Zig: il controllo e la potenza del C, senza spararsi sui piedi 🔫

- zig-bolognajs.vercel.app
- jackdbd/zig-bolognajs

Giacomo Debidda

Freelance full stack developer / web performance consultant

I write TypeScript / Clojure / Zig

I like:

- 📚 - add me on goodreads 🔗
- 🛹 - surfskating 🔗 actually
- 🛼 - it's time for a rollerblading emoji

jackdbd

jackdbd

giacomodebidda.com

Why should we care about compiled languages?

> 1. **Performance:** Compiled languages like C++, Rust, or Go often offer better performance than interpreted languages. This can be important for CPU-intensive tasks, algorithms, or systems programming.
> 2. **Understanding of low-level details:** Learning a compiled language can give you a better understanding of what's happening under the hood in your system. This can help you write more efficient code, even in higher-level languages like JavaScript.
> 3. **Interoperability:** Node.js supports native addons, which are dynamically-linked shared objects written in C++. These addons can be used to perform tasks that are either more efficiently done in C++, or simply not possible in JavaScript.
> 4. **Type Safety:** Many compiled languages are statically typed, meaning type checking is done at compile time. This can help catch errors earlier in the development process. While JavaScript is dynamically typed, understanding static typing systems can still be beneficial, especially with the rise of TypeScript in the JavaScript ecosystem.
> 5. **Career Flexibility:** Having experience in both interpreted and compiled languages can make you a more versatile developer and open up more job opportunities.
> 6. **Different Paradigms:** Many compiled languages use different programming paradigms (like functional or procedural programming) than JavaScript. Learning these can broaden your problem-solving skills and help you write better code in any language.
> 7. **WebAssembly:** This is a binary instruction format that allows code written in languages like C, C++, and Rust to run in the browser alongside JavaScript. As a Node.js developer, learning about WebAssembly can be beneficial as it becomes more prevalent in web development.
>
> —GPT 4
>
> Source: Why should a Node.js developer care about a compiled language?

Why not C or C++?

C

- Footguns everywhere. See banned.h for a list of functions banned in the git codebase
- Preprocessor macros. The C preprocessor is another language
- Cleanup code can be really messy
- A lot of undefined behavior

C++

- Complex, too many features
- Error handling tipically done using exceptions (the Google C++ Style Guide prohibits their use)
- Why should I have written ZeroMQ in C, not C++, part 1 and part 2
- OOP cargo cult: The religion of virtual methods, RAII (Resource Acquisition Is Initialization)

Why not Rust or Go?

Rust

- Complex, too many features
- Ownership and lifetimes are hard to understand
- Writing unsafe Rust is hard because it has a lot of nuanced rules about undefined behaviour
- Questionable policies

Go

- Garbage collected
- "Closed-world" language

> C++, Rust, and D have such a large number of features that they can be distracting from the actual meaning of the application you are working on. One finds oneself debugging one's knowledge of the programming language instead of debugging the application itself.
>
> Source: Why Zig When There is Already C++, D, and Rust?


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.

Keywords

1. `addrspace`
2. `align`
3. `allowzero`
4. `and`
5. `anyframe`
6. `anytype`
7. `asm`
8. `async`
9. `await`
10. `break`
11. `callconv`
12. `catch`
13. `comptime`
14. `const`
15. `continue`
16. `defer`
17. `else`
18. `enum`
19. `errdefer`
20. `error`
21. `export`
22. `extern`
23. `fn`
24. `for`
25. `if`
26. `inline`
27. `linksection`
28. `noalias`
29. `noinline`
30. `nosuspend`
31. `opaque`
32. `or`
33. `orelse`
34. `packed`
35. `pub`
36. `resume`
37. `return`
38. `struct`
39. `suspend`
40. `switch`
41. `test`
42. `threadlocal`
43. `try`
44. `union`
45. `unreachable`
46. `usingnamespace`
47. `var`
48. `volatile`
49. `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

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

In C++, shared_ptr uses automatic 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.SbrkAllocator`
- `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 1/2

How long is the string 北斗の拳 ?

