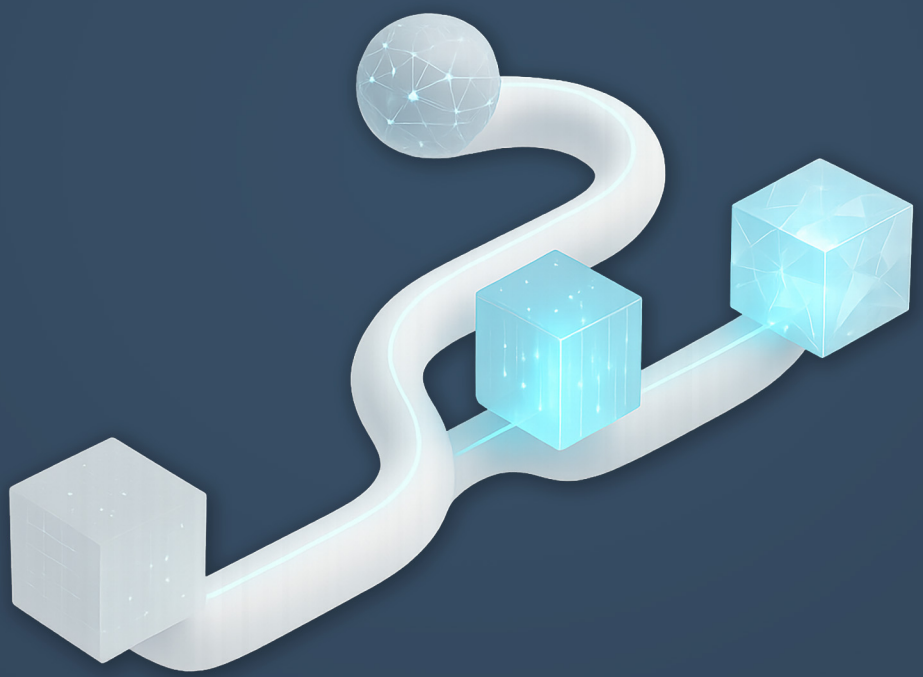
https://simplifycpp.org

Understanding the Compilation Pipeline in C++

Preprocessing -> Parsing -> IR (Intermediate Representation)

-> Optimization -> Linking



Prepared By Ayman Alheraki

Understanding the Compilation Pipeline in C++

Preprocessing → Parsing → IR (Intermediate Representation)
→ Optimization → Linking

Prepared by Ayman Alheraki

simplifycpp.org

December 2025

Contents

Author's Introduction

The C++ compilation pipeline is often presented as a simple arrow diagram:

$$\text{Preprocessing} \rightarrow \text{Parsing} \rightarrow \text{IR} \rightarrow \text{Optimization} \rightarrow \text{Linking}$$

However, in modern production compilers—such as Clang/LLVM, GCC, MSVC, and others—each arrow hides a rich world of sophisticated data structures, carefully ordered passes, and performance-critical heuristics.
This mini-booklet aims to provide a concise, practical, and modern (2020 and beyond) view of the compilation pipeline for C++ developers who:

- have already written non-trivial C++ code,
- care about performance, correctness, and debuggability,
- want to understand what really happens when they invoke g++, clang++, or cl.

Our focus is on the conceptual pipeline rather than a specific compiler implementation. Nonetheless, we will use real examples from Clang/LLVM, GCC, and common build workflows on modern systems (Linux, macOS, Windows).
Throughout the booklet you will find:

- short code snippets in C++20/23 that illustrate how modern features interact with the pipeline;

- practical compiler options to inspect intermediate stages (preprocessed source, IR, optimization reports);
- exercises (with solutions) at the end of each chapter to help you verify your understanding and connect theory to real tooling;
- references to books, standards, and technical articles if you wish to dive deeper.

This is not a book about writing a compiler from scratch. It is a practical guide for C++ developers who want to "see behind the curtain" and use this knowledge to write better, safer, and faster code.

