Zig
in Depth

A COMPLETE GUIDE TO MODERN SYSTEMS PROGRAMMING

- CLEAR, EXPLICIT DESIGN
- SAFE BY DEFAULT, WITHOUT OVERHEAD
- MAXIMUM PERFORMANCE
- POWERFUL COMPTIME
- CONCURRENCY MADE SIMPLE
- SEAMLESS C INTEROPERABILITY

Ivan Gebhart

Zig in Depth

A Complete Guide to Modern Systems Programming

Steve T. Publications

This book is available at https://leanpub.com/zigindepth

This version was published on 2026-07-14

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction: Why Zig?

The Systems Programming Landscape Today

Systems programming sits at the foundation of everything we compute. Operating systems, databases, game engines, embedded firmware, networking stacks, and compilers all share a common requirement: direct control over memory, CPU, and hardware resources without the overhead of a runtime system that makes decisions on your behalf. For decades, C has been the lingua franca of this domain. It is simple, ubiquitous, and produces efficient code. Yet C's simplicity comes at a steep price. The language provides no protection against buffer overflows, use-after-free bugs, integer overflow, or null pointer dereferences. These vulnerabilities are not theoretical concerns. They are the primary attack surface exploited in real-world security breaches, from Heartbleed to the countless remote code execution vulnerabilities found in network servers every year.

The response to C's shortcomings has been varied and sometimes contradictory. C++ added layers of abstraction, templates, exceptions, and a standard library that grew so large it became its own ecosystem, but the language retained backward compatibility with C, meaning all of C's pitfalls remain accessible. Rust emerged as the most ambitious alternative, offering memory safety through a borrow checker that tracks ownership at compile time. Rust has been enormously successful, adopted by Mozilla, Google, Amazon, and even integrated into the Linux kernel. But Rust's complexity is real: its lifetime system, trait bounds, and macro system create a steep learning curve, and the language continues to grow with new features each release.

Zig takes a different path entirely.

Andrew Kelley's Vision and the Origin of Zig

Zig was created by Andrew Kelley, first announced on February 8, 2016 [9], as a deliberate attempt to improve upon C without sacrificing its core virtues: simplicity, transparency, and minimalism. Kelley's frustration was not with C's

feature set, but with what C hides from the programmer. Consider the humble C for loop:

```
for (i = 0; i < n; i++) {
    // body
}
```

This seemingly simple construct involves branch prediction, stack manipulation, implicit integer promotion, and potential overflow behavior that varies by compiler optimization level. None of this is visible in the source code. In Zig, every cost must be explicit. If you want a loop, you write a while loop with all the control flow spelled out.

The language is named "Zig" because, as Kelley put it, it was meant to be the language that comes after C in the ASCII table, just as C succeeded B and B succeeded BCPL. The ambition is clear: Zig aims to replace C, not by adding features, but by removing the hidden behaviors that make C programs fragile.

Zig's development has been funded through the Zig Software Foundation, a 501(c)(3) non-profit organization founded in 2020 by Andrew Kelley, supported by corporate sponsorships and individual donations [4]. The project migrated from GitHub to Codeberg on November 26, 2025, citing reliability concerns with centralized hosting platforms and GitHub Actions bugs [8]. As of version 0.16.0, released on April 14, 2026, Zig is a self-hosted compiler with a comprehensive standard library, first-class cross-compilation support, and an integrated build system [3].

No Hidden Control Flow: The Core Philosophy

Zig's design can be summarized in one principle: if it is not written, it does not happen. This manifests in several concrete ways:

No hidden memory allocations. In many languages, string concatenation, exception throwing, or even function calls can silently allocate memory on the heap. In Zig, every allocation requires an explicit allocator parameter [4]. If a function needs to grow a buffer, you will see the allocator in its signature. The standard library never allocates behind your back. This means Zig code works seamlessly in environments where heap allocation is impossible or undesirable: real-time systems, operating system kernels, embedded devices with no memory manager, and WebAssembly modules targeting browsers.

No hidden control flow. Zig has no exceptions, no operator overloading, no property functions, no macros, and no preprocessor. When you read a Zig function, you can trace every possible execution path by reading the code from top to bottom. There is no way for a function call to jump to an exception handler defined elsewhere in the program. There is no way for the `+` operator to allocate memory, throw an error, or change control flow. It adds two numbers, and nothing else.

