← Back to the **Learn** section

In-depth Overview

Feature Highlights

Small, simple language

Focus on debugging your application rather than debugging your programming language knowledge.

Zig's entire syntax is specified with a 500-line PEG grammar file.

There is **no hidden control flow**, no hidden memory allocations, no preprocessor, and no macros. If Zig code doesn't look like it's jumping away to call a function, then it isn't. This means you can be sure that the following code calls only `foo()` and then `bar()`, and this is guaranteed without needing to know the types of anything:

```
var a = b + c.d;
foo();
bar();
```

Examples of hidden control flow:

- D has `@property` functions, which are methods that you call with what looks like field access, so in the above example, `c.d` might call a function.
- C++, D, and Rust have operator overloading, so the + operator might call a function.
- C++, D, and Go have throw/catch exceptions, so `foo()` might throw an exception, and prevent `bar()` from being called.

Zig promotes code maintenance and readability by making all control flow managed exclusively with language keywords and function calls.

Performance and Safety: Choose Two

Zig has four build modes, and they can all be mixed and matched all the way down to scope granularity.

Parameter	Debug	ReleaseSafe	ReleaseFast	ReleaseSmall
Optimizations - improve speed, harm debugging, harm compile time		-O3	-O3	-Os
Runtime Safety Checks - harm speed, harm size, crash instead of undefined behavior	On	On		

Here is what Integer Overflow looks like at compile time, regardless of the build mode:

test.zig

```
test "integer overflow at compile time" {
    const x: u8 = 255;
    _ = x + 1;
}
```

```
$ zig test test.zig
doctest-29df0a60/test.zig:3:11: error: overflow of integer type 'u8' with value '256'
    _ = x + 1;
        ~~^~~
```

Here is what it looks like at runtime, in safety-checked builds:

test.zig

```
test "integer overflow at runtime" {
    var x: u8 = 255;
    x += 1;
}
```

```
$ zig test test.zig
1/1 test.integer overflow at runtime... thread 2849 panic: integer overflow
/home/runner/work/www.ziglang.org/www.ziglang.org/doctest-d8c23029/test.zig:3:7: 0x102898e in test.integ
    x += 1;
      ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/test_runner.zig:181:28: 0x10335d0 in mainTerm
        } else test_fn.func();
                           ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/test_runner.zig:36:28: 0x10299ba in main (test
        return mainTerminal();
                           ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/start.zig:575:22: 0x1028ecc in posixCallMa
            root.main();
                     ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/start.zig:253:5: 0x1028a21 in _start (test
```

```
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/runner/.cache/zig/o/0580d7dc1520c628666630a9376f6452/test
```

Those stack traces work on all targets, including freestanding .

With Zig one can rely on a safety-enabled build mode, and selectively disable safety at the performance bottlenecks. For example the previous example could be modified like this:

```
test "actually undefined behavior" {
    @setRuntimeSafety(false);
    var x: u8 = 255;
    x += 1; // XXX undefined behavior!
}
```

Zig uses undefined behavior as a razor sharp tool for both bug prevention and performance enhancement.

Speaking of performance, Zig is faster than C.

- The reference implementation uses LLVM as a backend for state of the art optimizations.
- What other projects call “Link Time Optimization” Zig does automatically.
- For native targets, advanced CPU features are enabled (-march=native), thanks to the fact that Cross-compiling is a first-class use case.
- Carefully chosen undefined behavior. For example, in Zig both signed and unsigned integers have undefined behavior on overflow, contrasted to only signed integers in C. This facilitates optimizations that are not available in C .
- Zig directly exposes a SIMD vector type, making it easy to write portable vectorized code.

Please note that Zig is not a fully safe language. For those interested in following Zig’s safety story, subscribe to these issues:

- enumerate all kinds of undefined behavior, even that which cannot be safety-checked
- make Debug and ReleaseSafe modes fully safe

Zig competes with C instead of depending on it

The Zig Standard Library integrates with libc, but does not depend on it. Here’s Hello World:

hello.zig

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

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

```
$ zig build-exe hello.zig
$ ./hello
Hello, world!
```

When compiled with `-O ReleaseSmall`, debug symbols stripped, single-threaded mode, this produces a 9.8 KiB static executable for the x86_64-linux target:

```
$ zig build-exe hello.zig -O ReleaseSmall -fstrip -fsingle-threaded
$ wc -c hello
9944 hello
$ ldd hello
  not a dynamic executable
```

