Zig - Performance Matters

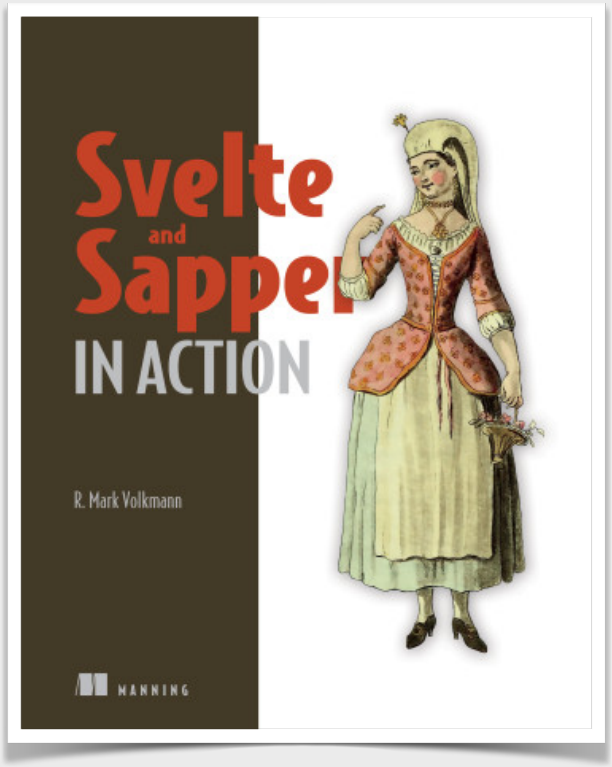
R. Mark Volkmann
Object Computing, Inc.
https://objectcomputing.com
mark@objectcomputing.com
@mark_volkmann

OBJECT COMPUTING
YOUR OUTCOMES ENGINEERED

Slides at **https://github.com/mvolkmann/talks/**

About Me

- Partner and Distinguished Software Engineer at Object Computing, Inc. in St. Louis, Missouri USA
- 43 years of professional software development experience
- Writer and speaker
- Blog at https://mvolkmann.github.io/blog/
- Author of Manning book “Svelte ... in Action”



Language Categories

Have a favorite in each category!

- **Scripting**
  - JavaScript
  - **Lua**
  - Python
  - Ruby
  - TypeScript
- **Compiled w/ GC**
  - C#
  - Go
  - Java
  - Kotlin
  - **Swift**
- **Compiled w/o GC**
  - C
  - C++
  - Rust
  - **Zig**

systems programming

These categories ignore distinctions like procedural, object-oriented, and functional. Zig is procedural.

Zig Overview

- Free, open source, high performance, systems programming language
- Modern alternative to C with ability to use C libraries

can also use C++ libraries with a bit more work

- Provides **complete LLVM-based toolchain** for creating, developing, building, and testing apps written in Zig, C, and C++
  - uses clang to compile C and clang++ to compile C++
- Suitable for apps that care deeply about **performance**, **memory usage**, or **binary size**
  - these concerns justify the tedium of manual memory management
- Seen as **simpler** than C++ and Rust and safer than C

Zig Goals

- **No hidden control flow**
  - no exception handling, operator overloading, destructors, or decorators
- **No hidden memory allocations**
  - all memory allocation is performed by allocators selected by developer
  - each kind of allocator implements a different allocation strategy
  - does not support closures, so allocations do not outlive their scope
- **No preprocessors or macros**
  - instead Zig uses code that runs at compile-time, indicated by **`comptime`** keyword
- **One obvious way to accomplish each task**

Zig Provides

- Package manager for managing dependencies
- Build system
  - simpler than combinations of built tools used with C and C++
  - includes API used in **`build.zig`** files
- Cross compilation support
  - can build executables for platforms other than current
- Test runner
- LLVM targets
  - can target all platforms supported by LLVM, including WebAssembly

Pros of Zig

- Run-time speed
- Integrated build system
- Fast compiler compared to C++ and Rust
- Manual memory management for great control
- Integration with C and C++
- Compiler enforced null handling
- Integrated test framework
- SIMD support with vectors

Cons of Zig

- Not yet 1.0 and not expected until 2025
- Manual memory management can be tedious
- String handling is tedious, but libraries are available
- Labeled **`break`** syntax is odd
- No checking for use of variables with **`undefined`** value
- Some stack traces do not include offending line

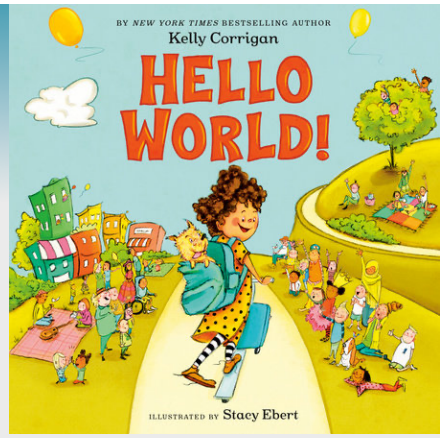
Where Used

- **Bun** - JS/TS run-time and toolchain
  - has many advantages over Node.js and Deno, especially performance
- **TigerBeetle** - “worlds fastest financial accounting database”
- **Roc** - “a fast, friendly, functional language”
  - “Roc's compiler has always been written in Rust.”
  - “Roc's standard library was briefly written in Rust, but was soon rewritten in Zig.”
- **Mach** - game engine and graphics toolkit
- **Uber** - uses Zig only to build its C++ applications
- **Ghosty** - terminal emulator

Installing

- To install Zig
  - download platform-specific zip or tar file from https://ziglang.org/getting-started/
    - choose a tagged release or a nightly build
  - expand it
  - move created directory to desired location
  - set environment variable **`ZIG_PATH`** to point to this directory
  - add **`ZIG_PATH`** to **`PATH`** environment variable
- To see version installed, enter **`zig version`**
- To see list of Zig guiding principles, enter **`zig zen`**

Hello World




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

pub fn main() void {
    // s for string, d for decimal
    print("Hello {s}! {d}\n", .{"Zig", 2023});
}
```

`@import` is a builtin function that returns a struct instance contains the entire standard library

if `main` can return an error, return type must be `!void`

syntax `.{ ... }` creates an anonymous tuple or struct

Builtin Functions

- Zig provides over 100 builtin functions
  - 117 as of November 2023
- Known to compiler and do not require importing
- All names begins with `@`
  - followed by an uppercase letter if function returns a type
- Some have behavior that normal functions cannot replicate
- Unofficial categories
  - Atomic and Memory, Bitwise, Cast and Conversion, Introspection, Math, Metaprogramming, Programming, Runtime and Async, and Other

Math builtin functions include **`@abs`**, **`@ceil`**, **`@cos`**, **`@floor`**, **`@log`**, **`@log10`**, **`@max`**, **`@min`**, **`@mod`**, **`@round`**, **`@sin`**, **`@sqrt`**, **`@tan`**, and **`@trunc`**.

C and C++

- Zig currently use **clang** to compile C code and **clang++** for C++
  - can find bugs that other C/C++ compilers do not
  - can build platform-specific executables for a specified platform
- To build **`hello.c`**, enter **`zig cc hello.c -o hello`**
- To build **`hello.cpp`**, enter **`zig c++ hello.cpp -o hello`**
- To run resulting executables, enter **`./hello`**

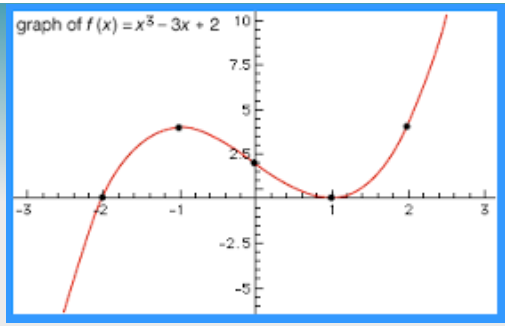
```c
// hello.c
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

```cpp
// hello.cpp
#include <iostream>

int main() {
  std::cout << "Hello World!" << std::endl;
  return 0;
}
```

