How Zig Do?

March 15, 2018

> *1/13/19 Update: This post was written using Zig 0.2.0, and some of the code is out of date. The repository has been updated to work with 0.3.0, inculding (for the moment) trunk. Refer to that to see a simplified use of stdout.*

Hello and good morning or whatever! Let's write a brainfuck interpreter. "Why are you doing this?" you might say, but you won't find that answer here.

I'm going to make it in Zig.

Zig is….

…a new programming language. It is still very much in beta, and moving quickly. If you've seen any Zig previously, the code in this post might look quite different. It is different! Zig 0.2.0 has just been released, coinciding with the release of LLVM 6 a few weeks ago, and includes a lot of changes to the syntax and general improvements to the language. Most notably, many of the sigils have been replaced by keywords. See here for a more in depth explanations of all the changes!

Zig is designed to be readable, and as such it's relatively intuitive if you are familiar with similarly compiled, (~)typed languages like C, C++, and in some cases Rust.

This code was all compiled and tested with Zig 0.2.0, which is available right now, via different channels, including homebrew if you're on a OSX with `brew install zig`.

Ok go

For info about how brainfuck works, look here. There's almost nothing to it, but it *is* turing complete, which means you can use it to write anything.

I've committed this code here if you want to see the finished product or dig into earlier commits.

Zig is a compiled language. When you compile a program, the resulting binary (if you are building an executable binary, as opposed to a library) needs a `main` function that denotes the entry point.

So…

```
// main.zig
fn main() void { }
```

…and running…

```
$ zig build-exe main.zig
```

…gives me…

```
/zig/std/special/bootstrap.zig:70:33: error: 'main' is private
/zigfuck/main.zig:2:1: note: declared here
```

`main` must be declared public in order to be visible outside of its module…

```
// main.zig
pub fn main() void { }
```

A brainfuck program is supposed to use a 30,000 byte array of memory, so I'll make one.

```
// main.zig
pub fn main() void {
  const mem: [30000]u8;
}
```

I can make a variable `const` or `var`. Here, I'm declaring `mem` as an array of `30000` unsigned (`u`) bytes (8 bits).

This does not compile.

```
/main.zig:3:5: error: variables must be initialized
```

The equivalent C would compile fine: I can declare a variable without initializing it, but Zig forces me to make this decision now, at the declaration site. Often, I don't care what the memory has in it, but I have to *say that*. I can state this intent clearly by initializing to `undefined`.

```
// main.zig
pub fn main() void {
  const mem: [30000]u8 = undefined;
}
```

Initializing a variable to `undefined` offers no guarantees about the values that may or may not be in the memory. This is just like an uninitialized declaration in C except the clarity of intent is enforced.

But maybe I *do* care what this memory is initialized to. Maybe I want to guarantee that it is zeroed out, or start it all off at some arbitrary value or something. In that case I can be explicit about *that* as well:

```
// main.zig
pub fn main() void {
  const mem = []u8{0} ** 30000;
}
```

This might look a little odd, but `**` is an operator used for array multiplication. I'm defining an array of a single `0` byte and then multiplying it by `30000` to get my final initialization value of an array of 30000 zeroed out bytes. This operation happens only once, at *compile time*. `comptime` is one of Zig's main Big Ideas, and I'll come back to it in a later post.

Now to write a brainfuck program that doesn't do anything except increment the first memory slot 5 times!

```
pub fn main() void {
  const mem = []u8{0} ** 30000;
  const src = "+++++";
}
```

In Zig, strings are just byte arrays. I don't have to declare `src` as a byte array because the compiler infers it. It is unneccesary to do so, but I am free to be explicit about that, too:

```
const src: [5]u8 = "+++++";
```