> 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!)

No string type 2/2

The encoding of a string in Zig is de-facto assumed to be UTF-8. See String Literals and Unicode Code Point Literals in the documentation.

- 北斗の拳 is 4 code points long.
- 北 is a Japanese Kanji and takes 3 bytes in UTF-8.
- 斗 and 拳 are also Japanese Kanji, so 3 bytes each.
- の is a Japanese Hiragana character and takes 3 bytes in UTF-8.

So 北斗の拳 12 `u8` long in UTF-8.

Trivia: there are many Japanese encoding standards

- JIS X 0208 is a 2-byte character set.
- ISO-2022-JP uses 7 bits.

No operator overloading

What does this Python code print?

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

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

```python
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)

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

Errors as values

Zig errors are values. Handle them like any other value.

```zig
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.
    }
}
```

We can use the same approach in our JS code by never throwing an `Error`, but always returning it.

For example, in Joi:

```js
const result = joi.validate(value, options)
const { error, value, warning, artifacts } = result
```

Error handling: vs

> 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

> 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

Boom (Hapi.js) represents all HTTP errors with a single class that extends `Error`.

```
exports.Boom = class extends Error {
  constructor(messageOrError, options = {}) {
    // ...
  }
}
```

@fastify/error defines this factory function.

```
function createError (
  code, message, statusCode = 500, Base = Error
) {
  // ...
}
```

Defining errors in Zig

Define an error set type

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

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

```
<error set type>!<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.

```javascript
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 error values.
We know what we caught.

```zig
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 });
}
```

`defer` and `errdefer`

Allocate, then `defer` a deallocation immediately afterwards.

Cleanup resources using `errdefer`.

```zig
fn createFoo(param: i32) !Foo {
    const foo = try tryToAllocateFoo();

    // Now we have allocated foo. We need to free it if the function fails.
    // But we want to return it if the function succeeds.
    errdefer deallocateFoo(foo);

    const tmp_buf = allocateTmpBuffer() orelse return error.OutOfMemory;
    // tmp_buf is truly a temporary resource, and we for sure want to clean it up
    // before this block leaves scope.
    defer deallocateTmpBuffer(tmp_buf);

    if (param > 1337) return error.InvalidParam;

    // The errdefer will not run since we're returning success from the function.
    // But the defer will run!
    return foo;
}
```

This passage of The Road to Zig 1.0 explains well how `errdefer` works.

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

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`.

- Error, panic or unreachable? - Loris Cro
- 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.

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

Even in recursive functions.

```zig
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.

```zig
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.

Build a JS project

`package.json` is a manifest.

You build your project using something else:

- npm scripts
- bash scripts
- make
- webpack
- esbuild
- pkg
- Single Executable Applications
- etc

Build a Zig project

`build.zig` is a program.

Zig is not the only language that takes this approach.

> The philosophy behind tools.build is that your project build is inherently a program - a series of instructions to create one or more project artifacts from your project source files.
>
> Source: Builds are programs

You build your project using the Zig compiler and toolchain. Building should be as simple as running:

```
zig build
# which stands for:
zig build --build-file build.zig
```

No need for extra build tools (make, Ninja) or meta-build tools (CMake, Meson, gn).

Let's use in JS with WA

WebAssembly WA

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 ReleaseFast
```

Generates `main.wasm`.