Conventions

- Names
  - functions: **`camelCase`**
  - variables: **`snake_case`**
  - types: **`PascalCase`**
- Indentation: 4 spaces
- Open braces on same line as statement
- Directory and file names: **`snake_case`**
  - but files that define a type should use **`PascalCase`** with same name as type

Functions



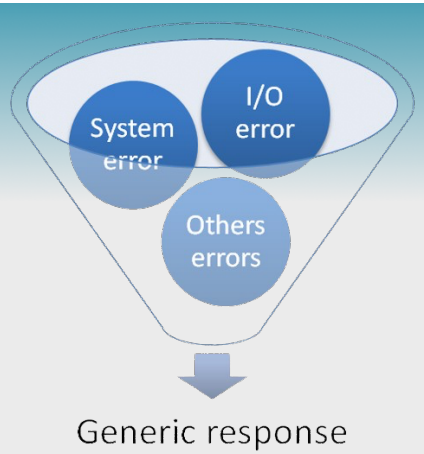
- To declare

```
[pub] fn {name}([parameter-list]) {return-type} {
    {body}
}
```

- Parameter list is comma-separated list of **`{name}: {type}`**
- Return type is **`{return-type}`** or **`[error-type]!{return-type}`**
  - will infer **`error-type`** if omitted
- Example

```
const EvalError = error{ Negative, TooHigh };

fn double(n: i8) EvalError!i8 {
    if (n < 0) return EvalError.Negative;
    if (n > 100) return EvalError.TooHigh;
    return n * 2;
}
```

Error Handling



- Functions can return an error value, not throw an error
- Must declare ias part of return type
  - ex. `fn double(n: i8) EvalError!i8 { ... }`

  or just `!i8` to infer possible error types
- Error types are declared with **`error`** keyword
  - ex. `const EvalError = error { Negative, TooHigh };`

  like **`enum`** values
- When calling a function that can return an error, must use **`try`** or **`catch`**
  - **`try`** returns error to caller of current function

    ```
    var result = try double(101);
    ```
  - **`catch`** provides value to use in place of error

    ```
    var result = double(-1) catch 0;
    ```

errdefer Keyword

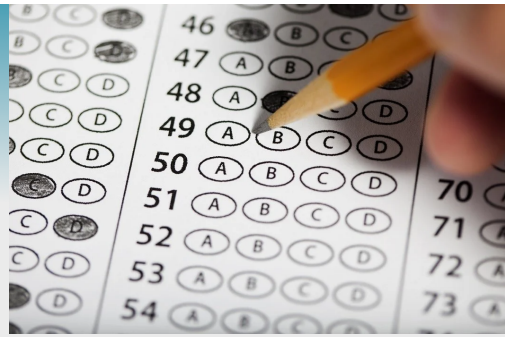
- **`errdefer`** keyword specifies an expression to evaluate if an error is returned from current scope

Tests ...

- Can include unit tests in same file as functions being tested
- **`std.testing`** module provides many **`expect`** functions
  - `expectEqual`
  - `expectEqualDeep`
  - `expectEqualStrings`
  - `expectStringStartsWith`
  - `expectStringEndsWith`
  - `expectEqualSlices`
  - `expectError`
  - `expectApproxEqAbs`
  - `expectApproxEqRel`
  - `expectEqualSentinel`
  - `expectFmt`

```
const std = @import("std");
const expectEqual = std.testing.expectEqual;

pub fn add(a: i32, b: i32) i32 {
    return a + b;
}

test add { // uses function name
    try expectEqual(add(1, 2), 3); // passes
}

test "add works" { // uses description string
    try expectEqual(add(1, 2), 3); // passes
    try expectEqual(add(2, 3), 50); // fails
}
```

... Tests



- To execute all tests in a specific file
  - **`zig test {file-name}.zig`**
- To execute tests in all reachable files (imported and used)
  - create a Zig project (described later) and enter **`zig build test`**
  - alternatively, see “Tests” section my Zig blog page

Projects

- To create a new Zig project, enter **`zig init-exe`**
  - creates **`build.zig`** file described on next slide
  - creates **`src`** directory containing **`main.zig`** file
    - app starting point that defines **`main`** function

Zig Build

- API used in **`build.zig`** file
  - similar to Node.js **`package.json`**, but “scripts” are referred to as “steps”
  - provided steps are **`install`**, **`uninstall`**, **`run`**, and **`test`** runs all tests
  - to run a step, enter **`zig build {step-name}`**
  - to build an executable, enter **`zig build`** creates **`zig-out/bin/{project-name}`**
  - modify to customize build process
  - for help, enter **`zig build --help`** or **`-h`**
  - to see available steps, enter **`zig build --list-steps`** or **`-l`**
  - can define custom steps in this file see my Zig blog page for details

Optimizations and Targets

- To build an executable with specific optimizations, enter **`zig build -Doptimize={value}`**

`-Doptimize` value	Run-time safety checks	Optimizations
`Debug`	Yes	No
`ReleaseSafe`	Yes	Yes (speed)
`ReleaseFast`	No	Yes (speed)
`ReleaseSmall`	No	Yes (size)

- To build an executable for a different target than current machine, enter **`zig build -Dtarget={target}`**
  - to see list of supported targets, enter **`zig targets`**
  - an example is **`x86_64-windows`**

Comment Syntax

- `//` for single-line comments
- `//!` for top-level module documentation
- `///` for documenting top-level variables, functions, and types
- no multiline comments
  - relies on editor support for applying single-line comments to multiple selected lines

Primitive Types

- signed integers - **`i{bits}`**
- unsigned integers - **`u{bits}`**
- floating point - **`f{n}`**
  where n is 16, 32, 64, 80, or 128
- **`isize`**, **`usize`**
  - uses pointer size of current CPU
- C types
  - like **`c_char`**, **`c_short`**, **`c_long`**, and many more

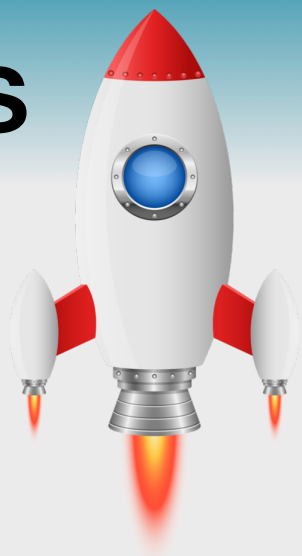
for interfacing with C

supports duck typing

- **`anyerror`**, **`anyopaque`**, **`anytype`**
- **`bool`** - values are **`true`** and **`false`**
- **`comptime_int`**, **`comptime_float`**
- **`noreturn`** -
  type of functions that never finish
- **`type`** - describes a type
- **`void`** - no value

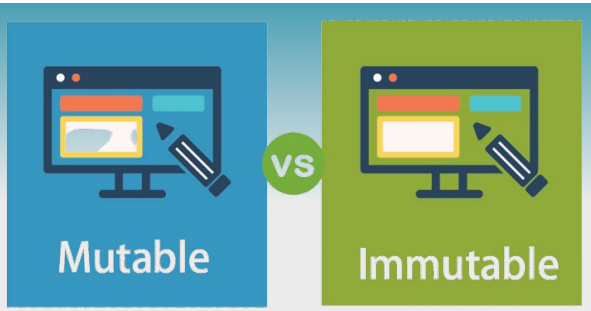
These are the types of compile-time known, literal integer and floating point values that can be any size. Their values are inlined in the generated assembly instructions, so they don't occupy memory.
This makes their byte size irrelevant.