A Windows build is even smaller, coming out to 4096 bytes:

```
$ zig build-exe hello.zig -O ReleaseSmall -fstrip -fsingle-threaded -target x86_64-windows
$ wc -c hello.exe
4096 hello.exe
$ file hello.exe
hello.exe: PE32+ executable (console) x86-64, for MS Windows
```

Order independent top level declarations

Top level declarations such as global variables are order-independent and lazily analyzed. The initialization values of global variables are evaluated at compile-time.

global_variables.zig

```
var y: i32 = add(10, x);
const x: i32 = add(12, 34);

test "global variables" {
    assert(x == 46);
    assert(y == 56);
}

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

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

```
$ zig test global_variables.zig
1/1 test.global variables... OK
All 1 tests passed.
```

Optional type instead of null pointers

In other programming languages, null references are the source of many runtime exceptions, and even stand accused of being the worst mistake of computer science .

Unadorned Zig pointers cannot be null:

```
test "null @intToPtr" {
    const foo: *i32 = @ptrFromInt(0x0);
    _ = foo;
}
```

```
$ zig test test.zig
doctest-8fe42237/test.zig:2:35: error: pointer type '*i32' does not allow address zero
    const foo: *i32 = @ptrFromInt(0x0);
                                  ^~~
```

However any type can be made into an optional type by prefixing it with ?:

optional_syntax.zig

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

test "null @intToPtr" {
    const ptr: ?*i32 = @ptrFromInt(0x0);
    assert(ptr == null);
}
```

```
$ zig test optional_syntax.zig
1/1 test.null @intToPtr... OK
All 1 tests passed.
```

To unwrap an optional value, one can use `orelse` to provide a default value:

```
// malloc prototype included for reference
extern fn malloc(size: size_t) ?*u8;

fn doAThing() ?*Foo {
    const ptr = malloc(1234) orelse return null;
    // ...
}
```

Another option is to use if:

```
fn doAThing(optional_foo: ?*Foo) void {
    // do some stuff

    if (optional_foo) |foo| {
        doSomethingWithFoo(foo);
    }

    // do some stuff
}
```

The same syntax works with while:

iterator.zig

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

pub fn main() void {
    const msg = "hello this is dog";
    var it = std.mem.tokenize(u8, msg, " ");
    while (it.next()) |item| {
        std.debug.print("{s}\n", .{item});
    }
}
```

```
$ zig build-exe iterator.zig
$ ./iterator
hello
this
is
dog
```

Manual memory management

A library written in Zig is eligible to be used anywhere:

- Desktop applications
- Low latency servers
- Operating System kernels
- Embedded devices
- Real-time software, e.g. live performances, airplanes, pacemakers
- In web browsers or other plugins with WebAssembly
- By other programming languages, using the C ABI

In order to accomplish this, Zig programmers must manage their own memory, and must handle memory allocation failure.

This is true of the Zig Standard Library as well. Any functions that need to allocate memory accept an allocator parameter. As a result, the Zig Standard Library can be used even for the freestanding target.

In addition to A fresh take on error handling, Zig provides defer and errdefer to make all resource management - not only memory - simple and easily verifiable.

For an example of `defer`, see Integration with C libraries without FFI/bindings. Here is an example of using `errdefer`:

```
const Device = struct {
    name: []u8,

    fn create(allocator: *Allocator, id: u32) !Device {
        const device = try allocator.create(Device);
        errdefer allocator.destroy(device);

        device.name = try std.fmt.allocPrint(allocator, "Device(id={d})", id);
        errdefer allocator.free(device.name);

        if (id == 0) return error.ReservedDeviceId;

        return device;
    }
};
```

A fresh take on error handling

Errors are values, and may not be ignored:

discard.zig

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

pub fn main() void {
    _ = std.fs.cwd().openFile("does_not_exist/foo.txt", .{});
}
```

```
$ zig build-exe discard.zig
doctest-16734c75/discard.zig:4:30: error: error is discarded
    _ = std.fs.cwd().openFile("does_not_exist/foo.txt", .{});
        ~~~~~~~~~~~~~~~~~~~~^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
doctest-16734c75/discard.zig:4:30: note: consider using 'try', 'catch', or 'if'
referenced by:
    callMain: zig/lib/std/start.zig:575:17
    initEventLoopAndCallMain: zig/lib/std/start.zig:519:34
    remaining reference traces hidden; use '-freference-trace' to see all reference traces
```