This will compile just fine. This, however…

```
const src: [6]u8= "+++++";
```

will not.

```
main.zig:5:22: error: expected type '[6]u8', found '[5]u8'
```

An additional note: as strings are just simple byte arrays, they are *not null terminated*. You can easily declare C style null terminated strings though! As literals, they look like this:

```
c"Hello I am a null terminated string";
```

for goodness's sake

I want to do *something* for each character in the source string. I can do that! At the top of `main.zig`, I can import some functionality from the standard library like so:

```
const warn = @import("std").debug.warn;
```

`@import`, and in fact anything that begins with an `@` sign, is a *compiler builtin function*. These functions are always available globally. Imports here actually feel quite a lot like javascript imports- you can simply assign anything to whatever, digging into a namespace to get at any publically exposed functions or variables. In the above example, I'm directly importing the warn function and assigning it to, surprise, the const `warn`. This is now callable. It is a common pattern to import the `std` namespace directly and then you may call `std.debug.warn()` *or* assign warn off of that. That looks like:

```
const std = @import("std");
const warn = std.debug.warn;
```

```
const warn = @import("std").debug.warn;
// main.zig
pub fn main() void {
  const mem = []u8{0} ** 30000;
  const src = "+++++";

  for (src) |c| {
      warn("{}", c);
  }
}
```

During debugging and initial development and testing, I just want to print something to the screen. Zig is fastidious about error handling and stdout is error prone. I don't want to mess around with that right now, so I can print straight to stderr with `warn`, above imported from the standard library.

`warn` takes a format string, just like `printf` in C does! The above prints:

```
4343434343
```

43 is the ascii code for the + char. I can also go:

```
warn("{c}", c);
```

and wouldn't you know it:

```
+++++
```

So, I've initialized the memory space, and written a program. Next, I must implement the language itself. I'll start with `+`, I'll replace the body of the `for` to `switch` on the character.

```
for (src) |c| {
    switch(c) {
        '+' => mem[0] += 1
    }
}
```

I get two errors from this program:

```
/main.zig:10:7: error: switch must handle all possibilities
      switch(c) {
      ^
/main.zig:11:25: error: cannot assign to constant
          '+' => mem[0] += 1
                        ^
```

Of course, I can't assign a new value to a variable that's been declared `constant`! `mem` needs to be `var`...

```
var mem = []u8{0} ** 30000;
```

as for the other error, my `switch` statement needs to know what to do for everything that's not a `+`, even if it's nothing. In my case, that's exactly what I want. I'll fulfill that case with an empty block:

```
for (src) |c| {
    switch(c) {
        '+' => mem[0] += 1,
        else => {}
    }
}
```

Now, I can compile the program. If I run it with a warn on the end of it,

```
const warn = @import("std").debug.warn;

pub fn main() void {
  var mem = []u8{0} ** 30000;
  const src = "+++++";
```

```
    for (src) |c| {
        switch(c) {
            '+' => mem[0] += 1,
            else => {}
        }
    }

    warn("{}", mem[0]);
}
```

I get 5 printed to stderr, like I would expect.

From here…

It becomes straightforward to support `-`.

```
switch(c) {
    '+' => mem[0] += 1,
    '-' => mem[0] -= 1,
    else => {}
}
```

To use > and <, I'll need a helper variable that represents a "pointer" into the memory I've allocated for brainfuck's "user space".

```
var memptr: u16 = 0;
```

an unsigned 16 bit number can be a maximum of 65,535, much more than enough to index the entire 30,000 byte address space.

> *Actually, all I really need is an unsigned 15 bit number, which would be enough for 32,767. Zig allows for fairly arbitrarily wide types, but not a u15 just yet.*
>
> > *Andy says: "You actually can make a u15 like this:*
> >
> > ```
> > const u15 = @IntType(false, 15);
> > ```

> *it's proposed to allow any [iu]\d+ type to be an integer type (I guess it wasn't, so I just made the above sentence be true)"*

Now, instead of indexing `mem[0]` for everything, I can use this variable.

```
'+' => mem[memptr] += 1,
'-' => mem[memptr] -= 1,
```

< and >, then, are simply incrementing and decrementing that pointer.

```
'>' => memptr += 1,
'<' => memptr -= 1,
```

Great. We can write "real" programs with this, sort of!

