




A tour of Zig

- <https://zig-tour.vercel.app/> 
- <https://github.com/jackdbd/zig-tour> 
- <https://raw.githubusercontent.com/jackdbd/zig-tour/main/assets/zig-tour.pdf> 
- <https://github.com/jackdbd/zig-demos> 

Giacomo Debidda

Freelance full stack developer

I like TypeScript / Clojure / Zig


I tried (almost) all JavaScript frameworks

I care about Web Performance



 jackdbd


 jackdbd

 giacomodebidda.com

Why Zig?

I know nothing about 'zig', but I do question this obsession with constantly inventing new programming languages. True skilled software engineering is very hard, and takes a lot of experience. Needlessly learning new syntax every few years to 'stay current' represents a huge cognitive drain on programmers.

—cliffski (C++ guy)

Source: [This tweet](#) 

Simple languages are not that bad IMO. My biggest gripe is when languages are constantly revising and adding new features. I don't care about that stuff. I only want to learn the language once.

—Daniel C

Source: [This reply to that tweet](#) 

I know nothing about 'C', but I do question this obsession with inventing higher-level languages. I've been coding about 142 years, coding in x89_66 assembly about 127 years. I'm quite good at it, but not an expert. That's just ONE asm lang.



—Michal Ziulek (zig-gamedev author)

Source: [This reply to the same tweet](#) 


Why not C / C++ / Rust?

Why Zig When There is Already C++, D, and Rust? 

C++

- Complex, too many features
- Error handling typically done using exceptions
- Why should I have written ZeroMQ in C, not C++, part 1  and part 2 

C

- Footguns everywhere
- Preprocessor macros
- Cleanup code can be really messy 

Rust

- Questionable policies 
- Complex, ownership and lifetimes are hard to understand

**SMALL LANGUAGE,
SIMPLE SYNTAX,
BUILD TOOLCHAIN**



**NO HIDDEN
CONTROL FLOW,
NICE ERROR HANDLING,
MEMORY ALLOCATORS**



**COMPILE-TIME
EVALUATION,
BUILT-IN
CROSS-COMPILATION**



**GREAT C-INTEROP,
FIRST-CLASS
WASM SUPPORT**



Learn Zig to learn how computers work


Learn C to learn about how computers work.

—A lot of people on the internet

Source: Reddit, Twitter, etc

Learn C to learn **more** about how computers work.

—Steve Klabnik

Source: Should you learn C to "learn how the computer works"? 

Learn Zig to learn **even more** about how computers work*.

—Me


Source: This slide

* Because you have to pick your own memory allocator, your own libc, etc.

The zen of Zig 🌸

C, but with the problems fixed.

—Andrew Kelley

Source: The Road to Zig 1.0 

Type `zig zen` and this is what you get:

```
1  * Communicate intent precisely.
2  * Edge cases matter.
3  * Favor reading code over writing code.
4  * Only one obvious way to do things.
5  * Runtime crashes are better than bugs.
6  * Compile errors are better than runtime crashes.
7  * Incremental improvements.
8  * Avoid local maximums.
9  * Reduce the amount one must remember.
10 * Focus on code rather than style.
11 * Resource allocation may fail; resource deallocation must succeed.
12 * Memory is a resource.
13 * Together we serve the users.
```

What Zig leaves out

We need to build **simple systems** if we want to build **good systems**.

The benefits of simplicity are: ease of understanding, ease of change, ease of debugging, flexibility.

—Rich Hickey

Source: Simple Made Easy 

Focusing is about **saying no**.

—Steve Jobs

Source: Apple's World Wide Developers Conference 1997 


No garbage collection

Modern garbage collectors (G1, Orinoco, etc) are really complex and they are basically a black box.

When, how, and whether garbage collection occurs is down to the implementation of any given JavaScript engine. Any behavior you observe in one engine may be different in another engine, in another version of the same engine, or **even in a slightly different situation with the same version of the same engine.**

Source: [FinalizationRegistry on mdn web docs](#) 

It is **not guaranteed** that `__del__()` methods are called for objects that still exist when the interpreter exits.

Source: [__del__ on docs.python.org](#) 

No reference counting

Several C libraries (GTK, Cairo) use GObject (GLib Object System). GObject's are reference counted. As long as their reference count is nonzero, they are "alive", and when their reference count drops to zero, they are deleted from memory.



GObject's use of GLib's `g_malloc()` memory allocation function will cause the program to **exit unconditionally upon memory exhaustion.**

Many programming languages choose to handle the possibility of heap allocation failure by unconditionally crashing. By convention, Zig programmers do not consider this to be a satisfactory solution.



Source: [Heap Allocation Failure on ziglang.org](#) 

Explicit allocations

Explicit memory management is hard, right? Not necessarily.

- [Code for Game Developers - Anatomy of a Memory Allocation \(Jorge Rodriguez\)](#) 
- [Introduction to General Purpose Allocation \(Casey Muratori\)](#) 




Understanding a generational garbage collector like Orinoco is much harder.

- [Orinoco: The new V8 Garbage Collector \(Peter Marshall\)](#) 
- [Garbage Collection Algorithms — Dmitry Soshnikov](#) 



Allocator interface

In Zig, functions which need to allocate accept an `Allocator` parameter.

