Cloud Contraptions
TRAINING & CONSULTING

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

Zig for C and C++ Programmers

Class Duration

14 hours of live training delivered over 2 days.

Student Prerequisites

- Significant professional experience writing C or C++
- Comfort with pointers, manual memory management, and a compiler toolchain
- No prior Zig experience required
- The C++-specific comparisons assume working knowledge of RAII, templates, and virtual dispatch

Target Audience

This course is for experienced C and C++ developers evaluating or adopting Zig. Rather than teaching programming fundamentals, every topic is presented as a translation: here is the C or C++ construct you know, here is the Zig equivalent, and here is why Zig made a different choice. It is ideal for systems programmers, embedded developers, and teams considering Zig for new components inside an existing C/C++ codebase.

Description

Zig is frequently described as a better C, and its interoperability story is unmatched: Zig speaks the C ABI natively, imports C headers directly with `@cImport`, machine-translates C source to Zig with `translate-c`, and compiles and links C and C++ files in the same build. This course leverages what C and C++ programmers already know. Each unit takes a familiar construct — raw pointers and arrays, `goto`-based cleanup or RAII destructors, `errno` or exceptions, `NULL`, preprocessor macros or templates, `malloc` or `new`, header files — and shows the Zig counterpart: slices with bounds, `defer` and `errdefer`, error unions, optionals, `comptime`, explicit allocators, and file-scoped modules with no preprocessor.

The course is honest about trade-offs. Students examine where C++ features like virtual dispatch and operator overloading map onto Zig idioms and where Zig deliberately declines to provide them, and they compare undefined behavior in C and C++ with Zig's safety-checked builds, where

the same classes of bugs become deterministic panics in Debug and ReleaseSafe modes. The second day focuses on migration: importing and linking C libraries, using `translate-c` as a starting point, and adding Zig to a real C/C++ codebase incrementally — one file, one component at a time — without a rewrite.

Learning Outcomes

- Translate C idioms — pointer arithmetic, sentinel-terminated strings, manual cleanup — into idiomatic Zig pointers, arrays, and slices
- Replace `goto` cleanup chains and RAII destructors with `defer` and `errdefer`, and articulate the trade-offs of each model
- Use error unions and `try` in place of `errno` conventions and C++ exceptions
- Eliminate `NULL`-related bugs with optionals and optional pointers
- Apply `comptime` where C uses macros and C++ uses templates, and explain what each approach can and cannot do
- Manage memory with explicit allocators instead of global `malloc`/`free` or `new`/`delete`
- Organize code with Zig's file-based modules and namespacing instead of headers and the preprocessor
- Import C headers with `@cImport`, link existing C libraries, and call Zig from C via the C ABI
- Use `translate-c` to bootstrap migration of C source files to Zig
- Plan and execute an incremental migration that mixes Zig and C in one codebase
- Contrast undefined behavior in C/C++ with Zig's safety-checked builds and illegal-behavior model
- Identify where C++ features such as virtual dispatch and operator overloading map to Zig patterns and where Zig deliberately omits them

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. Students will need permission to install Zig 0.16 and Visual Studio Code (with

the Zig language extension) on their computers. No separate C compiler is required — the Zig toolchain compiles the C and C++ code used in class. If students are unable to configure a local environment, a cloud-based environment can be provided.

Training Topics

Orientation: Why Zig Exists

- Zig's design goals: no hidden control flow, no hidden allocations
- What Zig keeps from C and what it removes
- The Zig toolchain as a C/C++ compiler
- Zig 0.16 at a glance and language stability before 1.0

Pointers, Arrays, and Slices

- Single-item pointers vs. many-item pointers vs. C's one pointer type
- Arrays with compile-time-known length
- Slices: pointer plus length as a first-class type
- Sentinel-terminated pointers and C string interop
- Pointer arithmetic: where it went and what replaces it
- Alignment and `@alignCast`

Cleanup: defer and errdefer vs. RAII and goto

- The `goto cleanup` pattern in C and its failure modes
- RAII in C++: destructors, scope, and hidden control flow
- `defer` for unconditional cleanup
- `errdefer` for error-path-only cleanup
- Partial initialization and multi-resource acquisition
- Trade-offs: explicitness vs. automation

Errors: Error Unions vs. errno and Exceptions

- `errno`, return codes, and out-parameters in C
- Exceptions in C++: cost, control flow, and exception safety
- Zig error sets and error union types
- `try`, `catch`, and `switch` on errors
- Error return traces
- Errors as values: required handling enforced by the compiler

Optionals vs. NULL

- The billion-dollar mistake in C and C++

Cloud Contraptions
TRAINING & CONSULTING

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

- Optional types and optional pointers in Zig
- `orelse`, `if` unwrapping, and `.?`
- Designing APIs that cannot receive a surprise null

comptime vs. Macros and Templates

- What the C preprocessor does and what replaces it in Zig
- C++ templates vs. Zig generic functions with `comptime` parameters
- Types as values and `@typeInfo` reflection
- Compile-time computation without a second language
- What templates can do that comptime will not, and vice versa

Memory: Allocators vs. malloc and new

- Why Zig has no global allocator
- The `Allocator` parameter convention
- Arena, fixed-buffer, and general-purpose allocators
- Ownership and lifetime without constructors and destructors
- Leak detection with the testing allocator

Code Organization: Modules vs. Headers

- Files as structs: namespacing without a preprocessor
- `@import` vs. `#include`
- Visibility with `pub` vs. header/implementation splits
- No forward declarations, no include guards, no ODR

C and C++ Interop and Migration

- `@cImport` and importing C headers directly
- Linking existing C libraries with the Zig build system
- `translate-c`: machine-translating C source to Zig
- Exporting Zig functions with the C ABI for C callers
- Compiling C and C++ sources inside `build.zig`
- Incremental migration strategy: one component at a time
- Case study: introducing Zig into a live C codebase

Safety: UB in C/C++ vs. Safety-Checked Builds

- The catalogue of undefined behavior in C and C++
- Zig build modes: Debug, ReleaseSafe, ReleaseFast, ReleaseSmall
- Safety checks: bounds, overflow, null unwrap, unreachable
- Illegal behavior vs. undefined behavior

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

- What `unreachable` and `@setRuntimeSafety` actually do

Where C++ Features Map — and Deliberately Don’t

- Virtual dispatch: vtables by hand, function pointers, and tagged unions
- Operator overloading: why Zig says no
- Constructors, destructors, and move semantics vs. init/deinit conventions
- Generic containers without template metaprogramming
- Choosing Zig vs. staying with C++ for a given component