Testing 1,2,3

Zig has a simple built in testing apparatus. Anywhere in any file I can write a test block:

```
test "Name of Test" {
  // test code
}
```

And then run the tests from the command line with `zig test $FILENAME`. There is nothing special about test blocks, except that they are executed only under these circumstances.

Look at this:

```
// test.zig
test "testing tests" {}
```

```
zig test test.zig
```

```
Test 1/1 testing tests...OK
```

Of course, an empty test is not useful. I can use `assert` to actually assert test cases.

```
const assert = @import("std").debug.assert;

test "test true" {
    assert(true);
}

test "test false" {
    assert(false);
}
```

```
zig test test.zig
```

```
"thing.zig" 10L, 127C written
:!zig test thing.zig

Test 1/2 test true...OK
Test 2/2 test false...assertion failure
 [37;1m_panic.7 [0m:  [2m0x0000000105260f34 in ??? (???) [0m
 [37;1m_panic [0m:  [2m0x0000000105260d6b in ??? (???) [0m
 [37;1m_assert [0m:  [2m0x0000000105260619 in ??? (???) [0m
 [37;1m_test false [0m:  [2m0x0000000105260cfb in ??? (???) [0m
 [37;1m_main.0 [0m:  [2m0x00000001052695ea in ??? (???) [0m
 [37;1m_callMain [0m:  [2m0x0000000105269379 in ??? (???) [0m
 [37;1m_callMainWithArgs [0m:  [2m0x00000001052692f9 in ??? (???) [0m
 [37;1m_main [0m:  [2m0x0000000105269184 in ??? (???) [0m
 [37;1m??? [0m:  [2m0x00007fff5c75c115 in ??? (???) [0m
 [37;1m??? [0m:  [2m0x0000000000000001 in ??? (???) [0m

Tests failed. Use the following command to reproduce the failure:
./zig-cache/test
```

Stack traces on Mac are currently a WIP.

In order to test this effectively, I need to break it up. Let's start with this;

```
fn bf(src: []const u8, mem: [30000]u8) void {
    var memptr: u16 = 0;
    for (src) |c| {
        switch(c) {
            '+' => mem[memptr] += 1,
            '-' => mem[memptr] -= 1,
            '>' => memptr += 1,
            '<' => memptr -= 1,
            else => {}
        }
    }
}

pub fn main() void {
    var mem = []u8{0} ** 30000;
    const src = "+++++";
    bf(src, mem);
}
```

This looks like it will work! All the types line up and everything, right?

and yet…

```
/main.zig:1:29: error: type '[30000]u8' is not copyable; cannot pass by value
```

> *this is addressed by https://github.com/zig-lang/zig/issues/733*

Zig is very strict about this. Complex types, basically anything that can possibly be variable in size, can't be passed by value. This makes stack allocations incredibly predictible and consistent, and can avoid unneccessary copying. If you want the semantics of pass by value in your program, you are free to implement them in user space using a custom allocation strategy, but the language itself is designed to discourage this under normal circumstances.

The natural way to get around this would be to pass a pointer instead (passing by reference). Zig prefers a different strategy, though, slices. A slice is sort of just a souped up pointer with bounds checking and a `len` property attached to it. The syntax looks like this in the function signiture:

```
fn bf(src: []const u8, mem: []u8) void { ... }
```

and this at the call site:

```
bf(src, mem[0..mem.len]);
```

It resembles taking a sliced index! Notice that I'm defining the upper bound by simply referencing the length of the array. There is a shorthand for this:

```
bf(src, mem[0..]);
```

Now I can start writing tests in earnest, unit testing the `bf()` function directly. I can just put test blocks at the bottom of this file, for now…

```
test "+" {
    var mem = []u8{0};
    const src = "+++";
    bf(src, mem[0..]);
    assert(mem[0] == 3);
}
```