Zig formatter in WebAssembly

Source code: jackdbd/zigfmt-web

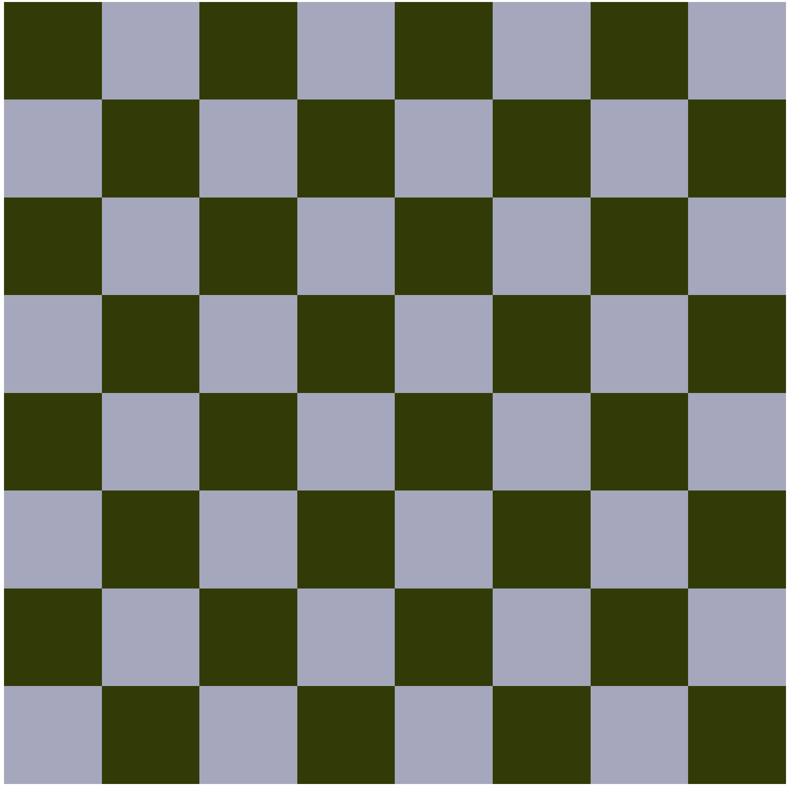
Paste some unformatted zig code in the `textarea` below and click 👇 to format it using `fmt.wasm`.

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

Logs...

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

Checkerboard



Adapted from: minimal-zig-wasm-canvas

Debug it!

- Open Chrome DevTools.
- Go to the `Sources` tab.
- Open the `checkerboard.wasm` file.
- Place a breakpoint in the `$colorCheckerboard` function, right after the local variables.
- Press F8 a few times to resume script execution when it pauses on the breakpoint.

Checkerboard (JS)

Instantiate a WebAssembly module and grant it an initial and maximum number of WebAssembly.Memory pages (64 KiB each).

```
let memory = new WebAssembly.Memory({
  initial: 2, maximum: 2
})

const importObject = { env: { memory } }

const {
  instance
} = await WebAssembly.instantiateStreaming(
  fetch('checkerboard.wasm'), importObject)
```

These functions are defined in `checkerboard.zig` using `export fn`.

```
const {
  colorCheckerboard,
  getCheckerboardBufferPointer,
  getCheckerboardSize,
} = instance.exports
```

Get the canvas from the DOM, get a memory slice representing a 1D RGBA array, use this slice as image data, draw the canvas.

Checkerboard (Zig)

Define functions and `export` them.

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

const checkerboard_size: usize = 8;

// each pixel is 4 bytes (rgba)
var checkerboard_buffer = std.mem.zeroes(
    [checkerboard_size][checkerboard_size][4]u8,
);

// The returned pointer will be used as an offset
// integer to the wasm memory
export fn getCheckerboardBufferPointer() [*]u8 {
    return @ptrCast(&checkerboard_buffer);
}

export fn getCheckerboardSize() usize {
    return checkerboard_size;
}

export fn colorCheckerboard(
    dark_value_red: u8,
    dark_value_green: u8,
    dark_value_blue: u8,
    light_value_red: u8,
    light_value_green: u8,
    light_value_blue: u8,
) void {
    // implementation not shown
}
```

Checkerboard (wasm)

We can inspect the sections available in the WASM binary using the WebAssembly Binary Toolkit.

The `Export` section contains zig functions we exported using `export fn`.

```
wasm-objdump public/checkerboard.wasm --details --section Export

Export[3]:
 - func[0] <getCheckerboardBufferPointer> → "getCheckerboardBufferPointer"
 - func[1] <getCheckerboardSize> → "getCheckerboardSize"
 - func[2] <colorCheckerboard> → "colorCheckerboard"
```

The `Import` section contains functions and `WebAssembly.Memory` objects we defined in JS.