Builtin Non-Primitive Types



- **`enum`** - enumeration of values
- **`error`** - defines an error set which is similar to an **`enum`**
- **Array** - contiguous memory with compile-time known, fixed length
- **Slice** - array-like view of an array subset whose length may not be known until run-time
- **`struct`** - custom type that holds a collection of fields, methods, namespaced constants, and namespaced functions
- **Tuple** - anonymous struct without specified field names
  - field names default to indexes starting from zero
  - field values can all be of different types
- **`union`** - set of fields where only one is active at a time
  - each can have a different type
- **Pointer** - address of a value in memory
  - often used to pass values by reference so a function can modify value

Variables



- Declared with
  `{const|var} {name}[: type] = {value};`
- **`const`** is immutable and **`var`** is mutable
- Names must begin with a letter and
  be composed of letters, numbers, and underscores
- Can omit type if it can be inferred from initial value
- All variables must be initialized, but can initialize to **`undefined`**
- Variable declarations can appear at file, function, and block scope
  - file scope is referred to as “container level”

Optionals

- Types of variables, **`struct`** fields, and function parameters can be made optional by preceding with **`?`**
  - allows assigning **`null`**
- Two ways to check for null

```
var opt: ?i32 = null;

const value = opt orelse 0;

if (opt) |value| { // captures unwrapped value
    // Use unwrapped value.
} else {
    // Handle null value.
    // Can panic using unreachable.
}
```

Pointers ...

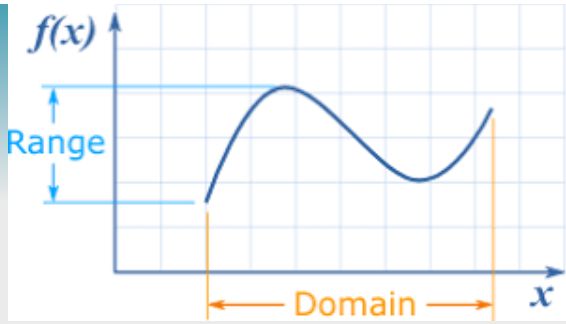
- To get a pointer to data in a variable, use **`&variable_name`**
- To dereference a pointer, use **`variable_name.*`**
- When value is a struct, can use chaining to access a field
  - ex. **`dog_ptr.*.name`**, but compiler treats **`dog_ptr.name`** as the same
- A **`const`** pointer cannot be changed to point to a different value
- A pointer to a non-**`const`** value can be used to modify the value regardless of whether the pointer is **`const`**
- A pointer cannot be set to **`null`** unless its type is optional

... Pointers

- Zig supports two kinds of pointers, single-item and many-item

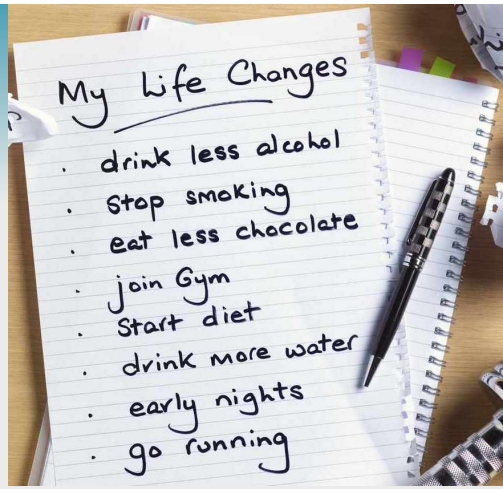
Type	Meaning
`*T`	pointer to a T value
`?*T`	optional pointer to a T value
`*?T`	pointer to an optional T value
`[*]T`	pointer to an unknown number of T values
`?[*]T`	optional pointer to an unknown number of T values
`[*]?T`	pointer to an unknown number of optional T values
`?[*]?T`	optional pointer to an unknown number of optional T values

Ranges



- Ranges of numbers have an inclusive lower bound and an upper bound that is either exclusive or inclusive
  - exclusive - `5..7` with two dots includes `5` and `6`
  - inclusive - `5...7` with three dots includes `5`, `6` and `7`
- Exclusive ranges can be used to create a slice from an array, but not inclusive
- Inclusive ranges can be used in **`switch`** branches, but not exclusive

Enumerations ...



- Typically defined at file scope rather than inside functions

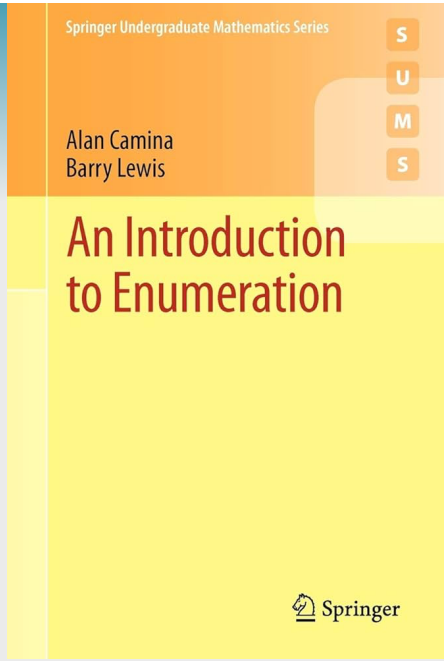
```
const Color = enum { red, yellow, blue, green };
```

- Must declare with **`const`** or **`comptime`**, not **`var`**
- Instances have unique ordinal values starting from zero by default
- Can get value an instance
  - `@intFromEnum(Color.yellow)` returns **`1`**
- Can get instance from a value
  - `const color: Color = @enumFromInt(1);` assigns **`Color.yellow`**

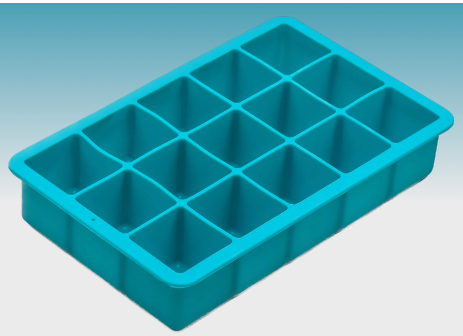
... Enumerations

A numeric type for ordinal values must be specified in order to access and override values.

```
var const Color = enum(u8) {
    red, // defaults to 0
    yellow, // assigned 1
    blue = 7, // overrides default of 2
    green, // assigned 8

    const favorite = Color.yellow;
    // const favorite: Color = .yellow; // alternatively

    const Self = @This();
    pub fn isPrimary(self: Self) bool {
        return self == Self.red or
               self == Self.yellow or
               self == Self.blue;
    }
};
```



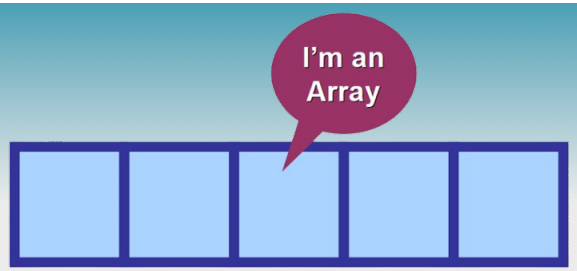
Arrays ...