No undefined behavior. In C, integer overflow, null pointer dereference, use-after-free, and many other conditions are classified as undefined behavior, meaning the compiler is free to assume they never happen and optimize accordingly. This leads to subtle bugs that manifest only under specific optimization levels. Zig classifies these conditions as illegal behavior and inserts runtime safety checks in Debug and ReleaseSafe build modes [1]. In ReleaseFast and ReleaseSmall modes, these checks are disabled for maximum performance, but the programmer has chosen this trade-off explicitly through the build configuration.

Compile-time code execution. Instead of a preprocessor with macro expansion, Zig provides `comptime`, a mechanism that allows any Zig code to run at compile time [1]. This enables generic programming, metaprogramming, and compile-time computation without introducing a second language or syntax. The same language you use at runtime is available at compile time, making the mental model simpler and the tooling more consistent.

Where Zig Fits Among C, C++, Rust, and Go

To understand Zig's place in the ecosystem, it helps to compare it directly with its closest relatives:

Zig versus C. Zig is designed to be a drop-in replacement for C in many contexts [6]. It can compile C code directly using `zig cc`, link against C libraries without FFI bindings, and export functions with C ABI compatibility. The syntax is familiar to C programmers: structs, enums, pointers, arrays, and explicit memory management all work in conceptually similar ways. Where Zig differs is in safety: optional types replace null pointers, error unions make error handling explicit, integer overflow is caught by default [1], and the build system eliminates the need for Makefiles, CMake, or Autotools. A C programmer can learn Zig in a day; a Zig program compiled for C ABI compatibility can replace a C library without changing the calling code.

Zig versus C++. C++ has grown into a massive language with templates, exceptions, smart pointers, move semantics, concepts, coroutines, and an enormous standard library. This breadth is both its strength and its weakness: C++ can do almost anything, but understanding any given C++ codebase requires knowledge of which subset of features it uses. Zig deliberately stays small. It has no operator overloading, no templates (instead using comptime generics), no exceptions, no virtual dispatch in the language itself, and no standard containers beyond what the standard library provides. The tradeoff is that Zig code is easier to read, reason about, and maintain, at the cost of some convenience features that C++ provides.

Zig versus Rust. Rust and Zig share many goals: memory safety, performance, modern tooling, and C interoperability [6]. They diverge sharply in how they achieve those goals. Rust uses a borrow checker to prevent data races and memory bugs at compile time, enforcing ownership rules that can be challenging to learn. Zig relies on the programmer for manual memory management, similar to C, but provides tools like arena allocators, debug allocators with leak detection, and explicit allocator parameters to make allocation patterns clear. Rust's type system is more expressive, supporting traits, lifetimes, and a powerful macro system. Zig's type system is simpler, using comptime evaluation for generic programming and tagged unions for variant types. In practice, Rust tends to be the better choice when writing large concurrent systems where data race prevention is critical, while Zig tends to be the better choice when writing small, focused tools, compilers, embedded systems, or libraries where simplicity and C interoperability matter more.

Zig versus Go. Go prioritizes developer productivity with garbage collection, channels, goroutines, and a minimal syntax. Zig prioritizes control and predictability with manual memory management, explicit concurrency primitives (mutexes, atomics, condition variables), and a richer type system. Go programs are generally easier to write quickly; Zig programs give you more control over resource usage and performance characteristics.

How This Book Is Organized

This book is structured to take you from beginner to advanced Zig proficiency through a progressive sequence of topics. The first chapters establish the foundation: installation, basic types, variables, control flow, and the core data structures (pointers, arrays, slices, structs, enums, unions). You will learn how

Zig handles memory explicitly, how error handling works with try and catch, and how optional types eliminate null pointer bugs.

The middle chapters dive into Zig's distinctive features: comptime programming for generics and metaprogramming, the allocator parameter pattern for memory management, string handling and formatting, the build system, C interoperability, testing, debugging, and concurrency. Each chapter builds on the previous ones, introducing increasingly sophisticated patterns while reinforcing the core philosophy of explicitness.

The final chapters bring everything together with practical case studies: building a JSON parser, an HTTP server, and a binary protocol encoder. These projects demonstrate how real Zig applications are structured, tested, and built, showing you how to apply all the language features you have learned in realistic scenarios.

