Looking into Zig

by Oren Eini, CEO RavenDB
August 5, 2021

I think that it was the Pragmatic Programmer that recommend that you should learn a new language a year. For me, in 2020 that was Rust. I read a bunch of books about the language, I read significant amount of code and wrote some non trivial amount of code in Rust. That was sufficient to get me to grok the language, I'm not a Rust developer by any mean, but I walked with Rusty shoes for long enough to get the feeling.

This year, I decided to look into Zig. Both Zig and Rust are more or less in the same space, replacing C. They couldn't be more different, however. Zig is a small language. I spent a couple of evenings going through the language reference and that was it, I had a pretty good idea about how to do things.

The learning curve is mostly flat, and that is pretty huge. This is especially because I can't help but compare Zig to Rust. I spent a *lot* of effort understanding Rust, but I had spent almost no cycles trying to grok Zig. It was simple, obvious and quite clear. In terms of power, mind, I would rate both languages on roughly the same spot. You can write *really nice* code in Zig, it is just that you don't need to bend your head into funny shapes to get the compiler to accept your code.

One of the key features for Zig is its *comptime* feature. This is a feature that allow Zig to run code at compilation time. That isn't a new or exciting feature, to be honest, C++ had it for many years. The key difference is that Zig can use this feature for *code generation*. For example, to create a generic list, you'll write the following code:

```
fn LinkedList(comptime T: type) type {
    return struct {
        pub const Node = struct {
            next: ?*Node,
            data: T,
        };

        first: ?*Node,
        len:   usize,

        pub const Self = @This(),

        fn push(self: *Self, val: T, allocator: *std.mem.allocator) error{OutOfMemory}!void {
          var node = try allocator.create(Node);
          node.next = self.first;
          node.data = val;

          self.first = node;
          self.len +=1 ;

        }
    };
}
```

list.zig hosted with ❤ by GitHub — view raw

Note that we are writing here a function that returns a *type*, which you can then use. That approach is incredibly powerful, but at the same time, this is *simple*, it is *obvious*.

Zig is easy to learn, because there isn't a whole lot more that is hidden from you behind the scenes.

That actually leads to another really important aspect in the design of Zig. There isn't anything behind the scenes. For example, you cannot allocate memory in Zig, there is no global function that will do that for you. You need to do so using an allocator. That means that the fact that memory allocations can fail is pervasive throughout the API, standard library and your code. There isn't a "*this may fail on rare occasions*" scenario that you somehow need to handle, this is clear and in your face.

At the same time, Zig does a lot more to make things easier than C. I want to focus on a few important points:

- Zig has the concept of Errors. In the code above, the function *push()* may fail because of an allocation failure. In this case, the function will fail with a return code. That is handled by the *try* keyword, which will abort the current function and return the error. Note that errors and regular values are separate channels in Zig (there is a union mark with ! at the function declaration).
- Zig has support for defer and errdefer keyword. The defer keyword works just as you would expect it to, at the function exit, it will run all the deferred statement in reverse order. The errdefer is a lot more interesting, because that will only run if the function exits with an error. This seemingly simple change has a huge impact on the code quality and the complexity that a developer need to keep in their head.
- Zig has built-in testing, to the point where *test* is a keyword in the language.

To give you some context, when I was writing C code, I literally wrote the exact same thing (manually, with macros and nastiness) in order to help me get things done.

In the same manner, the fact that allocation are explicit and managed from the top (all types that needs to allocate gets the allocator from their parents) means that you get to do some really cool things with memory. It is *easy* to say something like "this piece of code gets 10MB of memory only" and let it run like that. It also end up creating a more robust software system, I think, so memory allocations happen aren't a rare occurrence, they happen all the time.

In general, Zig feel like a lot better C, no additional mental overhead. Compared to Rust, you can get working almost immediately and the compilation speed is *excellent*, to the point where you don't really need to think about it. Rust makes you feel the slow compilation cost from the get go, basically, which is noticeable as your system grows bigger.

Thinking about this, I actually feel that we should compare Zig to Go, because it is closer in concept to what I think Go wanted to be. In fact, looking at the most common complaints people has against Go, Zig answers them all.

If you haven't noticed, I'm quite enjoying working with Zig.

And as an aside, the fact that a language can implement a language server and get automatic IDE support is *freaking amazing*. You can also debug Zig code inside VS Code, for example, pretty much with no more issues than you would for native code. Zig is implemented on top of LLVM and gains a lot of the benefits from it.

One thing that kept going through my mind when I looked at all that I got out of the package is: standing on the shoulders of giants.