I'm operating on the mem byte array (of a single byte) and then asserting that what I thought was going to happen (the byte is incremented three times) happened. It did!

```
Test 1/1 +...OK
```

The `-` case is similar:

```
test "-" {
    var mem = []u8{0};
    const src = "---";
    bf(src, mem[0..]);
    assert(mem[0] == 253);
}
```

But this fails! When I try to subtract `1` from `0` I get…

```
Test 2/2 -...integer overflow
```

`mem` is an array of *unsigned bytes*, so subtracting 1 from 0 overflows the type. Once again, Zig is forcing me to consider this possibility explicitly. In this case, it so happens that I don't care about this overflow- in fact I want it to default to dealing with it via modular arithmetic as per the brainfuck spec, such as it is. This means that decrementing a cell at `0` will give me `255`, and incrementing a value of `255` will give me `0`.

Zig has a set of auxiliary arithmetic operators that offer "guaranteed wrap around semantics"

```
'+' => mem[memptr] +%= 1,
'-' => mem[memptr] -%= 1,
```

This solves my integer overflow problem and does what I expect it to.

For < and >, I'll navigate a small array and then check the value of an incremented cell:

```
test ">" {
    var mem = []u8{0} ** 5;
    const src = ">>>+++";
    bf(src, mem[0..]);
    assert(mem[3] == 3);
}
```

and…

```
test "<" {
    var mem = []u8{0} ** 5;
    const src = ">>>+++<++<+";
    bf(src, mem[0..]);
    assert(mem[3] == 3);
    assert(mem[2] == 2);
    assert(mem[1] == 1);
}
```

For this last one, I can directly compare the result to a static array using…

```
const mem = std.mem;
```

Remember that I have imported `std` already. In the below example, I am calling into that namespace to use `mem.eql`:

```
test "<" {
    var storage = []u8{0} ** 5;
    const src = ">>>+++<++<+";
    bf(src, storage[0..]);
    assert(std.mem.eql(u8, storage, []u8{ 0, 1, 2, 3, 0 }));
}
```

…and remember, string literals are just `u8` arrays in zig, and I can put in hexadecimal literals inside them, so the following would work in the exact same way!

```
assert(mem.eql(u8, storage, "\x00\x01\x02\x03\x00"));
```

Let's add `.`! This simply prints the byte value as a character in the cell that is currently being pointed to. For now, I'll abuse `warn`, and revisit this later to properly handle `stdout` here.

```
'.' => warn("{c}", storage[memptr]),
```

For now, I'll ignore `,` as it's very simple conceptually but a little trickier to implement. I'll come back to it later!

Loops

`[` and `]` are where the magic happens….

```
[   if the value of current cell is zero skip to the matching bracket without executing t
]   if the value of the current cell is NOT zero go back to the opening bracket and execu
```

I'll *start* with the test case this time, testing them together (as it doesn't make sense to test them in isolation). The first test case- `storage[2]` should end up being empty even though the loop would increment it if it ran:

```
test "[] skips execution and exits" {
    var storage = []u8{0} ** 3;
    const src = "+++++>[>+++++<-]";
    bf(src, storage[0..]);
    assert(storage[0] == 5);
    assert(storage[1] == 0);
    assert(storage[2] == 0);
}
```

and I'll stub out the switch case:

```
'[' => if (storage[memptr] == 0) {
},
']' => if (storage[memptr] == 0) {
},
```

Now, *what goes here*? A naive approach presents itself. I will simply advance the src index forward until I find a `]`! But I cannot do this in a zig `for`, which is designed simply to iterate over elements of a collection, never to skip around them. The appropriate construct then here is `while`

from:

```
var memptr: u16 = 0;
for (src) |c| {
    switch(c) {
      ...
    }
}
```

to…

```
var memptr: u16 = 0;
var srcptr: u16 = 0;
while (srcptr < src.len) {
    switch(src[srcptr]) {
      ...
    }
    srcptr += 1;
}
```

Now, I am free to reassign the `srcptr` index mid block, and I will do so.

```
'[' => if (storage[memptr] == 0) {
    while (src[srcptr] != ']')
        srcptr += 1;
},
```

This satisfies the test "`[] skips execution and exits`", albeit flimsily, as we'll see.