Chapter 1

Overview of the C++ Compilation Pipeline

1.1 High-Level Stages

At a high level, the compilation of a C++ translation unit (.cpp file plus its included headers) goes through these stages:

1. Preprocessing (lexical transformations, macro expansion, file inclusion).
2. Parsing and semantic analysis (building an abstract syntax tree, type checking, template instantiation, overload resolution).
3. Intermediate Representation (IR) construction (e.g., LLVM IR, GCC GIMPLE/SSA).
4. Optimization on the IR and on lower-level representations.
5. Code generation to machine code and linking into an executable or library.

Most modern compilers separate the process into a front-end, a middle-end, and a back-end:

Front-end Preprocessing, parsing, semantic analysis, and initial IR generation.

Middle-end Machine-independent optimizations on IR.

Back-end Machine-dependent optimizations and instruction selection, followed by assembly or object code generation.

In practice, compilers also add auxiliary stages: diagnostics, static analysis, debug info generation, sanitizers, and link-time optimization (LTO).

1.2 A Small Running Example

We will use a small C++20 example as a running reference:

```
#include <vector>
#include <numeric>
#include <concepts>

template <std::integral T>
T sum_vector(const std::vector<T>& v)
{
    return std::accumulate(v.begin(), v.end(), T{0});
}

int main()
{
    std::vector<int> data{1, 2, 3, 4, 5};
    return sum_vector(data);
}
```

Even this simple example exercises:

- preprocessing (header inclusion),

- parsing (templates, concepts, standard library symbols),
- template instantiation for sum_vector<int>,
- optimization (inlining std::accumulate, loop unrolling, vectorization),
- linking against the C++ standard library.

1.3 Inspecting Stages with Clang/LLVM

On a modern toolchain with Clang/LLVM, you can inspect several stages:

```
# Preprocessed output
clang++ -std=c++20 -E main.cpp -o main.i

# LLVM IR (unoptimized)
clang++ -std=c++20 -O0 -S -emit-llvm main.cpp -o main.ll

# LLVM IR (optimized)
clang++ -std=c++20 -O2 -S -emit-llvm main.cpp -o main_O2.ll

# Object file and final executable
clang++ -std=c++20 -O2 main.cpp -o main
```

1.4 Exercises

E1 Run the compilation commands above on a modern Clang/LLVM toolchain and inspect:

- the preprocessed file (main.i),
- the unoptimized IR (main.ll),

- the optimized IR (main_O2.ll).

Note how different the IR becomes after optimization.

E2 Modify the example by adding constexpr and noexcept where possible. Regenerate the IR and observe any changes.

E3 Compare the assembly generated by -O0, -O2, and ;-O3 for the same source file. What are the most visible differences?

Solutions to Exercises

Solution to E1. The preprocessed file main.i is much larger and includes all headers expanded inline. The unoptimized IR still contains function calls to library routines and straightforward control flow, while the optimized IR tends to inline small functions, eliminate unused variables, and simplify loops. You may also see metadata for debug information and optimization hints.

Solution to E2. Adding constexpr and noexcept may allow the compiler to evaluate some expressions at compile time and simplify exception tables. In IR, some functions may disappear entirely (constant-folded), and exception-handling constructs may be reduced.

Solution to E3. At -O0, the assembly closely mirrors the source structure and retains many function calls and local variables. At -O2, you often see inlining, register allocation, and reduced branching. At ;-O3, you may see more aggressive inlining, loop unrolling, and vectorization, which can significantly change the control flow and structure of the emitted code.

Chapter 2

Preprocessing in Modern C++ Toolchains

2.1 What the Preprocessor Really Does

The C/C++ preprocessor is a textual transformation engine that operates before the actual C++ grammar is applied. It handles:

- macro expansion (#define),
- conditional compilation (#if, #ifdef, …),
- file inclusion (#include),
- line markers and diagnostics (#line, #error, …),
- feature-testing macros introduced by modern standards.

