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 tipically 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
imgflip.com

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:

```
 * Communicate intent precisely.
 * Edge cases matter.
 * Favor reading code over writing code.
 * Only one obvious way to do things.
 * Runtime crashes are better than bugs.
 * Compile errors are better than runtime crashes.
 * Incremental improvements.
 * Avoid local maximums.
 * Reduce the amount one must remember.
 * Focus on code rather than style.
 * Resource allocation may fail; resource deallocation must succeed.
 * Memory is a resource.
 * 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). GObjects 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 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`.

```
fn createFoo() !Foo {
    const foo = try tryToAllocateFoo();

    // this runs every time
    defer std.debug.print("runs every time", .{});

    // this runs ONLY when tryToAllocateFoo fails
    errdefer deallocateFoo(foo);

    return foo;
}
```

Errors can be handled like any other value.

```
fn doAThing(str: []u8) void {
    if (parseU64(str, 10)) |number| {
        doSomethingWithNumber(number);
    } else |err| switch (err) {
        error.Overflow => {
            // handle overflow...
        },
        error.InvalidChar => {
            // handle invalid char...
        },
        // Zig will not compile if we forgot to
        // handle all possible errors.
    }
}
```

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...

```
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).

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

pub fn main() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    defer std.debug.assert(!gpa.deinit());

    const allocator = gpa.allocator();

    // let's say way_too_many_characters depends on the content of a file, or
    // from a user's input. So it's runtime known.

    const s = try std.fmt.allocPrint(allocator, "foo {s} bar", .{ way_too_many_characters });
    defer allocator.free(s);

    const stdout = std.io.getStdOut().writer();
    try stdout.print("{s}\n", .{ s });
}
```

No operator overloading

What does this Python code print?

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

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

```
class Foo(object):
    def __init__(self, n):
        self.n = n
    def __add__(self, other):
        return self.n + other.n

class Bar(object):
    def __init__(self, n):
        self.n = n
    def __add__(self, other):
        return self.n - other.n
```

Solution:

```
5
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)

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 (22:20)

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.

```
const NumberNotInRangeError = error{
    TooSmall,
    TooBig,
};
```

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

```
<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.

```
const fn = () => {
  throw "I'm not an error object"
  // throw 42
  // throw true
  // throw { a: 1 }
  // throw undefined
}

const main = async () => {
  try {
    fn()
  } catch (ex) {
    console.trace(ex)
    console.log("message", ex.message)
    console.log("stack trace", ex.stack)
  }
}

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.

```
fn isNumInRange(n: u8) NumberNotInRangeError!bool {
    if (n <= 3) {
        return NumberNotInRangeError.TooSmall;
    } else if (n >= 7) {
        return NumberNotInRangeError.TooBig;
    } else {
        return true;
    }
}

pub fn main() void {
    var b = isNumInRange(5) catch false;
    std.debug.print("5 in range? {}\n", .{ b });

    b = isNumInRange(9) catch |err| blk: {
        std.debug.print("Error: {any}\n", .{err});
        break :blk false;
    };
    std.debug.print("9 in range? {}\n", .{ b });
}
```

`try` / `catch` / `@panic`

Often you don't `catch`. You simply `try`.

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

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 ≠ 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.

```
pub fn foo() !u32 {
  ...
}
```

Even in recursive functions.

```
const MyError = error{
    FourIsBadLuck,
};

fn factorial(n: usize) !usize {
    if (n == 1) return 1;
    if (n == 4) return MyError.FourIsBadLuck;
    return n * try factorial(n - 1);
}
```

Tips for error handling 2/2

❌ **Do not** use `anyerror` as the error set.

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

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.

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

while 1/4

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

pub fn main() void {
    std.log.info("while loop [0, 7)", .{});
    var i: usize = 0;
    while (i < 7) {
        std.log.debug(
            "i ({s}): {d}",
            .{ @typeName(@TypeOf(i)), i }
        );
        i += 1;
    }
}
```

Output:

```
info: for loop [0, 7)
debug: i (usize): 0
...
debug: i (usize): 7
```

while 2/4

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

pub fn main() void {
    std.log.info("while loop [0, 7)", .{});
    var i: usize = 0;
    while (i < 7) : (i += 1) {
        std.log.debug(
            "i ({s}): {d}",
            .{ @typeName(@TypeOf(i)), i }
        );

    }
}
```