What about the closing brace? I suppose the analog will be simple enough:

```
test "[] executes and exits" {
    var storage = []u8{0} ** 2;
    const src = "+++++[>+++++<-]";
    bf(src, storage[0..]);
    assert(storage[0] == 0);
    assert(storage[1] == 25);
}
```

```
']' => if (storage[memptr] != 0) {
    while (src[srcptr] != '[')
        srcptr -= 1;
},
```

You might see where this is going… the naive solution to both brackets has a fatal flaw in it completely breaks when there are nested loops of any kind. Consider:

```
++>[>++[-]++<-]
```

This should result in `{ 2, 0 }`, but the first opening bracket will dumbly jump to the first available closing bracket, and then get all confused. I need it to be able to jump to the *next closing bracket at the same nesting depth*. This is a bit fiddly but it's easy to add a depth count and keep track of it while going through the src string. Here, for both directions:

```
'[' => if (storage[memptr] == 0) {
    var depth:u16 = 1;
    srcptr += 1;
    while (depth > 0) {
        srcptr += 1;
        switch(src[srcptr]) {
            '[' => depth += 1,
            ']' => depth -= 1,
            else => {}
        }
    }
},
']' => if (storage[memptr] != 0) {
    var depth:u16 = 1;
    srcptr -= 1;
    while (depth > 0) {
        srcptr -= 1;
        switch(src[srcptr]) {
            '[' => depth -= 1,
            ']' => depth += 1,
            else => {}
        }
    }
},
```

and the corresponding tests- notice the `src` in both of these includes an internal loop.

```
test "[] skips execution with internal braces and exits" {
    var storage = []u8{0} ** 2;
    const src = "++>[>++[-]++<-]";
    try bf(src, storage[0..]);
    assert(storage[0] == 2);
    assert(storage[1] == 0);
}

test "[] executes with internal braces and exits" {
    var storage = []u8{0} ** 2;
    const src = "++[>++[-]++<-]";
    try bf(src, storage[0..]);
    assert(storage[0] == 0);
    assert(storage[1] == 2);
}
```

> *As an aside, `[-]` is a brainfuck idiom that means "zero out this cell". You can see that no matter the value of the cell you're on, it will decrement it until you get down to 0, then go on.*

the unhappy path

I've not accounted for possibly broken bf programs yet. What happens if I feed my interpreter malformed input? Like just

```
[
```

… with no matching closing brace, or

```
<
```

which immediately goes out of bounds of the memory space? (I could wrap this around, but I'd rather consider it an error.)

I'm going to jump ahead a bit and explain all the pertinent differences in this code. I'll pull the `bf` interpreter function into its own file and also pull out the seekBack and seekForward functionalities into their own little functions.

```
const warn = @import("std").debug.warn;
const sub = @import("std").math.sub;

fn seekBack(src: []const u8, srcptr: u16) !u16 {
    var depth:u16 = 1;
    var ptr: u16 = srcptr;
    while (depth > 0) {
        ptr = sub(u16, ptr, 1) catch return error.OutOfBounds;
        switch(src[ptr]) {
            '[' => depth -= 1,
            ']' => depth += 1,
            else => {}
        }
    }
    return ptr;
}

fn seekForward(src: []const u8, srcptr: u16) !u16 {
    var depth:u16 = 1;
    var ptr: u16 = srcptr;
    while (depth > 0) {
        ptr += 1;
        if (ptr >= src.len) return error.OutOfBounds;
        switch(src[ptr]) {
            '[' => depth += 1,
            ']' => depth -= 1,
            else => {}
        }
    }
    return ptr;
}

pub fn bf(src: []const u8, storage: []u8) !void {
    var memptr: u16 = 0;
    var srcptr: u16 = 0;
    while (srcptr < src.len) {
        switch(src[srcptr]) {
            '+' => storage[memptr] +%= 1,
            '-' => storage[memptr] -%= 1,
            '>' => memptr += 1,
            '<' => memptr -= 1,
            '[' => if (storage[memptr] == 0) srcptr = try seekForward(src, srcptr),
            ']' => if (storage[memptr] != 0) srcptr = try seekBack(src, srcptr),
            '.' => warn("{c}", storage[memptr]),
            else => {}
        }
        srcptr += 1;
    }
}
```