Throughout the book, every code example is complete and runnable. You can copy any example, save it to a file, and compile it with `zig build-exe` or test it with `zig test`. The examples are designed to be instructive rather than minimal, showing idiomatic patterns that you will encounter in real Zig codebases.

By the end of this book, you will understand not just how to write Zig code, but why Zig makes the design choices it does, when to use each feature, and how to structure Zig projects for maintainability and performance. You will be equipped to contribute to existing Zig projects, start your own, or make informed decisions about where Zig fits in your technology stack.

Chapter 1: Getting Started - Installation, Toolchain, and Your First Program

Installing Zig: Official Releases and Building from Source

Zig distributes as a single static binary that includes the compiler, linker, standard library, and all tooling. This is one of Zig's design advantages: there is no separate package manager to install, no runtime to configure, and no system dependencies beyond what your operating system provides [2]. The entire toolchain ships in one file, typically around 90 megabytes for modern releases.

To install Zig, visit the official download page at ziglang.org/download [2]. You will find prebuilt binaries for Windows, macOS, Linux, FreeBSD, NetBSD, and OpenBSD, supporting architectures including x86_64, aarch64, x86, ARM, RISC-V 64, and others. Each release is cryptographically signed using minisign, and the public key for verification is published on the download page [2].

For Linux users, the installation process looks like this:

```
# Download the tarball (adjust version and architecture as needed)
wget https://ziglang.org/download/0.16.0/zig-linux-x86_64-0.16.0.tar.xz

# Verify the signature
minisign -Vm zig-linux-x86_64-0.16.0.tar.xz -P
↪  RWSGOq2NVecA2UPNdBUZykf1CCb147pkmdtYxgb3Ti+JO/wCYvhbAb/U

# Extract and move to a location in your PATH
tar xf zig-linux-x86_64-0.16.0.tar.xz
sudo mv zig-linux-x86_64-0.16.0 /usr/local/lib/zig
sudo ln -snf /usr/local/lib/zig/zig /usr/local/bin/zig
```

For macOS users with Homebrew:

```
brew install zig
```

Windows users can download the zip archive, extract it, and add the directory to their PATH environment variable. The binary is named `zig.exe` on Windows.

If you need the latest development version or want to build from source, Zig's repository is hosted on Codeberg at codeberg.org/ziglang/zig [8]. Building from source requires a prebuilt bootstrap compiler (available on the download page) and works on all supported platforms. The build process produces a self-hosted compiler that can then be used to build itself.

To verify your installation, run:

```
zig version
```

This should output the installed version number, such as `0.16.0`. You can also check what targets your compiler supports:

```
zig targets
```

This command lists all supported CPU architectures, operating systems, and ABI combinations. The list is extensive: Zig ships with bundled libc for dozens of target platforms, which is why cross-compilation works out of the box without needing separate toolchains or sysroots.

The Zig Compiler Driver: `zig build-exe`, `zig run`, `zig test`

The `zig` binary is a multiplexing driver that routes commands to different subcommands based on the first argument. The most commonly used commands are:

`zig build-exe` compiles a Zig source file into an executable. This is the primary way to build applications:

```
zig build-exe main.zig
```

This produces an executable named `main` (or `main.exe` on Windows) in the current directory. By default, it compiles in Debug mode with safety checks enabled and no optimizations. You can change the optimization level:

```
zig build-exe main.zig -O ReleaseFast    # Maximum speed, no safety checks
zig build-exe main.zig -O ReleaseSafe    # Optimized with safety checks
zig build-exe main.zig -O ReleaseSmall   # Small binary size, no safety checks
```

The four build modes serve different purposes. Debug mode is fast to compile and includes all runtime safety checks (bounds checking, overflow detection, null pointer checks), making it ideal for development. ReleaseSafe enables compiler optimizations while keeping safety checks, useful for testing performance with safety guarantees. ReleaseFast maximizes speed by disabling safety checks. ReleaseSmall optimizes for binary size.

`zig run` compiles and immediately runs a source file without producing an intermediate executable:

```
zig run main.zig
```

This is convenient for quick experiments and scripts. The compilation happens in memory, and the output is executed directly. Any compilation errors are printed to standard error.