Output:

```
info: for loop [0, 7)
debug: i (usize): 0
...
debug: i (usize): 7
```

while 3/4

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

var numbers_left: u32 = undefined;
fn eventuallyNull() ?u32 {
    return if (numbers_left == 0) null else blk: {
        numbers_left -= 1;
        break :blk numbers_left;
    };
}

pub fn main() void {
    var tot: u32 = 0;
    numbers_left = 3;
    while (eventuallyNull()) |value| {
        std.log.debug("num left: {d}", .{value});
        tot += value;
    }
    std.log.debug("total: {d}", .{tot});
}
```

Output:

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

while 4/4

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

fn inRange(i0: usize, i1: usize, n: usize) bool {
    var i = begin;
    return while (i < end) : (i += 1) {
        if (i == n) {
            break true;
        }
    } else false;
}

pub fn main() void {
    std.log.debug(
        "is 3 within [1, 5) ? {}",
        .{inRange(1, 5, 3)}
    );
    std.log.debug(
        "is 7 within [1, 5) ? {}",
        .{inRange(1, 5, 7)}
    );
}
```

Output:

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

capture value and index

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

pub fn main() void {
    const colors = [_][]const u8{ "red", "green" };

    for (colors, 0..) |color, i| {
        log.debug(
            "colors[{d}] is {s}",
            .{ i, color }
        );
    }
}
```

Output:

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

inline for

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

fn typeNameLength(comptime T: type) usize {
    return @typeName(T).len;
}

pub fn main() void {
    const nums = [_]i32{ 2, 4, 6 };

    var sum: usize = 0;
    inline for (nums) |n| {
        const T = switch (n) {
            2 => f32,  // length 3
            4 => i8,   // length 2
            6 => bool, // length 4
            else => @panic("got unexpected n"),
        };
        sum += typeNameLength(T);
    }
    log.debug("sum is {d}", .{sum});
}
```

Output:

```
debug: sum is 9
```

Iterate over a slice

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

pub fn main() void {
    var items = [_]i32{ -1, 0, 1 };

    log.debug("items is {any}", .{items});

    for (&items) |*val| {
        log.debug(
            "val has type {s} and is {}",
            .{ @typeName(@TypeOf(val)), val }
        );

        log.debug(
            "val.* has type {s} and is {}",
            .{ @typeName(@TypeOf(val.*)), val.* }
        );

        // dereference and assign
        val.* += 1;
    }

    log.debug("items is {any}", .{items});
}
```

Output

```
debug: items is { -1, 0, 1 }

debug: value has type *i32 and is i32@7ffe311feadc
debug: value.* has type i32 and is -1

debug: value has type *i32 and is i32@7ffe311feae0
debug: value.* has type i32 and is 0

debug: value has type *i32 and is i32@7ffe311feae4
debug: value.* has type i32 and is 1

debug: items is { 0, 1, 2 }
```

```
comptime
```

Generics

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

```
fn max(comptime T: type, a: T, b: T) T {
    return if (a > b) a else b;
}
fn gimmeTheBiggerFloat(a: f32, b: f32) f32 {
    return max(f32, a, b);
}
fn gimmeTheBiggerInteger(a: u64, b: u64) u64 {
    return max(u64, a, b);
}
```

Compile-time type reflection

```zig
// demo-reflection.zig
const std = @import("std");

const Hello = struct {
    foo: u32,
    bar: []const u8,
};

pub fn main() void {
    printInfoAboutStruct(Hello);
}

fn printInfoAboutStruct(comptime T: type) void {
    const info = @typeInfo(T);
    inline for (info.Struct.fields, 0..) |field, i| {
        std.debug.print(
            "type {s} field {d} is called '{s}' and is of type {any}\n",
            .{ @typeName(T), i, field.name, field.type },
        );
    }
}
```

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

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

Read about `@typeInfo` in the documentation.

Compile-time defined types

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

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

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

```zig
pub fn Int(comptime signedness: std.builtin.Signedness, comptime bit_count: u16) type {
    return @Type(.{
        .Int = .{
            .signedness = signedness,
            .bits = bit_count,
        },
    });
}
```

How to solve software reuse?

We need a (modern-day) lingua franca. 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.

