Zig 0.13

Entry Point

```
pub fn main() void {
        std.debug.print("Hello
        ↪  world!\n", .{});
}
```

Basic

Variable

```
var n: u8 = 50;
```

Constant

```
const pi: u32 = 314159;
```

Array

```
var array = [_]u8{ 1,
↪  0b0000010, 0x03, 0o04 };
```

Array addition

```
var array_result = array_a
↪  ++ array_b;
```

Array repetition

```
var array_result = array_a
↪  ** 3;
```

Pointer

```
const pointer: *u8 = &n;
```

Pointer dereferencing

```
var n: u8 = pointer.*;
```

Pointer access

```
var n: u8 =
↪  struct_pointer.a;
```

Slice

```
var slice = array[0..3]; //
↪  1, 2, 3
```

Sentinel

```
const ptr: [*:0]u32 = &nums;
```

Tuple (anonymous struct)

```
const tuple = .{true, false,
↪  @as(i32, 42), @as(f32,
↪  3.141592), };
```

Anonymous list

```
const hello: [5]u8 = .{ 'h',
↪  'e', 'l', 'l', 'o' };
```

Bit manipulation

```
const res1 = numOne >> 4;
const res2 = numOne << 4;
const res3 = numOne &
↪  numTwo;
const res4 = numOne |
↪  numTwo;
const res5 = numOne ^
↪  numTwo;
```

Datatypes

Unsigned Integer `u8, u16, u32, u64`
Integer `i8, i16, i32, i64`
Float `f16, f32, f64, f80, f128`
String `[_]u8`
Bool `bool`
Pointer `*u8`
Pointer to Constant `*const u8`
Slice `[]const u8, []u8`
Many-Pointer(length is lost) `[*]const u8, [*]u8`

Strings

String `[_]u8`
Multiline String

```
var string =
  \\Line 1
  \\Line 2
```

Unions

Definitions

```
const Data = union {
  index: u16,
  link: bool,
};
```

Access/Reassignment

```
var node = Data{ .link =
↪  true };      //OK
const node_n = node.index;
↪  //Crash
```

```
node = Data{ .index = 1};
↪ //OK
```

Tagged Union

```
const Data =
↪ union(DataType){
  index: u16,
  link: bool,
};
const Data = union(enum){
  index: u16,
  link: bool,
};
```

Unpack Tagged Union (Values in switch statement are enum values)

```
switch (node) {
  .link => |l| ...,
  .index => |i|,
  inline else => |x| ...,
}
```

Optionals

Optional can be value or null **Defintion**

```
const value: ?u8 = null;
```

Assignment(0 if value is null)

```
const value_b: u8 = value
↪ orelse 0;
```

Forcing value to be not null

```
const value_b: u8 = value
↪ orelse unreachable;
```

Extraction

```
const value_b: u8 = value.?;
```

Error

Error Definition

```
const SpecialError = error{
        NoNumber,
        DivisionByZero,
        InfError,
};
```

Error Union Variable can be either error or datatype

```
var number_or_error:
↪ SpecialError!u8 = 5;
```

Error Catching

```
return funcWithError(n)
↪ catch |err| {
        if (err ==
        ↪ SpecialError.⌋
        ↪ DivisonByZero) {
            return 0;
        }
        return err;
};
```

Standard Error Catching

```
funcWithError(n) catch |err|
↪ return err;
try funcWithError(n);
```

Error Extraction

```
const n = funcWithError();
if (n) |value| {
} else |err| switch (err) {}
```

Error Packaging

```
const SpecialError =
↪ SpecialErrorA ||
↪ SpecialErrorB;
```

Enums

Definition

```
const Fruit = enum { APPLE,
↪ BANANA, STRAWBERRY,  };
const Fruit = enum(u8) {
↪ APPLE = 1, BANANA = 2,
↪ };
```

Structs

Defintion

```
const Picture = struct {
  width: u32,
  height: u32,
  data: [_]u32,
};
```

