Zig Language Reference

Zig Version

0.1.1 | 0.2.0 | 0.3.0 | 0.4.0 | 0.5.0 | 0.6.0 | 0.7.1 | 0.8.1 | 0.9.1 | 0.10.1 | 0.11.0 | 0.12.1 | 0.13.0 | 0.14.0 | master

Table of Contents

Introduction

Zig is a general-purpose programming language and toolchain for maintaining **robust**, **optimal**, and **reusable** software.

Robust
: Behavior is correct even for edge cases such as out of memory.

Optimal
: Write programs the best way they can behave and perform.

Reusable
: The same code works in many environments which have different constraints.

Maintainable
: Precisely communicate intent to the compiler and other programmers. The language imposes a low overhead to reading code and is resilient to changing requirements and environments.

Often the most efficient way to learn something new is to see examples, so this documentation shows how to use each of Zig's features. It is all on one page so you can search with your browser's search tool.

The code samples in this document are compiled and tested as part of the main test suite of Zig.

This HTML document depends on no external files, so you can use it offline.

Zig Standard Library

The Zig Standard Library has its own documentation.

Zig's Standard Library contains commonly used algorithms, data structures, and definitions to help you build programs or libraries. You will see many examples of Zig's Standard Library used in this documentation. To learn more about the Zig Standard Library, visit the link above.

Alternatively, the Zig Standard Library documentation is provided with each Zig distribution. It can be rendered via a local webserver with:

Shell

```
zig std
```

Hello World

hello.zig

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

pub fn main() !void {
    const stdout = std.io.getStdOut().writer();
    try stdout.print("Hello, {s}!\n", .{"world"});
}
```

Shell

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

Most of the time, it is more appropriate to write to stderr rather than stdout, and whether or not the message is successfully written to the stream is irrelevant. For this common case, there is a simpler API:

hello_again.zig

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

pub fn main() void {
```

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

Shell

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

In this case, the `!` may be omitted from the return type of `main` because no errors are returned from the function.

See also:

- Values
- Tuples
- @import
- Errors
- Entry Point
- Source Encoding
- try

Comments

Zig supports 3 types of comments. Normal comments are ignored, but doc comments and top-level doc comments are used by the compiler to generate the package documentation.

The generated documentation is still experimental, and can be produced with:

Shell

```
zig test -femit-docs main.zig
```

comments.zig

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

pub fn main() void {
    // Comments in Zig start with "//" and end at the next LF byte (end of line).
    // The line below is a comment and won't be executed.

    //print("Hello?", .{});

    print("Hello, world!\n", .{}); // another comment
}
```

Shell

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

There are no multiline comments in Zig (e.g. like `/* */` comments in C). This allows Zig to have the property that each line of code can be tokenized out of context.

Doc Comments

A doc comment is one that begins with exactly three slashes (i.e. `///` but not `////`); multiple doc comments in a row are merged together to form a multiline doc comment. The doc comment documents whatever immediately follows it.

doc_comments.zig

```
/// A structure for storing a timestamp, with nanosecond precision (this is a
/// multiline doc comment).
const Timestamp = struct {
    /// The number of seconds since the epoch (this is also a doc comment).
    seconds: i64, // signed so we can represent pre-1970 (not a doc comment)
    /// The number of nanoseconds past the second (doc comment again).
    nanos: u32,

    /// Returns a `Timestamp` struct representing the Unix epoch; that is, the
    /// moment of 1970 Jan 1 00:00:00 UTC (this is a doc comment too).
    pub fn unixEpoch() Timestamp {
        return Timestamp{
            .seconds = 0,
            .nanos = 0,
        };
    }
};
```

Doc comments are only allowed in certain places; it is a compile error to have a doc comment in an unexpected place, such as in the middle of an expression, or just before a non-doc comment.

invalid_doc-comment.zig

```
/// doc-comment
//! top-level doc-comment
const std = @import("std");
```

Shell

```
$ zig build-obj invalid_doc-comment.zig
doc/langref/invalid_doc-comment.zig:1:16: error: expected type expression, found 'a document cc
/// doc-comment
               ^
```

unattached_doc-comment.zig

```
pub fn main() void {}
```

```
/// End of file
```

Shell

```
$ zig build-obj unattached_doc-comment.zig
doc/langref/unattached_doc-comment.zig:3:1: error: unattached documentation comment
/// End of file
^~~~~~~~~~~~~~~
```

Doc comments can be interleaved with normal comments. Currently, when producing the package documentation, normal comments are merged with doc comments.

Top-Level Doc Comments

A top-level doc comment is one that begins with two slashes and an exclamation point: `//!`; it documents the current module.

It is a compile error if a top-level doc comment is not placed at the start of a container, before any expressions.

tldoc_comments.zig

```
//! This module provides functions for retrieving the current date and
//! time with varying degrees of precision and accuracy. It does not
//! depend on libc, but will use functions from it if available.

const S = struct {
    //! Top level comments are allowed inside a container other than a module,
    //! but it is not very useful.  Currently, when producing the package
    //! documentation, these comments are ignored.
};
```

Values

values.zig

```
// Top-level declarations are order-independent:
const print = std.debug.print;
const std = @import("std");
const os = std.os;
const assert = std.debug.assert;

pub fn main() void {
    // integers
    const one_plus_one: i32 = 1 + 1;
    print("1 + 1 = {}\n", .{one_plus_one});

    // floats
    const seven_div_three: f32 = 7.0 / 3.0;
    print("7.0 / 3.0 = {}\n", .{seven_div_three});
```

```zig
    // boolean
    print("{}\n{}\n{}\n", .{
        true and false,
        true or false,
        !true,
    });

    // optional
    var optional_value: ?[]const u8 = null;
    assert(optional_value == null);

    print("\noptional 1\ntype: {}\nvalue: {?s}\n", .{
        @TypeOf(optional_value), optional_value,
    });

    optional_value = "hi";
    assert(optional_value != null);

    print("\noptional 2\ntype: {}\nvalue: {?s}\n", .{
        @TypeOf(optional_value), optional_value,
    });

    // error union
    var number_or_error: anyerror!i32 = error.ArgNotFound;

    print("\nerror union 1\ntype: {}\nvalue: {!}\n", .{
        @TypeOf(number_or_error),
        number_or_error,
    });

    number_or_error = 1234;

    print("\nerror union 2\ntype: {}\nvalue: {!}\n", .{
        @TypeOf(number_or_error), number_or_error,
    });
}
```

Shell

```
$ zig build-exe values.zig
$ ./values
1 + 1 = 2
7.0 / 3.0 = 2.3333333e0
false
true
false

optional 1
type: ?[]const u8
value: null

optional 2
type: ?[]const u8
value: hi

error union 1
type: anyerror!i32
```

```
value: error.ArgNotFound

error union 2
type: anyerror!i32
value: 1234
```

Primitive Types

Primitive Types

Type	C Equivalent	Description
`i8`	`int8_t`	signed 8-bit integer
`u8`	`uint8_t`	unsigned 8-bit integer
`i16`	`int16_t`	signed 16-bit integer
`u16`	`uint16_t`	unsigned 16-bit integer
`i32`	`int32_t`	signed 32-bit integer
`u32`	`uint32_t`	unsigned 32-bit integer
`i64`	`int64_t`	signed 64-bit integer
`u64`	`uint64_t`	unsigned 64-bit integer
`i128`	`__int128`	signed 128-bit integer
`u128`	`unsigned __int128`	unsigned 128-bit integer
`isize`	`intptr_t`	signed pointer sized integer
`usize`	`uintptr_t`, `size_t`	unsigned pointer sized integer. Also see #5185
`c_char`	`char`	for ABI compatibility with C
`c_short`	`short`	for ABI compatibility with C
`c_ushort`	`unsigned short`	for ABI compatibility with C
`c_int`	`int`	for ABI compatibility with C
`c_uint`	`unsigned int`	for ABI compatibility with C

Type	C Equivalent	Description
`c_long`	`long`	for ABI compatibility with C
`c_ulong`	`unsigned long`	for ABI compatibility with C
`c_longlong`	`long long`	for ABI compatibility with C
`c_ulonglong`	`unsigned long long`	for ABI compatibility with C
`c_longdouble`	`long double`	for ABI compatibility with C
`f16`	`_Float16`	16-bit floating point (10-bit mantissa) IEEE-754-2008 binary16
`f32`	`float`	32-bit floating point (23-bit mantissa) IEEE-754-2008 binary32
`f64`	`double`	64-bit floating point (52-bit mantissa) IEEE-754-2008 binary64
`f80`	`long double`	80-bit floating point (64-bit mantissa) IEEE-754-2008 80-bit extended precision
`f128`	`_Float128`	128-bit floating point (112-bit mantissa) IEEE-754-2008 binary128
`bool`	`bool`	`true` or `false`
`anyopaque`	`void`	Used for type-erased pointers.
`void`	(none)	Always the value `void{}`
`noreturn`	(none)	the type of `break`, `continue`, `return`, `unreachable`, and `while (true) {}`
`type`	(none)	the type of types
`anyerror`	(none)	an error code
`comptime_int`	(none)	Only allowed for comptime-known values. The type of integer literals.

Type	C Equivalent	Description
`comptime_float`	(none)	Only allowed for comptime-known values. The type of float literals.

In addition to the integer types above, arbitrary bit-width integers can be referenced by using an identifier of `i` or `u` followed by digits. For example, the identifier `i7` refers to a signed 7-bit integer. The maximum allowed bit-width of an integer type is `65535`.

See also:

- Integers
- Floats
- void
- Errors
- @Type

Primitive Values

Primitive Values

Name	Description
`true` and `false`	`bool` values
`null`	used to set an optional type to `null`
`undefined`	used to leave a value unspecified

See also:

- Optionals
- undefined

String Literals and Unicode Code Point Literals

String literals are constant single-item Pointers to null-terminated byte arrays. The type of string literals encodes both the length, and the fact that they are null-terminated, and thus they can be coerced to both Slices and Null-Terminated Pointers. Dereferencing string literals converts them to Arrays.

Because Zig source code is UTF-8 encoded, any non-ASCII bytes appearing within a string literal in source code carry their UTF-8 meaning into the content of the string in the Zig program; the bytes are not modified by the compiler. It is possible to embed non-UTF-8 bytes into a string literal using `\xNN` notation.

Indexing into a string containing non-ASCII bytes returns individual bytes, whether valid UTF-8 or not.

Unicode code point literals have type `comptime_int`, the same as Integer Literals. All Escape Sequences are valid in both string literals and Unicode code point literals.

string_literals.zig

```
const print = @import("std").debug.print;
const mem = @import("std").mem; // will be used to compare bytes

pub fn main() void {
    const bytes = "hello";
    print("{}\n", .{@TypeOf(bytes)}); // *const [5:0]u8
    print("{d}\n", .{bytes.len}); // 5
    print("{c}\n", .{bytes[1]}); // 'e'
    print("{d}\n", .{bytes[5]}); // 0
    print("{}\n", .{'e' == '\x65'}); // true
    print("{d}\n", .{'\u{1f4a9}'}); // 128169
    print("{d}\n", .{'💯'}); // 128175
    print("{u}\n", .{'⚡'});
    print("{}\n", .{mem.eql(u8, "hello", "h\x65llo")}); // true
    print("{}\n", .{mem.eql(u8, "💯", "\xf0\x9f\x92\xaf")}); // also true
    const invalid_utf8 = "\xff\xfe"; // non-UTF-8 strings are possible with \xNN notation.
    print("0x{x}\n", .{invalid_utf8[1]}); // indexing them returns individual bytes...
    print("0x{x}\n", .{"💯"[1]}); // ...as does indexing part-way through non-ASCII characters
}
```

Shell

```
$ zig build-exe string_literals.zig
$ ./string_literals
*const [5:0]u8
5
e
0
true
128169
128175
⚡
true
true
0xfe
0x9f
```

See also:

- Arrays
- Source Encoding

Escape Sequences

Escape Sequences

Escape Sequence	Name
`\n`	Newline
`\r`	Carriage Return
`\t`	Tab
`\\`	Backslash
`\'`	Single Quote
`\"`	Double Quote
`\xNN`	hexadecimal 8-bit byte value (2 digits)
`\u{NNNNNN}`	hexadecimal Unicode scalar value UTF-8 encoded (1 or more digits)

Note that the maximum valid Unicode scalar value is `0x10ffff`.

Multiline String Literals

Multiline string literals have no escapes and can span across multiple lines. To start a multiline string literal, use the `\\` token. Just like a comment, the string literal goes until the end of the line. The end of the line is not included in the string literal. However, if the next line begins with `\\` then a newline is appended and the string literal continues.

multiline_string_literals.zig

```
const hello_world_in_c =
    \\#include <stdio.h>
    \\
    \\int main(int argc, char **argv) {
    \\    printf("hello world\n");
    \\    return 0;
    \\}
;
```

See also:

- @embedFile

Assignment

Use the `const` keyword to assign a value to an identifier:

constant_identifier_cannot_change.zig

```
const x = 1234;

fn foo() void {
    // It works at file scope as well as inside functions.
    const y = 5678;

    // Once assigned, an identifier cannot be changed.
    y += 1;
}

pub fn main() void {
    foo();
}
```

Shell

```
$ zig build-exe constant_identifier_cannot_change.zig
/home/andy/src/zig/doc/langref/constant_identifier_cannot_change.zig:8:7: error: cannot assign
    y += 1;
    ~~^~~~
referenced by:
    main: /home/andy/src/zig/doc/langref/constant_identifier_cannot_change.zig:12:8
    posixCallMainAndExit: /home/andy/src/zig/lib/std/start.zig:647:22
    4 reference(s) hidden; use '-freference-trace=6' to see all references
```

`const` applies to all of the bytes that the identifier immediately addresses. Pointers have their own const-ness.

If you need a variable that you can modify, use the `var` keyword:

mutable_var.zig

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

pub fn main() void {
    var y: i32 = 5678;

    y += 1;

    print("{d}", .{y});
}
```

Shell

```
$ zig build-exe mutable_var.zig
$ ./mutable_var
5679
```

Variables must be initialized:

var_must_be_initialized.zig

```zig
pub fn main() void {
    var x: i32;

    x = 1;
}
```

Shell

```
$ zig build-exe var_must_be_initialized.zig
/home/andy/src/zig/doc/langref/var_must_be_initialized.zig:2:15: error: expected '=', found ';'
    var x: i32;
              ^
```

undefined

Use `undefined` to leave variables uninitialized:

assign_undefined.zig

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

pub fn main() void {
    var x: i32 = undefined;
    x = 1;
    print("{d}", .{x});
}
```

Shell

```
$ zig build-exe assign_undefined.zig
$ ./assign_undefined
1
```

`undefined` can be coerced to any type. Once this happens, it is no longer possible to detect that the value is `undefined`. `undefined` means the value could be anything, even something that is nonsense according to the type. Translated into English, `undefined` means "Not a meaningful value. Using this value would be a bug. The value will be unused, or overwritten before being used."

In Debug mode, Zig writes `0xaa` bytes to undefined memory. This is to catch bugs early, and to help detect use of undefined memory in a debugger. However, this behavior is only an implementation feature, not a language semantic, so it is not guaranteed to be observable to code.

Destructuring

A destructuring assignment can separate elements of indexable aggregate types (Tuples, Arrays, Vectors):

destructuring_to_existing.zig

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

pub fn main() void {
    var x: u32 = undefined;
    var y: u32 = undefined;
    var z: u32 = undefined;

    const tuple = .{ 1, 2, 3 };

    x, y, z = tuple;

    print("tuple: x = {}, y = {}, z = {}\n", .{x, y, z});

    const array = [_]u32{ 4, 5, 6 };

    x, y, z = array;

    print("array: x = {}, y = {}, z = {}\n", .{x, y, z});

    const vector: @Vector(3, u32) = .{ 7, 8, 9 };

    x, y, z = vector;

    print("vector: x = {}, y = {}, z = {}\n", .{x, y, z});
}
```

Shell

```
$ zig build-exe destructuring_to_existing.zig
$ ./destructuring_to_existing
tuple: x = 1, y = 2, z = 3
array: x = 4, y = 5, z = 6
vector: x = 7, y = 8, z = 9
```

A destructuring expression may only appear within a block (i.e. not at container scope). The left hand side of the assignment must consist of a comma separated list, each element of which may be either an lvalue (for instance, an existing `var`) or a variable declaration:

destructuring_mixed.zig

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

pub fn main() void {
    var x: u32 = undefined;

    const tuple = .{ 1, 2, 3 };

    x, var y : u32, const z = tuple;

    print("x = {}, y = {}, z = {}\n", .{x, y, z});

    // y is mutable
    y = 100;

    // You can use _ to throw away unwanted values.
```

```
    _, x, _ = tuple;

    print("x = {}", .{x});
}
```

Shell

```
$ zig build-exe destructuring_mixed.zig
$ ./destructuring_mixed
x = 1, y = 2, z = 3
x = 2
```

A destructure may be prefixed with the `comptime` keyword, in which case the entire destructure expression is evaluated at comptime. All `var`s declared would be `comptime var`s and all expressions (both result locations and the assignee expression) are evaluated at comptime.

See also:

- Destructuring Tuples
- Destructuring Arrays
- Destructuring Vectors

Zig Test

Code written within one or more `test` declarations can be used to ensure behavior meets expectations:

testing_introduction.zig

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

test "expect addOne adds one to 41" {

    // The Standard Library contains useful functions to help create tests.
    // `expect` is a function that verifies its argument is true.
    // It will return an error if its argument is false to indicate a failure.
    // `try` is used to return an error to the test runner to notify it that the test failed.
    try std.testing.expect(addOne(41) == 42);
}

test addOne {
    // A test name can also be written using an identifier.
    // This is a doctest, and serves as documentation for `addOne`.
    try std.testing.expect(addOne(41) == 42);
}

/// The function `addOne` adds one to the number given as its argument.
fn addOne(number: i32) i32 {
    return number + 1;
}
```

Shell

```
$ zig test testing_introduction.zig
1/2 testing_introduction.test.expect addOne adds one to 41...OK
2/2 testing_introduction.decltest.addOne...OK
All 2 tests passed.
```

The **`testing_introduction.zig`** code sample tests the function `addOne` to ensure that it returns `42` given the input `41`. From this test's perspective, the `addOne` function is said to be *code under test*.

`zig test` is a tool that creates and runs a test build. By default, it builds and runs an executable program using the *default test runner* provided by the Zig Standard Library as its main entry point. During the build, **`test`** declarations found while resolving the given Zig source file are included for the default test runner to run and report on.

> This documentation discusses the features of the default test runner as provided by the Zig Standard Library. Its source code is located in **`lib/compiler/test_runner.zig`**.

The shell output shown above displays two lines after the `zig test` command. These lines are printed to standard error by the default test runner:

`1/2 testing_introduction.test.expect addOne adds one to 41...`
: Lines like this indicate which test, out of the total number of tests, is being run. In this case, `1/2` indicates that the first test, out of a total of two tests, is being run. Note that, when the test runner program's standard error is output to the terminal, these lines are cleared when a test succeeds.

`2/2 testing_introduction.decltest.addOne...`
: When the test name is an identifier, the default test runner uses the text decltest instead of test.

`All 2 tests passed.`
: This line indicates the total number of tests that have passed.

Test Declarations

Test declarations contain the keyword **`test`**, followed by an optional name written as a string literal or an identifier, followed by a block containing any valid Zig code that is allowed in a function.

Non-named test blocks always run during test builds and are exempt from Skip Tests.

Test declarations are similar to Functions: they have a return type and a block of code. The implicit return type of **`test`** is the Error Union Type **`anyerror!void`**, and it cannot be changed. When a Zig source file is not built using the `zig test` tool, the test declarations are omitted from the build.

Test declarations can be written in the same file, where code under test is written, or in a separate Zig source file. Since test declarations are top-level declarations, they are order-independent and can be written before or after the code under test.

See also:

- The Global Error Set
- Grammar

Doctests

Test declarations named using an identifier are *doctests*. The identifier must refer to another declaration in scope. A doctest, like a doc comment, serves as documentation for the associated declaration, and will appear in the generated documentation for the declaration.

An effective doctest should be self-contained and focused on the declaration being tested, answering questions a new user might have about its interface or intended usage, while avoiding unnecessary or confusing details. A doctest is not a substitute for a doc comment, but rather a supplement and companion providing a testable, code-driven example, verified by `zig test`.

