Zig Quirks

Mar 27, 2023

I've been writing a lot of Zig lately (http.zig, log.zig and websocket.zig). I'm still in the early stages of learning, and often run into things that either surprise me or I don't understand. I figure I'm probably not the only one, so going through them might be useful.

Some of these things are probably really obvious to a lot of people.

1 - const Self = @This()

Once you start looking at Zig source code, it won't be long until you see:

```
const Self = @This();
```

First, functions that begin with `@`, like `@This()`, are builtin functions that are provided by the compiler. `@This()` returns the type of the inner most struct/enum/union. For example, the following prints "true":

```
const Tea = struct {
  const Self = @This();
};

pub fn main() !void {
  // we'll talk about the .{} syntax later!
  // prints "true"
  std.debug.print("{}\n", .{Tea == Tea.Self});
}
```

This is often used in a method to specify the receiver:

```
const Tea = struct {
  const Self = @This();

  fn drink(self: *Self) void {
    ...
  }
};
```

But this usage, while common, is superficial. We could just have easily written: `fn drink(self: *Tea) void {...}`.

Where it's really useful is when we have an anonymous structure:

```
fn beverage() type {
  return struct {
    full: bool = true,

    const Self = @This();
```

```
    fn drink(self: *Self) void {
      self.full = false;
    }
  };
}

pub fn main() !void {
  // beverage() returns a type, which we instantiate using {}
  // "full" has a default value of "true", so we don't have to specify it he
  var b = beverage(){};
  std.debug.print("Full? {}\n", .{b.full});

  b.drink();
  std.debug.print("Full? {}\n", .{b.full});
}
```

This will print "true" followed by "false".

This example is contrived: why do we need an anonymous structure here? We don't, but this is the foundation for how Zig implements generics. For generics, we do something similar to the above (with the addition of passing a `type` to our function), and thus need `@This()` to reference the anonymous struct from within the anonymous struct.

2 - Files are Structures

In Zig, files are structures.

Say we wanted a `Tea` structure. We could create a file named "tea.zig" and add the following:

```
// contents of tea.zig
pub const Tea = struct{
  full: bool = true,
  const Self = @This();

  pub fn drink(self: *Self) void {
    self.full = false;
  }
};
```

Callers could then use our `Tea` structure like so:

```
const tea = @import("tea.zig");
...

var t = tea.Tea{};
```

Or, with this superficial change:

```
// if we're only using the Tea structure from tea.zig,
// maybe we'd prefer to do it this way.
const Tea = @import("tea.zig").Tea;
...

var t = Tea{};
```

Since files are structures, our `Tea` is actually nested inside of the implicitly created file structure. As an alternative, the full contents of tea.zig could be:

```
full: bool = true,
const Self = @This();

pub fn drink(self: *Self) void {
  self.full = false;
}
```

Which we could import with:

```
const Tea = @import("tea.zig");
```

It looks weird, but if you imagine that the contents are wrapped in a `pub const default = struct { ... };` it makes sense. I was pretty confused the first time I saw one.

3 - Naming Convention

In general:

- Functions are camelCase
- Types are PascalCase
- Variables are lowercase_with_underscores

The main exception to this rule are functions that returns types (most commonly used with generics). These are PascalCade.

Normally, file names are lowercase_with_underscore. However, files that expose a type directly (like our last tea example), follow the type naming rule. Thus, the file should have been called "Tea.zig".

It's easy to follow but is more colorful than what I'm used to.

4 - .{...}

You see `.{...}` throughout Zig code. This is an anonymous structure. The following compiles and prints "keemun"::

```
pub fn main() !void {
  const tea = .{.type = "keemun"};
  std.debug.print("{s}\n", .{tea.type});
```

```
}
```

The above example actually has 2 anonymous structures. The first is the anonymous structure that we've assigned to the `tea` variable. The other is the second parameter we passed to `print`: i.e `.{tea.type}`. This second version is a special type of anonymous structure with implicit field names. The field names are "0", "1", "2", ... In Zig, this is called a tuple. We can confirm the implicit field names by accessing them directly:

```
pub fn main() !void {
  const tea = .{"keemun", 10};
  std.debug.print("Type: {s}, Quality: {d}\n", .{tea.@"0", tea.@"1"});
}
```

The `@"0"` syntax is necessary because `0` and `1` aren't standard identifiers (i.e. they don't begin with a letter), and thus must be quoted.