`zig test` compiles and runs all test blocks in a source file:

```
zig test main.zig
```

Zig has built-in testing support. Test blocks are declared with the `test` keyword and can be placed anywhere in your source files. The test runner collects all test blocks, executes them, and reports results. We will cover testing in detail in Chapter 11.

Other useful commands include:

- **`zig build-lib`** - compiles a shared or static library
- **`zig cc`** - invokes the C compiler (Zig can compile C code directly)
- **`zig fmt`** - formats source files according to Zig style conventions
- **`zig translate-c`** - translates C header files into equivalent Zig code

Your First Program: A Walkthrough of main.zig

Let us write and examine a complete Zig program from scratch. Create a file named `hello.zig` with the following content:

```
const std = @import("std");

pub fn main() void {
    std.debug.print("Hello, World!\n", .{});
}
```

Let us break down every line of this program.

The first line imports the standard library and assigns it to the constant `std`. The `@import` builtin is how Zig loads modules. Every `.zig` source file is implicitly a struct, and `@import("std")` returns the struct representing the root of the standard library. By convention, this is always aliased as `std` in Zig codebases.

The second line declares the entry point function. In Zig, an executable must have a function named `main`. The `pub` keyword makes it visible to the compiler's startup code. The return type is `void`, meaning the function does not return a value. Alternatively, `main` can return `!void` (an error union that may return an error) or `u8` (an exit code).

Inside the function body, we call `std.debug.print`, which writes formatted output to standard error. The first argument is a format string. The second argument is a tuple of values to substitute into the format string. The `.{}` syntax creates an empty anonymous struct (a tuple with zero fields), which is appropriate when there are no values to substitute. If we wanted to include a value:

```
const name = "Zig";
std.debug.print("Hello, {s}!\n", .{name});
```

Here `{s}` is the format specifier for strings, and `.{name}` creates a tuple containing the `name` variable. Format strings in Zig must be compile-time known values, which allows the compiler to verify that the number and types of arguments match the format specifiers. This catches formatting errors at compile time rather than at runtime.

Compile and run the program:

```
zig build-exe hello.zig
./hello
```

The output is:

```
Hello, World!
```

Note that `std.debug.print` writes to standard error by convention, not standard output. This is because debug prints are typically used for diagnostic information that should appear even when standard output is redirected. For production output, you would use `std.io.getStdOut().writer()` or the higher-level `std.fmt` functions.

Now let us examine a more substantial first program that demonstrates several Zig concepts:

```
const std = @import("std");

pub fn main() void {
    // Integer types with explicit sizes
    const count: u32 = 42;
    const negative: i64 = -100;

    // Floating point types
    const pi: f32 = 3.14159;

    // Boolean type
    const is_ready: bool = true;

    // Print with multiple format specifiers
    std.debug.print("Count: {}, Negative: {}, Pi: {}, Ready: {}\n", .{
        count, negative, pi, is_ready,
    });

    // String type is []const u8 (a slice of constant bytes)
    const greeting: []const u8 = "Hello from Zig!";
    std.debug.print("Greeting: {s}\n", .{greeting});

    // Arrays have compile-time known length
    const numbers = [_]u32{ 1, 2, 3, 4, 5 };
    std.debug.print("First number: {}\n", .{numbers[0]});

    // Loops use while or for (no traditional C-style for loops)
    var i: usize = 0;
    while (i < numbers.len) : (i += 1) {
```

```
30          std.debug.print("  numbers[{}] = {}\n", .{ i, numbers[i] });
31      }
32
33      // Modern for loop syntax with index
34      for (numbers, 0..) |value, index| {
35          std.debug.print("  Index {}: value = {}\n", .{ index, value });
36      }
37  }
```

This program demonstrates several important Zig patterns:

Explicit type sizes. In C, `int` has an implementation-defined size (typically 32 bits on modern platforms). In Zig, every integer type has an explicit size: `u32` is a 32-bit unsigned integer, `i64` is a 64-bit signed integer. This eliminates ambiguity and makes code portable across platforms. The sizes range from 1 bit to 65535 bits, so you can use exactly the precision you need.

The `usize` type. This is an unsigned integer whose size matches the platform's pointer width (64 bits on 64-bit systems, 32 bits on 32-bit systems). It is used for sizes, indices, and lengths. In C, this role is filled by `size_t`, which is a typedef rather than a primitive type.