Although modern C++ encourages minimizing heavy macro usage in favour of templates, constexpr, and generic programming, the preprocessor remains fundamental for:

- header inclusion and modularization boundaries,
- platform-specific code paths,
- configuration flags in large projects,
- bridging with legacy code.

2.2 Modern Uses and Misuses

2.2.1 Feature Detection and Configuration

Feature-testing macros (e.g., ___cpp_concepts, ___cpp_coroutines) allow conditional compilation depending on compiler support:

```
#if defined(___cpp_concepts) && ___cpp_concepts >= 201907L
   #define HAS_CONCEPTS 1
#else
   #define HAS_CONCEPTS 0
#endif
```

This style is still common in modern libraries that support multiple C++ standards on multiple compilers.

2.2.2 Header Guards vs. #pragma once

Traditional header guards:

```
#ifndef MY_LIB_WIDGET_HPP
#define MY_LIB_WIDGET_HPP

// header contents

#endif // MY_LIB_WIDGET_HPP
```

#pragma once is widely supported by modern compilers and simplifies the pattern, but header guards remain fully portable and explicit.

2.2.3 Interaction with Modules

C++20 modules reduce reliance on textual inclusion, but they do not eliminate the preprocessor entirely. Many compilers support a phased approach where:

- legacy headers are still preprocessed and included;
- module interface units may limit preprocessor use for better isolation;
- build systems gradually adopt header-unit and module concepts.

2.3 Inspecting Preprocessor Output

On a modern compiler:

```
# Clang or GCC
g++   -std=c++20 -E main.cpp -o main.i
clang++ -std=c++20 -E main.cpp -o main.i
```

You can then open main.i and observe:

- all #includes expanded to full contents,
- all macros expanded,
- conditional code stripped according to compile flags.

This is extremely useful for debugging macro-related issues and understanding header bloat.

2.4 Exercises

P1 Create a small project with two headers and one source file. Use header guards in one header and #pragma once in the other. Generate the preprocessed output and verify that each header is included only once.

P2 Write a macro that logs file name, line, and function name, then expand it in the preprocessed file to see the transformation.

P3 Create two different feature-detection branches (using ___cpp_concepts or ___cplusplus) and verify what code remains after preprocessing with different compiler versions.

Solutions to Exercises

Solution to P1. In the preprocessed file, each distinct header should appear once regardless of which inclusion style you use, as long as header guards or #pragma once are correct. If you see duplicate header bodies, your include guards are likely wrong.

Solution to P2. A macro such as:

```
#define LOG() log(__FILE__, __LINE__, __func__)
```

expands directly to the log call with string and integer literals in the preprocessed file. This helps you reason about macro expansions and debug mismatched parentheses or arguments.

Solution to P3. By changing the standard flag (-std=c++17 vs -std=c++20), you will see different branches remain in the preprocessed file. This illustrates how portability layers for multiple standards are implemented.

Chapter 3

Parsing and Semantic Analysis

3.1 From Tokens to Abstract Syntax Trees

After preprocessing, the compiler front-end:

1. Lexes the character stream into tokens (identifiers, keywords, literals, operators).
2. Parses tokens according to the C++ grammar to build an Abstract Syntax Tree (AST).
3. Performs semantic analysis:
    - name lookup,
    - type checking,
    - overload resolution,
    - template argument deduction and instantiation,
    - concept satisfaction checks (C++20),

- access control, etc.

C++ is notoriously difficult to parse due to:

- context-sensitive grammar (e.g., » in templates vs shift operator),
- templates and dependent types,
- overload resolution and ADL,
- new features such as concepts and modules.

3.2 Error Messages and Diagnostics

Modern compilers invest enormous effort in diagnostics:

- descriptive error messages,
- fix-it hints,
- notes showing template instantiation backtraces,
- references to standard library headers and constraints.

Understanding that these diagnostics are generated during parsing and semantic analysis helps you interpret them properly. Complex template errors, for example, often result from earlier mistakes in type deduction or concept satisfaction.

