Gorgo Continuing C++

April 11, 2026

Contents

Appendix A: Build Systems and Tooling

Throughout this book you have compiled programs with a single `g++` command. That works for small programs, but real projects have dozens or hundreds of source files, external dependencies, and platform-specific requirements. A **build system** automates compilation so you do not have to type long commands or remember which files changed. This appendix covers CMake (the most widely used C++ build system), compiler flags, sanitizers, static analysis, and basic debugging.

CMake Basics

CMake is a **build system generator** — it reads a `CMakeLists.txt` file and generates the actual build files (Makefiles on Linux/macOS, Visual Studio projects on Windows).

A Minimal Project

Create a directory with two files:

```
my_project/
  CMakeLists.txt
  main.cpp
```

`CMakeLists.txt`:

```
cmake_minimum_required(VERSION 3.20)
project(MyProject LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 23)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

add_executable(myapp main.cpp)
```

`main.cpp`:

```
#include <iostream>

int main()
{
    std::cout << "Built with CMake!\n";
    return 0;
}
```

Build it:

```
mkdir build && cd build
cmake ..
make
./myapp
```

Multiple Source Files

```
add_executable(myapp
    main.cpp
    playlist.cpp
    audio.cpp
)
```

Libraries

```
# Create a library
add_library(audio audio.cpp codec.cpp)

# Link it to the executable
add_executable(myapp main.cpp)
target_link_libraries(myapp PRIVATE audio)
```

`PRIVATE` means `audio` is only needed by `myapp`, not by anything that uses `myapp`. Use `PUBLIC` if the dependency is also needed by consumers of the target.

Compiler Warnings

```
target_compile_options(myapp PRIVATE -Wall -Wextra -pedantic)
```

Including Headers

```
target_include_directories(myapp PRIVATE ${CMAKE_SOURCE_DIR}/include)
```

External Dependencies with `find_package`

```
find_package(Threads REQUIRED)
target_link_libraries(myapp PRIVATE Threads::Threads)
```

For popular libraries (Boost, OpenSSL, etc.), CMake provides built-in `Find` modules. For others, use `FetchContent` to download them:

```
include(FetchContent)
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 10.2.1
)
FetchContent_MakeAvailable(fmt)
target_link_libraries(myapp PRIVATE fmt::fmt)
```

Tip: CMake has a steep learning curve, but it is the de facto standard for C++ projects. Start with the basics above and learn more as your projects grow.

Compiler Flags and Warnings

The compiler flags you choose affect correctness, performance, and debuggability.

Warning Flags

Always compile with warnings enabled:

```
g++ -Wall -Wextra -pedantic -Werror main.cpp
```

Flag	Effect
`-Wall`	Enable most common warnings
`-Wextra`	Enable additional warnings
`-pedantic`	Warn about non-standard extensions
`-Werror`	Treat warnings as errors

Tip: Use `-Werror` in CI/CD pipelines to prevent warnings from accumulating. In development, you may want warnings without the hard stop.

Standard Selection

```
g++ -std=c++23 main.cpp   # C++23
g++ -std=c++20 main.cpp   # C++20
g++ -std=c++17 main.cpp   # C++17
```

Optimization Levels

Flag	Effect
`-O0`	No optimization (fastest compile, best debugging)
`-O1`	Basic optimization
`-O2`	Standard optimization (good for release)
`-O3`	Aggressive optimization
`-Os`	Optimize for size
`-Og`	Optimize for debugging

Debug Information

```
g++ -g main.cpp           # Include debug symbols
g++ -g -O0 main.cpp       # Debug build (best for debuggers)
g++ -O2 -DNDEBUG main.cpp # Release build (disables assert)
```

Sanitizers

Sanitizers are compiler features that instrument your code to detect bugs at run time. They add overhead but catch problems that are otherwise invisible.

AddressSanitizer (ASan)

Detects memory errors: buffer overflows, use-after-free, double-free, memory leaks:

```
g++ -fsanitize=address -g main.cpp -o main
./main
```

If your program has a memory bug, ASan prints a detailed error report with the exact location.

UndefinedBehaviorSanitizer (UBSan)

Detects undefined behavior: signed integer overflow, null pointer dereference, misaligned access:

```
g++ -fsanitize=undefined -g main.cpp -o main
```

ThreadSanitizer (TSan)

Detects data races in multithreaded programs (Chapter 10):

```
g++ -fsanitize=thread -g main.cpp -o main -pthread
```

Tip: Run your tests with sanitizers regularly. Many bugs — especially memory and threading bugs — are silent until they corrupt data or crash under production load. Sanitizers catch them early.