- Contiguous memory with compile-time known, fixed length and zero-based indexes
- Represented by a pointer and a **`len`** field of type **`usize`**
- Strings are arrays of characters
- Multidimension arrays
  - created by nesting single-dimension arrays
- Use **`std.ArrayList`** for growable arrays

```
// Use const for immutable array.
const dice_rolls = [5]u8{ 4, 2, 5, 1, 2 };
const third = dice_rolls[2]; // 5

// Use _ for length to infer.
const dice_rolls = [_]u8{ 4, 2, 5, 1, 2 };

// Use var for mutable array.
// Use ** operator to repeat a value.
var dice_rolls = [_]u8{0} ** 5; // 5 zero elements
dice_rolls[2] = 6; // modifies third element

const santa = "Ho " ** 3; // "Ho Ho Ho "

// Use ++ operator to create new array
// by concatenating two existing arrays.
var name = "Ma" ++ "rk"; // "Mark"
```

... Arrays




```
// A for loop can iterate over elements in an array or slice.
// Can iterate over multiple arrays at the same time.
// For example, can iterate over elements AND their indices.
for (dice_rolls, 0..) |roll, index| {
    try expectEqual(roll, dice_rolls[index]);
}

// Can get a slice of an array.
const subset = dice_rolls[2..4];
```

more on **for** loops later

Slices

- Reference to a range of array elements
- Like arrays, represented by a pointer and a **`len`** field
- Specified with a range of indexes separated by two dots

```
var array = [_]u8{ 1, 2, 3, 4, 5 };

var fullSlice = array[0..]; // gets all elements

var slice = array[2..4]; // gets elements at indexes 2 & 3
slice[0] = 30; // changes array[2] from 3 to 30
array[3] = 40; // changes slice[1] from 4 to 40
```

Strings ...

- Zig does not provide a dedicated string type, but libraries do
  - see https://github.com/JakubSzark/zig-string
- Zig represents a string as an array of **`u8`** values
- Literal strings
  - for single-line delimit with double quotes
  - for multiline precede each line with **`\\`**
    - newline is added after each line but last
  - turned into array of type **`[]const u8`**
  - null (0) terminated for C compatibility

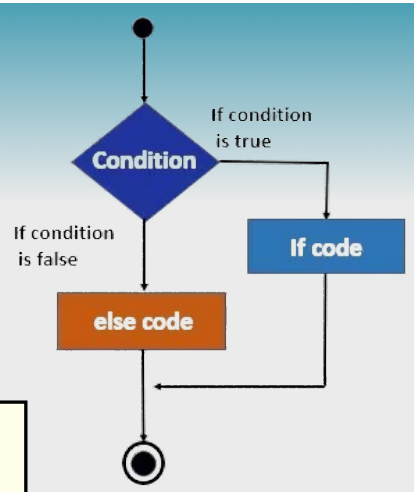
```
var name = "Mark";

const haiku =
    \\Out of memory.
    \\We wish to hold the whole sky,
    \\But we never will.
;
```

- See **`std.ascii`** and **`std.unicode`** namespaces

... Strings

- To define type
```
const String = []const u8;
```
optional
- To assign to a variable
```
var name: String = "Mark";
```
- To get a byte
```
const letter = name[1]; // a
```
- To modify a byte
```
name[1] = 'o'; // Mork
```
- To iterate over bytes
```
for (name) |byte| { ... }
```
- To compare
```
if (std.mem.eql(u8, name, "Mark")) { ... }
if (std.mem.startsWith(u8, name, "Ma")) { ... }
if (std.mem.endsWith(u8, name, "rk")) { ... }
```
compares arrays with a given element type, `u8` in this case
- To split
```
const colors = "red,green,blue";
var iter = std.mem.splitScalar(u8, colors, ',');
while (iter.next()) |color| { ... }
```
also see methods `splitSequence`, `splitAny`, and `token*`

if Expression



- Conditions must evaluate to a **`bool`** or optional value
  - other types are not interpreted as **`true`** or **`false`**

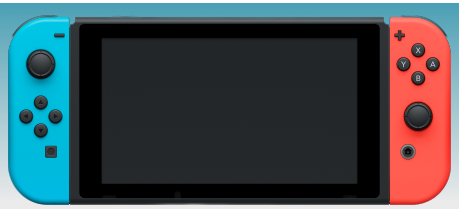
```
if (condition1) {
    ...
} else if (condition2) {
    ...
} else {
    ...
}
```

```
if (optional) |unwrapped| {
    // Use unwrapped value.
} else {
    // Handle case where optional is null.
}
```

- Zig doesn't support the ternary operator, but an **`if`** expression can be used instead

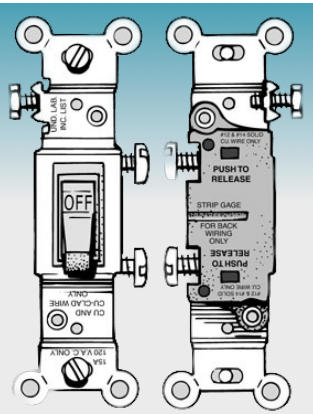
```
const value = if (condition) true_value else false_value;
```

switch Expression ...



- Expression that follows **`switch`** must evaluate to an integer, enum value, or **`bool`**, not a string
- Cases are referred to as “branches"
- Branches
  - can match a single value, a list of values, or a range of values
  - are followed by **=>**, an expression, and a statement or block
  - must be exhaustive (ex. matching all possible values of an **`enum`**) or include **`else`** branch
- Must be possible to coerce all branch values to a common type

... switch Expression




```
const print = std.debug.print;

fn log(comptime text: []const u8) void {
    print(text ++ "\n", .{});
}

switch (value) {
    1 => log("one"),
    2...5 => |capture| {
        print("got {}\n", .{capture});
    },
    6, 8, 10 => {
        log("six, eight, or ten");
    },
    else => print("unhandled {}\n", value);
}
```

Triple-dot ranges have an inclusive upper bound.
Double-dot ranges are not allowed here.

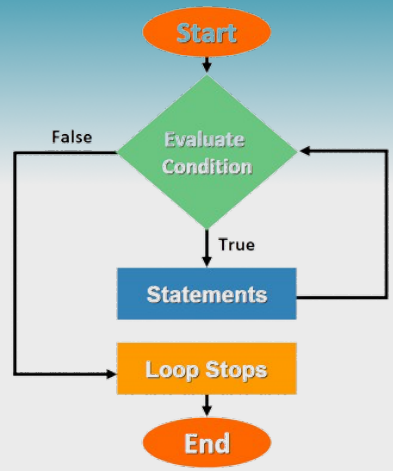
Capture is useful when **`switch`** expression is not just a variable ... perhaps a function call.

Braces are only required for multiple statements.

using **`switch`** as an expression

```
const result = switch (value) {
    1 => "single",
    2 => "couple",
    3 => "few",
    else => "many",
};
```

while Expression ...



- Nearly identical to C **`while`** statements
- Exits if expression evaluates to **`false`** or **`null`**
  - exiting on **`null`** is useful when expression is a call to a function that returns an optional value
- Can contain **`break`** and **`continue`** statements

```
const print = std.debug.print;

var value: u8 = 0;
// outputs 1, 2, 4, 5
while (value < 5) {
    value += 1;
    if (value == 3) continue;
    print("{}\n", .{value});
}
```

continue expression

```
var value: u8 = 1;
// outputs 1, 2, 3
while (value <= 3) : (value += 1) {
    print("{}\n", .{value});
}
```

... while Expression

- Can be used as an expression to obtain a value

```
var value: u8 = 0;
// result is "triple" if value starts at 1
// and "not found" if value starts at 0.
const result = while (value < 10) {
    if (value == 3) break "triple";
    value += 2;
} else "not found";
```

break exits the loop and uses value that follows

else value is used if **while** does not exit with a **break**

- Can catch errors when expression evaluates to an error

```
while (fetchCount()) |count| {
    print("{}\n", .{count});
} else |err| {
    print("err = {}\n", .{err});
}
```

fetchCount is a function that can return a **u8** value or an error

for Expression ...

- Can iterate over elements in an array or slice

```
const numbers = [_]u8{ 10, 20, 30 };
// outputs 10, 20, 30
for (numbers) |number| {
    print("{}\n", .{number});
}
```

captures current value

- Can iterate over a range of integers

```
// outputs 10, 11, 12, 13, 14
for (10..15) |number| {
    print("{}\n", .{number});
}
```

- Can include **`break`** and **`continue`** statements

... for Expression ...

