Zig Programming Language Cheatsheet

Updated on 6 January 2024 by Huzaif Sayyed

Zig is a statically-typed programming language designed for performance, safety, and simplicity. Here's a **Zig Programming Language Cheatsheet** to help you get started.

Links

Zig Official Website

Zig Playground

Zig Github

Zig Infographic

Hello World! Zig Program

Create a file called `main.zig`

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

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

Use `zig run main.zig` to build and run it. In this example will give output: Hello, World!

Basics

Variables

Mutable Variable

```
var x: i32 = 42;
```

Immutable Variable

```
const y: i32 = 123;
```

Type Inference

```
const message = "Hello";
```

Use var for mutable and const for immutable. Types can be explicit or inferred.

Arrays

```
const a = [5]u8{ 'h', 'e', 'l', 'l', 'o' };
const b = [_]u8{ 'w', 'o', 'r', 'l', 'd' };
```

To get the size of an array, simply access the array's len field.

```
const array = [_]u8{ 'h', 'e', 'l', 'l', 'o' };
const length = array.len; // 5
```

Data Type Conversion

```
const integerResult: i32 = @intCast(i64, 42);
const floatResult: f64 = @floatCast(f32, 3.14);

const intToFloat: f64 = @intToFloat(f64, 42);
const floatToInt: i32 = @floatToInt(i32, 3.14);

const charToInt: i32 = @intCast(i32, 'A');
const intToChar: char = @intCast(char, 65);
```

Use `@intCast` and `@floatCast` for numeric conversions, and `@intToFloat`, `@floatToInt` for broader numeric transformations.

Data Types

Integers

```
const integer: i32 = 42;
```

Floating Point

```
const floatingPoint: f64 = 3.14;
```

Boolean

```
const flag: bool = true;
```

Characters

```
const charVar: char = 'A';
```

Strings

```
const stringVar: [5]u8 = "Hello";
```

Loops and Control Statements

While Loop

```
var x: i32 = 0;
while x < 5 {
    // code
    x += 1;
}
```

For Loop

For Loop (Array Iteration)

```
const numbers = [1, 2, 3, 4, 5];
for (numbers) |value| {
    // code
}
```

For Loop (Range Iteration)

```
for (1 .. 5) |i| {
    // code
}
```

If Statement

```
if condition {
    // code
} else if anotherCondition {
    // code
} else {
    // code
}
```

Switch Statement

```
const result = someFunction();
switch (result) {
    Error => |err| {
        // code for error case
    },
    Ok => |value| {
        // code for success case
    },
}
```

Ternary Operator

```
const result = condition ? valueIfTrue : valueIfFalse;
```

Defer Statement

```
defer { // code }
```

Defer is used to execute a statement while exiting the current block.

Functions

Function Declaration

```
fn add(x: i32, y: i32) i32 {
    return x + y;
}
```

Function Invocation

```
const result = add(3, 4);
```

Function Parameters

```
fn greet(name: []const u8) void {
    std.debug.print("Hello, {}!\n", .{name});
}

greet("Alice");
```

Return Values

```
fn multiply(x: i32, y: i32) i32 {
    const result = x * y;
    return result;
}
```

Void Functions

```
fn printMessage() void {
    std.debug.print("This is a message.\n", .{});
}
```

Pointers

Creating Pointers

```
const x: i32 = 42;
const ptrToX: *i32 = &x;
```

Dereferencing Pointers

```
const valueAtPtr: i32 = *ptrToX;
```

Pointer Arithmetic

```
const array: [5]i32 = undefined;
const ptrToArray: *i32 = array.ptr;

const secondElement: i32 = *(ptrToArray + 1);
```

Pointers in Structs

```
const Point = struct {
    x: f64,
    y: f64,
};

const point: Point = .{ .x = 1.0, .y = 2.0 };
const ptrToPoint: *Point = &point;
```

Null Pointers

```
const nullPtr: ?i32 = null;
```

Error Handling

Result Type

```
const ResultType = struct {
    Error,
    Ok,
};

const divide = fn(x: i32, y: i32) ResultType {
    if (y == 0) {
        return ResultType.Error;
    }
    return ResultType.Ok;
}

const result = divide(10, 0);
switch (result) {
    ResultType.Error => |err| {
        // Handle error
    },
    ResultType.Ok => |value| {
        // Handle success
    },
}
```

Option Type

```
const OptionType = struct {
    Some,
    None,
};

const findElement = fn(arr: []i32, target: i32) OptionTyp
    for (arr) |element| {
        if (element == target) {
            return OptionType.Some;
        }
    }
    return OptionType.None;
}

const option = findElement([1, 2, 3], 2);
switch (option) {
    OptionType.Some => {
        // Element found
    },
    OptionType.None => {
        // Element not found
    },
}
```

Custom Error Type

```
const MyError = struct {
    message: []const u8,
};

fn throwError() MyError {
    return MyError{ .message = "Something went wrong" };
}

const result = throwError();
if (result) |err| {
    std.debug.print("Error: {}\n", .{err.message});
}
```

Memory Management

```
const allocator = std.heap.page_allocator;

// Allocating Memory
const myBuffer = allocator.alloc(u8, 1024) catch unreachable;

// Using Allocated Memory
myBuffer[0] = 42;

// Freeing Memory
allocator.free(myBuffer);
```

Concurrency (Async/Await)

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

// Async Function
async fn asyncTask() !void {
    // Asynchronous code
}

// Async Main Function
pub fn main() void {
    const asyncMain = asyncTask();
    const result = asyncMain.await() catch unreachable;

    std.debug.print("Result: {}\n", .{result});
}
```

Zig provides support for asynchronous programming with async/await syntax, allowing developers to write concurrent code more easily.

Congratulations on reaching the end of this Zig Programming Language Cheatsheet! This resource is designed to make your coding experience easy and efficient. Feel free to bookmark this page or download the PDF for future reference. Happy Zig Programming!