The memory allocator interface is defined in `std/mem/Allocator.zig`  and `std/mem.zig` .

- [What's a Memory Allocator Anyway? - Benjamin Feng](#) 
- [Testing memory allocation failures with Zig](#) 
- [Choosing an Allocator](#) 

This is a good idea. In fact, others are taking notes:

- [fitzgen/bumpalo](#) 
- [Trait std::alloc::Allocator](#) 

std.heap

Memory allocators in `std/heap.zig`:

- `std.heap.ArenaAllocator`
- `std.heap.FixedBufferAllocator`
- `std.heap.GeneralPurposeAllocator`
- `std.heap.LoggingAllocator`
- `std.heap.LogToWriterAllocator`
- `std.heap.PageAllocator`
- `std.heap.ScopedLoggingAllocator`
- `std.heap.ThreadSafeAllocator`
- `std.heap.WasmAllocator`
- `std.heap.WasmPageAllocator`

std.testing

Memory allocators in `std/testing.zig`:

- `std.testing.allocator`
- `std.testing.FailingAllocator`


No string type

How long is the string 日本 ?

It does not make sense to have a string without knowing what **encoding** it uses.

—Joel Spolsky

Source: The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!) ↗

See String Literals and Unicode Code Point Literals  in the documentation.

What about Zig?

The encoding of a string in Zig is de-facto assumed to be UTF-8.

So 日本 is 2 code points long, and 6 `u8` long.

Speaking of strings...

This is valid Zig code, if...

```
1  const s = "foo " ++ way_too_many_characters ++ " bar";
```

If `way_too_many_characters` is **compile-time** known.

If `way_too_many_characters` is **runtime** known, we `try` to allocate (and handle allocation failure).

```
1  const std = @import("std");
2
3  pub fn main() !void {
4      var gpa = std.heap.GeneralPurposeAllocator(.{}){};
5      defer std.debug.assert(!gpa.deinit());
6
7      const allocator = gpa.allocator();
8
9      // Let's say way_too_many_characters depends on the content of a file,
10     // or from the user's input. So it's runtime known.
11
12     const s = try std.fmt.allocPrint(allocator, "foo {s} bar", .{ way_too_many_characters });
13     defer allocator.free(s);
14
15     const stdout = std.io.getStdOut().writer();
16     try stdout.print("{s}\n", .{ s });
17 }
```

No operator overloading

What does this Python code print?

```
1 a = Foo(2)
2 b = Bar(3)
3 print(a + b)
4 print(b + a)
```

We need to know what `+` means for `a` and `b`.




```
1 class Foo(object):
2     def __init__(self, n):
3         self.n = n
4     def __add__(self, other):
5         return self.n + other.n
6
7 class Bar(object):
8     def __init__(self, n):
9         self.n = n
10    def __add__(self, other):
11        return self.n - other.n
```

Solution:

```
1 5
2 1
```

Why not?


Arguments in favor of / against operator overloading.

- Proposal: Custom Operators / Infix Functions (issue #427) 
- Operator Overloading (issue #871) 
- New to Zig. I had some questions and comments (r/Zig) 

No exceptions


Exceptions make cleaning up resources problematic.

When you add a `throw` statement to an existing function, you must examine all of its transitive callers. Either they must make at least the basic exception safety guarantee, or they must never catch the exception and be happy with the program terminating as a result. For instance, if `f()` calls `g()` calls `h()`, and `h` throws an exception that `f` catches, `g` has to be careful or it may not clean up properly.

Source: [Google C++ Style Guide](#) 

Exceptions hide control flow.

[...] exceptions make the control flow of programs difficult to evaluate by looking at code: functions may return in places you don't expect. This causes maintainability and debugging difficulties.

Source: [Google C++ Style Guide](#) 

What about Zig?

Resource cleanup is done using `errdefer`.

```
1 fn createFoo() !Foo {
2     const foo = try tryToAllocateFoo();
3
4     // this runs every time
5     defer std.debug.print("runs every time", .{});
6
7     // this runs ONLY when tryToAllocateFoo fails
8     errdefer deallocateFoo(foo);
9
10    return foo;
11 }
```


Errors can be handled like any other value.

```
1 fn doAThing(str: []u8) void {
2     if (parseU64(str, 10)) |number| {
3         doSomethingWithNumber(number);
4     } else |err| switch (err) {
5         error.Overflow => {
6             // handle overflow...
7         },
8         error.InvalidChar => {
9             // handle invalid char...
10        },
11        // Zig will not compile if we forgot to
12        // handle all possible errors.
13    }
14 }
```

Error handling

In order to have high quality software, correct error handling has to be the **easiest, most straightforward path** for people to follow.

—Andrew Kelley


Source: The Road to Zig 1.0 

Defining errors in JS

Don't. Or do it only once.


Consider extending the `Error` object with additional properties, but be careful not to overdo it. It's generally a good idea to extend the built-in `Error` object **only once**.

—nodebestpractices

Source: Use only the built-in Error object 

Example: Hapi web apps/APIs use Boom .