- Can iterate over any number of arrays and slices at the same time

```
const letters = "ABC";
const numbers = [_]u8{ 10, 20, 30 };
// This outputs the ASCII code of each letter
// followed by the number at same index.
// The arrays must have the same length.
for (letters, numbers) |letter, number| {
    print("{} - {}\n", .{ letter, number });
}
```

captures a value from each array

- Use an open-ended range starting from zero to get index values

```
for (0.., numbers) |index, number| {
    print("{} - {}\n", .{ index, number });
}
```

... for Expression

- To mutate elements while iterating, iterate over pointers to elements

```
var mutable = [_]u8{ 1, 2, 3 };
for (&mutable) |*item| {
    item.* *= 2; // doubles
}
```

- Can be used as an expression to obtain a value

```
// result is "triple" if value starts at 1
// and "not found" if value starts at 0.
const result = for (1..10) {
    if (value == 3) break "triple";
    value += 2;
} else "not found";
```

Vectors ...

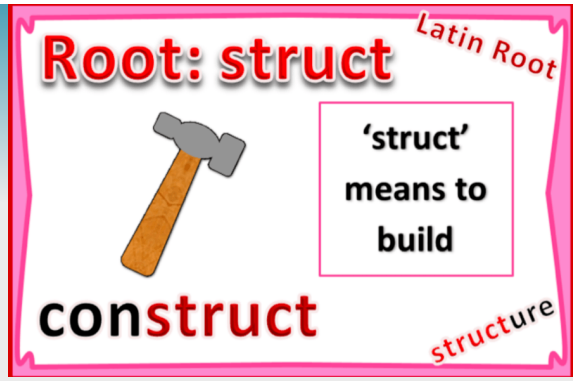
- Array-like collection of elements that all have same type
  - restricted to **`bool`**, integer types, float types, and pointers
  - fixed length
- Type is defined using builtin function **`@Vector`**
- Some operations can be performed in parallel
  - using some operators and builtin functions
  - very fast when processor supports "Single Instruction, Multiple Data" (SIMD)
  - also see standard library namespace **`std.simd`**

... Vectors


```
const std = @import("std");
const expectEqual = std.testing.expectEqual;

test "basic" {
    const MyVectorType = @Vector(5, f32); // defines a type
    const vector = MyVectorType{ 1.2, 2.3, 3.4, 4.5, 5.6 }; // creates an instance
    try expectEqual(vector[2], 3.4);
    try expectEqual(@reduce(.Min, vector), 1.2); // finds smallest element
    try expectEqual(@reduce(.Max, vector), 5.6); // finds largest element
    try expectEqual(@reduce(.Add, vector), 17); // adds all elements

    const vector2 = MyVectorType{ 1, 2, 3, 4, 5 }; // creates another instance
    const vector3 = vector + vector2; // creates instance by adding two of them
    try expectEqual(vector3[4], 10.6);
}
```

Structs ...

- Custom type that holds a collection of fields, methods, namespaced constants, and namespaced functions
- All members are “public”
- Improvement over C structs which only contain fields

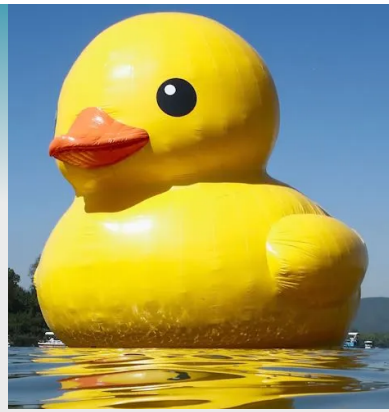


... Structs

```zig
const std = @import("std");
const sqrt = std.math.sqrt;
const expectEqual = std.testing.expectEqual;

fn square(n: f32) f32 {
    return std.math.pow(f32, n, 2); // or n * n
}

const Point = struct {
    pub const dimensions = 2; // namespaced constant

    x: f32 = 1, // default value
    y: f32 = 2, // default value

    // Defining an init function is optional.
    pub fn init(x: f32, y: f32) @This() {
        return Point{ .x = x, .y = y };
    }

    // Methods take an instance as first argument.
    pub fn distanceToOrigin(self: Point) f32 {
        return sqrt(square(self.x) + square(self.y));
    }

    pub fn distanceTo(self: Point, other: Point) f32 {
        const dx = self.x - other.x;
        const dy = self.y - other.y;
        return sqrt(square(dx) + square(dy));
    }
};
```

```zig
// Typically this would be a method in the Point struct,
// but we want to demonstrate passing a pointer to a struct
// to enable modifying fields.
fn translate(pt: *Point, dx: f32, dy: f32) void {
    pt.x += dx;
    pt.y += dy;
}

test "Point struct" {
    try expectEqual(Point.dimensions, 2);

    var p1 = Point{}; // uses default values for x and y
    try expectEqual(p1.x, 1);
    try expectEqual(p1.y, 2);

    const p2 = Point{ .x = 3, .y = 4 };
    // Two ways to call a method.
    try expectEqual(p2.distanceToOrigin(), 5);
    try expectEqual(Point.distanceToOrigin(p2), 5);

    const p3 = Point.init(6, 8);
    try expectEqual(p2.distanceTo(p3), 5);

    // Passing pointer so struct instance can be modified.
    translate(&p1, 2, 3);
    try expectEqual(p1.x, 3);
    try expectEqual(p1.y, 5);
}
```

Duck Typing ...



World's largest rubber duck

- Functions can have parameters with type **`anytype`**
  - allows any kind of value to be passed
  - compiler verifies that the value can be used correctly by function body
  - supports "duck typing"

```
const std = @import("std");
const String = []const u8;

const Animal = struct {
    name: String,
    top_speed: u32, // miles per hour
};

const Car = struct {
    make: String,
    model: String,
    year: u16,
    top_speed: u32, // miles per hour
};

const Wrong = struct {
    top_speed: f32, // not expected type u32
};
```

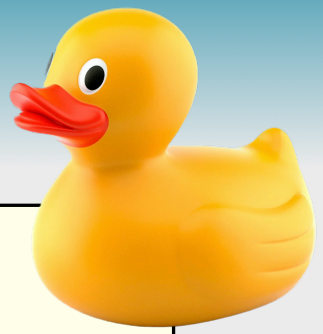
The compiler will verify that "thing" is a struct with "top_speed" field that is an integer because it is used that way here.

returns an error if **`thing`** is an incompatible type

```
fn travelTime(thing: anytype, distance: u32) !f32 {
    const s: f32 = @floatFromInt(thing.top_speed);
    const d: f32 = @floatFromInt(distance);
    return d / s;
}
```

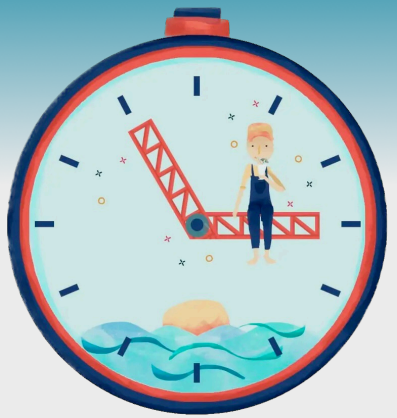
We can't eliminate the local variable d because @floatFromInt requires that we specify the result type.

