10/17/2023, 5:08:36 PM

1/57

In this presentation we will talk about:

1. Why Zig might be interesting for a JS developer. And why I think it's better than native languages in the same "league": C, C++, Rust.
2. Features that Zig decides to leave out of the language, and why.
3. Building a project: JS vs Zig.
4. Error handling: JS vs Zig.
5. How to develop hybrid JS/Zig projects (in browser, Node.js and other runtimes).
6. How to use Zig libraries, and the Zig package manager.
7. If there is enough time, we will see a bit of Zig syntax.

2/57

I tried almost all JS frontend frameworks:

- Cycle.js
- Marko.js
- Mithril.js

I haven't tried Angular though.

I care about Web Performance (that's why I tend to avoid JS frameworks if possible, and use 11ty instead).

It's ironic since this presentation is really heavy (but I didn't have much time to optimize it)

Why would a JS developer learn a native language? Performance, to know how computer works, interoperability.

In C, preprocessor macros transform your program **before** actual compilation.

I can't say much about Go. I wrote only a few hundred lines of Go.

Linus Torvalds' quote about C++: C++ is a horrible language.

The author of ZeroMQ did NOT use C++ exceptions, but in the constructor/destructur you kind of have to use them.

Explain what ZeroMQ is. And why C is a better candidate than C++ for this kind of fault-tolerant software.

Consider what happens when **initialisation** of an object can fail. Constructors have no return values, so failure can be reported only by throwing an exception. However, I've decided not to use exceptions.

Moreover, even if initialisation wasn't a problem, **termination** definitely is. You can't really throw exceptions in the destructor. Not because of some self-imposed artificial restrictions but because if the destructor is invoked in the process or unwinding the stack and it happens to throw an exception, it crashes the entire process.

The Google style guide prohibits the use of C++ exceptions.
https://google.github.io/styleguide/cppguide.html#Exceptions

With regards to RAII (as in constructors/destructors, not the stuff in the issue you linked), I think it simply didn't fit within Zig's goals. A big part of Zig is readability; what you read is what you get, and RAII is very much not that. Looking at a block of C++ code, there's no way to tell what happens unless you also know what the constructors/destructors of each data type in the block does.
https://news.ycombinator.com/item?id=27401371

https://nikhilism.com/post/2021/raii-footguns-rust-cpp/

Other names for RAII include Constructor Acquires, Destructor Releases (CADRe) and one particular style of use is called Scope-based Resource Management (SBRM). This latter term is for the special case of automatic variables. RAII ties resources to object lifetime, which may not coincide with entry and exit of a scope.

Coming from C++ I think I feel the need to encapsulate everything. I probably just simplify it to this so the usage is just creating the struct and leave any allocations to the caller. https://ziggit.dev/t/optionally-

reducing-memory-allocations/1948/2

The Go toolchain does not use the assembly language everyone else knows about. It does not use the linkers everyone else knows about. It does not let you use the debuggers everyone knows about, the memory checkers everyone knows about, or the calling conventions everyone else has agreed to suffer, in the interest of interoperability.

5/57

My reactions while I was reading the first post about Zig.

C and C++ basically need to use Emscripten to compile to WebAssembly. Rust can use either Emscripten or wasm-pack.

6/57

Ok, jokes aside, here is why Zig is worth learning.

C does not describe **how the computer works**, it describes how the **C abstract machine** works.

Runtime, **virtual machine**, and **abstract machine** are different words for the same fundamental thing. But they've since gained different connotations, due to non-essential variance in different implementations of these ideas.

C is fundamentally an abstraction of hardware, and abstractions are leaky.

Maybe cite this: https://github.com/cryptocode/bithacks

7/57

Other languages have this number of keywords:

- C has 32
- C++ has 63?
- Java has 64
- JavaScript has 48?
- Python has 33
- Rust has ~50?

8/57

Let's see a couple of features that Zig purposely leaves out.

- automatic memory management
- operator overloading
- exceptions
- interfaces, polymorphism

9/57 No garbage collection

Zig aims to be:

- **Robust**: Behavior is correct even for edge cases such as out of memory.
- **Optimal**: Write programs the best way they can behave and perform.
- **Reusable**: The same code works in many environments which have different constraints.

Using a garbage collector would imply some undeterministic behavior => not robust software

ARC: Automatic Reference Counting

Allocation might fail, so you have to handle failure. Deallocation must always succeed.

C has manual memory management, but many libraries do reference counting, so you don't really have to manage memory. C++ has shared pointers that use automatic reference counting.

C has a default allocator - malloc, realloc, and free.

10/57 Explicit allocations

At 27:45 Casey Muratori starts implementing an arena allocator.

11/57

Tip: read a few tests of the memory allocators (and the memory pools) in std.heap.

12/57 No string type 1/2

https://www.huy.rocks/everyday/01-04-2022-zig-strings-in-5-minutes

https://github.com/JakubSzark/zig-string

https://stackoverflow.com/questions/66527365/how-to-concat-two-string-literals-at-compile-time-in-zig

Kanji is U+4e00 to U+9faf, UTF8 3 bytes are U+0800 to U+FFFF.
https://stackoverflow.com/questions/3678752/are-all-kanji-characters-in-utf-8-3-bytes-long

The first 128 code points (ASCII) need one byte.

The next 1,920 code points need two bytes to encode, which covers the remainder of almost all Latin-script alphabets, and also IPA extensions, Greek, Cyrillic, Coptic, Armenian, Hebrew, Arabic, Syriac, Thaana and N'Ko alphabets, as well as Combining Diacritical Marks.