Defining errors in Zig



Use an Error Set .

```
1  const NumberNotInRangeError = error{
2      TooSmall,
3      TooBig,
4  };
```

The return type of a Zig function that might fail is:

```
1  <error set>!<expected type>
```

Zig errors cannot have a payload.

- Some people would want it 
- Some others would not 

Handling failures in JS

In JavaScript, `catch` catches **exceptions**.

JS functions can `throw` anything → An exception can be anything.

We **do not know** what we caught.

```
1  const fn = () => {
2    throw "I'm not an error object"
3    // throw 42
4    // throw true
5    // throw { a: 1 }
6    // throw undefined
7  }
8
9  const main = async () => {
10   try {
11     fn()
12   } catch (ex) {
13     console.trace(ex)
14     console.log("message", ex.message)
15     console.log("stack trace", ex.stack)
16   }
17 }
18
19 main()
```

Handling failures in Zig

In Zig, `catch` catches **errors**.

Zig functions can `return` possible error values →

An error type is a set of all possible values.

We **know** what we caught.

```
1  fn isNumInRange(n: u8) NumberNotInRangeError!bool {
2    if (n <= 3) {
3      return NumberNotInRangeError.TooSmall;
4    } else if (n >= 7) {
5      return NumberNotInRangeError.TooBig;
6    } else {
7      return true;
8    }
9  }
10
11 pub fn main() void {
12   var b = isNumInRange(5) catch false;
13   std.debug.print("5 in range? {}\n", .{ b });
14
15   b = isNumInRange(9) catch |err| blk: {
16     std.debug.print("Error: {any}\n", .{err});
17     break :blk false;
18   };
19   std.debug.print("9 in range? {}\n", .{ b });
20 }
```


try / catch / @panic

The keyword `try` is a shortcut for `catch |err| return err`. That `|err|` is called **capture**.



Often you don't `catch`. You simply `try`. You `catch` only **when you can handle** the error.

If you have **no idea how to handle** a runtime error and/or **want to crash** the program, use `@panic`.

You should (ideally) never use `@panic` in a library.

You can override the behavior  of `@panic`. I'm not sure it's a good idea though.


If you know already at **compile time** that something is wrong, use `@compileError`.

See also Error, panic or unreachable? - Loris Cro  and Zig / Handling errors .

error return trace \neq stack trace

When an error is returned, you get an **error return trace**.

When `@panic` is called, you get a **stack trace**.

This comparison  illustrates how an error return trace offers better debuggability.

Tips for error handling 1/2

✓ Do omit the error set of a function.

```
1 pub fn foo() !u32 {  
2     ...  
3 }
```


Even in recursive functions.

```
1 const MyError = error{  
2     FourIsBadLuck,  
3 };  
4  
5 fn factorial(n: usize) !usize {  
6     if (n == 1) return 1;  
7     if (n == 4) return MyError.FourIsBadLuck;  
8     return n * try factorial(n - 1);  
9 }
```

Tips for error handling 2/2

✗ Do not use `anyerror` as the error set.

```
1 pub fn foo() anyerror!u32 {  
2     ...  
3 }
```


The global error set `anyerror` should generally be avoided  because it prevents the compiler from knowing what errors are possible at compile-time.

Knowing the error set at compile-time is better for generated documentation and helpful error messages.

How to solve software reuse?

Let's review a few key terms.

An ABI defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format. [...] A common aspect of an ABI is the calling convention, which determines how data is provided as input to, or read as output from, computational routines.

Source: Application Binary Interface on Wikipedia 

A foreign function interface (FFI) is a mechanism by which a program written in one programming language can call routines or make use of services written or compiled in another one. An FFI is often used in contexts where calls are made into binary dynamic-link library.

Source: Foreign Function Interface on Wikipedia 

A popular FFI is libffi/libffi , which is used by Python (ctypes, cffi), Ruby, Haskell, etc.

C interop

Instead of running away from the C/C++ ecosystem, **we must find a way of moving forward that doesn't start by throwing in the trash everything that we have built in the last 40 years.**

—Loris Cro

Source: Maintain it With Zig [🔗](#)

Embrace, extend, and extinguish.

—Someone at Microsoft

Source: Embrace, extend, and extinguish on Wikipedia [🔗](#)

Step 0: no zig

Let's say we have a C program `singular.c` that uses a popular C library: `cairo` ².

```
1  #define WIDTH 400
2  #define HEIGHT 400
3  #include <cairo/cairo.h>
4  #include <math.h>
5  #include <stdio.h>
6
7  static void get_singular_values (const cairo_matrix_t *matrix, double *major, double *minor)
8  {...}
9
10 static void get_pen_axes (cairo_t *cr, double *major, double *minor)
11 {...}
12
13 static void draw (cairo_t *cr, int width, int height)
14 {...}
15
16 int main (int argc, char *argv[])
17 {
18     int width = WIDTH;
19     int height = HEIGHT;
20     cairo_surface_t *surface = cairo_image_surface_create (CAIRO_FORMAT_ARGB32, width, height);
21     cairo_t *cr = cairo_create (surface);
22     draw (cr, width, height);
23     cairo_status_t status = cairo_surface_write_to_png (surface, "singular.png");
24     printf("cairo surface status = %d (%s)\n", status, cairo_status_to_string (status) );
25     return 0;
26 }
```


We are compiling `singular.c` using gcc:

```
1  gcc singular.c -lm -lcairo -o singular
```

Step 1: replace gcc with zig cc

Replace gcc with zig cc:

```
1 zig cc singular.c -lm -lcairo -o singular
```

See [zig cc: a Powerful Drop-In Replacement for GCC/Clang](#) .