Errors can be handled with catch:

catch.zig

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

pub fn main() void {
    const file = std.fs.cwd().openFile("does_not_exist/foo.txt", .{}) catch |err| label: {
        std.debug.print("unable to open file: {}\n", .{err});
        const stderr = std.io.getStdErr();
        break :label stderr;
    };
    file.writeAll("all your codebase are belong to us\n") catch return;
}
```

```
$ zig build-exe catch.zig
$ ./catch
unable to open file: error.FileNotFound
all your codebase are belong to us
```

The keyword try is a shortcut for `catch |err| return err`:

try.zig

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

pub fn main() !void {
    const file = try std.fs.cwd().openFile("does_not_exist/foo.txt", .{});
    defer file.close();
    try file.writeAll("all your codebase are belong to us\n");
}
```

```
$ zig build-exe try.zig
$ ./try
error: FileNotFound
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/os.zig:1890:23: 0x1059e98 in openatZ (try)
            .NOENT => return error.FileNotFound,
                      ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/fs/Dir.zig:843:9: 0x102620d in openFileZ (
        try posix.openatZ(self.fd, sub_path, os_flags, 0);
        ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/fs/Dir.zig:775:5: 0x102370e in openFile (t
    return self.openFileZ(&path_c, flags);
    ^
/home/runner/work/www.ziglang.org/www.ziglang.org/doctest-66227843/try.zig:4:18: 0x102354d in main (try)
    const file = try std.fs.cwd().openFile("does_not_exist/foo.txt", .{});
                 ^
```

Note that is an Error Return Trace, not a stack trace. The code did not pay the price of unwinding the stack to come up with that trace.

The switch keyword used on an error ensures that all possible errors are handled:

test.zig

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

test "switch on error" {
    _ = parseInt("hi", 10) catch |err| switch (err) {};
}

fn parseInt(buf: []const u8, radix: u8) !u64 {
    var x: u64 = 0;

    for (buf) |c| {
        const digit = try charToDigit(c);

        if (digit >= radix) {
            return error.DigitExceedsRadix;
        }

        x = try std.math.mul(u64, x, radix);
        x = try std.math.add(u64, x, digit);
    }

    return x;
}
```

```
fn charToDigit(c: u8) !u8 {
    const value = switch (c) {
        '0'...'9' => c - '0',
        'A'...'Z' => c - 'A' + 10,
        'a'...'z' => c - 'a' + 10,
        else => return error.InvalidCharacter,
    };