```c
#define WIDTH 400
#define HEIGHT 400
#include <cairo/cairo.h>
#include <math.h>
#include <stdio.h>

static void get_singular_values (const cairo_matrix_t *matrix, double *major, double *minor)
{...}

static void get_pen_axes (cairo_t *cr, double *major, double *minor)
{...}

static void draw (cairo_t *cr, int width, int height)
{...}

int main (int argc, char *argv[])
{
    int width = WIDTH;
    int height = HEIGHT;
    cairo_surface_t *surface = cairo_image_surface_create (CAIRO_FORMAT_ARGB32, width, height);
    cairo_t *cr = cairo_create (surface);
    draw (cr, width, height);
    cairo_status_t status = cairo_surface_write_to_png (surface, "singular.png");
    printf("cairo surface status = %d (%s)\n", status, cairo_status_to_string (status) );
    return 0;
}
```

We are compiling `singular.c` using gcc:

```
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 an compile our project by typing `zig build`.

```
const std = @import("std");
const Build = std.build;

pub fn build(b: *Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    const exe = b.addExecutable(.{
        .name = "singular",
        .root_source_file = .{ .path = "singular.c" },
        .optimize = optimize,
        .target = target,
    });

    exe.linkSystemLibrary("c");
    exe.linkSystemLibrary("cairo");
    exe.install();

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

    const install_cairo = b.addSystemCommand(&.{ "sudo", "apt", "install", "libcairo2-dev" });
    const cairo_step = b.step("install-cairo", "Install cairo");
    cairo_step.dependOn(&install_cairo.step);
}
```

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

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

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

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

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

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

// The draw function is defined in singular.c, not here in singular.zig
extern "c" fn draw(cr: *c.struct__cairo, width: usize, height: usize) void;

pub fn main() !void {
    const width = 400;
    const height = 400;
    var surface = c.cairo_image_surface_create(c.CAIRO_FORMAT_ARGB32, width, height);
    var cr = c.cairo_create(surface);
    draw(cr.?, width, height);
    var status = c.cairo_surface_write_to_png(surface, "singular.png");
    std.debug.print("status {any}\n", .{status});
}
```

`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`

```
const std = @import("std");
const Build = std.build;

pub fn build(b: *Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    const exe = b.addExecutable(.{
        .name = "singular",
        .root_source_file = .{.path = "singular.zig"},
        .optimize = optimize,
        .target = target,
    });

    exe.linkSystemLibrary("c");
    exe.linkSystemLibrary("cairo");
    exe.addCSourceFile("cairo-singular.c", &[_][]const u8{});
    exe.install();

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

    const install_cairo = b.addSystemCommand(&.{ "sudo", "apt", "install", "libcairo2-dev" });
    const cairo_step = b.step("install-cairo", "Install cairo");
    cairo_step.dependOn(&install_cairo.step);
}
```

Step 3: replace some C with Zig: minor fixes

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

```
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:

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

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

We change this:

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

into this:

```
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 c.

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:

```c
#include <cairo/cairo.h>

int main (int argc, char *argv[])
{
    cairo_surface_t *surface =
      cairo_image_surface_create (
        CAIRO_FORMAT_ARGB32, 240, 80
    );
    cairo_t *cr = cairo_create (surface);

    cairo_select_font_face (
        cr,
        "serif",
        CAIRO_FONT_SLANT_NORMAL,
        CAIRO_FONT_WEIGHT_BOLD
    );
    cairo_set_font_size (cr, 32.0);
    cairo_set_source_rgb (cr, 0.0, 0.0, 1.0);
    cairo_move_to (cr, 10.0, 50.0);
    cairo_show_text (cr, "Hello, world");

    cairo_destroy (cr);
    cairo_surface_write_to_png (
        surface, "hello.png"
    );
    cairo_surface_destroy (surface);
    return 0;
}
```

zig translate-c 2/2

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

```
zig translate-c cairo_hello.c \
  -lc -lcairo \
  > cairo_hello.zig
```

Then we can try compiling it with this command:

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

The translation might not be complete, so check:

```
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.

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

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

```
zig build-lib src/lib.zig \
  -target wasm32-freestanding -dynamic \
  -O ReleaseSmall \
  --export format_zig_code \
  --export wasm_alloc \
  --export wasm_dealloc
```

Generates `lib.wasm`.

WASI runtimes (WASI support is under active development)

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

Generates `main.wasm`.

In Node.js, launch your app with `node app.js` (add `--experimental-wasi-unstable-preview1` no longer necessary for Node.js versions before `20.0.0`).

Check jackdbd/zigfmt-web for both examples.