Declaration

```
var pic = Picture {
  .width = 10,
  .height = 10,
  .data = {...},
};
```

Access `pic.data = {...};`
Method

```
const Picture = struct {
  width: u32,
  height: u32,
  pub fn empty() Picture {
    ...
  }
  pub fn mirrorX(self:
  ↪ *Picture ) void {
    ...
    }
  };
  Picture.empty();
  pic.mirrorX();
```

Anonymous struct

```
fn Circle(comptime T: type)
↪  type {
  return struct {
    center_x: T,
    center_y: T,
    radius: T,
  };
}
```

Flow Control

If Statement

```
if (foo) {
  std.debug.print("True!\n",
  ↪  .{});
} else {
std.debug.print("False!\n",
↪  .{});
}
```

If Assignment

```
const value: u8 = if
↪  (correct) 1 else 2;
```

While loop

```
while (condition) {}
```

While-Loop with continue expression

```
while (condition) : (n*=2)
↪  {}
```

Continue loop

```
while (condition) : (n*=2) {
  if (n % 2 == 0) continue;
}
```

Break loop

```
while (true) : (n*=2) {
  if (n % 2 == 0) break;
}
```

For-Loop

```
for (array) |a| {
  std.debug.print("{}",
  ↪  .{a});
}
for (array, 0..) |a, i| {
  std.debug.print("{} at
  ↪  index {}", .{a, i});
}
for (1..20) |n| {...}
for (hex_nums, dec_nums)
↪  |hn, dn| {...}
```

Switch-Statement

```
switch (c) {
  1 => std.debug.print("A",
  ↪  .{}),
  2 => std.debug.print("B",
  ↪  .{}),
  else =>
  ↪  std.debug.print("?",
  ↪  .{})
}
```

Switch-Assignment

```
const character: u8 = switch
↪  (c) {
  1 => 'A',
  2 => 'B',
  else => '!'
};
foo: switch (@as(u8, 1)) {
  1 => continue :foo 2,
  2 => continue :foo 3,
  3 => return,
  4 => {},
}
```

Loop-Assignment

```
const index: ?u8 = for
↪  (langs, 0..) |lang, i| {
  if (lang.len == 2) break
  ↪  i;
} else null;
```

Lables

```
const value = outer_loop:
↪  for (wave) |v| {
  for (v.frequency, 0..) |f,
  ↪  i| {
    if (f.frequency == 0)
    ↪  continue :food_loop;
  }
} else wave[0];
```

Functions

Function

```
fn func(argument: u32) u32 {
  return argument;
}
```

Pass By Reference

```
fn func(argument: *u32)
↪  void{
  argument = 0;
}
```

Function with possible Error

```
fn func(argument: u32)
↪  SpecialError!u32 {
  ...
  return
  ↪  u32SpecialError.InfError;
  ...
  return argument;
}
```

Generic function:

```
fn makeSequence(comptime T:
↪  anytype) void {}
```

?

Defer (Put an statement to end of block)

```
std.debug.print("(", .{});
defer std.debug.print(") ",
↪  .{});
printVector(vec);
```

Error defer

```
fn funcWithError()
↪  SpecialError!u32 {
  // print if function exits
  ↪  with an error:
  errdefer
  ↪  std.debug.print("failed!\n",
  ↪  .{});
}
```

Unreachable (Make specific blocks unreachable -> defined program crash)

```
switch (op) {
  else => unreachable
}
```

Undefined (Access of undefinied variables is not allowed)

```
var n: u8 = undefined;
```

Quoted Identifier (Put an statement to end of block) `@"123_nums"`

Tests

```
test "add" {
  try testing.expect(add(41,
  ↪  1) == 42);
  try testing.⌋
  ↪  expectError(error.⌋
  ↪  DivisionByZero,
  ↪  divide(15, 0));
}
```