But why?

- artifact caching
- cross compilation
- pick the libc you want to use / can use
- sane defaults (e.g. `-Wall` for warnings, `-fsanitize=undefined` for undefined behavior sanitizer)

Step 2: use the Zig toolchain to build

Use `build.zig` to compile `singular.c`. Now we can compile our project by typing `zig build`.

```
1  const std = @import("std");
2  const Build = std.build;
3
4  pub fn build(b: *Build) void {
5      const target = b.standardTargetOptions(.{});
6      const optimize = b.standardOptimizeOption(.{});
7
8      const exe = b.addExecutable(.{
9          .name = "singular",
10         .root_source_file = .{ .path = "singular.c" },
11         .optimize = optimize,
12         .target = target,
13     });
14
15     exe.linkSystemLibrary("c");
16     exe.linkSystemLibrary("cairo");
17     exe.install();
18
19     const run_cmd = exe.run();
20     run_cmd.step.dependOn(b.getInstallStep());
21     const run_step = b.step("run", "Run the app");
22     run_step.dependOn(&run_cmd.step);
23
24     const install_cairo = b.addSystemCommand(&.{ "sudo", "apt", "install", "libcairo2-dev" });
25     const cairo_step = b.step("install-cairo", "Install cairo");
26     cairo_step.dependOn(&install_cairo.step);
27 }
```

Step 3: replace **some** C with Zig: `c.zig`

We create a `c.zig` file where we import all C libraries.

```
1  pub usingnamespace @cImport({
2      // XCB is only required when using the XCB surface backend for Cairo.
3      @cInclude("xcb/xcb.h");
4      @cInclude("cairo/cairo-pdf.h");
5      @cInclude("cairo/cairo-script.h");
6      @cInclude("cairo/cairo-svg.h");
7      @cInclude("cairo/cairo-xcb.h");
8      @cInclude("cairo/cairo.h");
9      // Leave pango and pangocairo out for now. They would require
10     // additional dependencies and they are not used in our program.
11     // @cInclude("pango/pangocairo.h");
12 });
```

Step 3: replace **some** C with Zig: `singular.zig`

We create a `singular.zig` where we define the program.

```
1  const std = @import("std");
2  const c = @import("c.zig");
3
4  // The draw function is defined in singular.c, not here in singular.zig
5  extern "c" fn draw(cr: *c.struct__cairo, width: usize, height: usize) void;
6
7  pub fn main() !void {
8      const width = 400;
9      const height = 400;
10     var surface = c.cairo_image_surface_create(c.CAIRO_FORMAT_ARGB32, width, height);
11     var cr = c.cairo_create(surface);
12     draw(cr.?, width, height);
13     var status = c.cairo_surface_write_to_png(surface, "singular.png");
14     std.debug.print("status {any}\n", .{status});
15 }
```

`extern` can be used to declare a function or variable that will be resolved:

- at link time, when linking statically
- at runtime, when linking dynamically

Step 3: replace **some** C with Zig: new `build.zig`

```
1  const std = @import("std");
2  const Build = std.build;
3
4  pub fn build(b: *Build) void {
5      const target = b.standardTargetOptions(.{});
6      const optimize = b.standardOptimizeOption(.{});
7
8      const exe = b.addExecutable(.{
9          .name = "singular",
10         .root_source_file = .{.path = "singular.zig"},
11         .optimize = optimize,
12         .target = target,
13     });
14
15     exe.linkSystemLibrary("c");
16     exe.linkSystemLibrary("cairo");
17     exe.addCSourceFile("singular.c", &[_][]const u8{});
18     exe.install();
19
20     const run_cmd = exe.run();
21     run_cmd.step.dependOn(b.getInstallStep());
22     const run_step = b.step("run", "Run the app");
23     run_step.dependOn(&run_cmd.step);
24
25     const install_cairo = b.addSystemCommand(&.{ "sudo", "apt", "install", "libcairo2-dev" });
26     const cairo_step = b.step("install-cairo", "Install cairo");
27     cairo_step.dependOn(&install_cairo.step);
28 }
```

Step 3: replace **some** C with Zig: minor fixes

If we now try to compile our project with `zig build`, we get:

```
1 error: ld.lld: duplicate symbol: main
```

That's because we have two `main` functions: one in `singular.zig` and one in `singular.c`.

If we remove the `main` function in `singular.c` and try again, we get:

```
1 error: ld.lld: undefined symbol: draw
```

That's because we declared the `draw` function using `static`.

We change this:


```
1 static void draw (cairo_t *cr, int width, int height)
```

into this:

```
1 void draw (cairo_t *cr, int width, int height)
```

Now `zig build` can successfully compile the project.