... Duck Typing



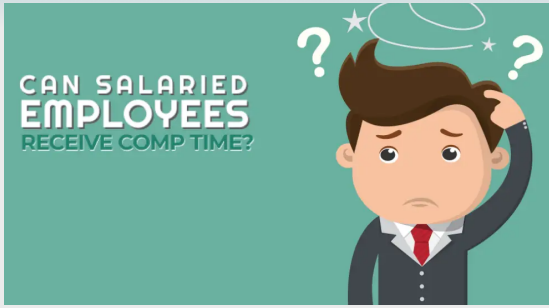
```
const expectApproxEqAbs = std.testing.expectApproxEqAbs;

test "anytype" {
    const cheetah = Animal{ .name = "cheetah", .top_speed = 75 };
    const distance = 20; // miles
    const tolerance = 0.001;
    try expectApproxEqAbs(try travelTime(cheetah, distance), 0.2667, tolerance);

    const ferrari = Car{ .make = "Ferrari", .model = "F40", .year = 1992, .top_speed = 201 };
    try expectApproxEqAbs(try travelTime(ferrari, distance), 0.0995, tolerance);

    // This results in a compile error which is good because
    // the first argument is struct whose top_speed field is not an integer.
    // const wrong = Wrong{ .top_speed = 1.0 };
    // _ = try travelTime(wrong, distance);

    // This results in a compile error which is good because
    // the first argument is not a struct with a "top_speed" field.
    // _ = try travelTime("wrong", distance);
}
```

comptime Keyword ...



- Marks items that must be known at compile-time
  - function parameters
  - variables declared inside functions that are initialized at compile-time
  - expressions such as function calls that are evaluated at compile-time
  - blocks of code that will be run at compile-time
- Takes place of **preprocessor directives** and **macros** in C and C++
- Some things are automatically evaluated at compile-time
  - initial values of variables declared at container level (outside any function)
  - type declarations of variables, functions (parameter and return types), enums, structs, and unions

... comptime Keyword



- Code executed at compile-time has several limitations
  - cannot have side-effects such as performing I/O operations or sending network requests
  - cannot perform more that a fixed number of branching operations such as loop iterations or recursive calls
    - limit can be set by calling `@setEvalBranchQuota(quota)`
    - defaults to 1000 and cannot be set lower
- One use of the **`comptime`** keyword is implementing generic types
  - discussed ahead

Compiler Explorer (godbolt.org)

This shows how a value (sum of numbers in an array in this can) can be computed at **compile-time** and the result can be hard-coded into the generated code.

godbolt.org

COMPILER EXPLORER Add... More Templates Sh

Zig source #1

```
1   fn sum(numbers: []const u32) u32 {
2       var _sum: u32 = 0;
3       for (numbers) |number| {
4           _sum += number;
5       }
6       return _sum;
7   }
8
9   const scores = [_]u32{ 10, 20, 30, 40, 50 };
10
11  // This is executed at compile-time.
12  const total = sum(&scores);
13
14  export fn demo() u32 {
15      return total;
16  }
```

Zig

zig 0.10.0 (Editor #1)

zig 0.10.0 Compiler options

```
1   demo:
2           push    rbp
3           mov     rbp, rsp
4           mov     eax, 150
5           pop     rbp
6           ret
7
8   example.sum:
9           push    rbp
10          mov     rbp, rsp
11          sub     rsp, 64
12          mov     qword ptr [rbp - 56], rsi
13          mov     qword ptr [rbp - 48], rdi
14          mov     dword ptr [rbp - 36], 0
15          mov     qword ptr [rbp - 32], 0
```

Generics ...

- Defined by a function that
  - takes compile-time known types as arguments
  - returns a struct definition
- Zig compiler generates a different version for every combination of types actually passed

... Generics


```zig
const std = @import("std");

fn makeNode(comptime T: type) type {
    return struct {
        const Self = @This(); // reference to containing struct

        // left and right are optional pointers to
        // another instance of this struct type.
        left: ?*Self,
        right: ?*Self,
        value: T,

        fn init(value: T) Self {
            return Self{
                .left = null,
                .right = null,
                .value = value,
            };
        }

        pub fn depthFirstPrint(self: *Self, indent: u8) void {
            // implementation omitted
        }
    };
}
```

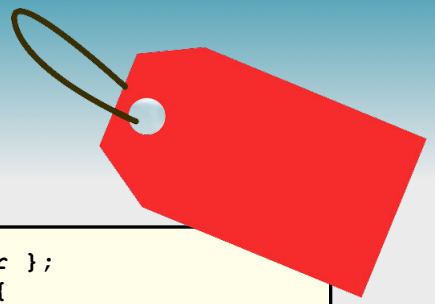
```zig
const Dog = struct {
    name: []const u8,
    breed: []const u8,
    age: u8,
};

fn treeOfDogs() void {
    const Node = makeNode(Dog);

    var node1 = Node.init(Dog{
        .name = "Maisey",
        .breed = "Treeing Walker Coonhound",
        .age = 3,
    });

    var node2 = Node.init(Dog{
        .name = "Ramsay",
        .breed = "Native American Indian Dog",
        .age = 3,
    });

    node1.left = &node2; // assigning a pointer

    node1.depthFirstPrint(0);
}
```

Bare Union

- Defines set of fields a value can have where only one field is active at a time
- Each field can have a different type

```
const Identifier = union {
    name: []const u8,
    number: i32,
};

const id1 = Identifier{ .name = "top secret" };
const id2 = Identifier{ .number = 1234 };
```

Tagged Union



- Adds use of enum that lists possible field names in **`union`**
- Allows **`union`** instances to be used in a **`switch`** statement
- To get **`enum`** value for a given tagged **`union`** instance call `std.meta.Tag(union_instance)`

```
const IdentifierTag = enum { name, number };
const Identifier = union(IdentifierTag) {
    name: []const u8,
    number: i32,
};

const ids = [_]Identifier{
    .{ .number = 1234 },
    .{ .name = "top secret" },
};

for (ids) |id| {
    switch (id) {
        .name => |name| try expectEqual(name, "top secret");
        .number => |number| try expectEqual(number, 1234),
    }
}

try expectEqual(
    std.meta.activeTag(ids[0]),
    IdentifierTag.number
);

try expectEqual(
    std.meta.activeTag(ids[1]),
    IdentifierTag.name
);
```

Inferred Enum Union

- Simpler alternative to tagged **`union`**
- Useful when a separate **`enum`** is not needed for other purposes

only difference from previous slide

```
const Identifier = union(enum) {
    name: []const u8,
    number: i32,
};

const ids = [_]Identifier{
    Identifier{ .number = 1234 },
    Identifier{ .name = "top secret" },
};

for (ids) |id| {
    switch (id) {
        .name => |name| try expectEqual(name, "top secret"),
        .number => |number| try expectEqual(number, 1234),
    }
}
```

inline Keyword

- Applied to functions to place code inline at all call sites
- Applied to **`for`** and **`while`** loops to unroll them

Tuples

- Anonymous structs without specified field names
- Field names default to indexes starting from zero
- Field values can be of different types
- Field values are accessed with **`my_tuple[index]`**
- Casting literal numeric values to specific types is optional

```
const my_tuple = .{ true, @as(u8, 19), @as(f32, 3.14), 'A', "hello" };

try expectEqual(my_tuple.len, 5);
try expectEqual(tuple[0], true);
try expectEqual(tuple[1], 19);
try expectEqual(tuple[2], 3.14);
try expectEqual(tuple[3], 'A');
try expectEqual(tuple[4], "hello");
```

type is **u8**

for loops require all values to have same type. When they do not, use **inline for** instead.

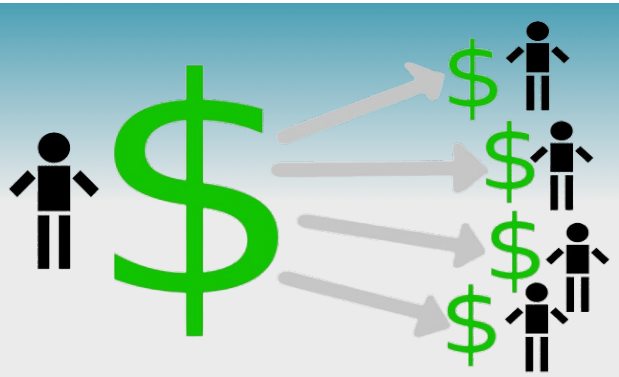
```
inline for (my_tuple) |value| {
    const T = @TypeOf(value);
    std.debug.print(
        "type of {any} is {}\n",
        .{ value, T }
    );
}
```

Standard Library Namespaces

Each defines types, constants, and functions.