Combining Sanitizers

You can combine ASan and UBSan:

```
g++ -fsanitize=address,undefined -g main.cpp
```

But ASan and TSan cannot be used together — they instrument memory differently.

Static Analysis

Static analysis examines your code without running it, catching bugs that the compiler's warnings miss.

clang-tidy

`clang-tidy` is the most popular C++ linter. It checks for common mistakes, style issues, and modernization opportunities:

```
clang-tidy main.cpp -- -std=c++23
```

Useful check categories:

Category	What it checks
`bugprone-*`	Common bug patterns
`modernize-*`	Suggest modern C++ replacements
`performance-*`	Performance issues
`readability-*`	Code readability
`cppcoreguidelines-*`	C++ Core Guidelines compliance

cppcheck

`cppcheck` is a standalone static analyzer:

```
cppcheck --enable=all main.cpp
```

It catches issues like unused variables, null pointer dereferences, and resource leaks.

Compiler Warnings as Analysis

With `-Wall -Wextra -pedantic -Werror`, the compiler itself is a basic static analyzer. Start there before adding external tools.

Debugging with gdb/lldb

When your program crashes or produces wrong results, a debugger lets you step through the code line by line, inspect variables, and examine the call stack.

Basic gdb Commands

```
g++ -g -O0 main.cpp -o main
gdb ./main
```

Command	Effect
`run`	Start the program
`break main`	Set a breakpoint at `main`
`break file.cpp:42`	Breakpoint at line 42
`next`	Execute next line (step over)
`step`	Step into function call
`continue`	Run until next breakpoint
`print x`	Print the value of `x`
`backtrace`	Show the call stack
`info locals`	Show local variables
`quit`	Exit gdb

lldb

`lldb` is the LLVM debugger, used primarily on macOS. Its commands are similar:

gdb	lldb
`run`	`run`
`break main`	`breakpoint set --name main`
`next`	`next`
`step`	`step`
`print x`	`frame variable x` or `p x`
`backtrace`	`thread backtrace`

Debugging Tips

- Compile with `-g -O0` for the best debugging experience. Optimizations can reorder code and eliminate variables.
- Use `valgrind` as an alternative to ASan for memory debugging: `valgrind ./main`
- Core dumps: if a program crashes, the OS can save a core dump. Load it with `gdb ./main core` to examine the state at the time of the crash.

Tip: Learn to use a debugger early. `std::cout` debugging is tempting but slow and unreliable. A debugger shows you exactly what is happening, where, and why.

Key Points

- **CMake** is the standard C++ build system. `CMakeLists.txt` defines targets, sources, and dependencies.
- Use `add_executable` for programs, `add_library` for libraries, and `target_link_libraries` to connect them.
- **Compiler flags**: `-Wall -Wextra -pedantic` for warnings, `-std=c++23` for the standard, `-O2` for release, `-g -O0` for debug.
- **Sanitizers** catch runtime bugs: ASan (memory), UBSan (undefined behavior), TSan (data races). Use them in testing.
- **Static analysis** tools like `clang-tidy` and `cppcheck` catch bugs without running the code.
- **gdb/lldb** let you step through code, set breakpoints, and inspect variables. Compile with `-g -O0` for best results.

Exercises

1. **Think about it:** Why is CMake called a "build system generator" rather than a "build system"? What does it generate?
2. **Write a CMakeLists.txt** for a project with `main.cpp`, `audio.cpp`, and `audio.h`. Set the C++ standard to 23 and enable `-Wall -Wextra -pedantic`.
3. **Think about it:** Why should you compile with `-Wall -Wextra -pedantic` from the start of a project rather than adding them later?
4. **Calculation:** You have a program with a buffer overflow that only corrupts memory silently. Which sanitizer would catch it? What compiler flag would you use?
5. **Think about it:** AddressSanitizer and ThreadSanitizer cannot be used together. Why might that be? How would you test for both memory and threading bugs?
6. **Write a gdb session** (sequence of commands) that:
   - Sets a breakpoint at `main`
   - Runs the program
   - Steps through three lines
   - Prints a local variable called `count`
   - Continues to the end
7. **Think about it:** What is the difference between `-O0`, `-O2`, and `-O3`? When would you use each?
8. **Where is the problem?**

   ```
   add_executable(myapp main.cpp)
   target_link_libraries(myapp fmt)
   ```

   What is missing compared to the example in this chapter?
9. **Think about it:** Why does the text recommend running tests with sanitizers enabled? What kinds of bugs do sanitizers catch that tests alone miss?
10. **Set up a project** with CMake that has a `main.cpp` and a `math_utils.cpp/math_utils.h` library. The library should have a function `int factorial(int n)`. Build it with CMake, run it, and then compile with AddressSanitizer enabled and verify it runs cleanly.