Test Failure

The default test runner checks for an error returned from a test. When a test returns an error, the test is considered a failure and its error return trace is output to standard error. The total number of failures will be reported after all tests have run.

testing_failure.zig

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

test "expect this to fail" {
    try std.testing.expect(false);
}

test "expect this to succeed" {
    try std.testing.expect(true);
}
```

Shell

```
$ zig test testing_failure.zig
1/2 testing_failure.test.expect this to fail...FAIL (TestUnexpectedResult)
/home/andy/src/zig/lib/std/testing.zig:580:14: 0x104899f in expect (test)
    if (!ok) return error.TestUnexpectedResult;
             ^
/home/andy/src/zig/doc/langref/testing_failure.zig:4:5: 0x1048a35 in test.expect this to fail (
    try std.testing.expect(false);
    ^
2/2 testing_failure.test.expect this to succeed...OK
```

```
1 passed; 0 skipped; 1 failed.
error: the following test command failed with exit code 1:
/home/andy/src/zig/.zig-cache/o/4b55be885f04d4406910bf905e0a160c/test --seed=0x5dd97878
```

Skip Tests

One way to skip tests is to filter them out by using the `zig test` command line parameter `--test-filter [text]`. This makes the test build only include tests whose name contains the supplied filter text. Note that non-named tests are run even when using the `--test-filter [text]` command line parameter.

To programmatically skip a test, make a `test` return the error `error.SkipZigTest` and the default test runner will consider the test as being skipped. The total number of skipped tests will be reported after all tests have run.

testing_skip.zig

```
test "this will be skipped" {
    return error.SkipZigTest;
}
```

Shell

```
$ zig test testing_skip.zig
1/1 testing_skip.test.this will be skipped...SKIP
0 passed; 1 skipped; 0 failed.
```

Report Memory Leaks

When code allocates [Memory](#) using the [Zig Standard Library](#)'s testing allocator, `std.testing.allocator`, the default test runner will report any leaks that are found from using the testing allocator:

testing_detect_leak.zig

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

test "detect leak" {
    var list = std.ArrayList(u21).init(std.testing.allocator);
    // missing `defer list.deinit();`
    try list.append('☔');

    try std.testing.expect(list.items.len == 1);
}
```

Shell

```
$ zig test testing_detect_leak.zig
1/1 testing_detect_leak.test.detect leak...OK
[gpa] (err): memory address 0x7f7537720000 leaked:
/home/andy/src/zig/lib/std/array_list.zig:474:67: 0x10695a2 in ensureTotalCapacityPrecise (test
                const new_memory = try self.allocator.alignedAlloc(T, alignment, new_capacity);
```

```
                                                                  ^
/home/andy/src/zig/lib/std/array_list.zig:450:51: 0x104ea80 in ensureTotalCapacity (test)
            return self.ensureTotalCapacityPrecise(better_capacity);
                                                  ^
/home/andy/src/zig/lib/std/array_list.zig:500:41: 0x104cdaf in addOne (test)
            try self.ensureTotalCapacity(newlen);
                                        ^
/home/andy/src/zig/lib/std/array_list.zig:261:49: 0x104a8cd in append (test)
            const new_item_ptr = try self.addOne();
                                                ^
/home/andy/src/zig/doc/langref/testing_detect_leak.zig:6:20: 0x1048d05 in test.detect leak (tes
    try list.append('☔');
                   ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10f7c35 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10f1c8d in main (test)
        return mainTerminal();
                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10f1212 in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10f0ded in _start (test)
    asm volatile (switch (native_arch) {
    ^

All 1 tests passed.
1 errors were logged.
1 tests leaked memory.
error: the following test command failed with exit code 1:
/home/andy/src/zig/.zig-cache/o/9073e84ae632f507d6f6e265f9c82f56/test --seed=0xe9497de3
```

See also:

- defer
- Memory

Detecting Test Build

Use the compile variable `@import("builtin").is_test` to detect a test build:

testing_detect_test.zig

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

test "builtin.is_test" {
    try expect(isATest());
}

fn isATest() bool {
    return builtin.is_test;
}
```

Shell

```
$ zig test testing_detect_test.zig
1/1 testing_detect_test.test.builtin.is_test...OK
All 1 tests passed.
```

Test Output and Logging

The default test runner and the Zig Standard Library's testing namespace output messages to standard error.

The Testing Namespace

The Zig Standard Library's `testing` namespace contains useful functions to help you create tests. In addition to the `expect` function, this document uses a couple of more functions as exemplified here:

testing_namespace.zig

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

test "expectEqual demo" {
    const expected: i32 = 42;
    const actual = 42;

    // The first argument to `expectEqual` is the known, expected, result.
    // The second argument is the result of some expression.
    // The actual's type is casted to the type of expected.
    try std.testing.expectEqual(expected, actual);
}

test "expectError demo" {
    const expected_error = error.DemoError;
    const actual_error_union: anyerror!void = error.DemoError;

    // `expectError` will fail when the actual error is different than
    // the expected error.
    try std.testing.expectError(expected_error, actual_error_union);
}
```

Shell

```
$ zig test testing_namespace.zig
1/2 testing_namespace.test.expectEqual demo...OK
2/2 testing_namespace.test.expectError demo...OK
All 2 tests passed.
```

The Zig Standard Library also contains functions to compare Slices, strings, and more. See the rest of the `std.testing` namespace in the Zig Standard Library for more available functions.

Test Tool Documentation

`zig test` has a few command line parameters which affect the compilation. See `zig test --help` for a full list.

Variables

A variable is a unit of Memory storage.

It is generally preferable to use `const` rather than `var` when declaring a variable. This causes less work for both humans and computers to do when reading code, and creates more optimization opportunities.

The `extern` keyword or @extern builtin function can be used to link against a variable that is exported from another object. The `export` keyword or @export builtin function can be used to make a variable available to other objects at link time. In both cases, the type of the variable must be C ABI compatible.

See also:

- Exporting a C Library

Identifiers

Variable identifiers are never allowed to shadow identifiers from an outer scope.

Identifiers must start with an alphabetic character or underscore and may be followed by any number of alphanumeric characters or underscores. They must not overlap with any keywords. See Keyword Reference.

If a name that does not fit these requirements is needed, such as for linking with external libraries, the `@""` syntax may be used.

identifiers.zig

```
const @"identifier with spaces in it" = 0xff;
const @"1SmallStep4Man" = 112358;

const c = @import("std").c;
pub extern "c" fn @"error"() void;
pub extern "c" fn @"fstat$INODE64"(fd: c.fd_t, buf: *c.Stat) c_int;

const Color = enum {
    red,
    @"really red",
};
const color: Color = .@"really red";
```

Container Level Variables

Container level variables have static lifetime and are order-independent and lazily analyzed. The initialization value of container level variables is implicitly comptime. If a container level variable is `const` then its value is `comptime`-known, otherwise it is runtime-known.

test_container_level_variables.zig

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

test "container level variables" {
    try expect(x == 46);
    try expect(y == 56);
}

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

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

Shell

```
$ zig test test_container_level_variables.zig
1/1 test_container_level_variables.test.container level variables...OK
All 1 tests passed.
```

Container level variables may be declared inside a struct, union, enum, or opaque:

test_namespaced_container_level_variable.zig

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

test "namespaced container level variable" {
    try expect(foo() == 1235);
    try expect(foo() == 1236);
}

const S = struct {
    var x: i32 = 1234;
};

fn foo() i32 {
    S.x += 1;
    return S.x;
}
```

Shell

```
$ zig test test_namespaced_container_level_variable.zig
1/1 test_namespaced_container_level_variable.test.namespaced container level variable...OK
All 1 tests passed.
```

Static Local Variables

It is also possible to have local variables with static lifetime by using containers inside functions.

test_static_local_variable.zig

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

test "static local variable" {
    try expect(foo() == 1235);
    try expect(foo() == 1236);
}

fn foo() i32 {
    const S = struct {
        var x: i32 = 1234;
    };
    S.x += 1;
    return S.x;
}
```

Shell

```
$ zig test test_static_local_variable.zig
1/1 test_static_local_variable.test.static local variable...OK
All 1 tests passed.
```

Thread Local Variables

A variable may be specified to be a thread-local variable using the `threadlocal` keyword, which makes each thread work with a separate instance of the variable:

test_thread_local_variables.zig

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

threadlocal var x: i32 = 1234;

test "thread local storage" {
    const thread1 = try std.Thread.spawn(.{}, testTls, .{});
    const thread2 = try std.Thread.spawn(.{}, testTls, .{});
    testTls();
    thread1.join();
    thread2.join();
}

fn testTls() void {
    assert(x == 1234);
    x += 1;
    assert(x == 1235);
}
```

Shell

```
$ zig test test_thread_local_variables.zig
1/1 test_thread_local_variables.test.thread local storage...OK
All 1 tests passed.
```

For Single Threaded Builds, all thread local variables are treated as regular Container Level Variables.

Thread local variables may not be `const`.

Local Variables

Local variables occur inside Functions, comptime blocks, and @cImport blocks.

When a local variable is `const`, it means that after initialization, the variable's value will not change. If the initialization value of a `const` variable is comptime-known, then the variable is also `comptime`-known.

A local variable may be qualified with the `comptime` keyword. This causes the variable's value to be `comptime`-known, and all loads and stores of the variable to happen during semantic analysis of the program, rather than at runtime. All variables declared in a `comptime` expression are implicitly `comptime` variables.

test_comptime_variables.zig

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

test "comptime vars" {
    var x: i32 = 1;
    comptime var y: i32 = 1;

    x += 1;
    y += 1;

    try expect(x == 2);
    try expect(y == 2);

    if (y != 2) {
        // This compile error never triggers because y is a comptime variable,
        // and so `y != 2` is a comptime value, and this if is statically evaluated.
        @compileError("wrong y value");
    }
}
```

Shell

```
$ zig test test_comptime_variables.zig
1/1 test_comptime_variables.test.comptime vars...OK
All 1 tests passed.
```

Integers

Integer Literals

integer_literals.zig

```
const decimal_int = 98222;
const hex_int = 0xff;
const another_hex_int = 0xFF;
const octal_int = 0o755;
const binary_int = 0b11110000;

// underscores may be placed between two digits as a visual separator
const one_billion = 1_000_000_000;
const binary_mask = 0b1_1111_1111;
const permissions = 0o7_5_5;
const big_address = 0xFF80_0000_0000_0000;
```

Runtime Integer Values

Integer literals have no size limitation, and if any Illegal Behavior occurs, the compiler catches it.

However, once an integer value is no longer known at compile-time, it must have a known size, and is vulnerable to safety-checked Illegal Behavior.

runtime_vs_comptime.zig

```
fn divide(a: i32, b: i32) i32 {
    return a / b;
}
```

In this function, values `a` and `b` are known only at runtime, and thus this division operation is vulnerable to both Integer Overflow and Division by Zero.

Operators such as `+` and `-` cause Illegal Behavior on integer overflow. Alternative operators are provided for wrapping and saturating arithmetic on all targets. `+%` and `-%` perform wrapping arithmetic while `+|` and `-|` perform saturating arithmetic.

Zig supports arbitrary bit-width integers, referenced by using an identifier of `i` or `u` followed by digits. For example, the identifier `i7` refers to a signed 7-bit integer. The maximum allowed bit-width of an integer type is `65535`. For signed integer types, Zig uses a two's complement representation.

See also:

- Wrapping Operations

Floats

Zig has the following floating point types:

- `f16` - IEEE-754-2008 binary16
- `f32` - IEEE-754-2008 binary32
- `f64` - IEEE-754-2008 binary64
- `f80` - IEEE-754-2008 80-bit extended precision
- `f128` - IEEE-754-2008 binary128
- `c_longdouble` - matches `long double` for the target C ABI

Float Literals

Float literals have type `comptime_float` which is guaranteed to have the same precision and operations of the largest other floating point type, which is `f128`.

Float literals coerce to any floating point type, and to any integer type when there is no fractional component.

float_literals.zig

```
const floating_point = 123.0E+77;
const another_float = 123.0;
const yet_another = 123.0e+77;

const hex_floating_point = 0x103.70p-5;
const another_hex_float = 0x103.70;
const yet_another_hex_float = 0x103.70P-5;

// underscores may be placed between two digits as a visual separator
const lightspeed = 299_792_458.000_000;
const nanosecond = 0.000_000_001;
const more_hex = 0x1234_5678.9ABC_CDEFp-10;
```

There is no syntax for NaN, infinity, or negative infinity. For these special values, one must use the standard library:

float_special_values.zig

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

const inf = std.math.inf(f32);
const negative_inf = -std.math.inf(f64);
const nan = std.math.nan(f128);
```

Floating Point Operations

By default floating point operations use `Strict` mode, but you can switch to `Optimized` mode on a per-block basis:

float_mode_obj.zig

```
const std = @import("std");
const big = @as(f64, 1 << 40);

export fn foo_strict(x: f64) f64 {
    return x + big - big;
}

export fn foo_optimized(x: f64) f64 {
    @setFloatMode(.optimized);
    return x + big - big;
}
```

Shell

```
$ zig build-obj float_mode_obj.zig -O ReleaseFast
```

For this test we have to separate code into two object files - otherwise the optimizer figures out all the values at compile-time, which operates in strict mode.

float_mode_exe.zig

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

extern fn foo_strict(x: f64) f64;
extern fn foo_optimized(x: f64) f64;

pub fn main() void {
    const x = 0.001;
    print("optimized = {}\n", .{foo_optimized(x)});
    print("strict = {}\n", .{foo_strict(x)});
}
```

See also:

- @setFloatMode
- Division by Zero

Operators

There is no operator overloading. When you see an operator in Zig, you know that it is doing something from this table, and nothing else.

Table of Operators

Name	Syntax	Types	Remarks	Exam

Name	Syntax	Types	Remarks	Exam
Addition	`a + b` `a += b`	Integers Floats	Can cause overflow for integers. Invokes Peer Type Resolution for the operands. See also @addWithOverflow.	`2 + 5 == 7`
Wrapping Addition	`a +% b` `a +%= b`	Integers	Twos-complement wrapping behavior. Invokes Peer Type Resolution for the operands. See also @addWithOverflow.	`@as(u32, 0xffffffff) +`
Saturating Addition	`a +| b` `a +|= b`	Integers	Invokes Peer Type Resolution for the operands.	`@as(u8, 255) +| 1 == @`
Subtraction	`a - b` `a -= b`	Integers Floats	Can cause overflow for integers. Invokes Peer Type Resolution for the operands. See also @subWithOverflow.	`2 - 5 == -3`
Wrapping Subtraction	`a -% b` `a -%= b`	Integers	Twos-complement wrapping behavior. Invokes Peer Type	`@as(u8, 0) -% 1 == 255`

Name	Syntax	Types	Remarks	Exam
			Resolution for the operands. See also @subWithOverflow.	
Saturating Subtraction	`a -| b` `a -|= b`	Integers	Invokes Peer Type Resolution for the operands.	`@as(u32, 0) -| 1 == 0`
Negation	`-a`	Integers Floats	Can cause overflow for integers.	`-1 == 0 - 1`
Wrapping Negation	`-%a`	Integers	Twos-complement wrapping behavior.	`-%@as(i8, -128) == -12`
Multiplication	`a * b` `a *= b`	Integers Floats	Can cause overflow for integers. Invokes Peer Type Resolution for the operands. See also @mulWithOverflow.	`2 * 5 == 10`

Name	Syntax	Types	Remarks	Exam
Wrapping Multiplication	`a *% b` `a *%= b`	Integers	Twos-complement wrapping behavior. Invokes Peer Type Resolution for the operands. See also @mulWithOverflow.	`@as(u8, 200) *% 2 == 1`
Saturating Multiplication	`a *| b` `a *|= b`	Integers	Invokes Peer Type Resolution for the operands.	`@as(u8, 200) *| 2 == 2`
Division	`a / b` `a /= b`	Integers Floats	Can cause overflow for integers. Can cause Division by Zero for integers. Can cause Division by Zero for floats in FloatMode.Optimized Mode. Signed integer operands must be comptime-known and positive. In other cases, use @divTrunc, @divFloor, or @divExact instead. Invokes Peer Type Resolution for the operands.	`10 / 5 == 2`
Remainder Division	`a % b` `a %= b`	Integers	Can cause Division by Zero for integers.	`10 % 3 == 1`

Name	Syntax	Types	Remarks	Exam
		Floats	Can cause Division by Zero for floats in FloatMode.Optimized Mode. Signed or floating-point operands must be comptime-known and positive. In other cases, use @rem or @mod instead. Invokes Peer Type Resolution for the operands.	
Bit Shift Left	`a << b` `a <<= b`	Integers	Moves all bits to the left, inserting new zeroes at the least-significant bit. `b` must be comptime-known or have a type with log2 number of bits as `a`. See also @shlExact. See also @shlWithOverflow.	`0b1 << 8 == 0b1000000`
Saturating Bit Shift Left	`a <<| b` `a <<|= b`	Integers	See also @shlExact. See also @shlWithOverflow.	`@as(u8, 1) <<| 8 == 25`
Bit Shift Right	`a >> b` `a >>= b`	Integers	Moves all bits to the right, inserting zeroes at the most-significant bit.	`0b1010 >> 1 == 0b101`

Name	Syntax	Types	Remarks	Exam
			`b` must be comptime-known or have a type with log2 number of bits as `a`. See also @shrExact.	
Bitwise And	`a & b` `a &= b`	Integers	Invokes Peer Type Resolution for the operands.	`0b011 & 0b101 == 0b001`
Bitwise Or	`a | b` `a |= b`	Integers	Invokes Peer Type Resolution for the operands.	`0b010 | 0b100 == 0b110`
Bitwise Xor	`a ^ b` `a ^= b`	Integers	Invokes Peer Type Resolution for the operands.	`0b011 ^ 0b101 == 0b110`
Bitwise Not	`~a`	Integers		`~@as(u8, 0b10101111) =`

Defaulting Optional Unwrap	`a orelse b`	Optionals	If `a` is `null`, returns `b` ("default value"), otherwise returns the unwrapped value of `a`. Note that `b` may be a value of type noreturn.	`const value: ?u32 = nu`
				`const unwrapped = valu`
				`unwrapped == 1234`

Optional Unwrap	`a.?`	Optionals	Equivalent to: `a orelse unreachable`	`const value: ?u32 = 56` `value.? == 5678`

Name	Syntax	Types	Remarks	Exam

Defaulting Error Unwrap	`a catch b`
	`a catch |err| b`
		Error Unions	If `a` is an `error`, returns `b` ("default value"), otherwise returns the unwrapped value of `a`. Note that `b` may be a value of type noreturn. `err` is the `error` and is in scope of the expression `b`.	`const value: anyerror!`
				`const unwrapped = valu`
				`unwrapped == 1234`

Logical And	`a and b`	bool	If `a` is `false`, returns `false` without evaluating `b`. Otherwise, returns `b`.	`(false and true) == fa`
Logical Or	`a or b`	bool	If `a` is `true`, returns `true` without evaluating `b`. Otherwise, returns `b`.	`(false or true) == tru`
Boolean Not	`!a`	bool		`!false == true`
Equality	`a == b`	Integers Floats bool type	Returns `true` if a and b are equal, otherwise returns `false`. Invokes Peer Type Resolution for the operands.	`(1 == 1) == true`
Null Check	`a == null`	Optionals	Returns `true` if a is `null`, otherwise returns `false`.	`const value: ?u32 = nu` `(value == null) == tru`
Inequality	`a != b`	Integers	Returns `false` if a and b are equal,	`(1 != 1) == false`

Name	Syntax	Types	Remarks	Exam
		Floats bool type	otherwise returns `true`. Invokes Peer Type Resolution for the operands.	
Non-Null Check	`a != null`	Optionals	Returns `false` if a is `null`, otherwise returns `true`.	`const value: ?u32 = nu` `(value != null) == fal`
Greater Than	`a > b`	Integers Floats	Returns `true` if a is greater than b, otherwise returns `false`. Invokes Peer Type Resolution for the operands.	`(2 > 1) == true`
Greater or Equal	`a >= b`	Integers Floats	Returns `true` if a is greater than or equal to b, otherwise returns `false`. Invokes Peer Type Resolution for the operands.	`(2 >= 1) == true`
Less Than	`a < b`	Integers Floats	Returns `true` if a is less than b, otherwise returns `false`. Invokes Peer Type Resolution for the operands.	`(1 < 2) == true`
Lesser or Equal	`a <= b`	Integers Floats	Returns `true` if a is less than or equal to b, otherwise returns `false`. Invokes Peer Type Resolution for the operands.	`(1 <= 2) == true`

Name	Syntax	Types	Remarks	Exam

Array Concatenation	`a ++ b`	Arrays	Only available when the lengths of both `a` and `b` are compile-time known.	`const mem = @import("s`
				`const array1 = [_]u32{`
				`const array2 = [_]u32{`
				`const together = array`
				`mem.eql(u32, &togethe`

Array Multiplication	`a ** b`	Arrays	Only available when the length of `a` and `b` are compile-time known.	`const mem = @import("s`
				`const pattern = "ab" *`
				`mem.eql(u8, pattern, '`

Pointer Dereference	`a.*`	Pointers	Pointer dereference.	`const x: u32 = 1234;`
				`const ptr = &x;`
				`ptr.* == 1234`

Address Of	`&a`	All types		`const x: u32 = 1234;`
				`const ptr = &x;`
				`ptr.* == 1234`

Error Set Merge	`a || b`	Error Set Type	Merging Error Sets	`const A = error{One};`
				`const B = error{Two};`
				`(A || B) == error{One,`

Precedence

```
x() x[] x.y x.* x.?
a!b
x{}
!x -x -%x ~x &x ?x
* / % ** *% *| ||
+ - ++ +% -% +| -|
<< >> <<|
& ^ | orelse catch
== != < > <= >=
and
or
= *= *%= *|= /= %= += +%= +|= -= -%= -|= <<= <<|= >>= &= ^= |=
```

Arrays

test_arrays.zig

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

// array literal
const message = [_]u8{ 'h', 'e', 'l', 'l', 'o' };