```
wasm-objdump public/checkerboard.wasm --details --section Import

Import[1]:
 - memory[0] pages: initial=2 max=2 ← env.memory
```

The `Type` section contains the data types available in the wasm module.

```
wasm-objdump public/checkerboard.wasm --details

Type[2]:
 - type[0] () → i32
 - type[1] (i32, i32, i32, i32, i32, i32) → nil
Function[3]:
 - func[0] sig=0 <getCheckerboardBufferPointer>
 - func[1] sig=0 <getCheckerboardSize>
 - func[2] sig=1 <colorCheckerboard>
```

wasm-api

Render DOM elements in WASM using wasm-api. Kind of how Yew (Rust) works.

Click to create a canvas and remove it from the DOM

wasm-api (JS/TS)

```js
import {WasmBridge} from "@thi.ng/wasm-api"
import {WasmDom} from "@thi.ng/wasm-api-dom"
import {WasmCanvas2D} from "@thi.ng/wasm-api-canvas"
import WASM_URL from "/canvas.wasm?url"

const bridge = new WasmBridge([
  new WasmCanvas2D(), new WasmDom()]
)

await bridge.instantiate(fetch(WASM_URL))

// call WASM main function to kick off
bridge.exports.start()
```

wasm-api (Zig)

```zig
const canvas2d = @import("wasm-api-canvas");
const wasm = @import("wasm-api");
const dom = @import("wasm-api-dom");

// expose thi.ng/wasm-api core API
// (incl. panic handler & allocation fns)
pub usingnamespace wasm;

fn initApp() !void {
    const canvas = dom.createCanvas(&.{
        .width = 640,
        .height = 480,
        .parent = dom.body,
        .dpr = 1,
        .index = 0,
    });

    canvas2d.beginCtx(canvas);

    canvas2d.setFont("100px Menlo");
    canvas2d.setTextBaseline(.top);
    canvas2d.fillText("Ciao", 10, 10, 0);
}

export fn start() void {
    wasm.printStr("started canvas-wasm");
    initApp() catch |e| @panic(@errorName(e));
}
```

Let's use in

Calling Zig from Node.js

Possible approaches:

1. Direct use of internal V8, libuv, and Node.js libraries
2. Native Abstractions for Node.js (nan)
3. Node-API (formerly known as N-API)

Node-API versions are additive and versioned independently from Node.js. See Node-API version matrix

Version 9 is an extension to version 8 in that it has all of the APIs from version 8 with some additions.

> Unless there is a need for direct access to functionality which is not exposed by Node-API, use **Node-API**.
>
> Source: Node.js documentation

zig translate-c 1/4

Let's say we have downloaded the Node.js `18.17.0` header files to `deps/node-v18.17.0`.

```
.
└── include
    └── node
        ├── common.gypi
        ├── config.gypi
        ├── cppgc
        ├── js_native_api.h
        ├── js_native_api_types.h
        ├── ...
        ├── node_api.h
        ├── node_api_types.h
        ├── ...
        ├── uv.h
        ├── v8.h
        ├── ...
        └── zlib.h
```

We can try translating the header files from C to Zig:

```
zig translate-c \
  deps/node-v18.17.0/include/node/node_api.h \
  > node_api.zig
```

zig translate-c 2/4

The C enum napi_status...

```c
typedef enum {
  napi_ok,
  napi_invalid_arg,
  napi_object_expected,
  napi_string_expected,
  // etc...
} napi_status;
```

...becomes a bunch of Zig constants:

```zig
pub const napi_ok: c_int = 0;
pub const napi_invalid_arg: c_int = 1;
pub const napi_object_expected: c_int = 2;
pub const napi_string_expected: c_int = 3;
// etc...
pub const napi_status = c_uint;
```

zig translate-c 3/4

The C function napi_create_string_utf8...