    return value;
}


$ zig test test.zig
doctest-392cb4a4/test.zig:4:40: error: switch must handle all possibilities
    _ = parseInt("hi", 10) catch |err| switch (err) {};
                                       ^~~~~~~~~~~~~~~
doctest-392cb4a4/test.zig:4:40: note: unhandled error value: 'error.InvalidCharacter'
doctest-392cb4a4/test.zig:4:40: note: unhandled error value: 'error.DigitExceedsRadix'
doctest-392cb4a4/test.zig:4:40: note: unhandled error value: 'error.Overflow'
```

The keyword unreachable is used to assert that no errors will occur:

unreachable.zig

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

pub fn main() void {
    const file = std.fs.cwd().openFile("does_not_exist/foo.txt", .{}) catch unreachable;
    file.writeAll("all your codebase are belong to us\n") catch unreachable;
}


$ zig build-exe unreachable.zig
$ ./unreachable
thread 3092 panic: attempt to unwrap error: FileNotFound
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/os.zig:1890:23: 0x105c588 in openatZ (unre
            .NOENT => return error.FileNotFound,
                      ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/fs/Dir.zig:843:9: 0x102750d in openFileZ (
        try posix.openatZ(self.fd, sub_path, os_flags, 0);
        ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/fs/Dir.zig:775:5: 0x10256ae in openFile (u
    return self.openFileZ(&path_c, flags);
    ^
/home/runner/work/www.ziglang.org/www.ziglang.org/doctest-f6031fb5/unreachable.zig:4:77: 0x10239d9 in ma
    const file = std.fs.cwd().openFile("does_not_exist/foo.txt", .{}) catch unreachable;
                                                                            ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/start.zig:575:22: 0x10231ec in posixCallMa
            root.main();
                     ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/start.zig:253:5: 0x1022d41 in _start (unre
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

This invokes undefined behavior in the unsafe build modes, so be sure to use it only when success is guaranteed.

Stack traces on all targets

The stack traces and error return traces shown on this page work on all Tier 1 Support and some Tier 2 Support targets. Even freestanding !

In addition, the standard library has the ability to capture a stack trace at any point and then dump it to standard error later:

stack_traces.zig

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

var address_buffer: [8]usize = undefined;

var trace1 = std.builtin.StackTrace{
    .instruction_addresses = address_buffer[0..4],
    .index = 0,
};

var trace2 = std.builtin.StackTrace{
    .instruction_addresses = address_buffer[4..],
    .index = 0,
};

pub fn main() void {
    foo();
    bar();

    std.debug.print("first one:\n", .{});
    std.debug.dumpStackTrace(trace1);
    std.debug.print("\n\nsecond one:\n", .{});
    std.debug.dumpStackTrace(trace2);
}

fn foo() void {
    std.debug.captureStackTrace(null, &trace1);
}

fn bar() void {
    std.debug.captureStackTrace(null, &trace2);
}
```

```
$ zig build-exe stack_traces.zig
$ ./stack_traces
first one:
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/debug.zig:356:29: 0x10271f5 in captureStac
            addr.* = it.next() orelse {
                            ^
/home/runner/work/www.ziglang.org/www.ziglang.org/doctest-4b4e56a8/stack_traces.zig:26:32: 0x10256bc in
    std.debug.captureStackTrace(null, &trace1);
                               ^
/home/runner/work/www.ziglang.org/www.ziglang.org/doctest-4b4e56a8/stack_traces.zig:16:8: 0x1023b08 in r
    foo();
       ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/start.zig:575:22: 0x10233cc in posixCallMa
            root.main();
                     ^


second one:
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/debug.zig:356:29: 0x10271f5 in captureStac
            addr.* = it.next() orelse {
                            ^
/home/runner/work/www.ziglang.org/www.ziglang.org/doctest-4b4e56a8/stack_traces.zig:30:32: 0x10256dc in
    std.debug.captureStackTrace(null, &trace2);
                               ^
/home/runner/work/www.ziglang.org/www.ziglang.org/doctest-4b4e56a8/stack_traces.zig:17:8: 0x1023b0d in r
```

```
    bar();
       ^
/home/runner/work/www.ziglang.org/www.ziglang.org/zig/lib/std/start.zig:575:22: 0x10233cc in posixCallMa
            root.main();
                     ^
```

You can see this technique being used in the ongoing GeneralPurposeDebugAllocator project .

Generic data structures and functions

Types are values that must be known at compile-time:

types.zig

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

test "types are values" {
    const T1 = u8;
    const T2 = bool;
    assert(T1 != T2);

    const x: T2 = true;
    assert(x);
}
```

```
$ zig test types.zig
1/1 test.types are values... OK
All 1 tests passed.
```

A generic data structure is simply a function that returns a `type`:

generics.zig

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

fn List(comptime T: type) type {
    return struct {
        items: []T,
        len: usize,
    };
}

pub fn main() void {
    var buffer: [10]i32 = undefined;
    var list = List(i32){
        .items = &buffer,
        .len = 0,
    };
    list.items[0] = 1234;
    list.len += 1;

    std.debug.print("{d}\n", .{list.items.len});
}
```

```
$ zig build-exe generics.zig
$ ./generics
10
```

Compile-time reflection and compile-time code execution

The @typeInfo builtin function provides reflection:

reflection.zig

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

const Header = struct {
    magic: u32,
    name: []const u8,
};

pub fn main() void {
    printInfoAboutStruct(Header);
}

fn printInfoAboutStruct(comptime T: type) void {
    const info = @typeInfo(T);
    inline for (info.Struct.fields) |field| {
        std.debug.print(
            "{s} has a field called {s} with type {s}\n",
            .{
                @typeName(T),
                field.name,
                @typeName(field.type),
            },
        );
    }
}
```