// alternative initialization using result location
const alt_message: [5]u8 = .{ 'h', 'e', 'l', 'l', 'o' };

comptime {
    assert(mem.eql(u8, &message, &alt_message));
}

// get the size of an array
comptime {
    assert(message.len == 5);
}

// A string literal is a single-item pointer to an array.
const same_message = "hello";

comptime {
    assert(mem.eql(u8, &message, same_message));
}

test "iterate over an array" {
    var sum: usize = 0;
    for (message) |byte| {
        sum += byte;
    }
    try expect(sum == 'h' + 'e' + 'l' * 2 + 'o');
}

// modifiable array
var some_integers: [100]i32 = undefined;

test "modify an array" {
    for (&some_integers, 0..) |*item, i| {
        item.* = @intCast(i);
    }
    try expect(some_integers[10] == 10);
    try expect(some_integers[99] == 99);
}

// array concatenation works if the values are known
// at compile time
const part_one = [_]i32{ 1, 2, 3, 4 };
const part_two = [_]i32{ 5, 6, 7, 8 };
const all_of_it = part_one ++ part_two;
comptime {
    assert(mem.eql(i32, &all_of_it, &[_]i32{ 1, 2, 3, 4, 5, 6, 7, 8 }));
```

```
}

// remember that string literals are arrays
const hello = "hello";
const world = "world";
const hello_world = hello ++ " " ++ world;
comptime {
    assert(mem.eql(u8, hello_world, "hello world"));
}

// ** does repeating patterns
const pattern = "ab" ** 3;
comptime {
    assert(mem.eql(u8, pattern, "ababab"));
}

// initialize an array to zero
const all_zero = [_]u16{0} ** 10;

comptime {
    assert(all_zero.len == 10);
    assert(all_zero[5] == 0);
}

// use compile-time code to initialize an array
var fancy_array = init: {
    var initial_value: [10]Point = undefined;
    for (&initial_value, 0..) |*pt, i| {
        pt.* = Point{
            .x = @intCast(i),
            .y = @intCast(i * 2),
        };
    }
    break :init initial_value;
};
const Point = struct {
    x: i32,
    y: i32,
};

test "compile-time array initialization" {
    try expect(fancy_array[4].x == 4);
    try expect(fancy_array[4].y == 8);
}

// call a function to initialize an array
var more_points = [_]Point{makePoint(3)} ** 10;
fn makePoint(x: i32) Point {
    return Point{
        .x = x,
        .y = x * 2,
    };
}
test "array initialization with function calls" {
    try expect(more_points[4].x == 3);
    try expect(more_points[4].y == 6);
    try expect(more_points.len == 10);
```

```
}
```

Shell

```
$ zig test test_arrays.zig
1/4 test_arrays.test.iterate over an array...OK
2/4 test_arrays.test.modify an array...OK
3/4 test_arrays.test.compile-time array initialization...OK
4/4 test_arrays.test.array initialization with function calls...OK
All 4 tests passed.
```

See also:

- for
- Slices

Multidimensional Arrays

Multidimensional arrays can be created by nesting arrays:

test_multidimensional_arrays.zig

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

const mat4x4 = [4][4]f32{
    [_]f32{ 1.0, 0.0, 0.0, 0.0 },
    [_]f32{ 0.0, 1.0, 0.0, 1.0 },
    [_]f32{ 0.0, 0.0, 1.0, 0.0 },
    [_]f32{ 0.0, 0.0, 0.0, 1.0 },
};
test "multidimensional arrays" {
    // Access the 2D array by indexing the outer array, and then the inner array.
    try expect(mat4x4[1][1] == 1.0);

    // Here we iterate with for loops.
    for (mat4x4, 0..) |row, row_index| {
        for (row, 0..) |cell, column_index| {
            if (row_index == column_index) {
                try expect(cell == 1.0);
            }
        }
    }

    // initialize a multidimensional array to zeros
    const all_zero: [4][4]f32 = .{.{0} ** 4} ** 4;
    try expect(all_zero[0][0] == 0);
}
```

Shell

```
$ zig test test_multidimensional_arrays.zig
1/1 test_multidimensional_arrays.test.multidimensional arrays...OK
All 1 tests passed.
```

Sentinel-Terminated Arrays

The syntax `[N:x]T` describes an array which has a sentinel element of value `x` at the index corresponding to the length `N`.

test_null_terminated_array.zig

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

test "0-terminated sentinel array" {
    const array = [_:0]u8{ 1, 2, 3, 4 };

    try expect(@TypeOf(array) == [4:0]u8);
    try expect(array.len == 4);
    try expect(array[4] == 0);
}

test "extra 0s in 0-terminated sentinel array" {
    // The sentinel value may appear earlier, but does not influence the compile-time 'len'.
    const array = [_:0]u8{ 1, 0, 0, 4 };

    try expect(@TypeOf(array) == [4:0]u8);
    try expect(array.len == 4);
    try expect(array[4] == 0);
}
```

Shell

```
$ zig test test_null_terminated_array.zig
1/2 test_null_terminated_array.test.0-terminated sentinel array...OK
2/2 test_null_terminated_array.test.extra 0s in 0-terminated sentinel array...OK
All 2 tests passed.
```

See also:

- Sentinel-Terminated Pointers
- Sentinel-Terminated Slices

Destructuring Arrays

Arrays can be destructured:

destructuring_arrays.zig

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

fn swizzleRgbaToBgra(rgba: [4]u8) [4]u8 {
    // readable swizzling by destructuring
    const r, const g, const b, const a = rgba;
    return .{ b, g, r, a };
}

pub fn main() void {
```

```
    const pos = [_]i32{ 1, 2 };
    const x, const y = pos;
    print("x = {}, y = {}\n", .{x, y});

    const orange: [4]u8 = .{ 255, 165, 0, 255 };
    print("{any}\n", .{swizzleRgbaToBgra(orange)});
}
```

Shell

```
$ zig build-exe destructuring_arrays.zig
$ ./destructuring_arrays
x = 1, y = 2
{ 0, 165, 255, 255 }
```

See also:

- Destructuring
- Destructuring Tuples
- Destructuring Vectors

Vectors

A vector is a group of booleans, Integers, Floats, or Pointers which are operated on in parallel, using SIMD instructions if possible. Vector types are created with the builtin function @Vector.

Vectors support the same builtin operators as their underlying base types. These operations are performed element-wise, and return a vector of the same length as the input vectors. This includes:

- Arithmetic (`+`, `-`, `/`, `*`, `@divFloor`, `@sqrt`, `@ceil`, `@log`, etc.)
- Bitwise operators (`>>`, `<<`, `&`, `|`, `~`, etc.)
- Comparison operators (`<`, `>`, `==`, etc.)

It is prohibited to use a math operator on a mixture of scalars (individual numbers) and vectors. Zig provides the @splat builtin to easily convert from scalars to vectors, and it supports @reduce and array indexing syntax to convert from vectors to scalars. Vectors also support assignment to and from fixed-length arrays with comptime-known length.

For rearranging elements within and between vectors, Zig provides the @shuffle and @select functions.

Operations on vectors shorter than the target machine's native SIMD size will typically compile to single SIMD instructions, while vectors longer than the target machine's native SIMD size will compile to multiple SIMD instructions. If a given operation doesn't have SIMD support on the target architecture, the compiler will default to operating on each vector element one at a time.

Zig supports any comptime-known vector length up to 2^32-1, although small powers of two (2-64) are most typical. Note that excessively long vector lengths (e.g. 2^20) may result in compiler crashes on current versions of Zig.

test_vector.zig

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

test "Basic vector usage" {
    // Vectors have a compile-time known length and base type.
    const a = @Vector(4, i32){ 1, 2, 3, 4 };
    const b = @Vector(4, i32){ 5, 6, 7, 8 };

    // Math operations take place element-wise.
    const c = a + b;

    // Individual vector elements can be accessed using array indexing syntax.
    try expectEqual(6, c[0]);
    try expectEqual(8, c[1]);
    try expectEqual(10, c[2]);
    try expectEqual(12, c[3]);
}

test "Conversion between vectors, arrays, and slices" {
    // Vectors and fixed-length arrays can be automatically assigned back and forth
    const arr1: [4]f32 = [_]f32{ 1.1, 3.2, 4.5, 5.6 };
    const vec: @Vector(4, f32) = arr1;
    const arr2: [4]f32 = vec;
    try expectEqual(arr1, arr2);

    // You can also assign from a slice with comptime-known length to a vector using .*
    const vec2: @Vector(2, f32) = arr1[1..3].*;

    const slice: []const f32 = &arr1;
    var offset: u32 = 1; // var to make it runtime-known
    _ = &offset; // suppress 'var is never mutated' error
    // To extract a comptime-known length from a runtime-known offset,
    // first extract a new slice from the starting offset, then an array of
    // comptime-known length
    const vec3: @Vector(2, f32) = slice[offset..][0..2].*;
    try expectEqual(slice[offset], vec2[0]);
    try expectEqual(slice[offset + 1], vec2[1]);
    try expectEqual(vec2, vec3);
}
```

Shell

```
$ zig test test_vector.zig
1/2 test_vector.test.Basic vector usage...OK
2/2 test_vector.test.Conversion between vectors, arrays, and slices...OK
All 2 tests passed.
```

TODO talk about C ABI interop

TODO consider suggesting std.MultiArrayList

See also:

- @splat
- @shuffle
- @select
- @reduce

Destructuring Vectors

Vectors can be destructured:

destructuring_vectors.zig

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

// emulate punpckldq
pub fn unpack(x: @Vector(4, f32), y: @Vector(4, f32)) @Vector(4, f32) {
    const a, const c, _, _ = x;
    const b, const d, _, _ = y;
    return .{ a, b, c, d };
}

pub fn main() void {
    const x: @Vector(4, f32) = .{ 1.0, 2.0, 3.0, 4.0 };
    const y: @Vector(4, f32) = .{ 5.0, 6.0, 7.0, 8.0 };
    print("{}", .{unpack(x, y)});
}
```

Shell

```
$ zig build-exe destructuring_vectors.zig
$ ./destructuring_vectors
{ 1e0, 5e0, 2e0, 6e0 }
```

See also:

- Destructuring
- Destructuring Tuples
- Destructuring Arrays

Pointers

Zig has two kinds of pointers: single-item and many-item.

- `*T` - single-item pointer to exactly one item.
  - Supports deref syntax: `ptr.*`
  - Supports slice syntax: `ptr[0..1]`
  - Supports pointer subtraction: `ptr - ptr`

- `[*]T` - many-item pointer to unknown number of items.
  - Supports index syntax: `ptr[i]`
  - Supports slice syntax: `ptr[start..end]` and `ptr[start..]`
  - Supports pointer-integer arithmetic: `ptr + int`, `ptr - int`
  - Supports pointer subtraction: `ptr - ptr`

  `T` must have a known size, which means that it cannot be `anyopaque` or any other opaque type.

These types are closely related to Arrays and Slices:

- `*[N]T` - pointer to N items, same as single-item pointer to an array.
  - Supports index syntax: `array_ptr[i]`
  - Supports slice syntax: `array_ptr[start..end]`
  - Supports len property: `array_ptr.len`
  - Supports pointer subtraction: `array_ptr - array_ptr`

- `[]T` - is a slice (a fat pointer, which contains a pointer of type `[*]T` and a length).
  - Supports index syntax: `slice[i]`
  - Supports slice syntax: `slice[start..end]`
  - Supports len property: `slice.len`

Use `&x` to obtain a single-item pointer:

test_single_item_pointer.zig

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

test "address of syntax" {
    // Get the address of a variable:
    const x: i32 = 1234;
    const x_ptr = &x;

    // Dereference a pointer:
    try expect(x_ptr.* == 1234);

    // When you get the address of a const variable, you get a const single-item pointer.
    try expect(@TypeOf(x_ptr) == *const i32);

    // If you want to mutate the value, you'd need an address of a mutable variable:
    var y: i32 = 5678;
    const y_ptr = &y;
    try expect(@TypeOf(y_ptr) == *i32);
    y_ptr.* += 1;
    try expect(y_ptr.* == 5679);
}

test "pointer array access" {
    // Taking an address of an individual element gives a
    // single-item pointer. This kind of pointer
    // does not support pointer arithmetic.
```

```
    var array = [_]u8{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    const ptr = &array[2];
    try expect(@TypeOf(ptr) == *u8);

    try expect(array[2] == 3);
    ptr.* += 1;
    try expect(array[2] == 4);
}

test "slice syntax" {
    // Get a pointer to a variable:
    var x: i32 = 1234;
    const x_ptr = &x;

    // Convert to array pointer using slice syntax:
    const x_array_ptr = x_ptr[0..1];
    try expect(@TypeOf(x_array_ptr) == *[1]i32);

    // Coerce to many-item pointer:
    const x_many_ptr: [*]i32 = x_array_ptr;
    try expect(x_many_ptr[0] == 1234);
}
```

Shell

```
$ zig test test_single_item_pointer.zig
1/3 test_single_item_pointer.test.address of syntax...OK
2/3 test_single_item_pointer.test.pointer array access...OK
3/3 test_single_item_pointer.test.slice syntax...OK
All 3 tests passed.
```

Zig supports pointer arithmetic. It's better to assign the pointer to `[*]T` and increment that variable. For example, directly incrementing the pointer from a slice will corrupt it.

test_pointer_arithmetic.zig

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

test "pointer arithmetic with many-item pointer" {
    const array = [_]i32{ 1, 2, 3, 4 };
    var ptr: [*]const i32 = &array;

    try expect(ptr[0] == 1);
    ptr += 1;
    try expect(ptr[0] == 2);

    // slicing a many-item pointer without an end is equivalent to
    // pointer arithmetic: `ptr[start..] == ptr + start`
    try expect(ptr[1..] == ptr + 1);

    // subtraction between any two pointers except slices based on element size is supported
    try expect(&ptr[1] - &ptr[0] == 1);
}

test "pointer arithmetic with slices" {
    var array = [_]i32{ 1, 2, 3, 4 };
    var length: usize = 0; // var to make it runtime-known
```

```
    _ = &length; // suppress 'var is never mutated' error
    var slice = array[length..array.len];

    try expect(slice[0] == 1);
    try expect(slice.len == 4);

    slice.ptr += 1;
    // now the slice is in an bad state since len has not been updated

    try expect(slice[0] == 2);
    try expect(slice.len == 4);
}
```

Shell

```
$ zig test test_pointer_arithmetic.zig
1/2 test_pointer_arithmetic.test.pointer arithmetic with many-item pointer...OK
2/2 test_pointer_arithmetic.test.pointer arithmetic with slices...OK
All 2 tests passed.
```

In Zig, we generally prefer Slices rather than Sentinel-Terminated Pointers. You can turn an array or pointer into a slice using slice syntax.

Slices have bounds checking and are therefore protected against this kind of Illegal Behavior. This is one reason we prefer slices to pointers.

test_slice_bounds.zig

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

test "pointer slicing" {
    var array = [_]u8{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
    var start: usize = 2; // var to make it runtime-known
    _ = &start; // suppress 'var is never mutated' error
    const slice = array[start..4];
    try expect(slice.len == 2);

    try expect(array[3] == 4);
    slice[1] += 1;
    try expect(array[3] == 5);
}
```

Shell

```
$ zig test test_slice_bounds.zig
1/1 test_slice_bounds.test.pointer slicing...OK
All 1 tests passed.
```

Pointers work at compile-time too, as long as the code does not depend on an undefined memory layout:

test_comptime_pointers.zig

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

```
test "comptime pointers" {
    comptime {
        var x: i32 = 1;
        const ptr = &x;
        ptr.* += 1;
        x += 1;
        try expect(ptr.* == 3);
    }
}
```

Shell

```
$ zig test test_comptime_pointers.zig
1/1 test_comptime_pointers.test.comptime pointers...OK
All 1 tests passed.
```

To convert an integer address into a pointer, use `@ptrFromInt`. To convert a pointer to an integer, use `@intFromPtr`:

test_integer_pointer_conversion.zig

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

test "@intFromPtr and @ptrFromInt" {
    const ptr: *i32 = @ptrFromInt(0xdeadbee0);
    const addr = @intFromPtr(ptr);
    try expect(@TypeOf(addr) == usize);
    try expect(addr == 0xdeadbee0);
}
```

Shell

```
$ zig test test_integer_pointer_conversion.zig
1/1 test_integer_pointer_conversion.test.@intFromPtr and @ptrFromInt...OK
All 1 tests passed.
```

Zig is able to preserve memory addresses in comptime code, as long as the pointer is never dereferenced:

test_comptime_pointer_conversion.zig

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

test "comptime @ptrFromInt" {
    comptime {
        // Zig is able to do this at compile-time, as long as
        // ptr is never dereferenced.
        const ptr: *i32 = @ptrFromInt(0xdeadbee0);
        const addr = @intFromPtr(ptr);
        try expect(@TypeOf(addr) == usize);
        try expect(addr == 0xdeadbee0);
    }
}
```

Shell

```
$ zig test test_comptime_pointer_conversion.zig
1/1 test_comptime_pointer_conversion.test.comptime @ptrFromInt...OK
All 1 tests passed.
```

@ptrCast converts a pointer's element type to another. This creates a new pointer that can cause undetectable Illegal Behavior depending on the loads and stores that pass through it. Generally, other kinds of type conversions are preferable to `@ptrCast` if possible.

test_pointer_casting.zig

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

test "pointer casting" {
    const bytes align(@alignOf(u32)) = [_]u8{ 0x12, 0x12, 0x12, 0x12 };
    const u32_ptr: *const u32 = @ptrCast(&bytes);
    try expect(u32_ptr.* == 0x12121212);

    // Even this example is contrived - there are better ways to do the above than
    // pointer casting. For example, using a slice narrowing cast:
    const u32_value = std.mem.bytesAsSlice(u32, bytes[0..])[0];
    try expect(u32_value == 0x12121212);

    // And even another way, the most straightforward way to do it:
    try expect(@as(u32, @bitCast(bytes)) == 0x12121212);
}

test "pointer child type" {
    // pointer types have a `child` field which tells you the type they point to.
    try expect(@typeInfo(*u32).pointer.child == u32);
}
```

Shell

```
$ zig test test_pointer_casting.zig
1/2 test_pointer_casting.test.pointer casting...OK
2/2 test_pointer_casting.test.pointer child type...OK
All 2 tests passed.
```

See also:

- Optional Pointers
- @ptrFromInt
- @intFromPtr
- C Pointers

volatile

Loads and stores are assumed to not have side effects. If a given load or store should have side effects, such as Memory Mapped Input/Output (MMIO), use **`volatile`**. In the following code, loads and stores with `mmio_ptr` are guaranteed to all happen and in the same order as in source code:

test_volatile.zig

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

test "volatile" {
    const mmio_ptr: *volatile u8 = @ptrFromInt(0x12345678);
    try expect(@TypeOf(mmio_ptr) == *volatile u8);
}
```

Shell

```
$ zig test test_volatile.zig
1/1 test_volatile.test.volatile...OK
All 1 tests passed.
```

Note that `volatile` is unrelated to concurrency and Atomics. If you see code that is using `volatile` for something other than Memory Mapped Input/Output, it is probably a bug.

Alignment

Each type has an **alignment** - a number of bytes such that, when a value of the type is loaded from or stored to memory, the memory address must be evenly divisible by this number. You can use @alignOf to find out this value for any type.

Alignment depends on the CPU architecture, but is always a power of two, and less than `1 << 29`.

In Zig, a pointer type has an alignment value. If the value is equal to the alignment of the underlying type, it can be omitted from the type:

test_variable_alignment.zig

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

test "variable alignment" {
    var x: i32 = 1234;
    const align_of_i32 = @alignOf(@TypeOf(x));
    try expect(@TypeOf(&x) == *i32);
    try expect(*i32 == *align(align_of_i32) i32);
    if (builtin.target.cpu.arch == .x86_64) {
        try expect(@typeInfo(*i32).pointer.alignment == 4);
    }
}
```

Shell

```
$ zig test test_variable_alignment.zig
1/1 test_variable_alignment.test.variable alignment...OK
All 1 tests passed.
```

In the same way that a `*i32` can be coerced to a `*const i32`, a pointer with a larger alignment can be implicitly cast to a pointer with a smaller alignment, but not vice versa.