Another place that you'll see the `.{...}` syntax is when the structure can be inferred. You'll commonly see this inside of a structure's `init` function:

```
pub const Tea = struct {
  full: bool,

  const Self = @This();

  fn init() Self {
    // struct type is inferred by the return type of the function
    return .{
      .full = true,
    };

  }
};
```

You also see it as a function parameter:

```
var server = httpz.Server().init(allocator, .{});
```

The second parameter is an `httpz.Config`, which Zig can infer. Zig requires that every field be initialized, but `httpz.Config` has default values for each field, so an empty struct intializer is fine. You could also specify one or more fields explicitly:

```
var server = httpz.Server().init(allocator, .{.port = 5040});
```

Zig's `.{...}` is like telling the compiler "make this fit".

5 - .field = value

In the above code, we used `.full = true` and `.port = 5040`. This is how fields are set when a struct is initialized. I don't know if this was the intention, but it's actually consistent with how fields are generally set.

I think the following example shows how the `.field = value` syntax makes sense:

```
var tea = Tea{.full = true};

// hey, look, it's pretty similar!
tea.full = false;
```

6 - Private Struct Fields

Speaking of structure fields, they're always public. Structures and functions are private by default with an option to make them public. But struct fields can only be public. The recommendation is to document allowed/proper usage of each field.

I don't want to editorialize this post too much, but it's already caused the type of issues that you'd expect, and I think it'll only cause more difficulties in a 1.x world.

7 - const *tmp

Prior to Zig 0.10, the first line of this code:

```
const r = std.rand.DefaultPrng.init(0).random();
std.debug.print("{d}\n", .{r.uintAtMost(u16, 1000)});
```

Would have been equivalent to:

```
var t = std.rand.DefaultPrng.init(0);
const r = t.random();
```

From 0.10 onward however, the same line is now equivalent to:

```
const t = std.rand.DefaultPrng.init(0);
const r = t.random();
```

Notice that `t` has gone from being a `var` to a `const`. The difference is important since `random()` requires a mutable value. In other words, our original code no longer works. You'll get an error saying that a `*rand.Xoshiro256` was expected, but a `*const rand.Xoshiro256` was found instead. To make it work, we need to split the original code and explicitly introduce the temporary variable as a `var`:

```
var t = std.rand.DefaultPrng.init(0);
const r = t.random();
```

8 - comptime_int

Zig has a powerful "comptime" feature that enables developers to do things at compile time. Logically, compile time execution can only operate on compile-time known data. To support this, Zig has a `comptime_int` and `comptime_float` type. Consider the following example:

```
var x = 0;
while (true) {
  if (someCondition()) break;
  x += 2;
}
```

This won't compile. `x's` type is inferred as being a `comptime_int` since the value, `0`, is known at compile time. The problem here is that a `comptime_int` must be a `const`. Of course, if we change the declaration to `const x = 0;` we'll get a different error because we're trying to add 2 to a `const`.

The solution is to explicitly define x as a `usize` (or some other runtime integer type, like `u64`):

```
var x: usize = 0;
```

9 - std.testing.expectEqual

Possibly the first test that you write will result in a surprising compilation error. Consider this code:

```
fn add(a: i64, b: i64) i64 {
  return a + b;
}

test "add" {
  try std.testing.expectEqual(5, add(2, 3));
}
```

If I showed you `expectEqual's` signature, can you tell why it doesn't compile:

```
pub fn expectEqual(expected: anytype, actual: @TypeOf(expected)) !void
```

It might be hard to tell, but the "actual" value is coerced to the same type as the "expected" value. Our above "add" test fails to compile because an `i64` cannot be coereced to a `comptime_int`.

One simple solution is to swap our parameters:

```
test "add" {
  try std.testing.expectEqual(add(2, 3), 5);
}
```

And this *does* work, and a lot of people do this. The main downside is that the failure message has the expected and actual values mixed up.

The correct way to solve this is to cast our expected value to the actual type, using the `@as()` builtin:

```
test "add" {
  try std.testing.expectEqual(@as(i64, 5), add(2, 3));
}
```

Now, you might think this is only a problem with `comptime_int`, but you'll run into this over and over for other types as well. This test also doesn't compile:

```
test "hashmap: get" {
  var m = std.StringHashMap([]const u8).init(std.testing.allocator);
  defer m.deinit();
  try std.testing.expectEqual(null, m.get("teg"));
}
```

The return value of `get` is `?[]const u8`, which is an optional (aka nullable) string. But our expected value is [correctly] null and a `?[]const u8` cannot be coerced to `null`. To fix this, we must coerce `null` to a `?[]const u8`:

```
try std.testing.expectEqual(@as(?[]const u8, null), m.get("teg"));
```

10 - Shadowing

The Zig documentation states that "Identifiers are never allowed to "hide" other identifiers by using the same name." So if you have `const reader = @import("reader.zig");` at the top your file, you can't have anything else named `reader` in the same file.

You'll have to be more creative when coming up with variables that don't shadow existing ones (which, for me, generally means using more obscure names).