3.3 Inspecting ASTs

Clang provides tools to dump ASTs:

```
clang++ -std=c++20 -Xclang -ast-dump -fsyntax-only main.cpp
```

This prints a tree-like representation of parsed constructs. It is very useful for:

- understanding how templates are instantiated,
- exploring how certain language constructs map to AST nodes,
- writing static analysis or refactoring tools.

3.4 Exercises

S1 Write a small template function using concepts and intentionally break a constraint. Observe the diagnostic produced by your compiler.

S2 Use Clang's -Xclang -ast-dump -fsyntax-only to inspect the AST of a simple class template with two member functions. Identify where each member appears in the AST.

S3 Experiment with a tricky C++ construct (e.g., dependent names, typename keyword) and see how the compiler's diagnostics guide you to a correct fix.

Solutions to Exercises

Solution to S1. A broken concept constraint usually yields an error message indicating which requirement failed, often with a note showing the location of the concept definition and the failed substitution. This arises during semantic analysis after the AST is constructed and constraints are checked.

Solution to S2. In the AST dump, class templates appear as nodes with child nodes for each member. You can identify the constructor, methods, and data members as separate nodes with associated types.

Solution to S3. Dependent name errors often mention missing typename or template keywords. These messages show that the parser and semantic analyzer require explicit disambiguation in contexts where the name could be either a type or a value.

Chapter 4

Intermediate Representation (IR)

4.1 Why IR Exists

Intermediate Representation (IR) acts as a bridge between:

- the language-specific front-end (C++ in our case),
- and the machine-specific back-end (x86-64, ARM, RISC-V, …).

Benefits of using IR include:

- reusing the same optimization passes for multiple languages,
- applying machine-independent optimizations once,
- enabling sophisticated analyses (data-flow, alias analysis, etc.),
- serving as a portable, lower-level “virtual assembly”.

4.2 LLVM IR in Practice

LLVM IR is a typed, SSA-based (Static Single Assignment) IR. At a very rough level, it looks like a RISC-style assembly language with infinite registers.
Example: a simplified IR for a function that sums an array may look like:

```
define i32 @sum(i32* %data, i64 %n) {
entry:
  %i = alloca i64
  %acc = alloca i32
  store i64 0, i64* %i
  store i32 0, i32* %acc
  br label %loop

loop:
  %i_val = load i64, i64* %i
  %cmp = icmp ult i64 %i_val, %n
  br i1 %cmp, label %body, label %exit

body:
  %elem_ptr = getelementptr inbounds i32, i32* %data, i64 %i_val
  %elem = load i32, i32* %elem_ptr
  %acc_val = load i32, i32* %acc
  %new_acc = add i32 %acc_val, %elem
  store i32 %new_acc, i32* %acc
  %next_i = add i64 %i_val, 1
  store i64 %next_i, i64* %i
  br label %loop
```

```
exit:
  %final_acc = load i32, i32* %acc
  ret i32 %final_acc
}
```

4.3 GCC's GIMPLE and RTL

GCC uses multiple IRs:

- GIMPLE: a simplified, three-address code-like tree representation amenable to high-level optimizations.
- SSA form on top of GIMPLE for data-flow optimizations.
- RTL (Register Transfer Language) closer to the target machine, used in later optimization and code generation passes.

Conceptually, GIMPLE plays a similar role to LLVM IR in the middle-end, while RTL corresponds to a lower-level representation before actual machine code emission.

4.4 Generating IR from C++20/23 Code

Using Clang:

```
clang++ -std=c++20 -O0 -S -emit-llvm main.cpp -o main.ll
clang++ -std=c++20 -O2 -S -emit-llvm main.cpp -o main_O2.ll
```

Using GCC with GIMPLE dumps (options may vary by version):

```
g++ -std=c++20 -O2 -fdump-tree-gimple main.cpp
```

Inspecting these dumps reveals how complex templates, lambdas, range-based for loops, and other modern constructs map to lower-level control flow and operations.