You can specify alignment on variables and functions. If you do this, then pointers to them get the specified alignment:

test_variable_func_alignment.zig

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

var foo: u8 align(4) = 100;

test "global variable alignment" {
    try expect(@typeInfo(@TypeOf(&foo)).pointer.alignment == 4);
    try expect(@TypeOf(&foo) == *align(4) u8);
    const as_pointer_to_array: *align(4) [1]u8 = &foo;
    const as_slice: []align(4) u8 = as_pointer_to_array;
    const as_unaligned_slice: []u8 = as_slice;
    try expect(as_unaligned_slice[0] == 100);
}

fn derp() align(@sizeOf(usize) * 2) i32 {
    return 1234;
}
fn noop1() align(1) void {}
fn noop4() align(4) void {}

test "function alignment" {
    try expect(derp() == 1234);
    try expect(@TypeOf(derp) == fn () i32);
    try expect(@TypeOf(&derp) == *align(@sizeOf(usize) * 2) const fn () i32);

    noop1();
    try expect(@TypeOf(noop1) == fn () void);
    try expect(@TypeOf(&noop1) == *align(1) const fn () void);

    noop4();
    try expect(@TypeOf(noop4) == fn () void);
    try expect(@TypeOf(&noop4) == *align(4) const fn () void);
}
```

Shell

```
$ zig test test_variable_func_alignment.zig
1/2 test_variable_func_alignment.test.global variable alignment...OK
2/2 test_variable_func_alignment.test.function alignment...OK
All 2 tests passed.
```

If you have a pointer or a slice that has a small alignment, but you know that it actually has a bigger alignment, use @alignCast to change the pointer into a more aligned pointer. This is a no-op at runtime, but inserts a safety check:

test_incorrect_pointer_alignment.zig

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

```
test "pointer alignment safety" {
    var array align(4) = [_]u32{ 0x11111111, 0x11111111 };
    const bytes = std.mem.sliceAsBytes(array[0..]);
    try std.testing.expect(foo(bytes) == 0x11111111);
}
fn foo(bytes: []u8) u32 {
    const slice4 = bytes[1..5];
    const int_slice = std.mem.bytesAsSlice(u32, @as([]align(4) u8, @alignCast(slice4)));
    return int_slice[0];
}
```

Shell

```
$ zig test test_incorrect_pointer_alignment.zig
1/1 test_incorrect_pointer_alignment.test.pointer alignment safety...thread 211638 panic: incor
/home/andy/src/zig/doc/langref/test_incorrect_pointer_alignment.zig:10:68: 0x1048c12 in foo (te
    const int_slice = std.mem.bytesAsSlice(u32, @as([]align(4) u8, @alignCast(slice4)));
                                                                   ^
/home/andy/src/zig/doc/langref/test_incorrect_pointer_alignment.zig:6:31: 0x1048abf in test.poi
    try std.testing.expect(foo(bytes) == 0x11111111);
                              ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10ef185 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10e771d in main (test)
        return mainTerminal();
                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10e6ca2 in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e687d in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/08ad046baf4e682f3eb12b3ecaa07a72/test --seed=0xaf095082
```

allowzero

This pointer attribute allows a pointer to have address zero. This is only ever needed on the freestanding OS target, where the address zero is mappable. If you want to represent null pointers, use Optional Pointers instead. Optional Pointers with `allowzero` are not the same size as pointers. In this code example, if the pointer did not have the `allowzero` attribute, this would be a Pointer Cast Invalid Null panic:

test_allowzero.zig

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

test "allowzero" {
    var zero: usize = 0; // var to make to runtime-known
    _ = &zero; // suppress 'var is never mutated' error
```

```
    const ptr: *allowzero i32 = @ptrFromInt(zero);
    try expect(@intFromPtr(ptr) == 0);
}
```

Shell

```
$ zig test test_allowzero.zig
1/1 test_allowzero.test.allowzero...OK
All 1 tests passed.
```

Sentinel-Terminated Pointers

The syntax `[*:x]T` describes a pointer that has a length determined by a sentinel value. This provides protection against buffer overflow and overreads.

sentinel-terminated_pointer.zig

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

// This is also available as `std.c.printf`.
pub extern "c" fn printf(format: [*:0]const u8, ...) c_int;

pub fn main() anyerror!void {
    _ = printf("Hello, world!\n"); // OK

    const msg = "Hello, world!\n";
    const non_null_terminated_msg: [msg.len]u8 = msg.*;
    _ = printf(&non_null_terminated_msg);
}
```

Shell

```
$ zig build-exe sentinel-terminated_pointer.zig -lc
/home/andy/src/zig/doc/langref/sentinel-terminated_pointer.zig:11:16: error: expected type '[*:
    _ = printf(&non_null_terminated_msg);
               ^~~~~~~~~~~~~~~~~~~~~~~~
/home/andy/src/zig/doc/langref/sentinel-terminated_pointer.zig:11:16: note: destination pointer
/home/andy/src/zig/doc/langref/sentinel-terminated_pointer.zig:4:34: note: parameter type decla
pub extern "c" fn printf(format: [*:0]const u8, ...) c_int;
                                 ^~~~~~~~~~~~~
referenced by:
    main: /home/andy/src/zig/lib/std/start.zig:656:37
    comptime: /home/andy/src/zig/lib/std/start.zig:58:30
    2 reference(s) hidden; use '-freference-trace=4' to see all references
```

See also:

- Sentinel-Terminated Slices
- Sentinel-Terminated Arrays

Slices

A slice is a pointer and a length. The difference between an array and a slice is that the array's length is part of the type and known at compile-time, whereas the slice's length is known at runtime. Both can be accessed with the `len` field.

test_basic_slices.zig

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

test "basic slices" {
    var array = [_]i32{ 1, 2, 3, 4 };
    var known_at_runtime_zero: usize = 0;
    _ = &known_at_runtime_zero;
    const slice = array[known_at_runtime_zero..array.len];

    // alternative initialization using result location
    const alt_slice: []const i32 = &.{ 1, 2, 3, 4 };

    try expectEqualSlices(i32, slice, alt_slice);

    try expect(@TypeOf(slice) == []i32);
    try expect(&slice[0] == &array[0]);
    try expect(slice.len == array.len);

    // If you slice with comptime-known start and end positions, the result is
    // a pointer to an array, rather than a slice.
    const array_ptr = array[0..array.len];
    try expect(@TypeOf(array_ptr) == *[array.len]i32);

    // You can perform a slice-by-length by slicing twice. This allows the compiler
    // to perform some optimisations like recognising a comptime-known length when
    // the start position is only known at runtime.
    var runtime_start: usize = 1;
    _ = &runtime_start;
    const length = 2;
    const array_ptr_len = array[runtime_start..][0..length];
    try expect(@TypeOf(array_ptr_len) == *[length]i32);

    // Using the address-of operator on a slice gives a single-item pointer.
    try expect(@TypeOf(&slice[0]) == *i32);
    // Using the `ptr` field gives a many-item pointer.
    try expect(@TypeOf(slice.ptr) == [*]i32);
    try expect(@intFromPtr(slice.ptr) == @intFromPtr(&slice[0]));

    // Slices have array bounds checking. If you try to access something out
    // of bounds, you'll get a safety check failure:
    slice[10] += 1;

    // Note that `slice.ptr` does not invoke safety checking, while `&slice[0]`
    // asserts that the slice has len > 0.

    // Empty slices can be created like this:
    const empty1 = &[0]u8{};
    // If the type is known you can use this short hand:
    const empty2: []u8 = &.{};
    try expect(empty1.len == 0);
```

```
    try expect(empty2.len == 0);

    // A zero-length initialization can always be used to create an empty slice, even if the sl
    // This is because the pointed-to data is zero bits long, so its immutability is irrelevant
}
```

Shell

```
$ zig test test_basic_slices.zig
1/1 test_basic_slices.test.basic slices...thread 222617 panic: index out of bounds: index 10, l
/home/andy/src/zig/doc/langref/test_basic_slices.zig:41:10: 0x104b5a1 in test.basic slices (tes
    slice[10] += 1;
         ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10f2645 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10eabdd in main (test)
        return mainTerminal();
                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10ea162 in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e9d3d in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/d7a037b9b130e4363980ed7df085d9dc/test --seed=0x830cf24c
```

This is one reason we prefer slices to pointers.

test_slices.zig

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

test "using slices for strings" {
    // Zig has no concept of strings. String literals are const pointers
    // to null-terminated arrays of u8, and by convention parameters
    // that are "strings" are expected to be UTF-8 encoded slices of u8.
    // Here we coerce *const [5:0]u8 and *const [6:0]u8 to []const u8
    const hello: []const u8 = "hello";
    const world: []const u8 = "世界";

    var all_together: [100]u8 = undefined;
    // You can use slice syntax with at least one runtime-known index on an
    // array to convert an array into a slice.
    var start: usize = 0;
    _ = &start;
    const all_together_slice = all_together[start..];
    // String concatenation example.
    const hello_world = try fmt.bufPrint(all_together_slice, "{s} {s}", .{ hello, world });

    // Generally, you can use UTF-8 and not worry about whether something is a
    // string. If you don't need to deal with individual characters, no need
```

```
    // to decode.
    try expect(mem.eql(u8, hello_world, "hello 世界"));
}

test "slice pointer" {
    var array: [10]u8 = undefined;
    const ptr = &array;
    try expect(@TypeOf(ptr) == *[10]u8);

    // A pointer to an array can be sliced just like an array:
    var start: usize = 0;
    var end: usize = 5;
    _ = .{ &start, &end };
    const slice = ptr[start..end];
    // The slice is mutable because we sliced a mutable pointer.
    try expect(@TypeOf(slice) == []u8);
    slice[2] = 3;
    try expect(array[2] == 3);

    // Again, slicing with comptime-known indexes will produce another pointer
    // to an array:
    const ptr2 = slice[2..3];
    try expect(ptr2.len == 1);
    try expect(ptr2[0] == 3);
    try expect(@TypeOf(ptr2) == *[1]u8);
}
```

Shell

```
$ zig test test_slices.zig
1/2 test_slices.test.using slices for strings...OK
2/2 test_slices.test.slice pointer...OK
All 2 tests passed.
```

See also:

- Pointers
- for
- Arrays

Sentinel-Terminated Slices

The syntax `[:x]T` is a slice which has a runtime-known length and also guarantees a sentinel value at the element indexed by the length. The type does not guarantee that there are no sentinel elements before that. Sentinel-terminated slices allow element access to the `len` index.

test_null_terminated_slice.zig

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

test "0-terminated slice" {
    const slice: [:0]const u8 = "hello";
```

```
    try expect(slice.len == 5);
    try expect(slice[5] == 0);
}
```

Shell

```
$ zig test test_null_terminated_slice.zig
1/1 test_null_terminated_slice.test.0-terminated slice...OK
All 1 tests passed.
```

Sentinel-terminated slices can also be created using a variation of the slice syntax `data[start..end :x]`, where `data` is a many-item pointer, array or slice and `x` is the sentinel value.

test_null_terminated_slicing.zig

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

test "0-terminated slicing" {
    var array = [_]u8{ 3, 2, 1, 0, 3, 2, 1, 0 };
    var runtime_length: usize = 3;
    _ = &runtime_length;
    const slice = array[0..runtime_length :0];

    try expect(@TypeOf(slice) == [:0]u8);
    try expect(slice.len == 3);
}
```

Shell

```
$ zig test test_null_terminated_slicing.zig
1/1 test_null_terminated_slicing.test.0-terminated slicing...OK
All 1 tests passed.
```

Sentinel-terminated slicing asserts that the element in the sentinel position of the backing data is actually the sentinel value. If this is not the case, safety-checked [Illegal Behavior](#) results.

test_sentinel_mismatch.zig

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

test "sentinel mismatch" {
    var array = [_]u8{ 3, 2, 1, 0 };

    // Creating a sentinel-terminated slice from the array with a length of 2
    // will result in the value `1` occupying the sentinel element position.
    // This does not match the indicated sentinel value of `0` and will lead
    // to a runtime panic.
    var runtime_length: usize = 2;
    _ = &runtime_length;
    const slice = array[0..runtime_length :0];

    _ = slice;
```

```
}
```

Shell

```
$ zig test test_sentinel_mismatch.zig
1/1 test_sentinel_mismatch.test.sentinel mismatch...thread 212416 panic: sentinel mismatch: exp
/home/andy/src/zig/doc/langref/test_sentinel_mismatch.zig:13:24: 0x1048af1 in test.sentinel mis
    const slice = array[0..runtime_length :0];
                       ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10eeee5 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10e747d in main (test)
        return mainTerminal();
                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10e6a02 in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e65dd in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/9dfcdd59e16f1035ebfa67da73c4ad20/test --seed=0xb1e7629e
```

See also:

- Sentinel-Terminated Pointers
- Sentinel-Terminated Arrays

struct

test_structs.zig

```
// Declare a struct.
// Zig gives no guarantees about the order of fields and the size of
// the struct but the fields are guaranteed to be ABI-aligned.
const Point = struct {
    x: f32,
    y: f32,
};

// Declare an instance of a struct.
const p: Point = .{
    .x = 0.12,
    .y = 0.34,
};

// Functions in the struct's namespace can be called with dot syntax.
const Vec3 = struct {
    x: f32,
    y: f32,
    z: f32,

    pub fn init(x: f32, y: f32, z: f32) Vec3 {
```

```
        return Vec3{
            .x = x,
            .y = y,
            .z = z,
        };
    }

    pub fn dot(self: Vec3, other: Vec3) f32 {
        return self.x * other.x + self.y * other.y + self.z * other.z;
    }
};

test "dot product" {
    const v1 = Vec3.init(1.0, 0.0, 0.0);
    const v2 = Vec3.init(0.0, 1.0, 0.0);
    try expect(v1.dot(v2) == 0.0);

    // Other than being available to call with dot syntax, struct methods are
    // not special. You can reference them as any other declaration inside
    // the struct:
    try expect(Vec3.dot(v1, v2) == 0.0);
}

// Structs can have declarations.
// Structs can have 0 fields.
const Empty = struct {
    pub const PI = 3.14;
};
test "struct namespaced variable" {
    try expect(Empty.PI == 3.14);
    try expect(@sizeOf(Empty) == 0);

    // Empty structs can be instantiated the same as usual.
    const does_nothing: Empty = .{};

    _ = does_nothing;
}

// Struct field order is determined by the compiler, however, a base pointer
// can be computed from a field pointer:
fn setYBasedOnX(x: *f32, y: f32) void {
    const point: *Point = @fieldParentPtr("x", x);
    point.y = y;
}
test "field parent pointer" {
    var point = Point{
        .x = 0.1234,
        .y = 0.5678,
    };
    setYBasedOnX(&point.x, 0.9);
    try expect(point.y == 0.9);
}

// Structs can be returned from functions.
fn LinkedList(comptime T: type) type {
    return struct {
        pub const Node = struct {
```

```
            prev: ?*Node,
            next: ?*Node,
            data: T,
        };

        first: ?*Node,
        last: ?*Node,
        len: usize,
    };
}

test "linked list" {
    // Functions called at compile-time are memoized.
    try expect(LinkedList(i32) == LinkedList(i32));

    const list = LinkedList(i32){
        .first = null,
        .last = null,
        .len = 0,
    };
    try expect(list.len == 0);

    // Since types are first class values you can instantiate the type
    // by assigning it to a variable:
    const ListOfInts = LinkedList(i32);
    try expect(ListOfInts == LinkedList(i32));

    var node = ListOfInts.Node{
        .prev = null,
        .next = null,
        .data = 1234,
    };
    const list2 = LinkedList(i32){
        .first = &node,
        .last = &node,
        .len = 1,
    };

    // When using a pointer to a struct, fields can be accessed directly,
    // without explicitly dereferencing the pointer.
    // So you can do
    try expect(list2.first.?.data == 1234);
    // instead of try expect(list2.first.?.*.data == 1234);
}

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

Shell

```
$ zig test test_structs.zig
1/4 test_structs.test.dot product...OK
2/4 test_structs.test.struct namespaced variable...OK
3/4 test_structs.test.field parent pointer...OK
4/4 test_structs.test.linked list...OK
All 4 tests passed.
```

Default Field Values

Each struct field may have an expression indicating the default field value. Such expressions are executed at comptime, and allow the field to be omitted in a struct literal expression:

struct_default_field_values.zig

```
const Foo = struct {
    a: i32 = 1234,
    b: i32,
};

test "default struct initialization fields" {
    const x: Foo = .{
        .b = 5,
    };
    if (x.a + x.b != 1239) {
        comptime unreachable;
    }
}
```

Shell

```
$ zig test struct_default_field_values.zig
1/1 struct_default_field_values.test.default struct initialization fields...OK
All 1 tests passed.
```

Faulty Default Field Values

Default field values are only appropriate when the data invariants of a struct cannot be violated by omitting that field from an initialization.

For example, here is an inappropriate use of default struct field initialization:

bad_default_value.zig

```
const Threshold = struct {
    minimum: f32 = 0.25,
    maximum: f32 = 0.75,

    const Category = enum { low, medium, high };

    fn categorize(t: Threshold, value: f32) Category {
        assert(t.maximum >= t.minimum);
        if (value < t.minimum) return .low;
        if (value > t.maximum) return .high;
        return .medium;
    }
};

pub fn main() !void {
    var threshold: Threshold = .{
        .maximum = 0.20,
    };
```

```
    const category = threshold.categorize(0.90);
    try std.io.getStdOut().writeAll(@tagName(category));
}

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

Shell

```
$ zig build-exe bad_default_value.zig
$ ./bad_default_value
thread 216489 panic: reached unreachable code
/home/andy/src/zig/lib/std/debug.zig:522:14: 0x1048e4d in assert (bad_default_value)
    if (!ok) unreachable; // assertion failure
             ^
/home/andy/src/zig/doc/langref/bad_default_value.zig:8:15: 0x10de7b9 in categorize (bad_default
        assert(t.maximum >= t.minimum);
              ^
/home/andy/src/zig/doc/langref/bad_default_value.zig:19:42: 0x10de6fa in main (bad_default_valu
    const category = threshold.categorize(0.90);
                                         ^
/home/andy/src/zig/lib/std/start.zig:656:37: 0x10de60a in posixCallMainAndExit (bad_default_val
            const result = root.main() catch |err| {
                                    ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10de1bd in _start (bad_default_value)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Above you can see the danger of ignoring this principle. The default field values caused the data invariant to be violated, causing illegal behavior.

To fix this, remove the default values from all the struct fields, and provide a named default value:

struct_default_value.zig

```
const Threshold = struct {
    minimum: f32,
    maximum: f32,

    const default: Threshold = .{
        .minimum = 0.25,
        .maximum = 0.75,
    };
};
```

If a struct value requires a runtime-known value in order to be initialized without violating data invariants, then use an initialization method that accepts those runtime values, and populates the remaining fields.

extern struct

An `extern struct` has in-memory layout matching the C ABI for the target.

If well-defined in-memory layout is not required, struct is a better choice because it places fewer restrictions on the compiler.

See packed struct for a struct that has the ABI of its backing integer, which can be useful for modeling flags.

See also:

- extern union
- extern enum

packed struct

Unlike normal structs, `packed` structs have guaranteed in-memory layout:

- Fields remain in the order declared, least to most significant.
- There is no padding between fields.
- Zig supports arbitrary width Integers and although normally, integers with fewer than 8 bits will still use 1 byte of memory, in packed structs, they use exactly their bit width.
- `bool` fields use exactly 1 bit.
- An enum field uses exactly the bit width of its integer tag type.
- A packed union field uses exactly the bit width of the union field with the largest bit width.
- Packed structs support equality operators.

This means that a `packed struct` can participate in a @bitCast or a @ptrCast to reinterpret memory. This even works at comptime:

test_packed_structs.zig

```
const std = @import("std");
const native_endian = @import("builtin").target.cpu.arch.endian();
const expect = std.testing.expect;

const Full = packed struct {
    number: u16,
};
const Divided = packed struct {
    half1: u8,
    quarter3: u4,
    quarter4: u4,
};

test "@bitCast between packed structs" {
    try doTheTest();
    try comptime doTheTest();
}
```

```
fn doTheTest() !void {
    try expect(@sizeOf(Full) == 2);
    try expect(@sizeOf(Divided) == 2);
    const full = Full{ .number = 0x1234 };
    const divided: Divided = @bitCast(full);
    try expect(divided.half1 == 0x34);
    try expect(divided.quarter3 == 0x2);
    try expect(divided.quarter4 == 0x1);

    const ordered: [2]u8 = @bitCast(full);
    switch (native_endian) {
        .big => {
            try expect(ordered[0] == 0x12);
            try expect(ordered[1] == 0x34);
        },
        .little => {
            try expect(ordered[0] == 0x34);
            try expect(ordered[1] == 0x12);
        },
    }
}
```

Shell

```
$ zig test test_packed_structs.zig
1/1 test_packed_structs.test.@bitCast between packed structs...OK
All 1 tests passed.
```

The backing integer is inferred from the fields' total bit width. Optionally, it can be explicitly provided and enforced at compile time:

test_missized_packed_struct.zig

```
test "missized packed struct" {
    const S = packed struct(u32) { a: u16, b: u8 };
    _ = S{ .a = 4, .b = 2 };
}
```

Shell

```
$ zig test test_missized_packed_struct.zig
doc/langref/test_missized_packed_struct.zig:2:29: error: backing integer type 'u32' has bit siz
    const S = packed struct(u32) { a: u16, b: u8 };
                            ^~~
```

Zig allows the address to be taken of a non-byte-aligned field:

test_pointer_to_non-byte_aligned_field.zig

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

const BitField = packed struct {
    a: u3,
    b: u3,
```

```
    c: u2,
};

var foo = BitField{
    .a = 1,
    .b = 2,
    .c = 3,
};

test "pointer to non-byte-aligned field" {
    const ptr = &foo.b;
    try expect(ptr.* == 2);
}
```

Shell

```
$ zig test test_pointer_to_non-byte_aligned_field.zig
1/1 test_pointer_to_non-byte_aligned_field.test.pointer to non-byte-aligned field...OK
All 1 tests passed.
```