```
$ zig build-exe reflection.zig
$ ./reflection
reflection.Header has a field called magic with type u32
reflection.Header has a field called name with type []const u8
```

The Zig Standard Library uses this technique to implement formatted printing. Despite being a Small, simple language, Zig's formatted printing is implemented entirely in Zig. Meanwhile, in C, compile errors for printf are hard-coded into the compiler. Similarly, in Rust, the formatted printing macro is hard-coded into the compiler.

Zig can also evaluate functions and blocks of code at compile-time. In some contexts, such as global variable initializations, the expression is implicitly evaluated at compile-time. Otherwise, one can explicitly evaluate code at compile-time with the comptime keyword. This can be especially powerful when combined with assertions:

test.zig

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

fn fibonacci(x: u32) u32 {
    if (x <= 1) return x;
    return fibonacci(x - 1) + fibonacci(x - 2);
}

test "compile-time evaluation" {
    var array: [fibonacci(6)]i32 = undefined;

    @memset(&array, 42);

    comptime {
        assert(array.len == 12345);
    }
}
```

```
$ zig test test.zig
zig/lib/std/debug.zig:403:14: error: reached unreachable code
    if (!ok) unreachable; // assertion failure
             ^~~~~~~~~~~
doctest-d91293c1/test.zig:15:15: note: called from here
        assert(array.len == 12345);
        ~~~~~~^~~~~~~~~~~~~~~~~~~~
```

Integration with C libraries without FFI/bindings

@cImport directly imports types, variables, functions, and simple macros for use in Zig. It even translates inline functions from C into Zig.

Here is an example of emitting a sine wave using libsoundio :

sine.zig

```
const c = @cImport(@cInclude("soundio/soundio.h"));
const std = @import("std");

fn sio_err(err: c_int) !void {
    switch (err) {
        c.SoundIoErrorNone => {},
        c.SoundIoErrorNoMem => return error.NoMem,
        c.SoundIoErrorInitAudioBackend => return error.InitAudioBackend,
        c.SoundIoErrorSystemResources => return error.SystemResources,
        c.SoundIoErrorOpeningDevice => return error.OpeningDevice,
        c.SoundIoErrorNoSuchDevice => return error.NoSuchDevice,
        c.SoundIoErrorInvalid => return error.Invalid,
        c.SoundIoErrorBackendUnavailable => return error.BackendUnavailable,
        c.SoundIoErrorStreaming => return error.Streaming,
        c.SoundIoErrorIncompatibleDevice => return error.IncompatibleDevice,
        c.SoundIoErrorNoSuchClient => return error.NoSuchClient,
        c.SoundIoErrorIncompatibleBackend => return error.IncompatibleBackend,
        c.SoundIoErrorBackendDisconnected => return error.BackendDisconnected,
        c.SoundIoErrorInterrupted => return error.Interrupted,
        c.SoundIoErrorUnderflow => return error.Underflow,
        c.SoundIoErrorEncodingString => return error.EncodingString,
        else => return error.Unknown,
    }
}

var seconds_offset: f32 = 0;

fn write_callback(
    maybe_outstream: ?[*]c.SoundIoOutStream,
    frame_count_min: c_int,
    frame_count_max: c_int,
) callconv(.C) void {
    _ = frame_count_min;
    const outstream: *c.SoundIoOutStream = &maybe_outstream.?[0];
    const layout = &outstream.layout;
    const float_sample_rate: f32 = @floatFromInt(outstream.sample_rate);
    const seconds_per_frame = 1.0 / float_sample_rate;
    var frames_left = frame_count_max;

    while (frames_left > 0) {
        var frame_count = frames_left;

        var areas: [*]c.SoundIoChannelArea = undefined;
        sio_err(c.soundio_outstream_begin_write(
            maybe_outstream,
            @ptrCast(&areas),
```

```
            &frame_count,
        )) catch |err| std.debug.panic("write failed: {s}", .{@errorName(err)});

        if (frame_count == 0) break;

        const pitch = 440.0;
        const radians_per_second = pitch * 2.0 * std.math.pi;
        var frame: c_int = 0;
        while (frame < frame_count) : (frame += 1) {
            const float_frame: f32 = @floatFromInt(frame);
            const sample = std.math.sin((seconds_offset + float_frame *
                seconds_per_frame) * radians_per_second);
            {
                var channel: usize = 0;
                while (channel < @as(usize, @intCast(layout.channel_count))) : (channel += 1) {
                    const channel_ptr = areas[channel].ptr;
                    const sample_ptr: *f32 = @alignCast(@ptrCast(&channel_ptr[@intCast(areas[channel].st
                    sample_ptr.* = sample;
                }
            }
        }
        const float_frame_count: f32 = @floatFromInt(frame_count);
        seconds_offset += seconds_per_frame * float_frame_count;

        sio_err(c.soundio_outstream_end_write(maybe_outstream)) catch |err| std.debug.panic("end write

        frames_left -= frame_count;
    }
}

pub fn main() !void {
    const soundio = c.soundio_create();
    defer c.soundio_destroy(soundio);

    try sio_err(c.soundio_connect(soundio));

    c.soundio_flush_events(soundio);

    const default_output_index = c.soundio_default_output_device_index(soundio);
    if (default_output_index < 0) return error.NoOutputDeviceFound;

    const device = c.soundio_get_output_device(soundio, default_output_index) orelse return error.OutOfM
    defer c.soundio_device_unref(device);

    std.debug.print("Output device: {s}\n", .{device.*.name});

    const outstream = c.soundio_outstream_create(device) orelse return error.OutOfMemory;
    defer c.soundio_outstream_destroy(outstream);

    outstream.*.format = c.SoundIoFormatFloat32NE;
    outstream.*.write_callback = write_callback;

    try sio_err(c.soundio_outstream_open(outstream));

    try sio_err(c.soundio_outstream_start(outstream));

    while (true) c.soundio_wait_events(soundio);
}
```

```
$ zig build-exe sine.zig -lsoundio -lc
$ ./sine
```

```
Output device: Built-in Audio Analog Stereo
^C
```

