Zig Cheat Sheet

Table of Contents

Introduction

Zig is a general-purpose programming language designed for robustness, optimality, and maintainability. It's statically typed and provides low-level control with high-level features.

Installation

To install Zig, follow these steps:

1. Visit the official Zig website: https://ziglang.org/
2. Go to the "Download" section
3. Choose the appropriate version for your operating system
4. Extract the downloaded archive
5. Add the Zig binary to your system's PATH

Alternatively, you can use package managers:

- On macOS with Homebrew:

  ```
  brew install zig
  ```

- On Linux with your package manager (e.g., for Ubuntu):

  ```
  sudo apt-get install zig
  ```

Verify the installation by running:

```
zig version
```

Hello, World!

Let's start with a simple "Hello, World!" program in Zig:

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

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

Save this in a file named `hello.zig` and run it with:

```
zig run hello.zig
```

Basic Syntax

Zig's syntax is designed to be clear and unambiguous. Here are some key points:

- Statements don't need semicolons at the end

- Blocks are denoted by `{}`
- Comments start with `//` for single-line and `//\\` for multi-line
- Variables are declared with `var` (mutable) or `const` (immutable)

Example:

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

pub fn main() void {
    // This is a comment
    const x = 5; // Immutable
    var y = 10; // Mutable

    //\\ This is a
    multi-line comment //\\

    std.debug.print("x = {}, y = {}\n", .{x, y});
}
```

Variables and Data Types

Zig has several built-in types:

- Integers: `i8`, `u8`, `i16`, `u16`, `i32`, `u32`, `i64`, `u64`, `i128`, `u128`
- Floating-point: `f16`, `f32`, `f64`, `f128`
- Boolean: `bool`
- Null: `null`
- Undefined: `undefined`

Examples:

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

pub fn main() void {
    const a: i32 = 42;
    var b: f64 = 3.14;
    const c: bool = true;
    var d: ?i32 = null; // Optional type

    std.debug.print("a = {}, b = {}, c = {}, d = {?}\n", .{a, b, c, d});
}
```

Control Flow

Zig provides familiar control flow structures:

If statement

```
const x = 10;
if (x > 5) {
    std.debug.print("x is greater than 5\n", .{});
} else if (x == 5) {
    std.debug.print("x is equal to 5\n", .{});
} else {
    std.debug.print("x is less than 5\n", .{});
}
```

While loop

```
var i: u32 = 0;
while (i < 5) : (i += 1) {
    std.debug.print("{} ", .{i});
}
// Prints: 0 1 2 3 4
```

For loop

```
const items = [_]i32{ 4, 5, 6 };
for (items) |item, index| {
    std.debug.print("items[{}] = {}\n", .{index, item});
}
```

Functions

Functions in Zig are defined using the `fn` keyword:

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

pub fn main() void {
    const result = add(5, 3);
    std.debug.print("5 + 3 = {}\n", .{result});
}
```

Error Handling

Zig uses a unique error handling approach:

```
const FileOpenError = error{
    AccessDenied,
    OutOfMemory,
    FileNotFound,
};

fn openFile(filename: []const u8) FileOpenError!File {
    if (outOfMemory()) return FileOpenError.OutOfMemory;
    if (accessDenied()) return FileOpenError.AccessDenied;
    if (fileNotFound()) return FileOpenError.FileNotFound;
    return File{};
}

pub fn main() void {
    const file = openFile("test.txt") catch |err| {
        std.debug.print("Error: {}\n", .{err});
        return;
    };
    // Use file...
}
```

Memory Management

Zig gives you fine-grained control over memory:

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

pub fn main() void {
    var general_purpose_allocator = std.heap.GeneralPurposeAllocator(.{}){};
    const gpa = general_purpose_allocator.allocator();

    const bytes = gpa.alloc(u8, 100) catch |err| {
        std.debug.print("Failed to allocate memory: {}\n", .{err});
        return;
    };
    defer gpa.free(bytes);

    // Use bytes...
}
```

Structs and Enums

Structs and enums are key to organizing data in Zig:

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

const Color = enum {
    Red,
    Green,
    Blue,
};

const Person = struct {
    name: []const u8,
    age: u32,
    favorite_color: Color,

    fn introduce(self: Person) void {
        std.debug.print("Hi, I'm {s}, I'm {} years old, and my favorite color is {}\n", .{
            self.name, self.age, self.favorite_color,
        });
    }
};

pub fn main() void {
    const bob = Person{
        .name = "Bob",
        .age = 30,
        .favorite_color = Color.Blue,
    };
    bob.introduce();
}
```

Pointers and Slices

Zig provides low-level control with pointers and high-level convenience with slices:

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

pub fn main() void {
    var x: i32 = 42;
    var y: *i32 = &x; // Pointer to x

    std.debug.print("x = {}, *y = {}\n", .{x, y.*});

    var arr = [_]i32{ 1, 2, 3, 4, 5 };
    var slice = arr[1..4]; // Slice of arr from index 1 to 3

    for (slice) |item| {
        std.debug.print("{} ", .{item});
    }
    // Prints: 2 3 4
}
```

Packages and Imports

Zig uses a straightforward module system:

```
// In math.zig
pub fn add(a: i32, b: i32) i32 {
    return a + b;
}

// In main.zig
const std = @import("std");
const math = @import("math.zig");

pub fn main() void {
    const result = math.add(5, 3);
    std.debug.print("5 + 3 = {}\n", .{result});
}
```

Testing

Zig has built-in support for testing:

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

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

test "basic addition" {
    try expect(add(3, 4) == 7);
}

test "negative numbers" {
    try expect(add(-1, -1) == -2);
}
```

Run tests with:

```
zig test your_file.zig
```

Concurrency

Zig provides low-level primitives for concurrency:

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

fn printNumbersThread(context: void) void {
    for (0..5) |i| {
        std.debug.print("Thread: {}\n", .{i});
        std.time.sleep(1 * std.time.ns_per_s);
    }
}

pub fn main() !void {
    var thread = try std.Thread.spawn(.{}, printNumbersThread, .{});
    for (0..5) |i| {
        std.debug.print("Main: {}\n", .{i});
        std.time.sleep(1 * std.time.ns_per_s);
    }
    thread.join();
}
```

Interoperability with C

Zig can easily interoperate with C code:

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

pub fn main() void {
    _ = c.printf("Hello from C!\n");
}
```

Compile with:

```
zig build-exe your_file.zig -lc
```

Build System

Zig comes with its own build system. Here's a simple `build.zig`:

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

pub fn build(b: *std.Build) void {
    const target = b.standardTargetOptions(.{});
    const optimize = b.standardOptimizeOption(.{});

    const exe = b.addExecutable(.{
        .name = "my-project",
        .root_source_file = .{ .path = "src/main.zig" },
        .target = target,
        .optimize = optimize,
    });

    b.installArtifact(exe);

    const run_cmd = b.addRunArtifact(exe);
    run_cmd.step.dependOn(b.getInstallStep());

    const run_step = b.step("run", "Run the app");
    run_step.dependOn(&run_cmd.step);
}
```

Build and run with:

```
zig build run
```

Useful Resources

- Official Zig Documentation
- Zig Learn
- Zig Standard Library Documentation

Remember, Zig is a rapidly evolving language, so always refer to the latest official documentation for the most up-to-date information.