4.5 Exercises

IR1 Generate LLVM IR for a function template that uses std::span and std::ranges::for_each. Inspect how the range-based construct is lowered.

IR2 Use GCC's GIMPLE dump to examine how a range-based for loop over std::vector<int> is transformed.

IR3 Compare IR generated for a function marked inline versus one not marked inline at -O0 and -O2. What changes do you observe?

Solutions to Exercises

Solution to IR1. In LLVM IR, range-based loops are typically translated into explicit loops with indices or iterators. The std::ranges::for_each call may get inlined and replaced by a plain loop if optimization is enabled.

Solution to IR2. The GIMPLE dump shows a for loop with explicit iterator increments or index increments, plus calls to operator[] or begin()/end(). Templates are instantiated before GIMPLE generation, so you usually see concrete types rather than templates in the IR.

Solution to IR3. At -O0, the presence or absence of inline rarely affects IR generation, as no inlining happens. At -O2, functions marked inline or used in a header may be more aggressively inlined. In the IR, this appears as the body of the function being merged into the caller and the call instruction disappearing.

Chapter 5

Optimization

5.1 Goals of Optimization

Optimizations aim to improve:

- performance (speed, memory, energy),
- code size,
- sometimes debug information quality (or at least preserve it as much as possible).

Common families of optimizations include:

- dead code elimination,
- constant propagation and folding,
- loop transformations (unrolling, fusion, vectorization),
- inlining,

- common subexpression elimination,
- alias analysis and memory optimization,
- interprocedural optimization (IPO).

5.2 Optimization Levels

Modern compilers provide preset optimization levels:

-O0 No optimization; maximize compile speed and debug friendliness.

-O1 Simple and fast optimizations.

-O2 Aggressive optimizations suitable for production builds.

-O3 More aggressive optimizations, including heavier loop transformations.

-Os Optimize for size.

-Ofast Enable non-standard, potentially unsafe optimizations (e.g., violate strict IEEE floating-point rules).

5.3 Profile-Guided Optimization (PGO)

Profile-Guided Optimization uses runtime profile data:

1. Compile the program with instrumentation.
2. Run representative workloads to collect execution profiles.
3. Rebuild the program, feeding the profile to the compiler.

On Clang/LLVM, for example:

```
# Instrumented build
clang++ -std=c++20 -O2 -fprofile-instr-generate main.cpp -o main_instrumented

# Run to collect profile data
LLVM_PROFILE_FILE="profile.profraw" ./main_instrumented

# Convert and use profile
llvm-profdata merge -output=profile.profdata profile.profraw
clang++ -std=c++20 -O2 -fprofile-instr-use=profile.profdata main.cpp -o main_pgo
```

5.4 Link-Time Optimization (LTO)

LTO performs optimization across translation units at link time. Modern toolchains support:

- Full LTO: whole-program optimization at link stage.
- Thin LTO: scalable LTO that trades some precision for faster builds.

Example with Clang:

```
clang++ -std=c++20 -O2 -flto -c a.cpp -o a.o
clang++ -std=c++20 -O2 -flto -c b.cpp -o b.o
clang++ -std=c++20 -O2 -flto a.o b.o -o app
```

5.5 Optimization Reports

Many compilers now offer optimization reports to help developers understand which transformations were applied:

```
# Clang: loop and inline reports
clang++ -std=c++20 -O2 -Rpass=loop-vectorize -Rpass=inline main.cpp

# GCC: -fopt-info
g++ -std=c++20 -O2 -fopt-info-vec -fopt-info-inline main.cpp
```

These reports bridge the gap between source and IR-level optimization decisions.

5.6 Exercises

O1 Build a small numerical kernel (e.g., dot product) with and without PGO. Measure runtime on a large input to see the effect.

O2 Enable vectorization reports on Clang or GCC and adjust your code to help the compiler auto-vectorize a loop (e.g., remove unnecessary branches).

O3 Experiment with LTO on a multi-file C++ project and observe compile-time and runtime changes.