Watch [Understanding the Extern Keyword in C](#)  if you need a refresher about `static` and `extern`.

Step 4 (opt.): create a Zig wrapper

But why?

- namespaces
- easier and safer to use (e.g. less risk of buffer overflow)
- better error handling
- extend it with higher level abstractions (e.g. slices instead of pointers)

How to do it?

- Wrapping a C Library with Zig 
- Iterative Replacement of C with Zig 
- How I built zig-sqlite 

Examples:

- libgeos.zig 
- zgui 
- zig-cairo 
- zig-sqlite 
- zig-v8 
- zstbi 

zig translate-c 1/2

Let's say we have this `cairo_hello.c` file:

```
1  #include <cairo/cairo.h>
2
3  int main (int argc, char *argv[])
4  {
5      cairo_surface_t *surface =
6          cairo_image_surface_create (
7              CAIRO_FORMAT_ARGB32, 240, 80
8          );
9      cairo_t *cr = cairo_create (surface);
10
11      cairo_select_font_face (
12          cr,
13          "serif",
14          CAIRO_FONT_SLANT_NORMAL,
15          CAIRO_FONT_WEIGHT_BOLD
16      );
17      cairo_set_font_size (cr, 32.0);
18      cairo_set_source_rgb (cr, 0.0, 0.0, 1.0);
19      cairo_move_to (cr, 10.0, 50.0);
20      cairo_show_text (cr, "Hello, world");
21
22      cairo_destroy (cr);
23      cairo_surface_write_to_png (
24          surface, "hello.png"
25      );
26      cairo_surface_destroy (surface);
27      return 0;
28  }
```

zig translate-c 2/2

We can convert it into a Zig file using this command:

```
1  zig translate-c cairo_hello.c \
2      -lc -lcairo \
3      > cairo_hello.zig
```

Then we can try compiling it with this command:

```
1  zig build-exe cairo_hello.zig \
2      -lc -lcairo \
3      -O ReleaseSmall
```

The translation might not be complete, so check:




```
1  cat cairo_hello.zig | grep 'unable to translate'
```

You can use `zig translate-c` to:

- understand weird C code 🚩
- learn about the symbols exported by a C library
- produce Zig code before editing it into more idiomatic code (when you want to create a Zig wrapper for a C library)

Calling Zig from higher level
languages

Calling Zig from Python

- Zig dynamic library exported to C  as Python extension module  (CPython only)
- ctypes 
- cffi 

Example: How to escape Python and write more Zig 





- Python Limited API  (recommended approach? 🤔)

ABI stable across versions, backward/forward compatibility.

```
1  const py = @cImport({  
2      @cDefine("PY_SSIZE_T_CLEAN", {});  
3      @cInclude("Python.h");  
4  });
```

Watch: Using Zig to write native extension modules for Python - Adam Serafini 

Calling Zig from Node.js

- Zig dynamic library that calls V8, libuv, Node.js built-ins, exported as C++ addon 
- Native Abstractions for Node.js (nan) 
- Node-API (formerly known as N-API)  (recommended approach )


ABI stable across versions, backward/forward compatibility.

- lib/wasi.js  (Node.js WebAssembly System Interface, WASI)

Calling Zig from JVM languages

- [Java Native Interface \(JNI\)](#) 

The [Android NDK](#)  uses the JNI. Watch [Create an Android Application with Zig](#) 

- [Project Panama](#)  (Java 19+)

The presentation [Project Panama: Say Goodbye to JNI](#)  shows both approaches.

- [GraalWasm](#)  (GraalVM WASI runtime)


WebAssembly

Zig supports building for WebAssembly out of the box 

Browsers

```
1  zig build-lib src/lib.zig \  
2    -target wasm32-freestanding -dynamic \  
3    -O ReleaseSmall \  
4    --export format_zig_code \  
5    --export wasm_alloc \  
6    --export wasm_dealloc
```

Generates `lib.wasm`.

WASI runtimes (WASI support is under active development )

```
1  zig build-exe src/main.zig \  
2    -target wasm32-wasi-musl \  
3    -O ReleaseSmall
```


Generates `main.wasm`.

In Node.js, launch your app with `node app.js` (add `-experimental-wasi-unstable-preview1` for Node.js versions before `20.0.0`).

Check [jackdbd/zigfmt-web](https://github.com/jackdbd/zigfmt-web)  for both examples.

WebAssembly (demo)

Source code: [jackdbd/zigfmt-web](https://github.com/jackdbd/zigfmt-web) 

Paste some unformatted zig code in the `textarea` below and [Click me](#)  to format it using `lib.wasm`.