This makes the switch statement much easier to read, in my opinion. `seekForward` and `seekBack` look and act *very similar*, and I am tempted to refactor them into something cleverer and more compact, but in the end, they are doing different things, and deal with their error cases slightly differently. It is easier to copy paste and tweak here, and it is clearer. Also I will be factoring out `seekForward` later, at some point, probably in a follow up post.

I've added a few important things, though! Notice that all three of these functions now return a `!` type. This is new syntax for what used to be a `%T` error union type. It's saying that the function can either return a specified type, or some type of error. Whenever I attempt to call a function like this, I must *either* `try` it (with `try` in front of the function call), which will propogate the error up the call stack if it encounters one, or `catch` it like:

```
const x = functionCall() catch {}
```

Where I deal with the error appropriately in the catch block. As written, this catch would swallow any error into the void. This is Bad Practice, but once again I'll point out that Zig *makes me do this explicitly.* If I've caught an error with an empty block, I'm saying that I don't think I'll ever see an error or that I don't need to deal with it. In practice, this should probably always be like a `TODO`, and in fact it is quite easy to make that explicit, too!

```
const x = functionCall() catch { @panic("TODO") }
```

Recall that *this case should never happen in production code.* I'm essentially assuring the compiler that I know what I'm doing, here. If it *could* happen, I should add *proper* error handling.

So what are the errors that I could be returning from seekBack or seekForward?

In seekBack:

```
ptr = sub(u16, ptr, 1) catch return error.OutOfBounds;
```

I've changed this pointer decrement to use the std lib function `sub` which will throw a `Overflow` error if overflow does occur. I want to catch that error and instead return an `OutOfBounds` error, which I am instantiating here simply by using it.

> *Zig errors are basically a global array of error codes that are generated by the compiler when you use `error.Whatever`. They are guaranteed to be unique, and can be `switched` on in switch blocks.*

I want to treat this as `OutOfBounds` because, semantically, if the memory pointer goes under zero, it means I've ask the runtime to point outside of the memory space I've alloted on the low end.

Similarly, in the seekForward function:

```
if (ptr >= src.len) return error.OutOfBounds;
```

In the case that the pointer is larger than the src.len is, I can catch that here and return the same error.

at the call site:

```
'[' => if (storage[memptr] == 0) srcptr = try seekForward(src, srcptr),
']' => if (storage[memptr] != 0) srcptr = try seekBack(src, srcptr),
```

I `try` these functions. If they succeed, they have executed correctly and try returns the new srcptr value. If they fail, `try` aborts the whole function and returns the error to the caller *of `bf` itself.*

That caller would be `main`, now!

```
const bf = @import("./bf.zig").bf;

// yes, hello
const hello_world = "++++++++++[>+++++++>++++++++++>+++>+<<<<-]>++.>+.+++++++..+++.>++

pub fn main() void {
    storage = []u8{0} ** 30000;
    bf(hello_world, storage[0..]) catch {};
}
```

I'm swallowing this error here now, which I should not be doing, but the important point to note is how easy zig makes it to properly handle errors up the call stack. It is not the reponisibility of the caller to check for any particular error state, but the compiler enforces calling any errorable function with try and annotating the correct return values all the way up. It has to be dealt with eventually, *even if it's being ignored*!