- array_hash_map
- **ascii**
- atomic
- base64
- bit_set
- builtin
- c
- coff
- compress
- comptime_string_map
- crypto
- caster
- **debug**
- dwarf
- elf
- enums
- event
- fifo
- **fmt**
- **fs**
- hash
- **hash_map**
- **heap**
- **http**
- **io**
- **json**
- leb
- **log**
- macho
- **math**
- **mem**
- **meta**
- **net**
- **options**
- **os**
- packed_int_array
- pdb
- **process**
- **rand**
- **simd**
- **sort**
- start
- tar
- **testing**
- **time**
- **tz**
- **unicode**
- valgrind
- was
- zig

Allocators



- All memory allocation is done through allocators
- There are many provided allocators that each use a different memory management strategy
- New allocators can be defined to implement custom memory management strategies
- A program can use any number of allocators for different groups of memory allocations
  - provides more flexibility than languages that only have one memory allocation strategy or only support selecting one for an entire program
- For guidelines on selecting an allocator, see https://ziglang.org/documentation/master/#Choosing-an-Allocator

Standard Library Allocators

- **`std.heap.ArenaAllocator`**
  - uses an “arena” to handle freeing memory of everything allocated by it when it goes out of scope
  - allows allocating memory for many things that don’t need to be individually freed
- **`std.heap.FixedBufferAllocator`**
  - allocates memory from a fixed size buffer which avoids allocating memory at run-time
  - requires determining maximum amount of memory needed at compile-time
  - if more than that amount is requested, an **`OutOfMemory`** error occurs
- **`std.heap.GeneralPurposeAllocator`**
  - configurable allocator that can detect certain errors while using heap memory
- **`std.heap.LoggingAllocator`**
  - wraps another allocator and logs all the allocations and frees for debugging purposes
- **`std.heap.LogToWriterAllocator`**
  - similar to **`std.heap.LoggingAllocator`**, but allows specifying where log messages are written (such as a file)
- **`std.heap.MemoryPool`**
  - allocates memory for only one type and is very fast
  - for allocating a large number of instances of one type
- **`std.heap.page_allocator`**
  - allocates memory in chunks of OS page size
- **`std.testing.allocator`**
  - only used inside test blocks
  - detects memory leaks

and many more!

Generic Collections

- The standard library provides many generic collections
  - **`ArrayList`**: “a contiguous, growable list of items in memory”
  - **`MultiArrayList`**: elements are structs or unions;
    each field is stored in a separate array
  - **`AutoHashMap`** and **`StringHashMap`**: collections of key/value pairs
  - **`BufSet`** and **`EnumSet`**: sets of strings and enum values
  - **`SinglyLinkedList`**: linked list where nodes have pointers to next
  - **`DoublyLinkedList`**: linked list where nodes have pointers to prev and next
  - and many more

defer Keyword

- **`defer`** keyword specifies an expression to evaluate when containing block exits
  - often used to deallocate memory allocated on preceding line or perform another kind of cleanup
  - cannot use **`return`** keyword
- Many struct types define **`init`** and **`deinit`** methods
  - always called explicitly, never implicitly
  - can be given any names, but those names are used by convention

```
var allocator = std.heap.page_allocator;
var myList = std.ArrayList(10).init(allocator);
defer myList.deinit();
```

more on allocators and **`ArrayList`** later

Keeping code that allocates and frees memory together is **less error-prone** than allocating memory, writing a bunch of code that uses it, and having to remember to free it after all that code.

Using **`defer`** is better than placing cleanup code at the end of a function because it may have multiple ways to exit.

ArrayList

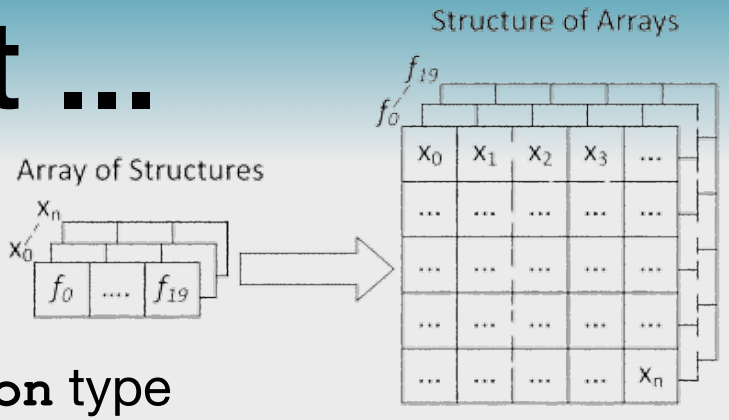
- Instances have
  - fields **`items`**, **`capacity`**, and **`allocator`**
  - methods **`append`**, **`appendSlice`**, **`clearAndFree`**, **`clone`**, **`deinit`**, **`getLast`**, **`getLastOrNull`**, **`init`**, **`initCapacity`**, **`insert`**, **`insertSlice`**, **`orderedRemove`**, **`pop`**, **`popOrNull`**, **`replaceRange`**, **`writer`**, and many more

```
const std = @import("std");
const print = std.debug.print;
const String = []const u8;

var list = std.ArrayList(String).init(allocator);
defer list.deinit();

try list.append("red");
try list.appendSlice(&[_]String{ "green", "blue" });
print("{d}\n", .{list.items.len}); // 3

for (list.items) |value| {
    print("{s}\n", .{value});
}
```

need to set this

```
// Beginning with a specified capacity
// can return an error.
var list =
    try std.ArrayList(String).
        initCapacity(allocator, 500);
```

MultiArrayList ...



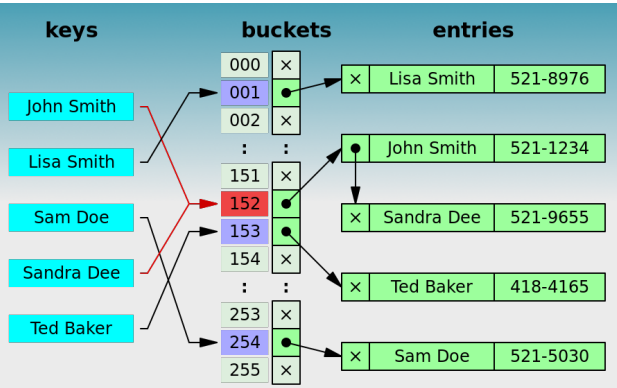
- Similar to **`ArrayList`** in that it stores a sequence of elements
- Elements must be instances of a **`struct`** or **`union`** type
- Each field is stored in a separate array (not a vector)
  - easy to obtain a slice containing all values for a given field
  - slice can used to create a vector which supports SIMD operations
- Instances have
  - fields **`bytes`**, **`let`**, and **`capacity`**
  - methods **`append`**, **`clone`**, **`deinit`**, **`get`**, **`insert`**, **`items`**, **`orderedRemove`**, **`pop`**, **`popOrNull`**, **`set`**, **`slice`**, and many more

... MultiArrayList

```
const std = @import("std");
const allocator = std.testing.allocator;
const expectEqual = std.testing.expectEqual;

const Range = struct {
    min: f32,
    max: f32,
    current: f32,
};

test "MultiArrayList" {
    var list = std.MultiArrayList(Range){};
    defer list.deinit(allocator);

    const r1 = Range{ .min = 0, .max = 100, .current = 50 };
    try list.append(allocator, r1);

    try list.append(allocator, Range{ .min = 10, .max = 50, .current = 25 });

    // The "items" method gets a slice of values for a given field.
    const currents: []f32 = list.items(.current);

    const vector: @Vector(3, f32) = currents[0..3].*;
    const sum = @reduce(.Add, vector);
    try expectEqual(sum, 75);
}
```

Unlike **`ArrayList`** instances, **`MultiArrayList`** instances do not store an allocator in order to optimize memory used. This is why an allocator must be passed to methods like **`append`** and **`insert`**.