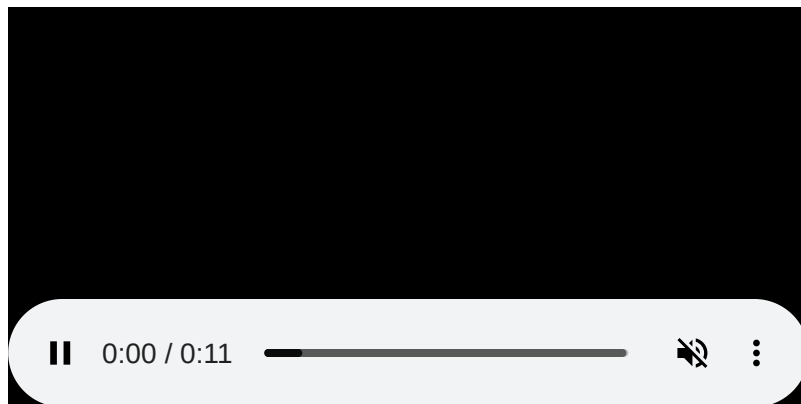
```
const std = @import("std"); pub fn main() void { std.debug.warn("Hello World\n");
}
```

Logs...



Debug it! Open Chrome DevTools > `Sources` tab > open `lib.wasm` > place a breakpoint in the `$format_zig_code` function.

Compilation targets











Build modes (optimizations)

Mode	Compilation speed	Safety checks	Runtime performance	Binary size	Reproducible build
Debug (default)	fast	✓	slow	large	✗
ReleaseFast	slow	✗	fast	large	✓
ReleaseSafe	slow	✓	medium	large	✓
ReleaseSmall	slow	✗	medium	small	✓

You can also use `@setRuntimeSafety(false)` to disable runtime safety checks 🚩 for individual scopes.








```
1  fn foo() void {
2      var x: u8 = 255;
3      x += 1; // undefined behavior
4      {
5          // runtime safety checks enabled, even for ReleaseFast and ReleaseSmall
6          @setRuntimeSafety(true);
7          var x: u8 = 255;
8          x += 1;
9      }
10 }
```

Cool projects 1/2

- bun  - JavaScript runtime, bundler, transpiler, package manager
- buztd  - process killer daemon
- Cosmic  - cross-platform web/desktop framework (similar to tauri )
- CoWasm  - WebAssembly for servers and browsers (similar to Pyodide)
- Fun Dude  - Gameboy emulator
- futureproof  - live editor for WebGPU shaders
- Mach  - game engine & graphics toolkit
- MicroZig  - Hardware Abstraction Layer for microcontrollers. See also Zig Embedded Group



Cool projects 2/2


- ncdu  - disk usage analyzer
- river  - dynamic tiling Wayland compositor
- Tigerbeetle  - distributed financial accounting database
- TinyVG  - vector graphics format (alternative to SVG)
- zig-gamedev  - monorepo of graphics libraries, physics engines, Entity Component System
- Zig-PSP  - PSP emulator
- zigimg  - library to create, process, read, write different image formats

Syntax

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Source: The Zen of Python 

Keywords

All Zig keywords  in a single slide:

- | | | | |
|---------------------------|---------------------------|------------------------------|---------------------------------|
| 1. <code>addrspace</code> | 13. <code>const</code> | 25. <code>inline</code> | 37. <code>switch</code> |
| 2. <code>align</code> | 14. <code>continue</code> | 26. <code>linksection</code> | 38. <code>test</code> |
| 3. <code>allowzero</code> | 15. <code>defer</code> | 27. <code>noalias</code> | 39. <code>threadlocal</code> |
| 4. <code>and</code> | 16. <code>else</code> | 28. <code>nosuspend</code> | 40. <code>try</code> |
| 5. <code>anyframe</code> | 17. <code>enum</code> | 29. <code>or</code> | 41. <code>union</code> |
| 6. <code>anytype</code> | 18. <code>errdefer</code> | 30. <code>orelse</code> | 42. <code>unreachable</code> |
| 7. <code>asm</code> | 19. <code>error</code> | 31. <code>packed</code> | 43. <code>usingnamespace</code> |
| 8. <code>async</code> | 20. <code>export</code> | 32. <code>pub</code> | 44. <code>var</code> |
| 9. <code>await</code> | 21. <code>extern</code> | 33. <code>resume</code> | 45. <code>volatile</code> |
| 10. <code>break</code> | 22. <code>fn</code> | 34. <code>return</code> | 46. <code>while</code> |
| 11. <code>catch</code> | 23. <code>for</code> | 35. <code>struct</code> | |
| 12. <code>comptime</code> | 24. <code>if</code> | 36. <code>suspend</code> | |

while 1/4

```
1  const std = @import("std");
2
3  pub fn main() void {
4      std.log.info("while loop [0, 7)", .{});
5      var i: usize = 0;
6      while (i < 7) {
7          std.log.debug(
8              "i ({s}): {d}",
9              .{ @typeName(@TypeOf(i)), i }
10             );
11             i += 1;
12         }
13     }
```

Output:

```
1  info: for loop [0, 7)
2  debug: i (usize): 0
3  ...
4  debug: i (usize): 7
```

while 2/4

```
1  const std = @import("std");
2
3  pub fn main() void {
4      std.log.info("while loop [0, 7)", .{});
5      var i: usize = 0;
6      while (i < 7) : (i += 1) {
7          std.log.debug(
8              "i ({s}): {d}",
9              .{ @typeName(@TypeOf(i)), i }
10             );
11             }
12         }
13     }
```