WebAssembly (demo)

Source code: jackdbd/zigfmt-web

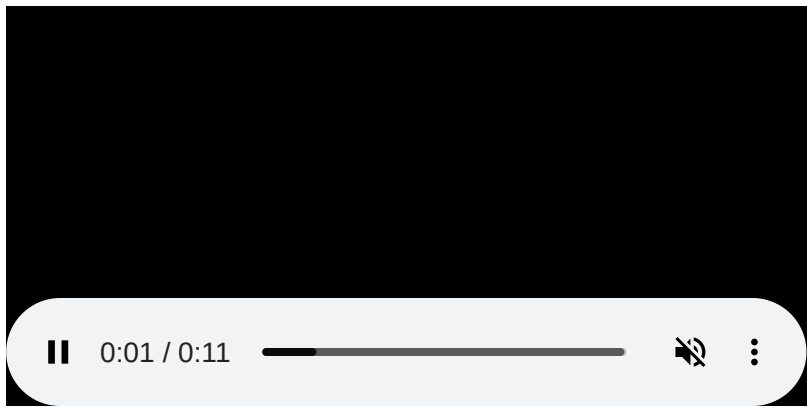
Paste some unformatted zig code in the `textarea` below and Click me 👇 to format it using `lib.wasm`.

```
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.

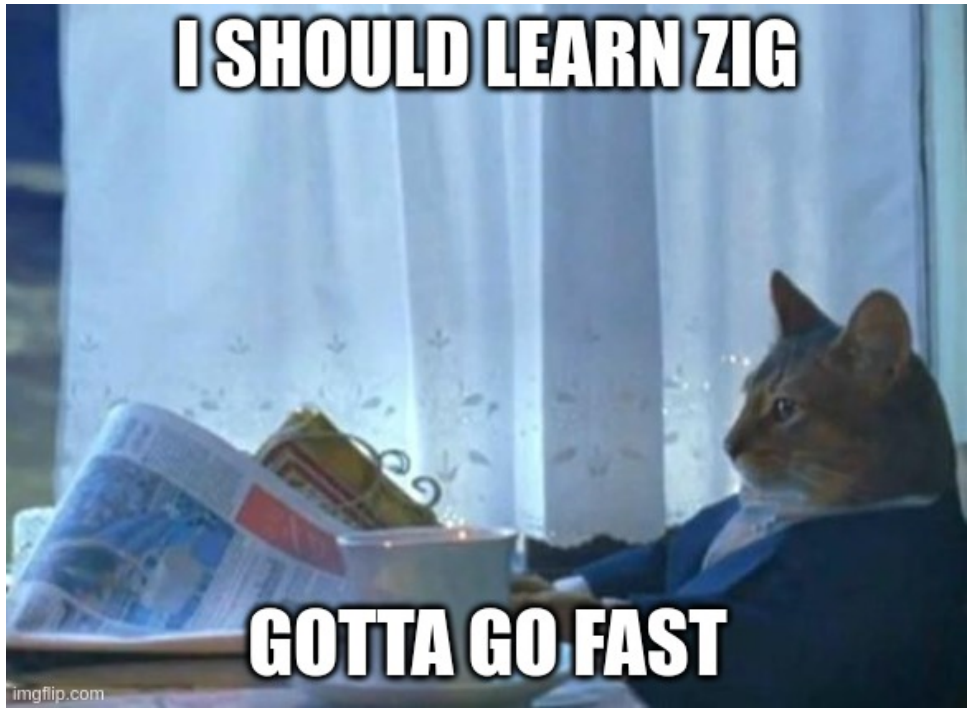
```
fn foo() void {
    var x: u8 = 255;
    x += 1; // undefined behavior
    {
        // runtime safety checks enabled, even for ReleaseFast and ReleaseSmall
        @setRuntimeSafety(true);
        var x: u8 = 255;
        x += 1;
    }
}
```

Cool projects 1/2

- bun - JavaScript runtime, bundler, transpiler, package manager
- buztd - process killer daemon
- Cosmic - Cross-platform (similar to tauri)
- CoWasm - WebAssembly for Servers and Browsers (similar to Pyodide)
- Fun Dude - Gameboy emulator
- futureproof - live editor for WebGPU fragment 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 (SVG alternative)
- zig-gamedev - monorepo containing graphics libraries, physics engines, Entity Component System
- Zig-PSP - PSP emulator
- zigimg - library to create, process, read, write different image formats


I SHOULD LEARN ZIG
GOTTA GO FAST
imgflip.com

How to get zig?

Download and manage zig compilers with zigup

Installation

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

Usage

```
zigup fetch master
zigup fetch 0.10.1

zigup list

zigup default 0.11.0-dev.2477+2ee328995
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`

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

`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. 😄