Arrays as first-class value types. The syntax `[_]u32{ 1, 2, 3, 4, 5 }` creates an array of five `u32` values. The `_` tells the compiler to infer the length from the number of elements. Arrays in Zig are value types, meaning they are copied when passed to functions. This is different from C, where arrays decay to pointers.

Modern for loop syntax. The `for (numbers, 0..) |value, index|` syntax iterates over both the values and indices of the array. The `0..` is a range expression starting from zero. This replaces the traditional C-style `for (i = 0; i < n; i++)` pattern with a cleaner, more expressive form.

While loops with continue expressions. The syntax `while (condition) : (increment)` separates the loop condition from the increment logic. The continue expression in parentheses is executed at the end of each iteration. This makes the control flow clearer than combining condition and increment in a single for-loop header.

Understanding the Zig Build System (`build.zig`)

For small programs, `zig build-exe` works perfectly. For larger projects, Zig provides an integrated build system that replaces Make, CMake, Autotools, and

similar tools [5]. The build system is written in Zig itself, meaning you use the same language for your build configuration as for your application code.

A typical project structure looks like this:

```
my-project/
├── build.zig          # Build configuration
├── src/
│   └── main.zig       # Application source
└── zig-cache/         # Build artifacts (generated)
```

The `build.zig` file defines how the project is compiled:

```zig
const std = @import("std");

pub fn build(b: *std.Build) void {
    // Standard options for target and optimization level
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    // Define the executable artifact
    const exe = b.addExecutable(.{
        .name = "my-project",
        .root_module = b.createModule(.{
            .root_source_file = b.path("src/main.zig"),
            .target = target,
            .optimize = optimize,
        }),
    });

    // Install the executable to the output directory
    b.installArtifact(exe);

    // Create a "run" step that builds and runs the executable
    const run_cmd = b.addRunArtifact(exe);
    run_cmd.step.dependOn(b.getInstallStep());

    const run_step = b.step("run", "Run the application");
    run_step.dependOn(&run_cmd.step);

    // Create a "test" step that runs all tests
    const unit_tests = b.addTest(.{
        .root_module = b.createModule(.{
            .root_source_file = b.path("src/main.zig"),
            .target = target,
            .optimize = optimize,
        }),
    });
```

```
36
37      const run_unit_tests = b.addRunArtifact(unit_tests);
38
39      const test_step = b.step("test", "Run unit tests");
40      test_step.dependOn(&run_unit_tests.step);
41  }
```

This build file defines three steps:

1. **`install`** (default) - builds the executable and installs it to `zig-out/bin/`
2. **`run`** - builds and runs the executable
3. **`test`** - runs all test blocks

Run the project with:

```
zig build              # Build and install
zig build run          # Build and run
zig build test         # Run tests
zig build -Doptimize=ReleaseFast  # Build with release optimization
```

The `b.standardTargetOptions(.{})` and `b.standardOptimizeOption(.{})` calls automatically register command-line flags (`-Dtarget=...` and `-Doptimize=...`) so users can customize the build. The build system caches compilation results, so rebuilding after small changes is fast.

Project Layout Conventions and the Standard Library

Zig projects follow some informal but widely adopted conventions:

Source files. Each `.zig` file is implicitly a struct containing its declarations. Files are organized by functionality, with related code grouped together. There is no strict requirement for directory structure, but `src/` is commonly used for application source and `lib/` or `src/lib/` for library code.

File naming. Files that define types (structs with fields) use TitleCase (e.g., `HttpServer.zig`). Files that serve as namespaces (collections of functions and constants without fields) use snake_case (e.g., `http_server.zig`). This convention helps readers understand the file's purpose from its name.

The standard library. Zig's standard library (`std`) provides commonly used functionality: memory allocation, formatting, file I/O, networking, threading,

JSON parsing, hashing, and more. Unlike C's standard library, which is part of libc, Zig's standard library is distributed with the compiler and linked automatically. You can use Zig without linking libc at all, making it suitable for freestanding environments like kernels and embedded systems.

Browse the standard library documentation locally with:

```
zig std
```

This starts a local web server serving the standard library documentation. The documentation is comprehensive and includes examples for every module.

With the toolchain installed, your first program running, and the project structure understood, you are ready to dive into Zig's type system and core language features.