Output:

```
1  info: for loop [0, 7)
2  debug: i (usize): 0
3  ...
4  debug: i (usize): 7
```

while 3/4

```
1  const std = @import("std");
2
3  var numbers_left: u32 = undefined;
4  fn eventuallyNull() ?u32 {
5      return if (numbers_left == 0) null else blk: {
6          numbers_left -= 1;
7          break :blk numbers_left;
8      };
9  }
10
11 pub fn main() void {
12     var tot: u32 = 0;
13     numbers_left = 3;
14     while (eventuallyNull()) |value| {
15         std.log.debug("num left: {d}", .{value});
16         tot += value;
17     }
18     std.log.debug("total: {d}", .{tot});
19 }
```

Output:

```
1  debug: num left: 2
2  debug: num left: 1
3  debug: num left: 0
4  debug: total: 3
```

while 4/4

```
1  const std = @import("std");
2
3  fn inRange(i0: usize, i1: usize, n: usize) bool {
4      var i = begin;
5      return while (i < end) : (i += 1) {
6          if (i == n) {
7              break true;
8          }
9      } else false;
10 }
11
12 pub fn main() void {
13     std.log.debug(
14         "is 3 within [1, 5) ? {}",
15         .{inRange(1, 5, 3)}
16     );
17     std.log.debug(
18         "is 7 within [1, 5) ? {}",
19         .{inRange(1, 5, 7)}
20     );
21 }
```

Output:

```
1  debug: is 3 within [1, 5) ? true
2  debug: is 7 within [1, 5) ? false
```

Capture value and index

```
1  const std = @import("std");
2  const log = std.log;
3
4  pub fn main() void {
5      const colors = [_][]const u8{ "red", "green" };
6
7      for (colors, 0..) |color, i| {
8          log.debug(
9              "colors[{d}] is {s}",
10             .{ i, color }
11         );
12     }
13 }
```

Output:

```
1  debug: colors[0] is red
2  debug: colors[1] is green
```

inline for

```
1  const std = @import("std");
2  const log = std.log;
3
4  fn typeNameLength(comptime T: type) usize {
5      return @typeName(T).len;
6  }
7
8  pub fn main() void {
9      const nums = [_]i32{ 2, 4, 6 };
10
11      var sum: usize = 0;
12      inline for (nums) |n| {
13          const T = switch (n) {
14              2 => f32, // length 3
15              4 => i8,  // length 2
16              6 => bool, // length 4
17              else => @panic("got unexpected n"),
18          };
19          sum += typeNameLength(T);
20      }
21      log.debug("sum is {d}", .{sum});
22  }
```

Output:

```
1  debug: sum is 9
```

Iterate over a slice

```
1  const std = @import("std");
2  const log = std.log;
3
4  pub fn main() void {
5      var items = [_]i32{ -1, 0, 1 };
6
7      log.debug("items is {any}", .{items});
8
9      for (&items) |*val| {
10         log.debug(
11             "val has type {s} and is {}",
12             .{ @typeName(@TypeOf(val)), val }
13         );
14
15         log.debug(
16             "val.* has type {s} and is {}",
17             .{ @typeName(@TypeOf(val.*)), val.* }
18         );
19
20         // dereference and assign
21         val.* += 1;
22     }
23
24     log.debug("items is {any}", .{items});
25 }
```

Output

```
1  debug: items is { -1, 0, 1 }
2
3  debug: val has type *i32 and is i32@7ffe311feadc
4  debug: val.* has type i32 and is -1
5
6  debug: val has type *i32 and is i32@7ffe311feae0
7  debug: val.* has type i32 and is 0
8
9  debug: val has type *i32 and is i32@7ffe311feae4
10 debug: val.* has type i32 and is 1
11
12 debug: items is { 0, 1, 2 }
```

comptime

Generics

Compile-time parameters is how Zig implements generics. It is compile-time duck typing.

```
1  fn max(comptime T: type, a: T, b: T) T {
2      return if (a > b) a else b;
3  }
4
5  fn gimmeTheBiggerFloat(a: f32, b: f32) f32 {
6      return max(f32, a, b);
7  }
8
9  fn gimmeTheBiggerInteger(a: u64, b: u64) u64 {
10     return max(u64, a, b);
11 }
```


Compile-time defined types

```
1  const std = @import("std");
2
3  pub fn main() void {
4      var i: u3 = 0; // 111 in binary is 7 in decimal
5      std.log.info("while loop [0, 7)", .{});
6      while (i < 7) {
7          std.log.debug("i ({s}): {d}", .{ @typeName(@TypeOf(i)), i });
8          i += 1;
9      }
10 }
```

`u3` is not a primitive type 🚩. It's defined at compile-time by this function in std/meta.zig 🔗

```
1  pub fn Int(comptime signedness: std.builtin.Signedness, comptime bit_count: u16) type {
2      return @Type(.{
3          .Int = .{
4              .signedness = signedness,
5              .bits = bit_count,
6          },
7      });
8  }
```