This Zig code is significantly simpler than the equivalent C code , as well as having more safety protections, and all this is accomplished by directly importing the C header file - no API bindings.

Zig is better at using C libraries than C is at using C libraries.

Zig is also a C compiler

Here's an example of Zig building some C code:

hello.c

```c
#include <stdio.h>

int main(int argc, char **argv) {
    printf("Hello world\n");
    return 0;
}
```

```
$ zig build-exe hello.c --library c
$ ./hello
Hello world
```

You can use `--verbose-cc` to see what C compiler command this executed:

```
$ zig build-exe hello.c --library c --verbose-cc
zig cc -MD -MV -MF zig-cache/tmp/42zL6fBH8fSo-hello.o.d -nostdinc -fno-spell-checking -isystem /home/and
```

Note that if you run the command again, there is no output, and it finishes instantly:

```
$ time zig build-exe hello.c --library c --verbose-cc

real    0m0.027s
user    0m0.018s
sys     0m0.009s
```

This is thanks to Build Artifact Caching. Zig automatically parses the .d file uses a robust caching system to avoid duplicating work.

Not only can Zig compile C code, but there is a very good reason to use Zig as a C compiler: Zig ships with libc.

Export functions, variables, and types for C code to depend on

One of the primary use cases for Zig is exporting a library with the C ABI for other programming languages to call into. The `export` keyword in front of functions, variables, and types causes them to be part of the library API:

mathtest.zig

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

To make a static library:

```
$ zig build-lib mathtest.zig
```

To make a shared library:

```
$ zig build-lib mathtest.zig -dynamic
```

Here is an example with the Zig Build System:

test.c

```c
#include "mathtest.h"
#include <stdio.h>

int main(int argc, char **argv) {
    int32_t result = add(42, 1337);
    printf("%d\n", result);
    return 0;
}
```

build.zig

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

pub fn build(b: *Builder) void {
    const lib = b.addSharedLibrary("mathtest", "mathtest.zig", b.version(1, 0, 0));

    const exe = b.addExecutable("test", null);
    exe.addCSourceFile("test.c", &[_][]const u8{"-std=c99"});
    exe.linkLibrary(lib);
    exe.linkSystemLibrary("c");

    b.default_step.dependOn(&exe.step);

    const run_cmd = exe.run();

    const test_step = b.step("test", "Test the program");
    test_step.dependOn(&run_cmd.step);
}
```

```
$ zig build test
1379
```

Cross-compiling is a first-class use case

Zig can build for any of the targets from the Support Table with Tier 3 Support or better. No “cross toolchain” needs to be installed or anything like that. Here’s a native Hello World:

hello.zig

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

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

```
$ zig build-exe hello.zig
$ ./hello
Hello, world!
```

Now to build it for x86_64-windows, x86_64-macos, and aarch64-linux:

```
$ zig build-exe hello.zig -target x86_64-windows
$ file hello.exe
hello.exe: PE32+ executable (console) x86-64, for MS Windows
$ zig build-exe hello.zig -target x86_64-macos
$ file hello
hello: Mach-O 64-bit x86_64 executable, flags:<NOUNDEFS|DYLDLINK|TWOLEVEL|PIE>
$ zig build-exe hello.zig -target aarch64-linux
```

```
$ file hello
hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), statically linked, with debug_info, not
```

This works on any Tier 3+ target, for any Tier 3+ target.

Zig ships with libc

You can find the available libc targets with `zig targets`:

```
...
 "libc": [
  "aarch64_be-linux-gnu",
  "aarch64_be-linux-musl",
  "aarch64_be-windows-gnu",
  "aarch64-linux-gnu",
  "aarch64-linux-musl",
  "aarch64-windows-gnu",
  "armeb-linux-gnueabi",
  "armeb-linux-gnueabihf",
  "armeb-linux-musleabi",
  "armeb-linux-musleabihf",
  "armeb-windows-gnu",
  "arm-linux-gnueabi",
  "arm-linux-gnueabihf",
  "arm-linux-musleabi",
  "arm-linux-musleabihf",
  "arm-windows-gnu",
  "i386-linux-gnu",
  "i386-linux-musl",
  "i386-windows-gnu",
  "mips64el-linux-gnuabi64",
  "mips64el-linux-gnuabin32",
  "mips64el-linux-musl",
  "mips64-linux-gnuabi64",
  "mips64-linux-gnuabin32",
  "mips64-linux-musl",
  "mipsel-linux-gnu",
  "mipsel-linux-musl",
  "mips-linux-gnu",
  "mips-linux-musl",
  "powerpc64le-linux-gnu",
  "powerpc64le-linux-musl",
  "powerpc64-linux-gnu",
  "powerpc64-linux-musl",
  "powerpc-linux-gnu",
  "powerpc-linux-musl",
  "riscv64-linux-gnu",
  "riscv64-linux-musl",
  "s390x-linux-gnu",
  "s390x-linux-musl",
  "sparc-linux-gnu",
  "sparcv9-linux-gnu",
  "wasm32-freestanding-musl",
  "x86_64-linux-gnu",
  "x86_64-linux-gnux32",
  "x86_64-linux-musl",
  "x86_64-windows-gnu"
 ],
```

What this means is that `--library c` for these targets *does not depend on any system files*!

Let's look at that C hello world example again:

```
$ zig build-exe hello.c --library c
$ ./hello
Hello world
$ ldd ./hello
        linux-vdso.so.1 (0x00007ffd03dc9000)
        libc.so.6 => /lib/libc.so.6 (0x00007fc4b62be000)
        libm.so.6 => /lib/libm.so.6 (0x00007fc4b5f29000)
        libpthread.so.0 => /lib/libpthread.so.0 (0x00007fc4b5d0a000)
        libdl.so.2 => /lib/libdl.so.2 (0x00007fc4b5b06000)
        librt.so.1 => /lib/librt.so.1 (0x00007fc4b58fe000)
        /lib/ld-linux-x86-64.so.2 => /lib64/ld-linux-x86-64.so.2 (0x00007fc4b6672000)
```

glibc does not support building statically, but musl does:

```
$ zig build-exe hello.c --library c -target x86_64-linux-musl
$ ./hello
Hello world
$ ldd hello
  not a dynamic executable
```

In this example, Zig built musl libc from source and then linked against it. The build of musl libc for x86_64-linux remains available thanks to the caching system, so any time this libc is needed again it will be available instantly.

This means that this functionality is available on any platform. Windows and macOS users can build Zig and C code, and link against libc, for any of the targets listed above. Similarly code can be cross compiled for other architectures:

```
$ zig build-exe hello.c --library c -target aarch64-linux-gnu
$ file hello
hello: ELF 64-bit LSB executable, ARM aarch64, version 1 (SYSV), dynamically linked, interpreter /lib/lc
```

In some ways, Zig is a better C compiler than C compilers!

This functionality is more than bundling a cross-compilation toolchain along with Zig. For example, the total size of libc headers that Zig ships is 22 MiB uncompressed. Meanwhile, the headers for musl libc + linux headers on x86_64 alone are 8 MiB, and for glibc are 3.1 MiB (glibc is missing the linux headers), yet Zig currently ships with 40 libcs. With a naive bundling that would be 444 MiB. However, thanks to this process_headers tool, and some good old manual labor, Zig binary tarballs remain roughly 30 MiB total, despite supporting libc for all these targets, as well as compiler-rt, libunwind, and libcxx, and despite being a clang-compatible C compiler. For comparison, the Windows binary build of clang 8.0.0 itself from llvm.org is 132 MiB.

Note that only the Tier 1 Support targets have been thoroughly tested. It is planned to add more libcs (including for Windows), and to add test coverage for building against all the libcs.

It's planned to have a Zig Package Manager, but it's not done yet. One of the things that will be possible is to create a package for C libraries. This will make the Zig Build System attractive for Zig programmers and C programmers alike.

Zig Build System

Zig comes with a build system, so you don't need make, cmake, or anything like that.