However, the pointer to a non-byte-aligned field has special properties and cannot be passed when a normal pointer is expected:

test_misaligned_pointer.zig

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

const BitField = packed struct {
    a: u3,
    b: u3,
    c: u2,
};

var bit_field = BitField{
    .a = 1,
    .b = 2,
    .c = 3,
};

test "pointer to non-byte-aligned field" {
    try expect(bar(&bit_field.b) == 2);
}

fn bar(x: *const u3) u3 {
    return x.*;
}
```

Shell

```
$ zig test test_misaligned_pointer.zig
doc/langref/test_misaligned_pointer.zig:17:20: error: expected type '*const u3', found '*align(
    try expect(bar(&bit_field.b) == 2);
                   ^~~~~~~~~~~~
doc/langref/test_misaligned_pointer.zig:17:20: note: pointer host size '1' cannot cast into poi
doc/langref/test_misaligned_pointer.zig:17:20: note: pointer bit offset '3' cannot cast into po
doc/langref/test_misaligned_pointer.zig:20:11: note: parameter type declared here
```

```
fn bar(x: *const u3) u3 {
          ^~~~~~~~~
```

In this case, the function `bar` cannot be called because the pointer to the non-ABI-aligned field mentions the bit offset, but the function expects an ABI-aligned pointer.

Pointers to non-ABI-aligned fields share the same address as the other fields within their host integer:

test_packed_struct_field_address.zig

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

const BitField = packed struct {
    a: u3,
    b: u3,
    c: u2,
};

var bit_field = BitField{
    .a = 1,
    .b = 2,
    .c = 3,
};

test "pointers of sub-byte-aligned fields share addresses" {
    try expect(@intFromPtr(&bit_field.a) == @intFromPtr(&bit_field.b));
    try expect(@intFromPtr(&bit_field.a) == @intFromPtr(&bit_field.c));
}
```

Shell

```
$ zig test test_packed_struct_field_address.zig
1/1 test_packed_struct_field_address.test.pointers of sub-byte-aligned fields share addresses..
All 1 tests passed.
```

This can be observed with @bitOffsetOf and offsetOf:

test_bitOffsetOf_offsetOf.zig

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

const BitField = packed struct {
    a: u3,
    b: u3,
    c: u2,
};

test "offsets of non-byte-aligned fields" {
    comptime {
        try expect(@bitOffsetOf(BitField, "a") == 0);
        try expect(@bitOffsetOf(BitField, "b") == 3);
```

```
        try expect(@bitOffsetOf(BitField, "c") == 6);

        try expect(@offsetOf(BitField, "a") == 0);
        try expect(@offsetOf(BitField, "b") == 0);
        try expect(@offsetOf(BitField, "c") == 0);
    }
}
```

Shell

```
$ zig test test_bitOffsetOf_offsetOf.zig
1/1 test_bitOffsetOf_offsetOf.test.offsets of non-byte-aligned fields...OK
All 1 tests passed.
```

Packed structs have the same alignment as their backing integer, however, overaligned pointers to packed structs can override this:

test_overaligned_packed_struct.zig

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

const S = packed struct {
    a: u32,
    b: u32,
};
test "overaligned pointer to packed struct" {
    var foo: S align(4) = .{ .a = 1, .b = 2 };
    const ptr: *align(4) S = &foo;
    const ptr_to_b: *u32 = &ptr.b;
    try expect(ptr_to_b.* == 2);
}
```

Shell

```
$ zig test test_overaligned_packed_struct.zig
1/1 test_overaligned_packed_struct.test.overaligned pointer to packed struct...OK
All 1 tests passed.
```

It's also possible to set alignment of struct fields:

test_aligned_struct_fields.zig

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

test "aligned struct fields" {
    const S = struct {
        a: u32 align(2),
        b: u32 align(64),
    };
    var foo = S{ .a = 1, .b = 2 };

    try expectEqual(64, @alignOf(S));
    try expectEqual(*align(2) u32, @TypeOf(&foo.a));
    try expectEqual(*align(64) u32, @TypeOf(&foo.b));
}
```

Shell

```
$ zig test test_aligned_struct_fields.zig
1/1 test_aligned_struct_fields.test.aligned struct fields...OK
All 1 tests passed.
```

Equating packed structs results in a comparison of the backing integer, and only works for the `==` and `!=` operators.

test_packed_struct_equality.zig

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

test "packed struct equality" {
    const S = packed struct {
        a: u4,
        b: u4,
    };
    const x: S = .{ .a = 1, .b = 2 };
    const y: S = .{ .b = 2, .a = 1 };
    try expect(x == y);
}
```

Shell

```
$ zig test test_packed_struct_equality.zig
1/1 test_packed_struct_equality.test.packed struct equality...OK
All 1 tests passed.
```

Using packed structs with [volatile](https://ziglang.org/documentation/0.14.0/) is problematic, and may be a compile error in the future. For details on this subscribe to [this issue](https://ziglang.org/documentation/0.14.0/). TODO update these docs with a recommendation on how to use packed structs with MMIO (the use case for volatile packed structs) once this issue is resolved. Don't worry, there will be a good solution for this use case in zig.

Struct Naming

Since all structs are anonymous, Zig infers the type name based on a few rules.

- If the struct is in the initialization expression of a variable, it gets named after that variable.
- If the struct is in the `return` expression, it gets named after the function it is returning from, with the parameter values serialized.
- Otherwise, the struct gets a name such as `(filename.funcname__struct_ID)`.
- If the struct is declared inside another struct, it gets named after both the parent struct and the name inferred by the previous rules, separated by a dot.

struct_name.zig

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

pub fn main() void {
    const Foo = struct {};
    std.debug.print("variable: {s}\n", .{@typeName(Foo)});
    std.debug.print("anonymous: {s}\n", .{@typeName(struct {})});
    std.debug.print("function: {s}\n", .{@typeName(List(i32))});
}

fn List(comptime T: type) type {
    return struct {
        x: T,
    };
}
```

Shell

```
$ zig build-exe struct_name.zig
$ ./struct_name
variable: struct_name.main.Foo
anonymous: struct_name.main__struct_24143
function: struct_name.List(i32)
```

Anonymous Struct Literals

Zig allows omitting the struct type of a literal. When the result is coerced, the struct literal will directly instantiate the result location, with no copy:

test_struct_result.zig

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

const Point = struct { x: i32, y: i32 };

test "anonymous struct literal" {
    const pt: Point = .{
        .x = 13,
        .y = 67,
    };
    try expect(pt.x == 13);
    try expect(pt.y == 67);
}
```

Shell

```
$ zig test test_struct_result.zig
1/1 test_struct_result.test.anonymous struct literal...OK
All 1 tests passed.
```

The struct type can be inferred. Here the result location does not include a type, and so Zig infers the type:

test_anonymous_struct.zig

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

test "fully anonymous struct" {
    try check(.{
        .int = @as(u32, 1234),
        .float = @as(f64, 12.34),
        .b = true,
        .s = "hi",
    });
}

fn check(args: anytype) !void {
    try expect(args.int == 1234);
    try expect(args.float == 12.34);
    try expect(args.b);
    try expect(args.s[0] == 'h');
    try expect(args.s[1] == 'i');
}
```

Shell

```
$ zig test test_anonymous_struct.zig
1/1 test_anonymous_struct.test.fully anonymous struct...OK
All 1 tests passed.
```

Tuples

Anonymous structs can be created without specifying field names, and are referred to as "tuples". An empty tuple looks like `.{}` and can be seen in one of the Hello World examples.

The fields are implicitly named using numbers starting from 0. Because their names are integers, they cannot be accessed with `.` syntax without also wrapping them in `@""`. Names inside `@""` are always recognised as identifiers.

Like arrays, tuples have a .len field, can be indexed (provided the index is comptime-known) and work with the ++ and ** operators. They can also be iterated over with inline for.

test_tuples.zig

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

```
test "tuple" {
    const values = .{
        @as(u32, 1234),
        @as(f64, 12.34),
        true,
        "hi",
    } ++ .{false} ** 2;
    try expect(values[0] == 1234);
    try expect(values[4] == false);
    inline for (values, 0..) |v, i| {
        if (i != 2) continue;
        try expect(v);
    }
    try expect(values.len == 6);
    try expect(values.@"3"[0] == 'h');
}
```

Shell

```
$ zig test test_tuples.zig
1/1 test_tuples.test.tuple...OK
All 1 tests passed.
```

Destructuring Tuples

Tuples can be destructured.

Tuple destructuring is helpful for returning multiple values from a block:

destructuring_block.zig

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

pub fn main() void {
    const digits = [_]i8 { 3, 8, 9, 0, 7, 4, 1 };

    const min, const max = blk: {
        var min: i8 = 127;
        var max: i8 = -128;

        for (digits) |digit| {
            if (digit < min) min = digit;
            if (digit > max) max = digit;
        }

        break :blk .{ min, max };
    };

    print("min = {}", .{ min });
    print("max = {}", .{ max });
}
```

Shell

```
$ zig build-exe destructuring_block.zig
```

```
$ ./destructuring_block
min = 0max = 9
```

Tuple destructuring is helpful for dealing with functions and built-ins that return multiple values as a tuple:

destructuring_return_value.zig

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

fn divmod(numerator: u32, denominator: u32) struct { u32, u32 } {
    return .{ numerator / denominator, numerator % denominator };
}

pub fn main() void {
    const div, const mod = divmod(10, 3);

    print("10 / 3 = {}\n", .{div});
    print("10 % 3 = {}\n", .{mod});
}
```

Shell

```
$ zig build-exe destructuring_return_value.zig
$ ./destructuring_return_value
10 / 3 = 3
10 % 3 = 1
```

See also:

- Destructuring
- Destructuring Arrays
- Destructuring Vectors

See also:

- comptime
- @fieldParentPtr

enum

test_enums.zig

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

// Declare an enum.
const Type = enum {
    ok,
    not_ok,
};
```

```
// Declare a specific enum field.
const c = Type.ok;

// If you want access to the ordinal value of an enum, you
// can specify the tag type.
const Value = enum(u2) {
    zero,
    one,
    two,
};
// Now you can cast between u2 and Value.
// The ordinal value starts from 0, counting up by 1 from the previous member.
test "enum ordinal value" {
    try expect(@intFromEnum(Value.zero) == 0);
    try expect(@intFromEnum(Value.one) == 1);
    try expect(@intFromEnum(Value.two) == 2);
}

// You can override the ordinal value for an enum.
const Value2 = enum(u32) {
    hundred = 100,
    thousand = 1000,
    million = 1000000,
};
test "set enum ordinal value" {
    try expect(@intFromEnum(Value2.hundred) == 100);
    try expect(@intFromEnum(Value2.thousand) == 1000);
    try expect(@intFromEnum(Value2.million) == 1000000);
}

// You can also override only some values.
const Value3 = enum(u4) {
    a,
    b = 8,
    c,
    d = 4,
    e,
};
test "enum implicit ordinal values and overridden values" {
    try expect(@intFromEnum(Value3.a) == 0);
    try expect(@intFromEnum(Value3.b) == 8);
    try expect(@intFromEnum(Value3.c) == 9);
    try expect(@intFromEnum(Value3.d) == 4);
    try expect(@intFromEnum(Value3.e) == 5);
}

// Enums can have methods, the same as structs and unions.
// Enum methods are not special, they are only namespaced
// functions that you can call with dot syntax.
const Suit = enum {
    clubs,
    spades,
    diamonds,
    hearts,

    pub fn isClubs(self: Suit) bool {
        return self == Suit.clubs;
```

```
    }
};
test "enum method" {
    const p = Suit.spades;
    try expect(!p.isClubs());
}

// An enum can be switched upon.
const Foo = enum {
    string,
    number,
    none,
};
test "enum switch" {
    const p = Foo.number;
    const what_is_it = switch (p) {
        Foo.string => "this is a string",
        Foo.number => "this is a number",
        Foo.none => "this is a none",
    };
    try expect(mem.eql(u8, what_is_it, "this is a number"));
}

// @typeInfo can be used to access the integer tag type of an enum.
const Small = enum {
    one,
    two,
    three,
    four,
};
test "std.meta.Tag" {
    try expect(@typeInfo(Small).@"enum".tag_type == u2);
}

// @typeInfo tells us the field count and the fields names:
test "@typeInfo" {
    try expect(@typeInfo(Small).@"enum".fields.len == 4);
    try expect(mem.eql(u8, @typeInfo(Small).@"enum".fields[1].name, "two"));
}

// @tagName gives a [:0]const u8 representation of an enum value:
test "@tagName" {
    try expect(mem.eql(u8, @tagName(Small.three), "three"));
}
```

Shell

```
$ zig test test_enums.zig
1/8 test_enums.test.enum ordinal value...OK
2/8 test_enums.test.set enum ordinal value...OK
3/8 test_enums.test.enum implicit ordinal values and overridden values...OK
4/8 test_enums.test.enum method...OK
5/8 test_enums.test.enum switch...OK
6/8 test_enums.test.std.meta.Tag...OK
7/8 test_enums.test.@typeInfo...OK
8/8 test_enums.test.@tagName...OK
All 8 tests passed.
```

See also:

- @typeInfo
- @tagName
- @sizeOf

extern enum

By default, enums are not guaranteed to be compatible with the C ABI:

enum_export_error.zig

```
const Foo = enum { a, b, c };
export fn entry(foo: Foo) void {
    _ = foo;
}
```

Shell

```
$ zig build-obj enum_export_error.zig -target x86_64-linux
doc/langref/enum_export_error.zig:2:17: error: parameter of type 'enum_export_error.Foo' not al
export fn entry(foo: Foo) void {
                ^~~~~~~~
doc/langref/enum_export_error.zig:2:17: note: enum tag type 'u2' is not extern compatible
doc/langref/enum_export_error.zig:2:17: note: only integers with 0, 8, 16, 32, 64 and 128 bits
doc/langref/enum_export_error.zig:1:13: note: enum declared here
const Foo = enum { a, b, c };
            ^~~~~~~~~~~~~~~~
referenced by:
    root: lib/std/start.zig:3:22
    comptime: lib/std/start.zig:27:9
    2 reference(s) hidden; use '-freference-trace=4' to see all references
```

For a C-ABI-compatible enum, provide an explicit tag type to the enum:

enum_export.zig

```
const Foo = enum(c_int) { a, b, c };
export fn entry(foo: Foo) void {
    _ = foo;
}
```

Shell

```
$ zig build-obj enum_export.zig
```

Enum Literals

Enum literals allow specifying the name of an enum field without specifying the enum type:

test_enum_literals.zig

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

const Color = enum {
    auto,
    off,
    on,
};

test "enum literals" {
    const color1: Color = .auto;
    const color2 = Color.auto;
    try expect(color1 == color2);
}

test "switch using enum literals" {
    const color = Color.on;
    const result = switch (color) {
        .auto => false,
        .on => true,
        .off => false,
    };
    try expect(result);
}
```

Shell

```
$ zig test test_enum_literals.zig
1/2 test_enum_literals.test.enum literals...OK
2/2 test_enum_literals.test.switch using enum literals...OK
All 2 tests passed.
```

Non-exhaustive enum

A non-exhaustive enum can be created by adding a trailing `_` field. The enum must specify a tag type and cannot consume every enumeration value.

@enumFromInt on a non-exhaustive enum involves the safety semantics of @intCast to the integer tag type, but beyond that always results in a well-defined enum value.

A switch on a non-exhaustive enum can include a `_` prong as an alternative to an `else` prong. With a `_` prong the compiler errors if all the known tag names are not handled by the switch.

test_switch_non-exhaustive.zig

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

const Number = enum(u8) {
    one,
    two,
    three,
    _,
};
```

```
test "switch on non-exhaustive enum" {
    const number = Number.one;
    const result = switch (number) {
        .one => true,
        .two, .three => false,
        _ => false,
    };
    try expect(result);
    const is_one = switch (number) {
        .one => true,
        else => false,
    };
    try expect(is_one);
}
```

Shell

```
$ zig test test_switch_non-exhaustive.zig
1/1 test_switch_non-exhaustive.test.switch on non-exhaustive enum...OK
All 1 tests passed.
```

union

A bare `union` defines a set of possible types that a value can be as a list of fields. Only one field can be active at a time. The in-memory representation of bare unions is not guaranteed. Bare unions cannot be used to reinterpret memory. For that, use @ptrCast, or use an extern union or a packed union which have guaranteed in-memory layout. Accessing the non-active field is safety-checked Illegal Behavior:

test_wrong_union_access.zig

```
const Payload = union {
    int: i64,
    float: f64,
    boolean: bool,
};
test "simple union" {
    var payload = Payload{ .int = 1234 };
    payload.float = 12.34;
}
```

Shell

```
$ zig test test_wrong_union_access.zig
1/1 test_wrong_union_access.test.simple union...thread 221884 panic: access of union field 'flo
/home/andy/src/zig/doc/langref/test_wrong_union_access.zig:8:12: 0x1048aef in test.simple unior
    payload.float = 12.34;
           ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10ef005 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10e759d in main (test)
        return mainTerminal();
```

```
                                ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10e6b22 in posixCallMainAndExit (test)
            root.main();
                      ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e66fd in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/cc43f9aad019d16b70a651087b598133/test --seed=0x94ba4c6a
```

You can activate another field by assigning the entire union:

test_simple_union.zig

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

const Payload = union {
    int: i64,
    float: f64,
    boolean: bool,
};
test "simple union" {
    var payload = Payload{ .int = 1234 };
    try expect(payload.int == 1234);
    payload = Payload{ .float = 12.34 };
    try expect(payload.float == 12.34);
}
```

Shell

```
$ zig test test_simple_union.zig
1/1 test_simple_union.test.simple union...OK
All 1 tests passed.
```

In order to use switch with a union, it must be a Tagged union.

To initialize a union when the tag is a comptime-known name, see @unionInit.

Tagged union

Unions can be declared with an enum tag type. This turns the union into a *tagged* union, which makes it eligible to use with switch expressions. Tagged unions coerce to their tag type: Type Coercion: Unions and Enums.

test_tagged_union.zig

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

const ComplexTypeTag = enum {
    ok,
    not_ok,
```

```
};
const ComplexType = union(ComplexTypeTag) {
    ok: u8,
    not_ok: void,
};

test "switch on tagged union" {
    const c = ComplexType{ .ok = 42 };
    try expect(@as(ComplexTypeTag, c) == ComplexTypeTag.ok);

    switch (c) {
        .ok => |value| try expect(value == 42),
        .not_ok => unreachable,
    }
}

test "get tag type" {
    try expect(std.meta.Tag(ComplexType) == ComplexTypeTag);
}
```

Shell

```
$ zig test test_tagged_union.zig
1/2 test_tagged_union.test.switch on tagged union...OK
2/2 test_tagged_union.test.get tag type...OK
All 2 tests passed.
```

In order to modify the payload of a tagged union in a switch expression, place a `*` before the variable name to make it a pointer:

test_switch_modify_tagged_union.zig

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

const ComplexTypeTag = enum {
    ok,
    not_ok,
};
const ComplexType = union(ComplexTypeTag) {
    ok: u8,
    not_ok: void,
};

test "modify tagged union in switch" {
    var c = ComplexType{ .ok = 42 };

    switch (c) {
        ComplexTypeTag.ok => |*value| value.* += 1,
        ComplexTypeTag.not_ok => unreachable,
    }

    try expect(c.ok == 43);
}
```

Shell

```
$ zig test test_switch_modify_tagged_union.zig
1/1 test_switch_modify_tagged_union.test.modify tagged union in switch...OK
All 1 tests passed.
```

Unions can be made to infer the enum tag type. Further, unions can have methods just like structs and enums.

test_union_method.zig

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

const Variant = union(enum) {
    int: i32,
    boolean: bool,

    // void can be omitted when inferring enum tag type.
    none,

    fn truthy(self: Variant) bool {
        return switch (self) {
            Variant.int => |x_int| x_int != 0,
            Variant.boolean => |x_bool| x_bool,
            Variant.none => false,
        };
    }
};

test "union method" {
    var v1: Variant = .{ .int = 1 };
    var v2: Variant = .{ .boolean = false };
    var v3: Variant = .none;

    try expect(v1.truthy());
    try expect(!v2.truthy());
    try expect(!v3.truthy());
}
```

Shell

```
$ zig test test_union_method.zig
1/1 test_union_method.test.union method...OK
All 1 tests passed.
```

[@tagName](#) can be used to return a [comptime](#) `[:0]const u8` value representing the field name:

test_tagName.zig

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

const Small2 = union(enum) {
    a: i32,
    b: bool,
    c: u8,
};
```

```
test "@tagName" {
    try expect(std.mem.eql(u8, @tagName(Small2.a), "a"));
}
```

Shell

```
$ zig test test_tagName.zig
1/1 test_tagName.test.@tagName...OK
All 1 tests passed.
```

extern union

An `extern union` has memory layout guaranteed to be compatible with the target C ABI.

See also:

- extern struct

packed union

A `packed union` has well-defined in-memory layout and is eligible to be in a packed struct.

Anonymous Union Literals

Anonymous Struct Literals syntax can be used to initialize unions without specifying the type:

test_anonymous_union.zig

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

const Number = union {
    int: i32,
    float: f64,
};

test "anonymous union literal syntax" {
    const i: Number = .{ .int = 42 };
    const f = makeNumber();
    try expect(i.int == 42);
    try expect(f.float == 12.34);
}

fn makeNumber() Number {
    return .{ .float = 12.34 };
}
```

Shell

```
$ zig test test_anonymous_union.zig
1/1 test_anonymous_union.test.anonymous union literal syntax...OK
All 1 tests passed.
```

opaque

`opaque {}` declares a new type with an unknown (but non-zero) size and alignment. It can contain declarations the same as structs, unions, and enums.