Solutions to Exercises

Solution to O1. PGO often improves hot paths (tight loops, frequently called functions) by reordering branches, improving code layout, and refining inlining decisions. You should see reduced runtime for representative workloads, especially in branch-heavy or cache-sensitive code.

Solution to O2. Vectorization reports will typically indicate why a loop was or was not vectorized (e.g., potential data dependencies, unknown alignment). By simplifying the loop structure and providing alignment hints, you help the compiler decide in favor of vectorization.

Solution to O3. LTO can inline functions across translation units and remove unused code, often reducing binary size and improving performance. However, link time may increase, so LTO is usually reserved for release builds.

Chapter 6

Code Generation and Linking

6.1 From IR to Machine Code

The back-end translates IR (or its lowered form) into target-specific assembly and then into object code. Key tasks include:

- instruction selection,
- register allocation,
- scheduling and pipeline-aware optimizations,
- inserting calling convention machinery,
- emitting debug information and unwind tables.

Example (Clang):

```
clang++ -std=c++20 -O2 -S main.cpp -o main.s   # assembly
clang++ -std=c++20 -O2 -c main.cpp -o main.o   # object file
```

6.2 Linking Basics

The linker is responsible for:

- resolving symbol references between object files and libraries,
- performing relocation and layout of code and data,
- constructing executable or shared library formats (ELF, PE, Mach-O),
- applying late optimizations in LTO scenarios.

From the C++ perspective, linking must handle:

- multiple definitions and ODR (One Definition Rule) violations,
- templates and inline functions defined in headers,
- static and dynamic libraries,
- symbol visibility and ABI compatibility.

6.3 Static vs. Dynamic Linking

Static linking Libraries are copied into the final binary at link time.

Dynamic linking Libraries are loaded at runtime (e.g., .so, .dll, .dylib).

Modern systems often combine both: core dependencies may be dynamic, while performance-critical or self-contained tools may be built statically.

6.4 Practical Linking Scenarios

6.4.1 Linking with the C++ Standard Library

On typical Linux systems:

```
g++ -std=c++20 main.cpp -o main      # links libstdc++
clang++ -std=c++20 main.cpp -o main  # links libc++ or libstdc++
```

On Windows with MSVC:

```
cl /std:c++20 /EHsc main.cpp
```

In each case, the driver automatically pulls the correct runtime and standard libraries.

6.4.2 Diagnostics at Link Time

Common link-time errors include:

- "undefined reference to" — symbol declared but not defined or not linked in;
- "multiple definition of" — ODR violation or conflicting objects;
- ABI mismatches between libraries and application (e.g., different standard library versions).

A good mental model of the linking stage helps you reason about these issues quickly.

6.5 Exercises

L1 Split a small C++ program across two source files, compile them separately, then link. Introduce a deliberate undefined symbol and observe the linker error.

L2 Build a static library and a shared library providing the same function, then create two executables linking against each variant. Measure binary size and startup time.

L3 Create a small project where you vary symbol visibility (e.g., using ___attribute___((visibility("hidden"))) or compiler-specific annotations) and observe resulting symbol tables with nm or objdump.

Solutions to Exercises

Solution to L1. The linker error will show which symbol is undefined and in which object file it is referenced. This illustrates how the linker connects declarations to definitions across objects.

Solution to L2. Executables linked statically tend to be larger but more self-contained; dynamically linked executables may have smaller on-disk size but depend on shared libraries at runtime. Startup time differences depend on OS and loader behavior.

Solution to L3. By reducing symbol visibility, you can shrink dynamic symbol tables and enable better inlining and dead code elimination at LTO time, especially for large shared libraries.

Chapter 7

Putting It All Together: A Modern Workflow

7.1 A Typical Release Build Pipeline

A modern C++ release build might involve:

1. Compiling each translation unit with -std=c++20, -O2 or -O3, and -DNDEBUG.
2. Enabling PGO on performance-critical components.
3. Using LTO for whole-program optimization.
4. Enabling warnings-as-errors (-Werror) and static analysis tools.
5. Building platform-specific binaries with appropriate ABIs and standard libraries.