```
$ zig init-exe
Created build.zig
Created src/main.zig

Next, try `zig build --help` or `zig build run`
```

src/main.zig

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

pub fn main() anyerror!void {
```

```
    std.debug.print("All your base are belong to us.\n");
}
```

build.zig

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

pub fn build(b: *Builder) void {
    const mode = b.standardReleaseOptions();
    const exe = b.addExecutable("example", "src/main.zig");
    exe.setBuildMode(mode);

    const run_cmd = exe.run();

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

    b.default_step.dependOn(&exe.step);
    b.installArtifact(exe);
}
```

Let's have a look at that `--help` menu.

```
$ zig build --help
Usage: zig build [steps] [options]

Steps:
  install (default)      Copy build artifacts to prefix path
  uninstall              Remove build artifacts from prefix path
  run                    Run the app

General Options:
  --help                 Print this help and exit
  --verbose              Print commands before executing them
  --prefix [path]        Override default install prefix
  --search-prefix [path] Add a path to look for binaries, libraries, headers

Project-Specific Options:
  -Dtarget=[string]      The CPU architecture, OS, and ABI to build for.
  -Drelease-safe=[bool]  optimizations on and safety on
  -Drelease-fast=[bool]  optimizations on and safety off
  -Drelease-small=[bool] size optimizations on and safety off

Advanced Options:
  --build-file [file]         Override path to build.zig
  --cache-dir [path]          Override path to zig cache directory
  --override-lib-dir [arg]    Override path to Zig lib directory
  --verbose-tokenize          Enable compiler debug output for tokenization
  --verbose-ast               Enable compiler debug output for parsing into an AST
  --verbose-link              Enable compiler debug output for linking
  --verbose-ir                Enable compiler debug output for Zig IR
  --verbose-llvm-ir           Enable compiler debug output for LLVM IR
  --verbose-cimport           Enable compiler debug output for C imports
  --verbose-cc                Enable compiler debug output for C compilation
  --verbose-llvm-cpu-features Enable compiler debug output for LLVM CPU features
```

You can see that one of the available steps is run.

```
$ zig build run
All your base are belong to us.
```

Here are some example build scripts:

- Build script of OpenGL Tetris game
- Build script of bare metal Raspberry Pi 3 arcade game
- Build script of self-hosted Zig compiler

Concurrency via Async Functions

Zig 0.5.0 introduced async functions. This feature has no dependency on a host operating system or even heap-allocated memory. That means async functions are available for the freestanding target.

Zig infers whether a function is async, and allows `async`/`await` on non-async functions, which means that **Zig libraries are agnostic of blocking vs async I/O**. Zig avoids function colors .

The Zig Standard Library implements an event loop that multiplexes async functions onto a thread pool for M:N concurrency. Multithreading safety and race detection are areas of active research.

Wide range of targets supported

Zig uses a “support tier” system to communicate the level of support for different targets.

Support Table as of Zig 0.11.0

Friendly toward package maintainers

The reference Zig compiler is not completely self-hosted yet, but no matter what, it will remain exactly 3 steps to go from having a system C++ compiler to having a fully self-hosted Zig compiler for any target. As Maya Rashish notes, porting Zig to other platforms is fun and speedy .

Non-debug build modes are reproducible/deterministic.

There is a JSON version of the download page.

Several members of the Zig team have experience maintaining packages.

- Daurnimator maintains the Arch Linux package
- Marc Tiehuis maintains the Visual Studio Code package.
- Andrew Kelley spent a year or so doing Debian and Ubuntu packaging , and casually contributes to nixpkgs .
- Jeff Fowler maintains the Homebrew package and started the Sublime package (now maintained by emekoi).