still requires representative measurement or human review?
6. Under what project constraints would a simpler alternative to dependency boundaries be the more responsible choice?

Definition of done

- The objective is demonstrated for the normal path and failure cases relevant to targets over global flags.
- The contract around translation units is explicit, smaller than its implementation, and exercised by evidence.
- The verification does not depend on hidden secrets, machine-specific paths, or uncontrolled state while testing build variants.
- A reviewer can trace the decision about dependency boundaries from requirement to implementation and evidence.
- Known limitations, operational signals, and follow-up work for “Engineering a Multi-Target C++ Project” are recorded with an owner or review trigger.