> *This new `try/catch` syntax also gets rid of a lot of `%%` and `%` sigils, which people really didn't like much.*

I've now implemented 7 of the 8 symbols in brainfuck, all the ones I need to run "meaningful" programs.

"Meaningful" programs

I've got a program here:

```
// our old friend, the fibonacci sequence.
const fib = "++++++++++++++++++++++++++++++++++++++++++++>+++++++++++++++++++++++++++
```

let me run it…

```
pub fn main() void {
    storage = []u8{0} ** 30000;
    bf(fib, storage[0..]) catch {};
}
```

voila!

```
1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 121, 98, 219,
```

> *I have this memory that always comes back to me when I think about the fibonacci sequence… I learned it from PBS in the 80's, and I've always remembered that. I thought it was lost to time but Youtube is amazing.*

How can I improve this?

I've already alluded to a few `TODO`s. I shouldn't be using stderr for output, I want to be using stdout.

Whenever I invoke the interpreter, I'll open up a stream to stdout and print to that instead:

```
const io = std.io;
...
pub fn bf(src: []const u8, storage: []u8) !void {
    const stdout = &(io.FileOutStream.init(&(io.getStdOut() catch unreachable)).stream
    ...
            '.' => stdout.print("{c}", storage[memptr]) catch unreachable,
            ...
```

What is happening here? I call `io.getStdOut()`, which is failable (and again I am explicitly swallowing that possible error with `catch unreachable`- if this function failed my program would crash!). I call `stream` on that result to initialize a stream, take a pointer to it, and initialize it as an outstream. This outstream is what I assign to stdout, and what I can write to by calling `print` on it. `print` accepts a formatted string just like warn, so that swap is straightforward. `print` can also fail, and I am also swallowing those errors here.

In a proper program, I should account for the potential failure of opening up stdout here, and also the possible errors of trying to write to stdout. I am not doing that, but I am made to say that I am not doing that. Zig makes it easy to ignore these errors as long as you promise you *know that you're ignoring them.*

What happens when I decide I want to turn my prototype into a proper release? Do I sit down with a cup of coffee and start the thankless work of error handling, relying on my decades of experience and knowledge to enumerate every possible error case and how I want to deal with it? What if I don't have decades of experience and knowledge? That's ok, Zig does!

I want to demonstrate a power feature now, error inference!

```
const bf = @import("./bf.zig").bf;
const warn = @import("std").debug.warn;

const serpinsky = "++++++++[>+>++++<<-]>++>>+<[-[>>+<<-]+>>]>+[ -<<<[ ->[+[-]+>++>>>-<

pub fn main() void {
    var storage = []u8{0} ** 30000;
    bf(serpinsky, storage[0..]) catch unreachable;
}
```

I know that `bf` can fail, because it returns `!void`. I am swallowing that error at the call site here in `main`. When I am ready to accept my fate and do the right thing, I can catch that possible error like this:

```
const bf = @import("./bf.zig").bf;
const warn = @import("std").debug.warn;

const serpinsky = "++++++++[>+>++++<<-]>++>>+<[-[>>+<<-]+>>]>+[ -<<<[ ->[+[-]+>++>>>-<

pub fn main() void {
    var storage = []u8{0} ** 30000;
    bf(serpinsky, storage[0..]) catch |err| switch (err) {
    };
}
```

The compiler is my friend now!

```
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.OutOfBounds not handled in switch
shell returned 1
```

This error should be familiar as the one that is bubbling up from `bf` and it's helper functions! *But wait there's more.* Let's say I'm looking at the `stdout` related error handling that I've been swallowing in `bf`. Instead of swallowing them, I'm going to kick them up the chain by using `try`. Remember, used without a catch block on a failable call, try will abort if it encounters an error, forcing *its* caller to deal with any potential errors.