Chapter 2: Types, Variables, and Constants - The Building Blocks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Primitive Types: Integers, Floats, Booleans, and the Void Type

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Variable Declarations: `var` vs `const` and Immutability by Default

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Type Inference and Explicit Typing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Integer Overflow Semantics: Safe, Wrapping, Saturating, and Unchecked

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

The Undefined Behavior Problem Zig Eliminates

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 3: Control Flow - Branching, Looping, and the Block Expression

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

If/Else as Expressions: Values Everywhere

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

The Switch Statement: Exhaustive Pattern Matching Without Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

While Loops: Tags, Break Values, and Control Flow Clarity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

For Loops: Arrays, Slices, Enumerators, and the New Syntax

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Block Expressions and Local Scoping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 4: Pointers, Arrays, and Slices - Memory in Zig

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Pointer Types: Single (*T) vs Multi ([*]T) Pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Const Correctness in Pointers: `*const T` and Nested Const

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Arrays as First-Class Value Types: Length Is Part of the Type

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Slices: Fat Pointers, Bounds Checking, and Ownership Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

How Zig's Pointer Model Differs from C

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 5: Structs, Enums, and Unions - Composite Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Struct Definitions: Fields, Default Values, and Initialization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Methods and the Self Parameter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Enums: Values, Tags, Sentinels, and Iteration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Unions and Tagged Unions (`union(enum)`): Safe Variant Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Packed Structs and Extern Structs for Binary Layout Control

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 6: Optionals, Error Sets, and Error Unions - Safety Without Exceptions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Optionals: The `T?` Type and Safe Null Handling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Error Sets as Compile-Time Types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

The `try` Operator: Propagating Errors Explicitly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Error Unions (`T!E`) and the `catch` Operator

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

`errdefer`: Cleanup on Error Return

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Why Zig Rejects Exceptions: Design Rationale

Chapter 7: Functions, Generics, and Compile-Time Programming - Zig's Superpower

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Function Declarations: Parameters, Return Types, and Conventions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Inline Functions and Call-Site Inlining

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

The `comptime` Keyword: When Code Runs at Compile Time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Type Parameters and Generic Functions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Building Generic Data Structures: A Linked List from Scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Comptime Blocks and Compile-Time Assertions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 8: Memory Management - Explicit Allocation, No Garbage Collector

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

The Allocator Parameter: Passing Memory Responsibility Explicitly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

GeneralPurposeAllocator and the Standard Library Allocators

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

ArenaAllocator: Scope-Based Allocation for Temporary Data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

FixedBufferAllocator and Stack-Backed Allocation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Manual Memory Management Patterns and Common Mistakes

Chapter 9: Strings, Formatting, and I/O - Working with Text

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Strings Are UTF-8 Byte Slices: Encoding Semantics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

String Comparison, Concatenation, and Allocation-Free Operations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Formatting: `std.fmt` Formatters and Custom Format Implementations

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

File I/O: Readers, Writers, and the Buffering Layer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Path Handling and Directory Traversal

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 10: The Build System, Package Management, and C Interop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

The Build System (`build.zig`) in Depth: Steps, Options, and Artifacts

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Cross-Compilation: Zig's Built-In Multiplatform Story

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Module System and Package Management

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Importing C Libraries: Build-System C Translation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Exporting to C: FFI in the Other Direction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 11: Testing, Debugging, and Tooling - Building Reliable Software

The Built-In Test Framework: `zig test` and Test Organization

Assertion Macros and Custom Error Messages

Debugging with GDB and LLDB

Sanitizers: AddressSanitizer, UndefinedBehaviorSanitizer, and ThreadSanitizer

Profiling and Performance Analysis Tools

Chapter 12: Concurrency - Threads, Atomics, and Synchronization

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Threads: `std.Thread` and the Spawn API

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Atomic Operations: Load, Store, Compare-and-Swap, and Ordering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Mutexes, RWLocks, and Condition Variables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Channel-Based Communication Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Concurrency Pitfalls and Safe Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Chapter 13: Real-World Projects - Case Studies in Idiomatic Zig

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Case Study 1: Building a Command-Line JSON Parser

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Case Study 2: A Simple HTTP Server from Scratch

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Case Study 3: A Binary Protocol Encoder and Decoder

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Project Structure Best Practices for Larger Codebases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Performance Optimization: Benchmarking, Profiling, and Tuning

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

Conclusion: The Zig Journey Ahead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.

References

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/zigindepth.