Compile-time type reflection

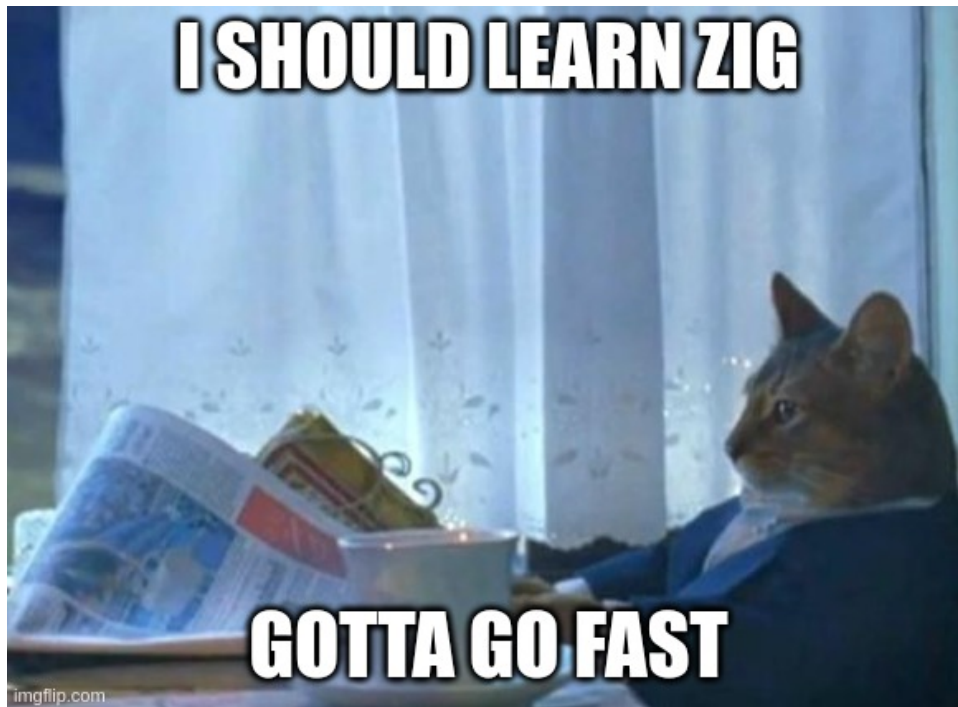
```
1 // demo-reflection.zig
2 const std = @import("std");
3
4 const Hello = struct {
5     foo: u32,
6     bar: []const u8,
7 };
8
9 pub fn main() void {
10     printInfoAboutStruct(Hello);
11 }
12
13 fn printInfoAboutStruct(comptime T: type) void {
14     const info = @typeInfo(T);
15     inline for (info.Struct.fields, 0..) |field, i| {
16         std.debug.print(
17             "type {s} field {d} is called '{s}' and is of type {any}\n",
18             .{ @typeName(T), i, field.name, field.type },
19         );
20     }
21 }
```

Compile and run it with `zig run demo-reflection.zig`

```
1 type demo-reflection.Hello field 0 is called 'foo' and is of type u32
2 type demo-reflection.Hello field 1 is called 'bar' and is of type []const u8
```

Read about `@typeInfo` in the [documentation](#).

I SHOULD LEARN ZIG



GOTTA GO FAST

How to get zig?

Download and manage zig compilers with `zigup` 

Installation

```
1 wget https://github.com/marler8997/zigup/releases/download/v2022_08_25/zigup.ubuntu-latest-x86_64.zip
2 unzip zigup.ubuntu-latest-x86_64.zip
3 chmod u+x zigup
4 mv zigup ~/bin/zigup
```

Usage

```
1 zigup fetch master
2 zigup fetch 0.10.1
3
4 zigup list
5
6 zigup default 0.11.0-dev.2477+2ee328995
7 zigup default 0.10.1
```

Double-check with `zig version`.

How to setup VS Code for Zig?

Install the VS Code extension [ziglang.vscode-zig](#) and declare it in your `.vscode/extensions.json`

```
1 {  
2   "recommendations": ["ziglang.vscode-zig"]  
3 }
```

`ziglang.vscode-zig` automatically installs the [Zig Language Server \(zls\)](#) for autocompletion, goto definition, formatting, etc.

If you prefer, you can also download zls from [zigtools/zls](#) and compile it yourself.

How to use libraries?

Current status: a bit messy. What's the proper way to install/use library? 🍷

Current solutions:





- nektro/zigmod 🔄
- mattnite/gyro 🔄
- marler8997/zig-build-repos 🔄
- git submodules
- just copy the source files in your project

The official package manager is almost here:








- Zig tips: v0.11 std.build API / package manager changes @
- build system terminology update: package, project, module, dependency (issue #14307) 🔄

How to learn Zig?

Learn the basics

1. Familiarize yourself with the syntax: ziglearn 
2. Fix tiny broken programs: ratfactor/ziglings 
3. Review the main features of the language 
4. Read a few funtions of the standard library 

Get better

- Watch Reading Zig's Standard Library 
 - Write tests, especially allocation failures usin std.testing.FailingAllocator 
 - Review Type/pointer cheatsheet 
 - Join r/Zig  and/or other communities  and read/ask/answer questions.
- ⚠ Basically all talks on Zig SHOWTIME  and Zig Meetups  are good. Try not to binge watch them. 😊