Cloud Contraptions
TRAINING & CONSULTING

To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

The Zig Build System and Cross-Compilation

Class Duration

14 hours of live training delivered over 2 days.

Student Prerequisites

- Experience building software with a compiled language toolchain (C, C++, or Zig)
- Comfort with the command line and at least one build tool (Make, CMake, or similar)
- Basic familiarity with CI systems is helpful
- No prior Zig language experience required — the small amount of Zig syntax needed to read `build.zig` is taught in class

Target Audience

This course serves two audiences. The first is C and C++ teams who may never write a line of Zig but want `zig cc` as a hermetic, zero-dependency cross-compiler and the Zig build system as a saner alternative to their current toolchain — for cross-compiling to musl targets, pinning glibc versions, or escaping toolchain drift across developer machines and CI. The second is Zig developers who want to master `build.zig`, the package manager, and multi-target release pipelines.

Description

The Zig toolchain is two products in one download: a programming language, and a self-contained C/C++ compiler and build system that works on projects containing no Zig code at all. `zig cc` and `zig c++` are zero-dependency, drop-in replacements for `cc` and `c++` that cross-compile out of the box — pass `-target aarch64-linux-musl` and get a static Linux binary from a macOS laptop, with no sysroots, no Docker images, and no separately installed cross-toolchains. This course teaches that capability systematically: target triples, glibc version selection and musl static linking, and replacing fragile cross-compilation setups with a single tool.

From there the course goes deep on the build system itself: the anatomy of `build.zig` — modules, artifacts, steps, and user options — declarative package management with `build.zig.zon`, including vendoring C

dependencies, and building existing C/C++ projects with Zig instead of (or alongside) CMake and Make. Students wire builds into CI with GitHub Actions, exploit Zig's content-addressed caching, learn to debug builds when something goes wrong, and look ahead to the optimized build-runner work in the in-development 0.17 cycle. Students leave able to cross-compile a real C/C++ or Zig project to multiple targets from one machine and one configuration file.

Learning Outcomes

- Use `zig cc` and `zig c++` as drop-in replacements for existing C/C++ compilers, locally and in CI
- Cross-compile to any supported target with `-target` triples, selecting CPU features, OS, and ABI deliberately
- Choose between glibc (including pinning specific versions) and musl static linking, and explain the consequences of each
- Read and write `build.zig`: modules, artifacts, steps, options, and the dependency graph between them
- Manage dependencies with `build.zig.zon`, including hashing, lazy dependencies, and vendoring C libraries
- Build existing C/C++ projects with the Zig build system, incrementally or wholesale
- Explain Zig's caching model and use it to speed up local and CI builds
- Set up GitHub Actions pipelines that build and test a matrix of targets from a single runner
- Compare the Zig build system with CMake and Make and make an informed adoption decision
- Debug failing builds: verbose output, inspecting compile commands, and diagnosing linker issues
- Describe the optimized build-runner improvements in the in-development Zig 0.17 cycle and what they mean for large builds

Training Materials

Comprehensive courseware is distributed online at the start of class. All students receive a downloadable MP4 recording of the training.

Software Requirements

Students will need a free, personal GitHub account to access the courseware and to run the CI labs. Students will need permission to install Zig 0.16 and

Cloud Contraptions
TRAINING & CONSULTING

To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

Visual Studio Code on their computers — the Zig download is the only compiler required, on Windows, macOS, or Linux. If students are unable to configure a local environment, a cloud-based environment can be provided.

Training Topics

The Zig Toolchain as Build Infrastructure

- One ~50 MB download: compiler, linker, libc headers, build system
- Why Zig can compile C and C++: Clang and LLD under the hood
- Hermetic toolchains: ending “works on my machine”
- Who uses `zig cc` without writing Zig
- Zig 0.16 today and the 0.17 development cycle

zig cc and zig c++ as Drop-In Compilers

- `zig cc` and `zig c++` flag compatibility with cc/clang/gcc
- Substituting CC and CXX in an existing Makefile or CMake project
- What zero-dependency means: no sysroots, no cross-toolchain packages
- Bundled libcs and headers for every supported target
- Limits and gotchas: flags that differ, when to fall back

Targets and Triples

- Anatomy of a target triple: arch, OS, ABI
- `-target` with `zig cc` and target options in `build.zig`
- CPU feature selection: baseline, native, and explicit feature sets
- Querying supported targets
- Windows, macOS, Linux, and freestanding targets

glibc, musl, and libc Selection

- Dynamic glibc linking and pinning a glibc version (e.g. `-target x86_64-linux-gnu.2.31`)
- musl and fully static Linux binaries
- Choosing between glibc and musl: compatibility vs. portability
- libc on Windows and macOS targets
- Common deployment scenarios: old distros, containers, scratch images

Anatomy of build.zig

- The build function and the step graph
- Modules: units of Zig and C code that can be imported

Cloud Contraptions
TRAINING & CONSULTING

To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

- Artifacts: executables, static and shared libraries, objects
- Steps: build, install, run, test, and custom steps
- User options: `-D` flags, `b.option`, and conditional builds
- Compiling C/C++ sources: `addCSourceFiles`, flags, and defines
- Just enough Zig syntax to read and modify a build script

Package Management with build.zig.zon

- `build.zig.zon` format: name, version, dependencies, paths
- `zig fetch` and content hashes
- Lazy dependencies and conditional fetching
- Vendoring C dependencies as Zig packages
- Wrapping a third-party C library with its own `build.zig`
- Private registries, mirrors, and offline builds

Building Existing C/C++ Projects

- Strategy 1: keep Make/CMake, swap in `zig cc` as the compiler
- Strategy 2: port the build to `build.zig` entirely
- Translating compiler and linker flags
- Handling generated headers, configure steps, and platform checks
- Case study: cross-compiling a real C library to multiple targets

Caching and Build Performance

- Zig's content-addressed cache: local and global cache directories
- What invalidates the cache and what survives it
- Sharing caches across CI runs
- Incremental compilation status in 0.16/0.17
- The optimized build-runner work in the 0.17 dev cycle (in development)

CI Integration with GitHub Actions

- Installing Zig in CI: setup actions and version pinning
- A cross-compilation matrix from a single Linux runner
- Caching strategies for CI
- Running tests for foreign targets with QEMU and emulation
- Producing and uploading release artifacts for many platforms

Comparison with CMake and Make

- Declarative step graph vs. Makefile rules vs. CMake's two-stage model

To discuss this course and customizations:
Call: 434-509-5680 or Email: sales@cloudcontraptions.com

- Dependency management: build.zig.zon vs. FetchContent vs. submodules
- Cross-compilation ergonomics compared honestly
- IDE and tooling integration
- Migration paths and coexistence strategies

Debugging Builds

- `--verbose`, `--verbose-cc`, and `--verbose-link`
- Inspecting the exact compiler invocations Zig issues
- Diagnosing linker errors and missing libc symbols
- `zig build --summary all` and reading the step graph
- When the cache lies: forcing rebuilds safely