A tour of Zig 

Giacomo Debidda

Freelance full stack developer

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

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.
```

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`

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

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

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.

Executables: dynamic vs static

todo

https://blog.wesleyac.com/posts/how-i-run-my-servers

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.

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 --experimental-wasi-unstable-preview1` (no longer necessary in Node.js 20?)

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.

Strings

How long is the string 日本 ?

- ○ 2
- ○ 6
- ○ We cannot answer this question

> 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.

String concatenation

The `++` and `**` operators are available **only** at compile-time.

```
1   const final_url = "https://github.com/" ++ user ++ "/reponame";
```

Arrays vs Slices

todo

See Type/pointer cheatsheet on zig.news.

Solving Common Pointer Conundrums - Loris Cro

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.

Compilation targets

todo

zigup

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`.

Setup VS Code

Just one line.

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.

Using libraries

No package manager yet.

gyro (package manager). Questo citalo e basta.

zigmod (package manager)

https://github.com/nektro/zigmod/blob/master/docs/tutorial.md

What's the proper way to install/use library? 

https://github.com/marler8997/zig-build-repos/blob/master/GitRepoStep.zig

Cool projects 1/2

- bun - JavaScript runtime, bundler, transpiler, package manager
- buztd - process killer daemon
- MicroZig - Hardware Abstraction Layer for microcontrollers. See also Zig Embedded Group
- ncdu - disk usage analyzer
- zigimg - library to create, process, read, write different image formats
- river - dynamic tiling Wayland compositor
- Tigerbeetle - distributed financial accounting database

Cool projects 2/2

- TinyVG - vector graphics format (SVG alternative)
- futureproof - live editor for WebGPU fragment shaders
- CoWasm - WebAssembly for Servers and Browsers (Pyodide alternative)
- Mach - game engine & graphics toolkit
- zig-gamedev - monorepo containing graphics libraries, physics engines, Entity Component System
- Fun Dude - Gameboy emulator
- Zig-PSP - PSP emulator

Memory allocation

Esempio di GObject e cairo. Ecco perche' secondo me e' meglio che la memory allocation sia esplicita in un low level language.

Good course on Garbage Collection Algorithms — Dmitry Soshnikov.

> 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

In Rust c'e' ma e' una knightly. [1]

Trait std::alloc::Allocator

Jorge Rodriguez's Code for Game Developers - Anatomy of a Memory Allocation.

Casey Muratori's Introduction to General Purpose Allocation. At 27:45 he starts implementing an arena allocator.

---

1. This slide is not endorsed by the Rust Foundation ↩

Memory allocator interface

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

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`

comptime

Example with generics

Example with factorial, not fibonacci https://youtu.be/Gv2I7qTux7g?t=882

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.

How do I learn Zig?

https://github.com/ratfactor/ziglings

https://ziglearn.org/https://github.com/Sobeston/ziglearn

https://medium.com/swlh/zig-the-introduction-dcd173a86975

https://ziglang.org/learn/overview/

A half-hour to learn Zig https://gist.github.com/ityonemo/769532c2017ed9143f3571e5ac104e50

https://ikrima.dev/dev-notes/zig/zig-crash-course/

https://ziglang.org/documentation/master/

https://ziglang.org/documentation/master/std/#A;std

Links da sistemare

todo

https://demo.sli.dev/composable-vue/1

Carini i numeri qui. https://prisma-talk.netlify.app/12

https://github.com/ziglang/zig/wiki/Community

Structs have the unique property that when given a pointer to a struct, one level of dereferencing is done automatically when accessing fields. Notice how in this example, self.x and self.y are accessed in the swap function without needing to dereference the self pointer. https://ziglearn.org/chapter-1/#structs

3 things you might like about Zig

https://zig.news/toxi/zig-projects-im-working-on-2704

https://zig.news/lupyuen/build-an-lvgl-touchscreen-app-with-zig-38lm

https://github.com/capy-ui/capy

Prova questo smallest echo binary (Linux only) https://blog.mandejan.nl/posts/smallest-echo.html

features Zig purposely leave out

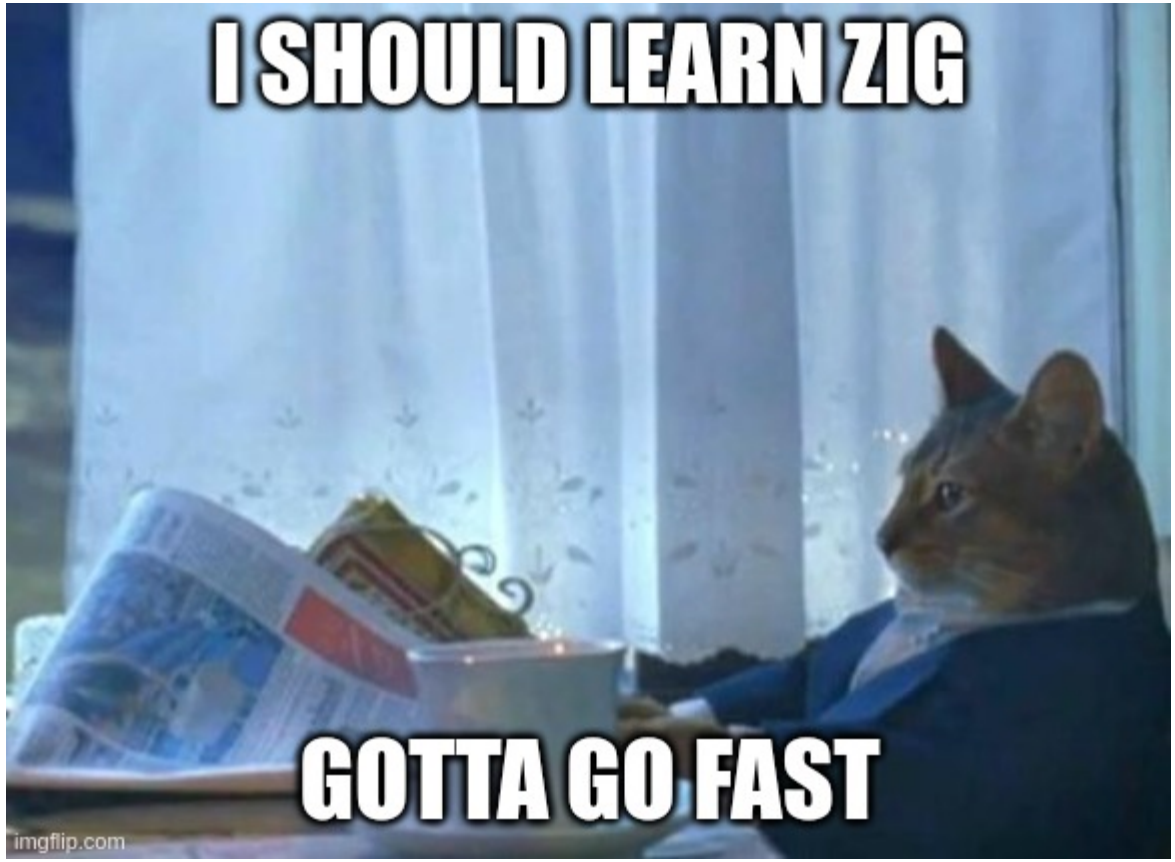
1. operator overloading
2. method overloading
3. RAII (relevant?)

core features

1. error handling
2. explicit memory management. Pick your own memory allocator
3. does not link libc by default. Pick your own libc

advanced features

1. reflection
2. comptime


I SHOULD LEARN ZIG
GOTTA GO FAST
imgflip.com