7.2 Debug vs. Release: Different Pipelines

Debug builds -O0 or -Og, full debug symbols, sanitizers enabled, fast incremental builds.

Release builds -O2 or -O3, LTO, PGO, minimal debug info or separate symbol files.

The underlying pipeline (preprocess, parse, IR, optimize, link) is the same, but the chosen options dramatically change what optimizations run and how much information is preserved.

7.3 Exercises

W1 Design separate debug and release build configurations (e.g., using CMake) that differ in optimization level, debug symbols, and LTO. Build and compare binary size and runtime.

W2 Add a sanitizer (e.g., AddressSanitizer) to your debug build and intentionally introduce a memory error to see how the instrumented binary reports it.

W3 Experiment with a C++20 modules build (if your compiler supports it) and observe any differences in compilation time and binary layout compared to heavy header-based builds.

Solutions to Exercises

Solution to W1. Release builds usually produce smaller, faster binaries due to optimizations and dead-code removal, but they take longer to compile and are harder to debug. Debug builds compile quickly and are easy to inspect in a debugger, but may run much slower.

Solution to W2. Sanitizers insert additional runtime checks at various stages of execution, often implemented via instrumentation at the IR level. They can dramatically improve bug detection at the cost of slower runtimes, making them ideal for testing but not for production deployment.

Solution to W3. Modules can reduce rebuild times by avoiding repeated preprocessing and parsing of large headers. This benefit appears especially in large projects with many translation units that depend on heavy standard or third-party headers.

Final Conclusion

The C++ compilation pipeline is much more than a black box that turns .cpp files into executables. It is a carefully engineered sequence of transformations:

Preprocessing → Parsing & Semantics → IR → Optimization → Codegen & Linking

Understanding these stages gives you practical power:

- You can reason about error messages in terms of where they arise (parsing, semantic analysis, linking).
- You can read and interpret IR or GIMPLE dumps to see how modern constructs are lowered.
- You can leverage optimization flags, PGO, and LTO in an informed way instead of blindly toggling options.
- You can design build configurations that balance compilation time, debug friendliness, and runtime performance.

As C++ continues to evolve (C++20, C++23, C++26 and beyond), compilers will grow more sophisticated. New language features, such as modules, coroutines, and ranges, introduce fresh challenges and opportunities in each stage of the pipeline.

However, the core structure of the pipeline remains stable and continues to be one of the main reasons C++ can scale from tiny embedded systems to massive, performance-critical applications.

If this booklet helps you "see" your compiler differently—as a series of understandable stages rather than a mysterious monolith—then it has accomplished its goal. From here, you can dive deeper into compiler internals, write your own tools on top of IR, or simply write C++ with greater confidence, knowing what truly happens between your source code and the machine.

References

The following references are recommended for deeper study of the C++ compilation pipeline, intermediate representations, and modern optimization techniques:

- Bjarne Stroustrup, A Tour of C++ (3rd Edition), Addison-Wesley.
- Bjarne Stroustrup, The C++ Programming Language (4th Edition), Addison-Wesley.
- Scott Meyers, Effective Modern C++, O'Reilly.
- Matt Godbolt, Compiler Explorer (online tool to inspect generated assembly and compiler differences).
- LLVM Project, LLVM Language Reference Manual and official LLVM documentation.
- Clang Documentation, Clang Command Line Argument Reference and articles describing Clang's driver phases.
- GCC Documentation, GCC Internals Manual, especially the sections on GIMPLE, SSA, and optimization passes.
- IBM, ARM, and other vendor documentation for Link-Time Optimization (LTO), PGO, and target-specific compiler extensions.

- Articles and talks on modern C++ compilation and optimization (2020 and beyond), including:
  - technical blogs explaining Clang/LLVM IR dumps and optimization passes;
  - guides to PGO and LTO in recent GCC and Clang releases;
  - resources on C++20/23 features and their impact on compilation and optimization.