This is typically used for type safety when interacting with C code that does not expose struct details. Example:

test_opaque.zig

```
const Derp = opaque {};
const Wat = opaque {};

extern fn bar(d: *Derp) void;
fn foo(w: *Wat) callconv(.C) void {
    bar(w);
}

test "call foo" {
    foo(undefined);
}
```

Shell

```
$ zig test test_opaque.zig
doc/langref/test_opaque.zig:6:9: error: expected type '*test_opaque.Derp', found '*test_opaque.
    bar(w);
        ^
doc/langref/test_opaque.zig:6:9: note: pointer type child 'test_opaque.Wat' cannot cast into pc
doc/langref/test_opaque.zig:2:13: note: opaque declared here
const Wat = opaque {};
            ^~~~~~~~~
doc/langref/test_opaque.zig:1:14: note: opaque declared here
const Derp = opaque {};
             ^~~~~~~~~
doc/langref/test_opaque.zig:4:18: note: parameter type declared here
extern fn bar(d: *Derp) void;
                 ^~~~~
```

Blocks

Blocks are used to limit the scope of variable declarations:

test_blocks.zig

```
test "access variable after block scope" {
    {
        var x: i32 = 1;
        _ = &x;
    }
    x += 1;
}
```

Shell

```
$ zig test test_blocks.zig
doc/langref/test_blocks.zig:6:5: error: use of undeclared identifier 'x'
    x += 1;
    ^
```

Blocks are expressions. When labeled, `break` can be used to return a value from the block:

test_labeled_break.zig

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

test "labeled break from labeled block expression" {
    var y: i32 = 123;

    const x = blk: {
        y += 1;
        break :blk y;
    };
    try expect(x == 124);
    try expect(y == 124);
}
```

Shell

```
$ zig test test_labeled_break.zig
1/1 test_labeled_break.test.labeled break from labeled block expression...OK
All 1 tests passed.
```

Here, `blk` can be any name.

See also:

- Labeled while
- Labeled for

Shadowing

Identifiers are never allowed to "hide" other identifiers by using the same name:

test_shadowing.zig

```
const pi = 3.14;
```

```
test "inside test block" {
    // Let's even go inside another block
    {
        var pi: i32 = 1234;
    }
}
```

Shell

```
$ zig test test_shadowing.zig
doc/langref/test_shadowing.zig:6:13: error: local variable shadows declaration of 'pi'
        var pi: i32 = 1234;
            ^~
doc/langref/test_shadowing.zig:1:1: note: declared here
const pi = 3.14;
^~~~~~~~~~~~~~~
```

Because of this, when you read Zig code you can always rely on an identifier to consistently mean the same thing within the scope it is defined. Note that you can, however, use the same name if the scopes are separate:

test_scopes.zig

```
test "separate scopes" {
    {
        const pi = 3.14;
        _ = pi;
    }
    {
        var pi: bool = true;
        _ = &pi;
    }
}
```

Shell

```
$ zig test test_scopes.zig
1/1 test_scopes.test.separate scopes...OK
All 1 tests passed.
```

Empty Blocks

An empty block is equivalent to `void{}`:

test_empty_block.zig

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

test {
    const a = {};
    const b = void{};
    try expect(@TypeOf(a) == void);
```

```
    try expect(@TypeOf(b) == void);
    try expect(a == b);
}
```

Shell

```
$ zig test test_empty_block.zig
1/1 test_empty_block.test_0...OK
All 1 tests passed.
```

switch

test_switch.zig

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

test "switch simple" {
    const a: u64 = 10;
    const zz: u64 = 103;

    // All branches of a switch expression must be able to be coerced to a
    // common type.
    //
    // Branches cannot fallthrough. If fallthrough behavior is desired, combine
    // the cases and use an if.
    const b = switch (a) {
        // Multiple cases can be combined via a ','
        1, 2, 3 => 0,

        // Ranges can be specified using the ... syntax. These are inclusive
        // of both ends.
        5...100 => 1,

        // Branches can be arbitrarily complex.
        101 => blk: {
            const c: u64 = 5;
            break :blk c * 2 + 1;
        },

        // Switching on arbitrary expressions is allowed as long as the
        // expression is known at compile-time.
        zz => zz,
        blk: {
            const d: u32 = 5;
            const e: u32 = 100;
            break :blk d + e;
        } => 107,

        // The else branch catches everything not already captured.
        // Else branches are mandatory unless the entire range of values
        // is handled.
        else => 9,
    };
```

```
    try expect(b == 1);
}

// Switch expressions can be used outside a function:
const os_msg = switch (builtin.target.os.tag) {
    .linux => "we found a linux user",
    else => "not a linux user",
};

// Inside a function, switch statements implicitly are compile-time
// evaluated if the target expression is compile-time known.
test "switch inside function" {
    switch (builtin.target.os.tag) {
        .fuchsia => {
            // On an OS other than fuchsia, block is not even analyzed,
            // so this compile error is not triggered.
            // On fuchsia this compile error would be triggered.
            @compileError("fuchsia not supported");
        },
        else => {},
    }
}
```

Shell

```
$ zig test test_switch.zig
1/2 test_switch.test.switch simple...OK
2/2 test_switch.test.switch inside function...OK
All 2 tests passed.
```

`switch` can be used to capture the field values of a Tagged union. Modifications to the field values can be done by placing a `*` before the capture variable name, turning it into a pointer.

test_switch_tagged_union.zig

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

test "switch on tagged union" {
    const Point = struct {
        x: u8,
        y: u8,
    };
    const Item = union(enum) {
        a: u32,
        c: Point,
        d,
        e: u32,
    };

    var a = Item{ .c = Point{ .x = 1, .y = 2 } };

    // Switching on more complex enums is allowed.
    const b = switch (a) {
        // A capture group is allowed on a match, and will return the enum
        // value matched. If the payload types of both cases are the same
        // they can be put into the same switch prong.
        Item.a, Item.e => |item| item,
```

```
        // A reference to the matched value can be obtained using `*` syntax.
        Item.c => |*item| blk: {
            item.*.x += 1;
            break :blk 6;
        },

        // No else is required if the types cases was exhaustively handled
        Item.d => 8,
    };

    try expect(b == 6);
    try expect(a.c.x == 2);
}
```

Shell

```
$ zig test test_switch_tagged_union.zig
1/1 test_switch_tagged_union.test.switch on tagged union...OK
All 1 tests passed.
```

See also:

- comptime
- enum
- @compileError
- Compile Variables

Exhaustive Switching

When a `switch` expression does not have an `else` clause, it must exhaustively list all the possible values. Failure to do so is a compile error:

test_unhandled_enumeration_value.zig

```
const Color = enum {
    auto,
    off,
    on,
};

test "exhaustive switching" {
    const color = Color.off;
    switch (color) {
        Color.auto => {},
        Color.on => {},
    }
}
```

Shell

```
$ zig test test_unhandled_enumeration_value.zig
doc/langref/test_unhandled_enumeration_value.zig:9:5: error: switch must handle all possibiliti
    switch (color) {
```

```
    ^~~~~~
doc/langref/test_unhandled_enumeration_value.zig:3:5: note: unhandled enumeration value: 'off'
    off,
    ^~~
doc/langref/test_unhandled_enumeration_value.zig:1:15: note: enum 'test_unhandled_enumeration_v
const Color = enum {
              ^~~~
```

Switching with Enum Literals

Enum Literals can be useful to use with `switch` to avoid repetitively specifying enum or union types:

test_exhaustive_switch.zig

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

const Color = enum {
    auto,
    off,
    on,
};

test "enum literals with switch" {
    const color = Color.off;
    const result = switch (color) {
        .auto => false,
        .on => false,
        .off => true,
    };
    try expect(result);
}
```

Shell

```
$ zig test test_exhaustive_switch.zig
1/1 test_exhaustive_switch.test.enum literals with switch...OK
All 1 tests passed.
```

Labeled switch

When a switch statement is labeled, it can be referenced from a `break` or `continue`. `break` will return a value from the `switch`.

A `continue` targeting a switch must have an operand. When executed, it will jump to the matching prong, as if the `switch` were executed again with the `continue`'s operand replacing the initial switch value.

test_switch_continue.zig

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

```
test "switch continue" {
    sw: switch (@as(i32, 5)) {
        5 => continue :sw 4,

        // `continue` can occur multiple times within a single switch prong.
        2...4 => |v| {
            if (v > 3) {
                continue :sw 2;
            } else if (v == 3) {

                // `break` can target labeled loops.
                break :sw;
            }

            continue :sw 1;
        },

        1 => return,

        else => unreachable,
    }
}
```

Shell

```
$ zig test test_switch_continue.zig
1/1 test_switch_continue.test.switch continue...OK
All 1 tests passed.
```

Semantically, this is equivalent to the following loop:

test_switch_continue_equivalent.zig

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

test "switch continue, equivalent loop" {
    var sw: i32 = 5;
    while (true) {
        switch (sw) {
            5 => {
                sw = 4;
                continue;
            },
            2...4 => |v| {
                if (v > 3) {
                    sw = 2;
                    continue;
                } else if (v == 3) {
                    break;
                }

                sw = 1;
                continue;
            },
            1 => return,
            else => unreachable,
```

```
        }
    }
}
```

Shell

```
$ zig test test_switch_continue_equivalent.zig
1/1 test_switch_continue_equivalent.test.switch continue, equivalent loop...OK
All 1 tests passed.
```

This can improve clarity of (for example) state machines, where the syntax `continue :sw .next_state` is unambiguous, explicit, and immediately understandable.

However, the motivating example is a switch on each element of an array, where using a single switch can improve clarity and performance:

test_switch_dispatch_loop.zig

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

const Instruction = enum {
    add,
    mul,
    end,
};

fn evaluate(initial_stack: []const i32, code: []const Instruction) !i32 {
    var stack = try std.BoundedArray(i32, 8).fromSlice(initial_stack);
    var ip: usize = 0;

    return vm: switch (code[ip]) {
        // Because all code after `continue` is unreachable, this branch does
        // not provide a result.
        .add => {
            try stack.append(stack.pop().? + stack.pop().?);

            ip += 1;
            continue :vm code[ip];
        },
        .mul => {
            try stack.append(stack.pop().? * stack.pop().?);

            ip += 1;
            continue :vm code[ip];
        },
        .end => stack.pop().?,
    };
}

test "evaluate" {
    const result = try evaluate(&.{ 7, 2, -3 }, &.{ .mul, .add, .end });
    try expectEqual(1, result);
}
```

Shell

```
$ zig test test_switch_dispatch_loop.zig
1/1 test_switch_dispatch_loop.test.evaluate...OK
All 1 tests passed.
```

If the operand to `continue` is comptime-known, then it can be lowered to an unconditional branch to the relevant case. Such a branch is perfectly predicted, and hence typically very fast to execute.

If the operand is runtime-known, each `continue` can embed a conditional branch inline (ideally through a jump table), which allows a CPU to predict its target independently of any other prong. A loop-based lowering would force every branch through the same dispatch point, hindering branch prediction.

Inline Switch Prongs

Switch prongs can be marked as `inline` to generate the prong's body for each possible value it could have, making the captured value comptime.

test_inline_switch.zig

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

fn isFieldOptional(comptime T: type, field_index: usize) !bool {
    const fields = @typeInfo(T).@"struct".fields;
    return switch (field_index) {
        // This prong is analyzed twice with `idx` being a
        // comptime-known value each time.
        inline 0, 1 => |idx| @typeInfo(fields[idx].type) == .optional,
        else => return error.IndexOutOfBounds,
    };
}

const Struct1 = struct { a: u32, b: ?u32 };

test "using @typeInfo with runtime values" {
    var index: usize = 0;
    try expect(!try isFieldOptional(Struct1, index));
    index += 1;
    try expect(try isFieldOptional(Struct1, index));
    index += 1;
    try expectError(error.IndexOutOfBounds, isFieldOptional(Struct1, index));
}

// Calls to `isFieldOptional` on `Struct1` get unrolled to an equivalent
// of this function:
fn isFieldOptionalUnrolled(field_index: usize) !bool {
    return switch (field_index) {
        0 => false,
        1 => true,
        else => return error.IndexOutOfBounds,
```

```
    };
}
```

Shell

```
$ zig test test_inline_switch.zig
1/1 test_inline_switch.test.using @typeInfo with runtime values...OK
All 1 tests passed.
```

The `inline` keyword may also be combined with ranges:

inline_prong_range.zig

```
fn isFieldOptional(comptime T: type, field_index: usize) !bool {
    const fields = @typeInfo(T).@"struct".fields;
    return switch (field_index) {
        inline 0...fields.len - 1 => |idx| @typeInfo(fields[idx].type) == .optional,
        else => return error.IndexOutOfBounds,
    };
}
```

`inline else` prongs can be used as a type safe alternative to `inline for` loops:

test_inline_else.zig

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

const SliceTypeA = extern struct {
    len: usize,
    ptr: [*]u32,
};
const SliceTypeB = extern struct {
    ptr: [*]SliceTypeA,
    len: usize,
};
const AnySlice = union(enum) {
    a: SliceTypeA,
    b: SliceTypeB,
    c: []const u8,
    d: []AnySlice,
};

fn withFor(any: AnySlice) usize {
    const Tag = @typeInfo(AnySlice).@"union".tag_type.?;
    inline for (@typeInfo(Tag).@"enum".fields) |field| {
        // With `inline for` the function gets generated as
        // a series of `if` statements relying on the optimizer
        // to convert it to a switch.
        if (field.value == @intFromEnum(any)) {
            return @field(any, field.name).len;
        }
    }
    // When using `inline for` the compiler doesn't know that every
    // possible case has been handled requiring an explicit `unreachable`.
    unreachable;
}
```

```
fn withSwitch(any: AnySlice) usize {
    return switch (any) {
        // With `inline else` the function is explicitly generated
        // as the desired switch and the compiler can check that
        // every possible case is handled.
        inline else => |slice| slice.len,
    };
}

test "inline for and inline else similarity" {
    const any = AnySlice{ .c = "hello" };
    try expect(withFor(any) == 5);
    try expect(withSwitch(any) == 5);
}
```

Shell

```
$ zig test test_inline_else.zig
1/1 test_inline_else.test.inline for and inline else similarity...OK
All 1 tests passed.
```

When using an inline prong switching on an union an additional capture can be used to obtain the union's enum tag value.

test_inline_switch_union_tag.zig

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

const U = union(enum) {
    a: u32,
    b: f32,
};

fn getNum(u: U) u32 {
    switch (u) {
        // Here `num` is a runtime-known value that is either
        // `u.a` or `u.b` and `tag` is `u`'s comptime-known tag value.
        inline else => |num, tag| {
            if (tag == .b) {
                return @intFromFloat(num);
            }
            return num;
        },
    }
}

test "test" {
    const u = U{ .b = 42 };
    try expect(getNum(u) == 42);
}
```

Shell

```
$ zig test test_inline_switch_union_tag.zig
1/1 test_inline_switch_union_tag.test.test...OK
```

```
All 1 tests passed.
```

See also:

- inline while
- inline for

while

A while loop is used to repeatedly execute an expression until some condition is no longer true.

test_while.zig

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

test "while basic" {
    var i: usize = 0;
    while (i < 10) {
        i += 1;
    }
    try expect(i == 10);
}
```

Shell

```
$ zig test test_while.zig
1/1 test_while.test.while basic...OK
All 1 tests passed.
```

Use `break` to exit a while loop early.

test_while_break.zig

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

test "while break" {
    var i: usize = 0;
    while (true) {
        if (i == 10)
            break;
        i += 1;
    }
    try expect(i == 10);
}
```

Shell

```
$ zig test test_while_break.zig
1/1 test_while_break.test.while break...OK
All 1 tests passed.
```

Use `continue` to jump back to the beginning of the loop.

test_while_continue.zig

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

test "while continue" {
    var i: usize = 0;
    while (true) {
        i += 1;
        if (i < 10)
            continue;
        break;
    }
    try expect(i == 10);
}
```

Shell

```
$ zig test test_while_continue.zig
1/1 test_while_continue.test.while continue...OK
All 1 tests passed.
```

While loops support a continue expression which is executed when the loop is continued. The `continue` keyword respects this expression.

test_while_continue_expression.zig

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

test "while loop continue expression" {
    var i: usize = 0;
    while (i < 10) : (i += 1) {}
    try expect(i == 10);
}

test "while loop continue expression, more complicated" {
    var i: usize = 1;
    var j: usize = 1;
    while (i * j < 2000) : ({
        i *= 2;
        j *= 3;
    }) {
        const my_ij = i * j;
        try expect(my_ij < 2000);
    }
}
```

Shell

```
$ zig test test_while_continue_expression.zig
1/2 test_while_continue_expression.test.while loop continue expression...OK
2/2 test_while_continue_expression.test.while loop continue expression, more complicated...OK
All 2 tests passed.
```

While loops are expressions. The result of the expression is the result of the `else` clause of a while loop, which is executed when the condition of the while loop is tested as false.

`break`, like `return`, accepts a value parameter. This is the result of the `while` expression. When you `break` from a while loop, the `else` branch is not evaluated.

test_while_else.zig

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

test "while else" {
    try expect(rangeHasNumber(0, 10, 5));
    try expect(!rangeHasNumber(0, 10, 15));
}

fn rangeHasNumber(begin: usize, end: usize, number: usize) bool {
    var i = begin;
    return while (i < end) : (i += 1) {
        if (i == number) {
            break true;
        }
    } else false;
}
```

Shell

```
$ zig test test_while_else.zig
1/1 test_while_else.test.while else...OK
All 1 tests passed.
```

Labeled while

When a `while` loop is labeled, it can be referenced from a `break` or `continue` from within a nested loop:

test_while_nested_break.zig

```
test "nested break" {
    outer: while (true) {
        while (true) {
            break :outer;
        }
    }
}

test "nested continue" {
    var i: usize = 0;
    outer: while (i < 10) : (i += 1) {
        while (true) {
            continue :outer;
        }
    }
}
```

Shell

```
$ zig test test_while_nested_break.zig
1/2 test_while_nested_break.test.nested break...OK
2/2 test_while_nested_break.test.nested continue...OK
```

```
All 2 tests passed.
```

while with Optionals

Just like if expressions, while loops can take an optional as the condition and capture the payload. When null is encountered the loop exits.

When the `|x|` syntax is present on a `while` expression, the while condition must have an Optional Type.

The `else` branch is allowed on optional iteration. In this case, it will be executed on the first null value encountered.

test_while_null_capture.zig

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

test "while null capture" {
    var sum1: u32 = 0;
    numbers_left = 3;
    while (eventuallyNullSequence()) |value| {
        sum1 += value;
    }
    try expect(sum1 == 3);

    // null capture with an else block
    var sum2: u32 = 0;
    numbers_left = 3;
    while (eventuallyNullSequence()) |value| {
        sum2 += value;
    } else {
        try expect(sum2 == 3);
    }

    // null capture with a continue expression
    var i: u32 = 0;
    var sum3: u32 = 0;
    numbers_left = 3;
    while (eventuallyNullSequence()) |value| : (i += 1) {
        sum3 += value;
    }
    try expect(i == 3);
}

var numbers_left: u32 = undefined;
fn eventuallyNullSequence() ?u32 {
    return if (numbers_left == 0) null else blk: {
        numbers_left -= 1;
        break :blk numbers_left;
    };
}
```

Shell

```
$ zig test test_while_null_capture.zig
1/1 test_while_null_capture.test.while null capture...OK
All 1 tests passed.
```

while with Error Unions

Just like if expressions, while loops can take an error union as the condition and capture the payload or the error code. When the condition results in an error code the else branch is evaluated and the loop is finished.

When the `else |x|` syntax is present on a `while` expression, the while condition must have an Error Union Type.

test_while_error_capture.zig

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

test "while error union capture" {
    var sum1: u32 = 0;
    numbers_left = 3;
    while (eventuallyErrorSequence()) |value| {
        sum1 += value;
    } else |err| {
        try expect(err == error.ReachedZero);
    }
}

var numbers_left: u32 = undefined;

fn eventuallyErrorSequence() anyerror!u32 {
    return if (numbers_left == 0) error.ReachedZero else blk: {
        numbers_left -= 1;
        break :blk numbers_left;
    };
}
```

Shell

```
$ zig test test_while_error_capture.zig
1/1 test_while_error_capture.test.while error union capture...OK
All 1 tests passed.
```

inline while

While loops can be inlined. This causes the loop to be unrolled, which allows the code to do some things which only work at compile time, such as use types as first class values.

test_inline_while.zig

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

test "inline while loop" {
```

```
    comptime var i = 0;
    var sum: usize = 0;
    inline while (i < 3) : (i += 1) {
        const T = switch (i) {
            0 => f32,
            1 => i8,
            2 => bool,
            else => unreachable,
        };
        sum += typeNameLength(T);
    }
    try expect(sum == 9);
}

fn typeNameLength(comptime T: type) usize {
    return @typeName(T).len;
}
```

Shell

```
$ zig test test_inline_while.zig
1/1 test_inline_while.test.inline while loop...OK
All 1 tests passed.
```

It is recommended to use `inline` loops only for one of these reasons:

- You need the loop to execute at comptime for the semantics to work.
- You have a benchmark to prove that forcibly unrolling the loop in this way is measurably faster.