```c
NAPI_EXTERN napi_status NAPI_CDECL
  napi_create_string_utf8(
    napi_env env,
    const char* str,
    size_t length,
    napi_value* result
);
```

...becomes this Zig function:

```zig
pub extern fn napi_create_string_utf8(
    env: napi_env,
    str: [*c]const u8,
    length: usize,
    result: [*c]napi_value
) napi_status;
```

zig translate-c 4/4

The translation might not be perfect, so check:

```
cat napi_api.zig | grep 'unable to translate'
```

We can use `zig translate-c` to:

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

For example, we could convert napi_status into this:

```zig
pub const Status = enum(u5) {
    ok = c.napi_ok,
    invalid_arg = c.napi_invalid_arg,
    object_expected = c.napi_object_expected,
    string_expected = c.napi_string_expected,
    // etc
};
```

Create a Zig wrapper of a C library

Why?

- Namespaces
- Better error handling
- Easier and safer to use (e.g. less risk of buffer overflow)
- Extend the original library with higher level abstractions (e.g. slices instead of pointers)

How?

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

Examples:

- zig-v8
- zig-cairo
- zig-sqlite
- duckdb.zig
- base64-simd

Node.js addon with Node-API 1/2

Possible project structure:

```
.
├── build.zig
├── deps
│   └── node-v18.17.0
├── dist
│   ├── debug
│   └── release
├── download-node-headers.sh
├── examples
│   ├── bare-minimum.cjs
│   └── greet.cjs
├── package.json
├── package-lock.json
├── README.md
├── src
│   ├── addon.zig
│   ├── c.zig
│   └── napi.zig
├── test
│   └── index.cjs
├── zig-cache
└── zig-out
```

Source: jackdbd/zig-nodeapi-example

Node.js addon with Node-API 2/2

In `c.zig`, import the C header file/s.

```
pub usingnamespace @cImport({
    @cInclude("node_api.h");
});
```

In `napi.zig`, import `c.zig` and implement all Zig data types / functions you want.

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

// implement Zig wrappers in this file.
```

In `addon.zig`, implement your addon functions and register them.

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

fn foo(env: c.napi_env, cbinfo: c.napi_callback_info) callconv(.C) c.napi_value {
    _ = cbinfo;
    return napi.create_string(env, "Hi from the native addon!") catch return null;
}

export fn napi_register_module_v1(env: c.napi_env, exports: c.napi_value) c.napi_value {
    napi.register_function(env, exports, "foo", foo) catch return null;
    return exports;
}
```

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;
    }
}
```

Let's use Z in JS with WA

WASI (JS)

```js
import { readFile } from "node:fs/promises"
import { WASI } from "node:wasi"

// argv[0] - node binary
// argv[1] - fullpath to this script
// argv[2] - first parameter to pass to the WASI program
const args = process.argv.slice(1);

const wasi = new WASI({
  version: "preview1",
  args,
  env: process.env,
  preopens: {
    "/docs": "/home/jack/repos/zig-bolognajs/assets"
  },
})

const wasm = await WebAssembly.compile(await readFile(new URL("main.wasm", import.meta.url)))

const instance = await WebAssembly.instantiate(wasm, wasi.getImportObject())

wasi.start(instance)
```

The WebAssembly System Interface (WASI) is still experimental.

There is a discussion on marking the Node.js WASI module as stable.

Launch with `node --experimental-wasi-unstable-preview1 app.js` in Node.js < `20.0.0`.