So, instead of:

```
const io = std.io;
...
pub fn bf(src: []const u8, storage: []u8) !void {
    const stdout = &(io.FileOutStream.init(&(io.getStdOut() catch unreachable)).stream
    ...
            '.' => stdout.print("{c}", storage[memptr]) catch unreachable,
            ...
```

I do:

```
const io = std.io;
...
pub fn bf(src: []const u8, storage: []u8) !void {
    const stdout = &(io.FileOutStream.init(&(try io.getStdOut())).stream);
    ...


            '.' => try stdout.print("{c}", storage[memptr]),
            ...
```

Now, compiling

```
const bf = @import("./bf.zig").bf;
const warn = @import("std").debug.warn;

const serpinsky = "++++++++[>+>++++<<-]>++>>+<[-[>>+<<-]+>>]>+[ -<<<[ ->[+[-]+>++>>>-<

pub fn main() void {
    var storage = []u8{0} ** 30000;
    bf(serpinsky, storage[0..]) catch |err| switch (err) {
    };
}
```

Provides me with an enumerated list of *all the possible errors I could get from calling this function!*

```
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.SystemResources not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.OperationAborted not handled in switc
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.IoPending not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.BrokenPipe not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.Unexpected not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.WouldBlock not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.FileClosed not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.DestinationAddressRequired not handle
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.DiskQuota not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.FileTooBig not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.InputOutput not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.NoSpaceLeft not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.AccessDenied not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.OutOfBounds not handled in switch
/Users/jfo/code/zigfuck/main.zig:7:46: error: error.NoStdHandles not handled in switch
shell returned 1
```

Zig empowers me to handle these cases meticulously if I need or want to! I switch on that `err` value, handle cases I want to, and can fall through if I want to.

```
pub fn main() void {
    var storage = []u8{0} ** 30000;
    bf(serpinsky, storage[0..]) catch |err| switch (err) {
        error.OutOfBounds => @panic("Out Of Bounds!"),
        else => @panic("IO error")
    };
}
```

This is still not *proper* error handling, stricly speaking, but I just wanted to demonstrate how clever Zig is about reporting possible error cases to the callsite! And when you encounter an error, you'll get an *error return trace* instead of just a stack trace! Cool stuff!

Todo

There are plenty of improvements I could make to this interpreter! I need to actually properly handle all error cases, obviously, and I need to implement the comma operator "," , which is brainfuck's `getc` function to allow for input into the program runtime. I also should probably make it possible to read a sourcefile into a buffer and interpret that, instead of hard coding all of the bf source code. There are also some improvements I have in mind that aren't strictly necessary but would illuminate some more of Zig itself. Instead of cramming all of that onto the end of this post, I'm going to try splitting them into some upcoming posts that may be smaller and more easily digestible. Stay tuned!

Conclusion

I hope this little half finished miniature project has given you some insight into how Zig code looks and what it might be used for. Zig is not a swiss army knife language, it is not the perfect tool for every job… it has a particular focus in mind, to be a pragmatic systems language that can be used along with and in lieu of the likes of C and C++. It forces you to be meticulous and specific about memory usage, memory management, and error handling. In constrained systems environments, this is a feature not a bug. Zig is deterministic, it's non-ambiguous, it's trying to make it easy to write robust code in environments where that has traditionlly been difficult to do.

I've only covered a very small amount of Zig's syntax and features here, there are a lot of exciting changes coming to the language in 0.2.0 and beyond! It's also worth noting that Zig has wildly diverging compile modes that optimize for different things… all of the compilations I've done here have been in debug mode, which optimizes for safety checks and fast compile times to make iterating easy! There are also currently `--release-fast` mode, and `--release-safe` mode, and there could potentially be more in the future For more about these differences and the original rationale behind them, see here.

I have been continuously impressed with the velocity and direction of Zig's development! It's very much in flux, and will be so until 1.0.0, so if you opt to give it a try just keep that in mind. There will likely be plenty of breaking changes coming up, and there will certainly be many bugs too, but there are a lot of good ideas in there, and I'm excited to see where it goes!

Give it a try, and pop into `#zig` on freenode anytime if you have questions.