See also:

- if
- Optionals
- Errors
- comptime
- unreachable

for

test_for.zig

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

test "for basics" {
    const items = [_]i32{ 4, 5, 3, 4, 0 };
    var sum: i32 = 0;

    // For loops iterate over slices and arrays.
    for (items) |value| {
        // Break and continue are supported.
        if (value == 0) {
```

```
            continue;
        }
        sum += value;
    }
    try expect(sum == 16);

    // To iterate over a portion of a slice, reslice.
    for (items[0..1]) |value| {
        sum += value;
    }
    try expect(sum == 20);

    // To access the index of iteration, specify a second condition as well
    // as a second capture value.
    var sum2: i32 = 0;
    for (items, 0..) |_, i| {
        try expect(@TypeOf(i) == usize);
        sum2 += @as(i32, @intCast(i));
    }
    try expect(sum2 == 10);

    // To iterate over consecutive integers, use the range syntax.
    // Unbounded range is always a compile error.
    var sum3: usize = 0;
    for (0..5) |i| {
        sum3 += i;
    }
    try expect(sum3 == 10);
}

test "multi object for" {
    const items = [_]usize{ 1, 2, 3 };
    const items2 = [_]usize{ 4, 5, 6 };
    var count: usize = 0;

    // Iterate over multiple objects.
    // All lengths must be equal at the start of the loop, otherwise detectable
    // illegal behavior occurs.
    for (items, items2) |i, j| {
        count += i + j;
    }

    try expect(count == 21);
}

test "for reference" {
    var items = [_]i32{ 3, 4, 2 };

    // Iterate over the slice by reference by
    // specifying that the capture value is a pointer.
    for (&items) |*value| {
        value.* += 1;
    }

    try expect(items[0] == 4);
    try expect(items[1] == 5);
    try expect(items[2] == 3);
```

```
}

test "for else" {
    // For allows an else attached to it, the same as a while loop.
    const items = [_]?i32{ 3, 4, null, 5 };

    // For loops can also be used as expressions.
    // Similar to while loops, when you break from a for loop, the else branch is not evaluated
    var sum: i32 = 0;
    const result = for (items) |value| {
        if (value != null) {
            sum += value.?;
        }
    } else blk: {
        try expect(sum == 12);
        break :blk sum;
    };
    try expect(result == 12);
}
```

Shell

```
$ zig test test_for.zig
1/4 test_for.test.for basics...OK
2/4 test_for.test.multi object for...OK
3/4 test_for.test.for reference...OK
4/4 test_for.test.for else...OK
All 4 tests passed.
```

Labeled for

When a `for` loop is labeled, it can be referenced from a `break` or `continue` from within a nested loop:

test_for_nested_break.zig

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

test "nested break" {
    var count: usize = 0;
    outer: for (1..6) |_| {
        for (1..6) |_| {
            count += 1;
            break :outer;
        }
    }
    try expect(count == 1);
}

test "nested continue" {
    var count: usize = 0;
    outer: for (1..9) |_| {
        for (1..6) |_| {
            count += 1;
            continue :outer;
```

```
        }
    }

    try expect(count == 8);
}
```

Shell

```
$ zig test test_for_nested_break.zig
1/2 test_for_nested_break.test.nested break...OK
2/2 test_for_nested_break.test.nested continue...OK
All 2 tests passed.
```

inline for

For loops can be inlined. This causes the loop to be unrolled, which allows the code to do some things which only work at compile time, such as use types as first class values. The capture value and iterator value of inlined for loops are compile-time known.

test_inline_for.zig

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

test "inline for loop" {
    const nums = [_]i32{ 2, 4, 6 };
    var sum: usize = 0;
    inline for (nums) |i| {
        const T = switch (i) {
            2 => f32,
            4 => i8,
            6 => bool,
            else => unreachable,
        };
        sum += typeNameLength(T);
    }
    try expect(sum == 9);
}

fn typeNameLength(comptime T: type) usize {
    return @typeName(T).len;
}
```

Shell

```
$ zig test test_inline_for.zig
1/1 test_inline_for.test.inline for loop...OK
All 1 tests passed.
```

It is recommended to use `inline` loops only for one of these reasons:

- You need the loop to execute at comptime for the semantics to work.
- You have a benchmark to prove that forcibly unrolling the loop in this way is measurably faster.

See also:

- while
- comptime
- Arrays
- Slices

if

test_if.zig

```
// If expressions have three uses, corresponding to the three types:
// * bool
// * ?T
// * anyerror!T

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

test "if expression" {
    // If expressions are used instead of a ternary expression.
    const a: u32 = 5;
    const b: u32 = 4;
    const result = if (a != b) 47 else 3089;
    try expect(result == 47);
}

test "if boolean" {
    // If expressions test boolean conditions.
    const a: u32 = 5;
    const b: u32 = 4;
    if (a != b) {
        try expect(true);
    } else if (a == 9) {
        unreachable;
    } else {
        unreachable;
    }
}

test "if error union" {
    // If expressions test for errors.
    // Note the |err| capture on the else.

    const a: anyerror!u32 = 0;
    if (a) |value| {
        try expect(value == 0);
    } else |err| {
        _ = err;
        unreachable;
    }

    const b: anyerror!u32 = error.BadValue;
    if (b) |value| {
        _ = value;
```

```
        unreachable;
    } else |err| {
        try expect(err == error.BadValue);
    }

    // The else and |err| capture is strictly required.
    if (a) |value| {
        try expect(value == 0);
    } else |_| {}

    // To check only the error value, use an empty block expression.
    if (b) |_| {} else |err| {
        try expect(err == error.BadValue);
    }

    // Access the value by reference using a pointer capture.
    var c: anyerror!u32 = 3;
    if (c) |*value| {
        value.* = 9;
    } else |_| {
        unreachable;
    }

    if (c) |value| {
        try expect(value == 9);
    } else |_| {
        unreachable;
    }
}
```

Shell

```
$ zig test test_if.zig
1/3 test_if.test.if expression...OK
2/3 test_if.test.if boolean...OK
3/3 test_if.test.if error union...OK
All 3 tests passed.
```

if with Optionals

test_if_optionals.zig

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

test "if optional" {
    // If expressions test for null.

    const a: ?u32 = 0;
    if (a) |value| {
        try expect(value == 0);
    } else {
        unreachable;
    }

    const b: ?u32 = null;
    if (b) |_| {
        unreachable;
```

```
    } else {
        try expect(true);
    }

    // The else is not required.
    if (a) |value| {
        try expect(value == 0);
    }

    // To test against null only, use the binary equality operator.
    if (b == null) {
        try expect(true);
    }

    // Access the value by reference using a pointer capture.
    var c: ?u32 = 3;
    if (c) |*value| {
        value.* = 2;
    }

    if (c) |value| {
        try expect(value == 2);
    } else {
        unreachable;
    }
}

test "if error union with optional" {
    // If expressions test for errors before unwrapping optionals.
    // The |optional_value| capture's type is ?u32.

    const a: anyerror!?u32 = 0;
    if (a) |optional_value| {
        try expect(optional_value.? == 0);
    } else |err| {
        _ = err;
        unreachable;
    }

    const b: anyerror!?u32 = null;
    if (b) |optional_value| {
        try expect(optional_value == null);
    } else |_| {
        unreachable;
    }

    const c: anyerror!?u32 = error.BadValue;
    if (c) |optional_value| {
        _ = optional_value;
        unreachable;
    } else |err| {
        try expect(err == error.BadValue);
    }

    // Access the value by reference by using a pointer capture each time.
    var d: anyerror!?u32 = 3;
    if (d) |*optional_value| {
```

```
        if (optional_value.*) |*value| {
            value.* = 9;
        }
    } else |_| {
        unreachable;
    }

    if (d) |optional_value| {
        try expect(optional_value.? == 9);
    } else |_| {
        unreachable;
    }
}
```

Shell

```
$ zig test test_if_optionals.zig
1/2 test_if_optionals.test.if optional...OK
2/2 test_if_optionals.test.if error union with optional...OK
All 2 tests passed.
```

See also:

- Optionals
- Errors

defer

Executes an expression unconditionally at scope exit.

test_defer.zig

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

fn deferExample() !usize {
    var a: usize = 1;

    {
        defer a = 2;
        a = 1;
    }
    try expect(a == 2);

    a = 5;
    return a;
}

test "defer basics" {
    try expect((try deferExample()) == 5);
}
```

Shell

```
$ zig test test_defer.zig
1/1 test_defer.test.defer basics...OK
All 1 tests passed.
```

Defer expressions are evaluated in reverse order.

defer_unwind.zig

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

test "defer unwinding" {
    print("\n", .{});

    defer {
        print("1 ", .{});
    }
    defer {
        print("2 ", .{});
    }
    if (false) {
        // defers are not run if they are never executed.
        defer {
            print("3 ", .{});
        }
    }
}
```

Shell

```
$ zig test defer_unwind.zig
1/1 defer_unwind.test.defer unwinding...
2 1 OK
All 1 tests passed.
```

Inside a defer expression the return statement is not allowed.

test_invalid_defer.zig

```
fn deferInvalidExample() !void {
    defer {
        return error.DeferError;
    }

    return error.DeferError;
}
```

Shell

```
$ zig test test_invalid_defer.zig
doc/langref/test_invalid_defer.zig:3:9: error: cannot return from defer expression
        return error.DeferError;
        ^~~~~~~~~~~~~~~~~~~~~~~
doc/langref/test_invalid_defer.zig:2:5: note: defer expression here
    defer {
    ^~~~~
```

See also:

- Errors

unreachable

In Debug and ReleaseSafe mode **`unreachable`** emits a call to `panic` with the message `reached unreachable code`.

In ReleaseFast and ReleaseSmall mode, the optimizer uses the assumption that **`unreachable`** code will never be hit to perform optimizations.

Basics

test_unreachable.zig

```
// unreachable is used to assert that control flow will never reach a
// particular location:
test "basic math" {
    const x = 1;
    const y = 2;
    if (x + y != 3) {
        unreachable;
    }
}
```

Shell

```
$ zig test test_unreachable.zig
1/1 test_unreachable.test.basic math...OK
All 1 tests passed.
```

In fact, this is how `std.debug.assert` is implemented:

test_assertion_failure.zig

```
// This is how std.debug.assert is implemented
fn assert(ok: bool) void {
    if (!ok) unreachable; // assertion failure
}

// This test will fail because we hit unreachable.
test "this will fail" {
    assert(false);
}
```

Shell

```
$ zig test test_assertion_failure.zig
1/1 test_assertion_failure.test.this will fail...thread 220435 panic: reached unreachable code
```

```
/home/andy/src/zig/doc/langref/test_assertion_failure.zig:3:14: 0x10489ad in assert (test)
    if (!ok) unreachable; // assertion failure
             ^
/home/andy/src/zig/doc/langref/test_assertion_failure.zig:8:11: 0x104897a in test.this will fai
    assert(false);
          ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10eedd5 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10e736d in main (test)
        return mainTerminal();
                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10e68f2 in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e64cd in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/51c9599b1e9a8f93217f914df57d48a3/test --seed=0xcd3102a8
```

At Compile-Time

test_comptime_unreachable.zig

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

test "type of unreachable" {
    comptime {
        // The type of unreachable is noreturn.

        // However this assertion will still fail to compile because
        // unreachable expressions are compile errors.

        assert(@TypeOf(unreachable) == noreturn);
    }
}
```

Shell

```
$ zig test test_comptime_unreachable.zig
doc/langref/test_comptime_unreachable.zig:10:16: error: unreachable code
        assert(@TypeOf(unreachable) == noreturn);
               ^~~~~~~~~~~~~~~~~~~~
doc/langref/test_comptime_unreachable.zig:10:24: note: control flow is diverted here
        assert(@TypeOf(unreachable) == noreturn);
                       ^~~~~~~~~~~
```

See also:

- Zig Test
- Build Mode

- comptime

noreturn

`noreturn` is the type of:

- `break`
- `continue`
- `return`
- `unreachable`
- `while (true) {}`

When resolving types together, such as `if` clauses or `switch` prongs, the `noreturn` type is compatible with every other type. Consider:

test_noreturn.zig

```
fn foo(condition: bool, b: u32) void {
    const a = if (condition) b else return;
    _ = a;
    @panic("do something with a");
}
test "noreturn" {
    foo(false, 1);
}
```

Shell

```
$ zig test test_noreturn.zig
1/1 test_noreturn.test.noreturn...OK
All 1 tests passed.
```

Another use case for `noreturn` is the `exit` function:

test_noreturn_from_exit.zig

```
const std = @import("std");
const builtin = @import("builtin");
const native_arch = builtin.cpu.arch;
const expect = std.testing.expect;

const WINAPI: std.builtin.CallingConvention = if (native_arch == .x86) .Stdcall else .C;
extern "kernel32" fn ExitProcess(exit_code: c_uint) callconv(WINAPI) noreturn;

test "foo" {
    const value = bar() catch ExitProcess(1);
    try expect(value == 1234);
}

fn bar() anyerror!u32 {
    return 1234;
}
```

Shell

```
$ zig test test_noreturn_from_exit.zig -target x86_64-windows --test-no-exec
```

Functions

test_functions.zig

```
const std = @import("std");
const builtin = @import("builtin");
const native_arch = builtin.cpu.arch;
const expect = std.testing.expect;

// Functions are declared like this
fn add(a: i8, b: i8) i8 {
    if (a == 0) {
        return b;
    }

    return a + b;
}

// The export specifier makes a function externally visible in the generated
// object file, and makes it use the C ABI.
export fn sub(a: i8, b: i8) i8 {
    return a - b;
}

// The extern specifier is used to declare a function that will be resolved
// at link time, when linking statically, or at runtime, when linking
// dynamically. The quoted identifier after the extern keyword specifies
// the library that has the function. (e.g. "c" -> libc.so)
// The callconv specifier changes the calling convention of the function.
const WINAPI: std.builtin.CallingConvention = if (native_arch == .x86) .Stdcall else .C;
extern "kernel32" fn ExitProcess(exit_code: u32) callconv(WINAPI) noreturn;
extern "c" fn atan2(a: f64, b: f64) f64;

// The @branchHint builtin can be used to tell the optimizer that a function is rarely called
fn abort() noreturn {
    @branchHint(.cold);
    while (true) {}
}

// The naked calling convention makes a function not have any function prologue or epilogue.
// This can be useful when integrating with assembly.
fn _start() callconv(.Naked) noreturn {
    abort();
}

// The inline calling convention forces a function to be inlined at all call sites.
// If the function cannot be inlined, it is a compile-time error.
inline fn shiftLeftOne(a: u32) u32 {
    return a << 1;
}

// The pub specifier allows the function to be visible when importing.
```

```
// Another file can use @import and call sub2
pub fn sub2(a: i8, b: i8) i8 {
    return a - b;
}

// Function pointers are prefixed with `*const `.
const Call2Op = *const fn (a: i8, b: i8) i8;
fn doOp(fnCall: Call2Op, op1: i8, op2: i8) i8 {
    return fnCall(op1, op2);
}

test "function" {
    try expect(doOp(add, 5, 6) == 11);
    try expect(doOp(sub2, 5, 6) == -1);
}
```

Shell

```
$ zig test test_functions.zig
1/1 test_functions.test.function...OK
All 1 tests passed.
```

There is a difference between a function *body* and a function *pointer*. Function bodies are comptime-only types while function Pointers may be runtime-known.

Pass-by-value Parameters

Primitive types such as Integers and Floats passed as parameters are copied, and then the copy is available in the function body. This is called "passing by value". Copying a primitive type is essentially free and typically involves nothing more than setting a register.

Structs, unions, and arrays can sometimes be more efficiently passed as a reference, since a copy could be arbitrarily expensive depending on the size. When these types are passed as parameters, Zig may choose to copy and pass by value, or pass by reference, whichever way Zig decides will be faster. This is made possible, in part, by the fact that parameters are immutable.

test_pass_by_reference_or_value.zig

```
const Point = struct {
    x: i32,
    y: i32,
};

fn foo(point: Point) i32 {
    // Here, `point` could be a reference, or a copy. The function body
    // can ignore the difference and treat it as a value. Be very careful
    // taking the address of the parameter - it should be treated as if
    // the address will become invalid when the function returns.
    return point.x + point.y;
}

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

```zig
test "pass struct to function" {
    try expect(foo(Point{ .x = 1, .y = 2 }) == 3);
}
```

Shell

```
$ zig test test_pass_by_reference_or_value.zig
1/1 test_pass_by_reference_or_value.test.pass struct to function...OK
All 1 tests passed.
```

For extern functions, Zig follows the C ABI for passing structs and unions by value.

Function Parameter Type Inference

Function parameters can be declared with `anytype` in place of the type. In this case the parameter types will be inferred when the function is called. Use @TypeOf and @typeInfo to get information about the inferred type.

test_fn_type_inference.zig

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

fn addFortyTwo(x: anytype) @TypeOf(x) {
    return x + 42;
}

test "fn type inference" {
    try expect(addFortyTwo(1) == 43);
    try expect(@TypeOf(addFortyTwo(1)) == comptime_int);
    const y: i64 = 2;
    try expect(addFortyTwo(y) == 44);
    try expect(@TypeOf(addFortyTwo(y)) == i64);
}
```

Shell

```
$ zig test test_fn_type_inference.zig
1/1 test_fn_type_inference.test.fn type inference...OK
All 1 tests passed.
```

inline fn

Adding the `inline` keyword to a function definition makes that function become *semantically inlined* at the callsite. This is not a hint to be possibly observed by optimization passes, but has implications on the types and values involved in the function call.

Unlike normal function calls, arguments at an inline function callsite which are compile-time known are treated as Compile Time Parameters. This can potentially propagate all the way to the return value:

inline_call.zig

```
test "inline function call" {
    if (foo(1200, 34) != 1234) {
        @compileError("bad");
    }
}

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

Shell

```
$ zig test inline_call.zig
1/1 inline_call.test.inline function call...OK
All 1 tests passed.
```

If `inline` is removed, the test fails with the compile error instead of passing.

It is generally better to let the compiler decide when to inline a function, except for these scenarios:

- To change how many stack frames are in the call stack, for debugging purposes.
- To force comptime-ness of the arguments to propagate to the return value of the function, as in the above example.
- Real world performance measurements demand it.

Note that `inline` actually *restricts* what the compiler is allowed to do. This can harm binary size, compilation speed, and even runtime performance.

