Zig Programming Language: Beginner to Expert

Table of Contents

Introduction to Zig
Zig is a modern programming language designed for robustness, performance, and clarity. It is particularly suited for systems programming, competing with languages like C and Rust. Zig emphasizes simplicity and gives developers fine-grained control over low-level operations.

Key Features:
- Manual memory management with safety features.
- Cross-compilation capabilities.
- No hidden control flow or memory allocations.
- Compile-time code execution and reflection.

Installing Zig
Downloading Zig
1. Visit the official [Zig download page](https://ziglang.org/download/).
2. Download the binary for your operating system.
3. Extract the archive and add Zig to your system's PATH.

Verifying Installation
Run the following command to verify Zig is installed:
```bash
zig version
```

Hello, World!
The canonical first program in Zig:

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

pub fn main() !void {
```

    const stdout = std.io.getStdOut().writer();
    try stdout.print("Hello, World!\n", .{});
}
~~~
### Explanation:
- `@import("std")`: Imports Zig's standard library.
- `main`: The entry point of the program.
- `try`: Handles errors gracefully by propagating them.
- `print`: Outputs text to the standard output.

### Compiling and Running
```bash
zig build-exe hello.zig
./hello
~~~

Basic Syntax and Data Types

Variables and Constants
```zig
const x = 10; // Immutable
var y = 20;   // Mutable
y += 5;
```

Data Types
- `i32`, `u32`: Signed and unsigned integers.
- `f64`: 64-bit floating point.
- `bool`: Boolean type.
- `[]u8`: Arrays of unsigned 8-bit integers.

Type Inference
```zig
const z = 42; // Type inferred as i32
```

Control Flow

If-Else
```zig
const age = 25;
if (age >= 18) {
    std.debug.print("Adult\n", .{});
} else {
    std.debug.print("Minor\n", .{});
}
```

Loops
- **While Loop:**
    ```zig
    var i = 0;
    while (i < 10) : (i += 1) {
    ```

```zig
        std.debug.print("{}\n", .{i});
    }
```

- **For Loop:**
    ```zig
    const arr = [_]i32{1, 2, 3};
    for (arr) |item| {
        std.debug.print("{}\n", .{item});
    }
    ```

Switch
```zig
const value = 2;
switch (value) {
    1 => std.debug.print("One\n", .{}),
    2 => std.debug.print("Two\n", .{}),
    else => std.debug.print("Other\n", .{}),
}
```

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

pub fn main() void {
    const result = add(5, 7);
    std.debug.print("Result: {}\n", .{result});
}
```

Inline Functions
```zig
inline fn multiply(a: i32, b: i32) i32 {
    return a * b;
}
```

Working with Arrays and Strings

Arrays
```zig
const arr = [_]i32{1, 2, 3};
const first = arr[0];
```

Strings
```zig
const name = "Zig";
std.debug.print("Hello, {}!\n", .{name});
```

Error Handling
Zig uses error unions for error handling.
```zig
fn mightFail(flag: bool) !void {
    if (!flag) {
        return error.Failure;
    }
}

pub fn main() !void {
    try mightFail(true);
}
```

Memory Management

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

pub fn main() !void {
    const allocator = std.heap.page_allocator;
    const ptr = try allocator.alloc(u8, 10);
    defer allocator.free(ptr);
}
```

Using Zig for Systems Programming

Direct Memory Access
```zig
const std = @import("std");

pub fn main() void {
    var buffer: [10]u8 = undefined;
    buffer[0] = 42;
    std.debug.print("Value: {}\n", .{buffer[0]});
}
```

Advanced Features

Generics
```zig
fn identity(comptime T: type, value: T) T {
    return value;
}
```

Compile-time Code Execution
```zig
const factorial = comptime fn(n: u32) u32 {
    return if (n == 0) 1 else n * factorial(n - 1);
};
```

Building and Linking Libraries
```bash
zig build-lib -dynamic mylib.zig
```

Zig Build System

`build.zig` Example
```zig
const std = @import("std");

pub fn build(b: *std.build.Builder) void {
    const exe = b.addExecutable("myapp", "src/main.zig");
    exe.install();
    b.default_step.dependOn(&exe.step);
}
```

Best Practices
1. Use `const` wherever possible.
2. Handle errors explicitly.
3. Prefer small, composable functions.
4. Leverage compile-time features.

Conclusion and Next Steps

You've now explored Zig from the basics to advanced features. To deepen your understanding:
- Explore Zig's [documentation](https://ziglang.org/documentation/).
- Contribute to open-source Zig projects.
- Experiment with building your own libraries and tools.