Async

BuiltIn

Get the innermost struct/enum/union

```
@This()
```

Typeinfo:

```
@typeInfo(Narcissus).⌋
↪  @"struct".fields;
pub const StructField =
↪  struct {
    name: []const u8,
    type: type,
    default_value:
    ↪  anytype,
    is_comptime: bool,
    alignment:
    ↪  comptime_int,
};
```

Compile Time logging

```
@compileLog("Count at
↪  compile time: ");
```

Compile Time Inheritance(?) (Returns true if type has a method with given name)

```
@hasDecl(Type, "function");
```

Import c header file

```
const c = @cImport({
    @cInclude("unistd.h");
});
```

Vector

```
@Vector(len: comptime_int,
↪  Element: type)
```

Absoulte Value

```
@abs(value: anytype)
```

Transform vector to scalar

```
@reduce(comptime op:
↪   std.builtin.ReduceOp,
↪   value: anytype)
```

Comptime

Compile time loop

```
inline for (fields) |field|{
  if (field.type != void) {
    print(" {s}",
    ↪   .{field.name});
  }
}
```

Compile time variable

```
comptime var scale: u32 =
↪   undefined;
```

Compile time function

```
fn makeSequence(comptime T:
↪   type, comptime size:
↪   usize) [size]T {}
```

Compile time block `comptime {...}`

C Interaction

Standard Library

Import Std

Index of

```
@import("std").mem.indexOf;
```

Std out

```
const stdout =
↪   std.io.getStdOut().writer();
stdout.print("Hello
↪   world!\n", .{});
```

Fmt (Variabletype:filler(Alignment:<>ˆ)Space)

```
print("{s:*^20}\n",
↪   .{"Hello!"});
```

Tokenizer

```
var it =
↪   std.mem.tokenizeAny(u8,
↪   poem, " ,;!\n");
```

Threads

```
const handle = try
↪   std.Thread.spawn(.{},
↪   thread_function, .{1})
defer handle.join();
```

Filesystem

```
const cwd: std.fs.Dir =
↪   std.fs.cwd()
cwd.makeDir("dir") catch |e|
↪   switch (e) {...}
var output_dir: std.fs.Dir =
↪   cwd.openDir("dir", .{});
defer output_dir.close();
const file: std.fs.File =
↪   try
↪   output_dir.createFile("file.txt",
↪   .{});
defer file.close();
const byte_written = try
↪   file.write("File
↪   Opened");
```

Allocation

Arena Allocator

```
var arena = std.heap.⌋
↪   ArenaAllocator.⌋
↪   init(std.heap.⌋
↪   page_allocator);
defer arena.deinit();
const allocator =
↪   arena.allocator();
const avg: []f64 = try
↪   allocator.alloc(f64, 5);
```

General purpose allocater

```
var gpa = heap.⌋
↪   GeneralPurposeAllocator(.⌋
↪   {}){};
defer if (gpa.detectLeaks())
↪   log.err("Memory leak
↪   detected!", .{});
const alloc =
↪   gpa.allocator();
```

Build System

Fetch Dependy

```
zig fetch --save=vaxis
↪ https://github.com/␣
↪ rockorager/libvaxis/␣
↪ archive/refs/tags/␣
↪ v0.5.1.tar.gz
```

Commands

New Project

```
zig init
```

Examples

Create map with names of enum

```
pub const entity_names: std.StaticStringMap(EntityType) = EnumStrMap(EntityType);
pub fn EnumStrMap(V: type) std.StaticStringMap(V) {
        comptime {
                const field = @typeInfo(V).Enum.fields;
                const array_type = struct { [:0]const u8, V };
                var array: [field.len]array_type = undefined;
                for (field, 0..) |f, i| {
                        array[i] = array_type{ f.name, @as(V, @enumFromInt(f.value)) };
                }
                return std.StaticStringMap(V).initComptime(array);
        }
}
```

Links/Documentation

- Zig Documentation 0.14
- Zig Standard Library Documentation 0.14
- Zig Guide
- Ziglings examples
- Zig cookbook
- Zig forum