Three bytes are needed for the remaining 61,440 code points of the Basic Multilingual Plane (BMP), which are virtually all the rest in common use, including most Chinese, Japanese and Korean characters.

Four bytes are needed for the 1,048,576 code points in the other planes of Unicode, which include less common CJK characters, various historic scripts, mathematical symbols, and emoji (pictographic symbols).

13/57 No operator overloading

Operator overloading can be very useful, but there is often a concern that it hinders the ability to understand code at first glance: not only may you have to check whether `+` really means `add`, but it **hides a function call**. One of Zig's main objectives is clarity, so this makes operator overloading a no-go.

JavaScript doesn't support operator overloading. But Python does support it. That's why I make a Python example in this slide.

15/57

Andrew in the talk The Road to Zig 1.0 shows that in C the easiest path is to not deal with errors at all. And this is obviously not good.

16/57

Boom errors contain additional payload and methods for returning HTTP status codes in a consistent way.

I don't like the Fastify approach. It does not enforce a single error type, so in theory I could create many subclasses of Error. This approach would be typical in Python (where it's standard practice to define many exceptions), but it's not a good idea in JS.

Zig errors are basically like C return codes.

17/57

In the `throw undefined` scenario, in the `catch` block `ex.message` throws a `TypeError`. So it's the worst.

We don't know what we caught. That's why in TypeScript we have:

- `catch(e: any)` and not
- `catch(e: Error | SomeOtherError)`

Robust Error Handling in Node.js Applications

- `isNumInRange(5)` prints true
- `isNumInRange(9)` prints false

The labeled block that starts with blk: nd ends with :blk is necessary because we have to return a value to be assigned to `b` (a boolean in this case), and we can't use `return` because we are still in the same function.

18/57

1st click: happy path 2nd click: error path OutOfMemory 3rd click: error path InvalidParam

I think in C++ we would use RAII to implement a function like createFoo(), but we wouldn't have this fine control on resource deallocation.

19/57 error return trace $\neq$ stack trace

The nodebestpractices repo is clear on the distinction between operational errors and programmer errors.

Operational errors refer to situations where you understand what happened and the impact of it – for example, a query to some HTTP service failed due to connection problem. Operational errors are relatively easy to handle.

Programmer errors (aka bugs) refer to cases where you have no idea why and sometimes where an error came from. With a programmer error there's nothing better you can do than to restart gracefully.

20/57 Tips for error handling 1/2

Regarding usize, read this and maybe cite it: https://github.com/ziglang/zig/issues/5185

28/57 wasm-api

Writing hybrid apps with Zig and JS requires quite a lot of glue code. We can avoid it using wasp-api.

31/57 Calling Zig from Node.js

Although Node-API provides an ABI stability guarantee, other parts of Node.js do not, and any external libraries used from the addon may not.

Node-API is backward and forward compatible.

Node-API versions are additive, no semantic versioning => no breaking changes.

A native addon author can take advantage of the Node-API forward compatibility guarantee by ensuring that the addon makes use only of APIs defined in node_api.h and data structures and constants defined in node_api_types.h.

34/57

I need to update zig-cairo when the package manager is ready. But also I need to understand whether it makes sense to use cairo when there is skia.

36/57

Then in JS import the addon:

```
const addon = require("./zig-out/lib/addon.node")
console.log(addon.foo())
```

38/57 Build modes (optimizations)

You can use `**@setRuntimeSafety()**` at any scope, so the value can be overridden at any scope.

43/57 JS+WASM runtimes

The ones within brackets are JS engines.

Clouflare implemented its own JS/WASM runtime based on V8. workerd is the open source version of the JS/WASM runtime that powers Cloudflare Workers. It has feature-parity with the Cloudflare Workers runtime.

JS + WASM => no need for containers.

Watch this talk: Fine-Grained Sandboxing with V8 Isolates

Beware of platform limitations. For example:

- A Cloudflare Worker cannot exceed 1MB in size (free plan) or 5MB (paid plan).

44/57 How to get zig?

Similar to nvm, volta or asdf https://asdf-vm.com/

45/57 How to setup VS Code for Zig?

Zig Language Server (zls) implements Microsoft's Language Server Protocol for Zig in Zig.

47/57

You learn Zig like you learn any other language: by writing it and reading the code of good libraries.

All talks are good. It's not like you are watching the nth intro tutorial on a new JS framework.

49/57

While loops support a **continue expression** which is executed when the loop is continued. The `**continue**` keyword respects this expression.

50/57

We can use a **capture** in a `**while**` loop. If the captured value is null, the while exits automatically.

Maybe cite:

- `**labeled while**`: https://ziglang.org/documentation/master/#Labeled-while
- `**inline while**`: https://ziglang.org/documentation/master/#inline-while

---

51/57

We can use a **capture** in a `**for**` loop. If the captured value is null, the for exits automatically. We can use a first capture for the value, and a second capture for its index.

- `**for**` is an expression, like `**while**`
- `**for**` can be labeled, like `**while**`
- `**for**` can have a `**inline**` keyword, like `**while**`

In the example with `**typeNameLength**`:

- f32 has length 3
- i8 has length 2
- bool has length 4

The capture value (n) and iterator value (nums) of inlined for loops are **compile-time** known.

We can use a **capture** in a `**for**` loop. If the captured value is null, the for exits automatically.

We can use a first capture for the value, and a second capture for its index.

- `**for**` is an expression, like `**while**`
- `**for**` can be labeled, like `**while**`
- `**for**` can have a `**inline**` keyword, like `**while**`