WASI (Zig) 1/2

Import the `wasi` module from the standard library.

```
const std = @import("std");
const wasi = std.fs.wasi;

pub fn main() !void {}
```

Implement the `main` function (1/2).

```
var gpa = std.heap.GeneralPurposeAllocator(){};
defer _ = gpa.deinit();
var allocator = gpa.allocator();

const args = try std.process.argsAlloc(allocator);
defer std.process.argsFree(allocator, args);

const stdout = std.io.getStdOut().writer();
const stderr = std.io.getStdErr().writer();
var preopens = try wasi.preopensAlloc(allocator);

const fd = preopens.find("/docs");
if (fd == null) {
    try std.fmt.format(
        stderr,
        "/docs is not an available preopen: {s}\n",
        .{preopens.names}
    );
    std.process.exit(1);
}
```

WASI (Zig) 2/2

Implement the `main` function (2/2).

```
var dir = std.fs.Dir{ .fd = fd orelse unreachable };
defer dir.close();

var file = try dir.openFile(input_filename, .{});
defer file.close();

const contents = try file.reader().readAllAlloc(
    allocator, std.math.maxInt(usize)
);
defer allocator.free(contents);

std.log.debug(
    "Contents of {s}\n{s}",
    .{ input_filename, contents }
);
```

Compile with:

```
zig build-exe main.zig -target wasm32-wasi
```

Every WASI-compliant runtime implements the file system interface with a libpreopen-like layer.

This group proposes an alternative: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes (PDF).

Let's use in other runtimes

JS+WASM runtimes

We can deploy to any environment that offers both a JS runtime and a WebAssembly runtime.

- AWS Lambda and AWS Lambda@Edge (V8)
- Bun (JavaScriptCore)
- Cloudflare Workers and Pages Functions (V8)
- Deno and Deno Deploy (V8)
- Fastly Compute@Edge (SpiderMonkey)
- WasmEdge (QuickJS)
- Lagon?

No need to change Zig code. Keep compiling it with `-target wasm32-wasi`.

These ones are just WASM runtimes:

- wasmtime
- WebAssembly Micro Runtime (WAMR)
- GraalWasm

WASI in Workers

```
import { WASI } from "@cloudflare/workers-wasi"
import demoWasm from "./demo.wasm"

export default {
  async fetch(request, _env, ctx) {
    const stdout = new TransformStream()

    const wasi = new WASI({
      args: [],
      // Cloudflare does not expose a filesystem API
      // ⇒ no WASI preopens except stdin/stdout
      // See Cloudflare security-model
      stdin: request.body,
      stdout: stdout.writable,
    })

    const instance = new WebAssembly.Instance(
        demoWasm,
        { wasi_snapshot_preview1: wasi.wasiImport }
    )

    // Keep our worker alive until the WASM has
    // finished executing.
    ctx.waitUntil(wasi.start(instance))

    // Finally, let's reply with the WASM's output.
    return new Response(stdout.readable)
  },
}
```

How to get zig?

Download and manage zig compilers with zigup

Installation

```
wget https://github.com/marler8997/zigup/releases/download/v2023_07_27/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 0.11.0

zigup list

zigup default 0.12.0-dev.881+42998e637
zigup default 0.11.0
```

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`

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

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

How to use libraries?

Past solutions

- nektro/zigmod
- mattnite/gyro
- marler8997/zig-build-repos

Evergreen solutions

- vendorization (i.e. copy external dependencies into your project itself)
- git subtree (basically vendorizing)
- git submodule

Package manager

Current status

Declare dependencies in a `build.zig.zon` file.

- Zig Package Manager -- WTF is Zon
- build system terminology update: package, project, module, dependency #14307

The Future

- Run build.zig logic in a WebAssembly sandbox #14286
- Package Manager GitHub project board

How to learn Zig?

Learn the basics

1. Familiarize yourself with the syntax: ziglearn
2. Fix tiny broken programs: ziglings/exercises
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
- Have a look at jackdbd/zig-demos
- Review Type/pointer cheatsheet
- Join ~~r/Zig~~ Ziggit and/or other communities and read/ask/answer questions
- Subscribe to Zig SHOWTIME and Zig Meetups

The zen of Zig 🪷

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.

Source: Zen

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

```zig
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

```zig
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

```
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

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

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

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

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

`comptime`

> A Lisp programmer knows the value of everything, but the cost of nothing.
>
> —Alan Perlis
>
> Source: Epigrams on Programming

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 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,
        },
    });
}
```

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.

The end

zig-bolognajs.vercel.app