HashMap ...



- Instances have
  - fields **`items`**, **`capacity`**, and **`allocator`**
  - methods **`append`**, **`appendSlice`**, **`clearAndFree`**, **`clone`**, **`deinit`**, **`getLast`**, **`getLastOrNull`**, **`init`**, **`initCapacity`**, **`insert`**, **`insertSlice`**, **`orderedRemove`**, **`pop`**, **`popOrNull`**, **`replaceRange`**, **`writer`**, and many more

... HashMap

Called “auto” because it automatically provides a hashing function for most key types.

```
const std = @import("std");
const print = std.debug.print;
const allocator = std.testing.allocator;
const expect = std.testing.expect;
const expectEqual =
    std.testing.expectEqual;
const expectEqualStrings =
    std.testing.expectEqualStrings;
const String = []const u8;
```

```
test "AutoHashMap" {
    var map = std.AutoHashMap(u8, String).init(allocator);
    defer map.deinit();

    try map.put(99, "Gretzky");
    try map.put(4, "Orr");
    try map.put(19, "Ratelle");
    try expectEqual(map.count(), 3);

    var iter2 = map.keyIterator();
    while (iter2.next()) |key| {
        const number = key.*;
        if (map.get(number)) |name| {
            print("{s} number is {d}.\n", .{ name, number });
        }
    }

    try expect(map.contains(99));

    // The get method returns an optional value.
    var name = map.get(99) orelse "";
    try expectEqualStrings("Gretzky", name);

    const removed = map.remove(99); // returns bool
    try expect(removed);
    try expectEqual(@as(?String, null), map.get(99));
}
```

Reflection

- Several builtin functions support compile-time reflection
  - **`@hasDecl`** determines if a **`struct`** contains a declaration with a given name
  - **`@hasField`** determines if a **`struct`** contains a field with a given name
  - **`@This`** when inside a **`struct`** definition, returns its type
  - **`@TypeOf`** returns the type of a given value or the common type of a list of values
  - **`@typeInfo`** returns a tagged union that describes a type
  - **`@typeName`** returns the name of a given type as a string
  - **`std.meta.fields`** returns information about fields in a **`struct`**
  - **`std.meta.hasFn`** determines if a **`struct`** contains a function with a given name
- For more see https://github.com/wrongnull/zigtrait

Functional Programming

- Zig is not a functional programming language
- Zig standard library collection types
  - do not provide methods like **`map`**, **`filter`**, and **`reduce`**
  - can be implemented, but not preferred due to excessive memory allocation

```
const fruits = [_]Fruit{
    .{ .name = "apple", .color = "red", .price = 1.5 },
    .{ .name = "banana", .color = "yellow", .price = 0.25 },
    .{ .name = "orange", .color = "orange", .price = 0.75 },
    .{ .name = "cherry", .color = "red", .price = 3.0 },
};
```

We can implement support for this style.

```
const redTotal = fruits
    .filter(isRed)
    .map(f32, PriceCollection, getPrice)
    .reduce(f32, add, 0.0);
```

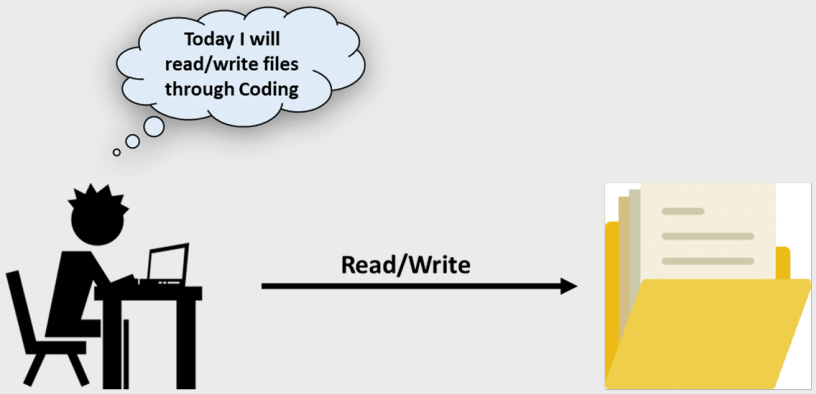
The `filter` and `map` methods must allocate space for new `Fruit` instances.

This style uses less memory and requires less code.

```
var redTotal: f32 = 0.0;
for (fruits) |fruit| {
    if (std.mem.eql(u8, fruit.color, "red")) {
        redTotal += fruit.price;
    }
}
```

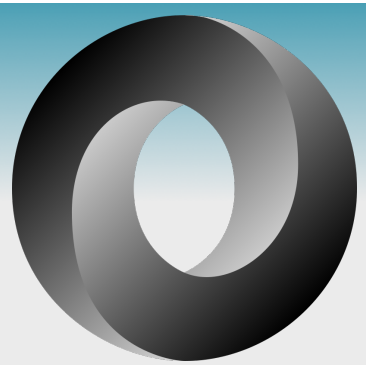
Writing and Reading Files

- **`std.fs`** and **`std.os`** namespaces define many I/O functions




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

// Creates and writes a file in the current working directory.
fn writeFile() !void {
    const dir = std.fs.cwd();
    const file = try dir.createFile("data.txt", .{});
    defer file.close();

    try file.writeAll("Hello, World!");
}

// Reads a file in the current working directory.
fn readFile() !void {
    const dir = std.fs.cwd();
    const file = try dir.openFile("data.txt", .{});
    defer file.close();

    var buffer: [100]u8 = undefined;
    const length = try file.readAll(&buffer);
    print("read {} bytes\n", .{length}); // 13
    const content = buffer[0..length];
    print("{s}\n", .{content}); // Hello, World!
}

pub fn main() !void {
    try writeFile();
    try readFile();
}
```

JSON



- **`std.json`** namespace provides functions for generating and parsing JSON strings

```
const std = @import("std");
const my_allocator = std.testing.allocator;
const expectEqual = std.testing.expectEqual;
const String = []const u8;

const Place = struct {
    lat: f32,
    long: f32,
};
```

```
fn fromJSON(
    T: anytype,
    allocator: std.mem.Allocator,
    json: String,
) !T {
    const parsed =
        try std.json.parseFromSlice(
            T, allocator, json, .{}
        );
    defer parsed.deinit();
    return parsed.value;
}
```

```
fn toJSON(
    allocator: std.mem.Allocator,
    value: anytype,
) !String {
    // The ArrayList will grow as needed.
    var out = std.ArrayList(u8).init(allocator);
    defer out.deinit();
    try std.json.stringify(value, .{}, out.writer());
    return try out.toOwnedSlice(); // empties ArrayList
}
```

```
test "json" {
    const place1 = Place{
        .lat = 51.997664,
        .long = -0.740687,
    };

    const json = try toJSON(my_allocator, place1);
    defer my_allocator.free(json);

    const place2 = try fromJSON(Place, my_allocator, json);
    try expectEqual(place1, place2);
}
```

Resources

- **My blog** - https://mvolkmann.github.io/blog/ (select Zig)
- **Zig home page** - https://ziglang.org/
- **My Zig example code** - https://github.com/mvolkmann/zig-examples/
- **Zig Showtime Youtube videos** - https://zig.show/
- **Ziglings exercises** - https://codeberg.org/ziglings/exercises/
- **Zig Discord server** - https://discord.com/servers/zig-programming-language-605571803288698900
- **Zig News** - https://zig.news/
- **Awesome Zig** - https://github.com/nrdmn/awesome-zig

Wrap Up

- Zig is a great option for apps that care deeply about performance, memory usage, or binary size
- Not yet 1.0, but already used by serious projects like Bun
- Has great interoperability with C and C++