Function Reflection

test_fn_reflection.zig

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

test "fn reflection" {
    try testing.expect(@typeInfo(@TypeOf(testing.expect)).@"fn".params[0].type.? == bool);
    try testing.expect(@typeInfo(@TypeOf(testing.tmpDir)).@"fn".return_type.? == testing.TmpDi

    try testing.expect(@typeInfo(@TypeOf(math.Log2Int)).@"fn".is_generic);
}
```

Shell

```
$ zig test test_fn_reflection.zig
1/1 test_fn_reflection.test.fn reflection...OK
All 1 tests passed.
```

Errors

Error Set Type

An error set is like an enum. However, each error name across the entire compilation gets assigned an unsigned integer greater than 0. You are allowed to declare the same error name more than once, and if you do, it gets assigned the same integer value.

The error set type defaults to a `u16`, though if the maximum number of distinct error values is provided via the `--error-limit [num]` command line parameter an integer type with the minimum number of bits required to represent all of the error values will be used.

You can coerce an error from a subset to a superset:

test_coerce_error_subset_to_superset.zig

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

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

const AllocationError = error{
    OutOfMemory,
};

test "coerce subset to superset" {
    const err = foo(AllocationError.OutOfMemory);
    try std.testing.expect(err == FileOpenError.OutOfMemory);
}

fn foo(err: AllocationError) FileOpenError {
    return err;
}
```

Shell

```
$ zig test test_coerce_error_subset_to_superset.zig
1/1 test_coerce_error_subset_to_superset.test.coerce subset to superset...OK
All 1 tests passed.
```

But you cannot coerce an error from a superset to a subset:

test_coerce_error_superset_to_subset.zig

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

const AllocationError = error{
    OutOfMemory,
};
```

```
test "coerce superset to subset" {
    foo(FileOpenError.OutOfMemory) catch {};
}

fn foo(err: FileOpenError) AllocationError {
    return err;
}
```

Shell

```
$ zig test test_coerce_error_superset_to_subset.zig
doc/langref/test_coerce_error_superset_to_subset.zig:16:12: error: expected type 'error{OutOfMe
    return err;
           ^~~
doc/langref/test_coerce_error_superset_to_subset.zig:16:12: note: 'error.AccessDenied' not a me
doc/langref/test_coerce_error_superset_to_subset.zig:16:12: note: 'error.FileNotFound' not a me
doc/langref/test_coerce_error_superset_to_subset.zig:15:28: note: function return type declared
fn foo(err: FileOpenError) AllocationError {
                           ^~~~~~~~~~~~~~~
```

There is a shortcut for declaring an error set with only 1 value, and then getting that value:

single_value_error_set_shortcut.zig

```
const err = error.FileNotFound;
```

This is equivalent to:

single_value_error_set.zig

```
const err = (error{FileNotFound}).FileNotFound;
```

This becomes useful when using Inferred Error Sets.

The Global Error Set

`anyerror` refers to the global error set. This is the error set that contains all errors in the entire compilation unit, i.e. it is the union of all other error sets.

You can coerce any error set to the global one, and you can explicitly cast an error of the global error set to a non-global one. This inserts a language-level assert to make sure the error value is in fact in the destination error set.

The global error set should generally be avoided because it prevents the compiler from knowing what errors are possible at compile-time. Knowing the error set at compile-time is better for generated documentation and helpful error messages, such as forgetting a possible error value in a switch.

Error Union Type

An error set type and normal type can be combined with the `!` binary operator to form an error union type. You are likely to use an error union type more often than an error set type by itself.

Here is a function to parse a string into a 64-bit integer:

error_union_parsing_u64.zig

```zig
const std = @import("std");
const maxInt = std.math.maxInt;

pub fn parseU64(buf: []const u8, radix: u8) !u64 {
    var x: u64 = 0;

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

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

        // x *= radix
        var ov = @mulWithOverflow(x, radix);
        if (ov[1] != 0) return error.OverFlow;

        // x += digit
        ov = @addWithOverflow(ov[0], digit);
        if (ov[1] != 0) return error.OverFlow;
        x = ov[0];
    }

    return x;
}

fn charToDigit(c: u8) u8 {
    return switch (c) {
        '0'...'9' => c - '0',
        'A'...'Z' => c - 'A' + 10,
        'a'...'z' => c - 'a' + 10,
        else => maxInt(u8),
    };
}

test "parse u64" {
    const result = try parseU64("1234", 10);
    try std.testing.expect(result == 1234);
}
```

Shell

```
$ zig test error_union_parsing_u64.zig
1/1 error_union_parsing_u64.test.parse u64...OK
All 1 tests passed.
```

Notice the return type is `!u64`. This means that the function either returns an unsigned 64 bit integer, or an error. We left off the error set to the left of the `!`, so the error set is inferred.

Within the function definition, you can see some return statements that return an error, and at the bottom a return statement that returns a `u64`. Both types coerce to `anyerror!u64`.

What it looks like to use this function varies depending on what you're trying to do. One of the following:

- You want to provide a default value if it returned an error.
- If it returned an error then you want to return the same error.
- You know with complete certainty it will not return an error, so want to unconditionally unwrap it.
- You want to take a different action for each possible error.

catch

If you want to provide a default value, you can use the `catch` binary operator:

catch.zig

```
const parseU64 = @import("error_union_parsing_u64.zig").parseU64;

fn doAThing(str: []u8) void {
    const number = parseU64(str, 10) catch 13;
    _ = number; // ...
}
```

In this code, `number` will be equal to the successfully parsed string, or a default value of 13. The type of the right hand side of the binary `catch` operator must match the unwrapped error union type, or be of type `noreturn`.

If you want to provide a default value with `catch` after performing some logic, you can combine `catch` with named Blocks:

handle_error_with_catch_block.zig.zig

```
const parseU64 = @import("error_union_parsing_u64.zig").parseU64;

fn doAThing(str: []u8) void {
    const number = parseU64(str, 10) catch blk: {
        // do things
        break :blk 13;
    };
    _ = number; // number is now initialized
}
```

try

Let's say you wanted to return the error if you got one, otherwise continue with the function logic:

catch_err_return.zig

```
const parseU64 = @import("error_union_parsing_u64.zig").parseU64;

fn doAThing(str: []u8) !void {
    const number = parseU64(str, 10) catch |err| return err;
    _ = number; // ...
}
```

There is a shortcut for this. The `try` expression:

try.zig

```
const parseU64 = @import("error_union_parsing_u64.zig").parseU64;

fn doAThing(str: []u8) !void {
    const number = try parseU64(str, 10);
    _ = number; // ...
}
```

`try` evaluates an error union expression. If it is an error, it returns from the current function with the same error. Otherwise, the expression results in the unwrapped value.

Maybe you know with complete certainty that an expression will never be an error. In this case you can do this:

`const number = parseU64("1234", 10) catch unreachable;`

Here we know for sure that "1234" will parse successfully. So we put the `unreachable` value on the right hand side. `unreachable` invokes safety-checked Illegal Behavior, so in Debug and ReleaseSafe, triggers a safety panic by default. So, while we're debugging the application, if there *was* a surprise error here, the application would crash appropriately.

You may want to take a different action for every situation. For that, we combine the if and switch expression:

handle_all_error_scenarios.zig

```
fn doAThing(str: []u8) void {
    if (parseU64(str, 10)) |number| {
        doSomethingWithNumber(number);
    } else |err| switch (err) {
        error.Overflow => {
            // handle overflow...
        },
        // we promise that InvalidChar won't happen (or crash in debug mode if it does)
        error.InvalidChar => unreachable,
    }
}
```

Finally, you may want to handle only some errors. For that, you can capture the unhandled errors in the `else` case, which now contains a narrower error set:

handle_some_error_scenarios.zig

```
fn doAnotherThing(str: []u8) error{InvalidChar}!void {
    if (parseU64(str, 10)) |number| {
        doSomethingWithNumber(number);
    } else |err| switch (err) {
        error.Overflow => {
            // handle overflow...
        },
        else => |leftover_err| return leftover_err,
    }
}
```

You must use the variable capture syntax. If you don't need the variable, you can capture with `_` and avoid the `switch`.

handle_no_error_scenarios.zig

```
fn doADifferentThing(str: []u8) void {
    if (parseU64(str, 10)) |number| {
        doSomethingWithNumber(number);
    } else |_| {
        // do as you'd like
    }
}
```

errdefer

The other component to error handling is defer statements. In addition to an unconditional defer, Zig has `errdefer`, which evaluates the deferred expression on block exit path if and only if the function returned with an error from the block.

Example:

errdefer_example.zig

```
fn createFoo(param: i32) !Foo {
    const foo = try tryToAllocateFoo();
    // now we have allocated foo. we need to free it if the function fails.
    // but we want to return it if the function succeeds.
    errdefer deallocateFoo(foo);

    const tmp_buf = allocateTmpBuffer() orelse return error.OutOfMemory;
    // tmp_buf is truly a temporary resource, and we for sure want to clean it up
    // before this block leaves scope
    defer deallocateTmpBuffer(tmp_buf);

    if (param > 1337) return error.InvalidParam;

    // here the errdefer will not run since we're returning success from the function.
```

```
    // but the defer will run!
    return foo;
}
```

The neat thing about this is that you get robust error handling without the verbosity and cognitive overhead of trying to make sure every exit path is covered. The deallocation code is always directly following the allocation code.

The `errdefer` statement can optionally capture the error:

test_errdefer_capture.zig

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

fn captureError(captured: *?anyerror) !void {
    errdefer |err| {
        captured.* = err;
    }
    return error.GeneralFailure;
}

test "errdefer capture" {
    var captured: ?anyerror = null;

    if (captureError(&captured)) unreachable else |err| {
        try std.testing.expectEqual(error.GeneralFailure, captured.?);
        try std.testing.expectEqual(error.GeneralFailure, err);
    }
}
```

Shell

```
$ zig test test_errdefer_capture.zig
1/1 test_errdefer_capture.test.errdefer capture...OK
All 1 tests passed.
```

A couple of other tidbits about error handling:

- These primitives give enough expressiveness that it's completely practical to have failing to check for an error be a compile error. If you really want to ignore the error, you can add `catch unreachable` and get the added benefit of crashing in Debug and ReleaseSafe modes if your assumption was wrong.
- Since Zig understands error types, it can pre-weight branches in favor of errors not occurring. Just a small optimization benefit that is not available in other languages.

See also:

- defer
- if
- switch

An error union is created with the `!` binary operator. You can use compile-time reflection to access the child type of an error union:

test_error_union.zig

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

test "error union" {
    var foo: anyerror!i32 = undefined;

    // Coerce from child type of an error union:
    foo = 1234;

    // Coerce from an error set:
    foo = error.SomeError;

    // Use compile-time reflection to access the payload type of an error union:
    try comptime expect(@typeInfo(@TypeOf(foo)).error_union.payload == i32);

    // Use compile-time reflection to access the error set type of an error union:
    try comptime expect(@typeInfo(@TypeOf(foo)).error_union.error_set == anyerror);
}
```

Shell

```
$ zig test test_error_union.zig
1/1 test_error_union.test.error union...OK
All 1 tests passed.
```

Merging Error Sets

Use the `||` operator to merge two error sets together. The resulting error set contains the errors of both error sets. Doc comments from the left-hand side override doc comments from the right-hand side. In this example, the doc comments for `C.PathNotFound` is `A doc comment`.

This is especially useful for functions which return different error sets depending on [comptime](#) branches. For example, the Zig standard library uses `LinuxFileOpenError || WindowsFileOpenError` for the error set of opening files.

test_merging_error_sets.zig

```
const A = error{
    NotDir,

    /// A doc comment
    PathNotFound,
};
const B = error{
    OutOfMemory,

    /// B doc comment
    PathNotFound,
};
```

```zig
const C = A || B;

fn foo() C!void {
    return error.NotDir;
}

test "merge error sets" {
    if (foo()) {
        @panic("unexpected");
    } else |err| switch (err) {
        error.OutOfMemory => @panic("unexpected"),
        error.PathNotFound => @panic("unexpected"),
        error.NotDir => {},
    }
}
```

Shell

```
$ zig test test_merging_error_sets.zig
1/1 test_merging_error_sets.test.merge error sets...OK
All 1 tests passed.
```

Inferred Error Sets

Because many functions in Zig return a possible error, Zig supports inferring the error set. To infer the error set for a function, prepend the `!` operator to the function's return type, like `!T`:

test_inferred_error_sets.zig

```zig
// With an inferred error set
pub fn add_inferred(comptime T: type, a: T, b: T) !T {
    const ov = @addWithOverflow(a, b);
    if (ov[1] != 0) return error.Overflow;
    return ov[0];
}

// With an explicit error set
pub fn add_explicit(comptime T: type, a: T, b: T) Error!T {
    const ov = @addWithOverflow(a, b);
    if (ov[1] != 0) return error.Overflow;
    return ov[0];
}

const Error = error{
    Overflow,
};

const std = @import("std");

test "inferred error set" {
    if (add_inferred(u8, 255, 1)) |_| unreachable else |err| switch (err) {
        error.Overflow => {}, // ok
    }
}
```

Shell

```
$ zig test test_inferred_error_sets.zig
1/1 test_inferred_error_sets.test.inferred error set...OK
All 1 tests passed.
```

When a function has an inferred error set, that function becomes generic and thus it becomes trickier to do certain things with it, such as obtain a function pointer, or have an error set that is consistent across different build targets. Additionally, inferred error sets are incompatible with recursion.

In these situations, it is recommended to use an explicit error set. You can generally start with an empty error set and let compile errors guide you toward completing the set.

These limitations may be overcome in a future version of Zig.

Error Return Traces

Error Return Traces show all the points in the code that an error was returned to the calling function. This makes it practical to use try everywhere and then still be able to know what happened if an error ends up bubbling all the way out of your application.

error_return_trace.zig

```zig
pub fn main() !void {
    try foo(12);
}

fn foo(x: i32) !void {
    if (x >= 5) {
        try bar();
    } else {
        try bang2();
    }
}

fn bar() !void {
    if (baz()) {
        try quux();
    } else |err| switch (err) {
        error.FileNotFound => try hello(),
    }
}

fn baz() !void {
    try bang1();
}

fn quux() !void {
    try bang2();
}
```

```
fn hello() !void {
    try bang2();
}

fn bang1() !void {
    return error.FileNotFound;
}

fn bang2() !void {
    return error.PermissionDenied;
}
```

Shell

```
$ zig build-exe error_return_trace.zig
$ ./error_return_trace
error: PermissionDenied
/home/andy/src/zig/doc/langref/error_return_trace.zig:34:5: 0x10de6f8 in bang1 (error_return_tr
    return error.FileNotFound;
    ^
/home/andy/src/zig/doc/langref/error_return_trace.zig:22:5: 0x10de723 in baz (error_return_trac
    try bang1();
    ^
/home/andy/src/zig/doc/langref/error_return_trace.zig:38:5: 0x10de748 in bang2 (error_return_tr
    return error.PermissionDenied;
    ^
/home/andy/src/zig/doc/langref/error_return_trace.zig:30:5: 0x10de7b3 in hello (error_return_tr
    try bang2();
    ^
/home/andy/src/zig/doc/langref/error_return_trace.zig:17:31: 0x10de858 in bar (error_return_tra
        error.FileNotFound => try hello(),
                              ^
/home/andy/src/zig/doc/langref/error_return_trace.zig:7:9: 0x10de8c0 in foo (error_return_trace
        try bar();
        ^
/home/andy/src/zig/doc/langref/error_return_trace.zig:2:5: 0x10de918 in main (error_return_trac
    try foo(12);
    ^
```

Look closely at this example. This is no stack trace.

You can see that the final error bubbled up was `PermissionDenied`, but the original error that started this whole thing was `FileNotFound`. In the `bar` function, the code handles the original error code, and then returns another one, from the switch statement. Error Return Traces make this clear, whereas a stack trace would look like this:

stack_trace.zig

```
pub fn main() void {
    foo(12);
}

fn foo(x: i32) void {
    if (x >= 5) {
        bar();
```

```
    } else {
        bang2();
    }
}

fn bar() void {
    if (baz()) {
        quux();
    } else {
        hello();
    }
}

fn baz() bool {
    return bang1();
}

fn quux() void {
    bang2();
}

fn hello() void {
    bang2();
}

fn bang1() bool {
    return false;
}

fn bang2() void {
    @panic("PermissionDenied");
}
```

Shell

```
$ zig build-exe stack_trace.zig
$ ./stack_trace
thread 216810 panic: PermissionDenied
/home/andy/src/zig/doc/langref/stack_trace.zig:38:5: 0x10df2bc in bang2 (stack_trace)
    @panic("PermissionDenied");
    ^
/home/andy/src/zig/doc/langref/stack_trace.zig:30:10: 0x10dfb38 in hello (stack_trace)
    bang2();
         ^
/home/andy/src/zig/doc/langref/stack_trace.zig:17:14: 0x10df290 in bar (stack_trace)
        hello();
             ^
/home/andy/src/zig/doc/langref/stack_trace.zig:7:12: 0x10df0b4 in foo (stack_trace)
        bar();
           ^
/home/andy/src/zig/doc/langref/stack_trace.zig:2:8: 0x10de84d in main (stack_trace)
    foo(12);
       ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de352 in posixCallMainAndExit (stack_trace)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf2d in _start (stack_trace)
```

```
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Here, the stack trace does not explain how the control flow in `bar` got to the `hello()` call. One would have to open a debugger or further instrument the application in order to find out. The error return trace, on the other hand, shows exactly how the error bubbled up.

This debugging feature makes it easier to iterate quickly on code that robustly handles all error conditions. This means that Zig developers will naturally find themselves writing correct, robust code in order to increase their development pace.

Error Return Traces are enabled by default in Debug and ReleaseSafe builds and disabled by default in ReleaseFast and ReleaseSmall builds.

There are a few ways to activate this error return tracing feature:

- Return an error from main
- An error makes its way to `catch unreachable` and you have not overridden the default panic handler
- Use errorReturnTrace to access the current return trace. You can use `std.debug.dumpStackTrace` to print it. This function returns comptime-known null when building without error return tracing support.

Implementation Details

To analyze performance cost, there are two cases:

- when no errors are returned
- when returning errors

For the case when no errors are returned, the cost is a single memory write operation, only in the first non-failable function in the call graph that calls a failable function, i.e. when a function returning `void` calls a function returning `error`. This is to initialize this struct in the stack memory:

stack_trace_struct.zig

```
pub const StackTrace = struct {
    index: usize,
    instruction_addresses: [N]usize,
};
```

Here, N is the maximum function call depth as determined by call graph analysis. Recursion is ignored and counts for 2.

A pointer to `StackTrace` is passed as a secret parameter to every function that can return an error, but it's always the first parameter, so it can likely sit in a register and stay there.

That's it for the path when no errors occur. It's practically free in terms of performance.

When generating the code for a function that returns an error, just before the `return` statement (only for the `return` statements that return errors), Zig generates a call to this function:

zig_return_error_fn.zig

```
// marked as "no-inline" in LLVM IR
fn __zig_return_error(stack_trace: *StackTrace) void {
    stack_trace.instruction_addresses[stack_trace.index] = @returnAddress();
    stack_trace.index = (stack_trace.index + 1) % N;
}
```

The cost is 2 math operations plus some memory reads and writes. The memory accessed is constrained and should remain cached for the duration of the error return bubbling.

As for code size cost, 1 function call before a return statement is no big deal. Even so, I have a plan to make the call to `__zig_return_error` a tail call, which brings the code size cost down to actually zero. What is a return statement in code without error return tracing can become a jump instruction in code with error return tracing.

Optionals

One area that Zig provides safety without compromising efficiency or readability is with the optional type.

The question mark symbolizes the optional type. You can convert a type to an optional type by putting a question mark in front of it, like this:

optional_integer.zig

```
// normal integer
const normal_int: i32 = 1234;

// optional integer
const optional_int: ?i32 = 5678;
```

Now the variable `optional_int` could be an `i32`, or `null`.

Instead of integers, let's talk about pointers. Null references are the source of many runtime exceptions, and even stand accused of being the worst mistake of computer science.

Zig does not have them.

Instead, you can use an optional pointer. This secretly compiles down to a normal pointer, since we know we can use 0 as the null value for the optional type. But the compiler can check your work and make sure you don't assign null to something that can't be null.

Typically the downside of not having null is that it makes the code more verbose to write. But, let's compare some equivalent C code and Zig code.

Task: call malloc, if the result is null, return null.

C code

call_malloc_in_c.c

```
// malloc prototype included for reference
void *malloc(size_t size);

struct Foo *do_a_thing(void) {
    char *ptr = malloc(1234);
    if (!ptr) return NULL;
    // ...
}
```

Zig code

call_malloc_from_zig.zig

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

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

Here, Zig is at least as convenient, if not more, than C. And, the type of "ptr" is `[*]u8` *not* `?[*]u8`. The `orelse` keyword unwrapped the optional type and therefore `ptr` is guaranteed to be non-null everywhere it is used in the function.

The other form of checking against NULL you might see looks like this:

checking_null_in_c.c

```
void do_a_thing(struct Foo *foo) {
    // do some stuff

    if (foo) {
        do_something_with_foo(foo);
    }

    // do some stuff
```

```
}
```

In Zig you can accomplish the same thing:

checking_null_in_zig.zig

```
const Foo = struct {};
fn doSomethingWithFoo(foo: *Foo) void {
    _ = foo;
}

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

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

    // do some stuff
}
```

Once again, the notable thing here is that inside the if block, `foo` is no longer an optional pointer, it is a pointer, which cannot be null.

One benefit to this is that functions which take pointers as arguments can be annotated with the "nonnull" attribute - `__attribute__((nonnull))` in GCC. The optimizer can sometimes make better decisions knowing that pointer arguments cannot be null.

Optional Type

An optional is created by putting `?` in front of a type. You can use compile-time reflection to access the child type of an optional:

test_optional_type.zig

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

test "optional type" {
    // Declare an optional and coerce from null:
    var foo: ?i32 = null;

    // Coerce from child type of an optional
    foo = 1234;

    // Use compile-time reflection to access the child type of the optional:
    try comptime expect(@typeInfo(@TypeOf(foo)).optional.child == i32);
}
```

Shell

```
$ zig test test_optional_type.zig
1/1 test_optional_type.test.optional type...OK
All 1 tests passed.
```

null

Just like undefined, `null` has its own type, and the only way to use it is to cast it to a different type:

null.zig

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

Optional Pointers

An optional pointer is guaranteed to be the same size as a pointer. The `null` of the optional is guaranteed to be address 0.

test_optional_pointer.zig

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

test "optional pointers" {
    // Pointers cannot be null. If you want a null pointer, use the optional
    // prefix `?` to make the pointer type optional.
    var ptr: ?*i32 = null;

    var x: i32 = 1;
    ptr = &x;

    try expect(ptr.?.* == 1);

    // Optional pointers are the same size as normal pointers, because pointer
    // value 0 is used as the null value.
    try expect(@sizeOf(?*i32) == @sizeOf(*i32));
}
```

Shell

```
$ zig test test_optional_pointer.zig
1/1 test_optional_pointer.test.optional pointers...OK
All 1 tests passed.
```

See also:

- while with Optionals
- if with Optionals

Casting

A **type cast** converts a value of one type to another. Zig has Type Coercion for conversions that are known to be completely safe and unambiguous, and Explicit Casts for conversions that one would not want to happen on accident. There is also a third kind of type conversion called Peer

Type Resolution for the case when a result type must be decided given multiple operand types.

Type Coercion

Type coercion occurs when one type is expected, but different type is provided:

test_type_coercion.zig

```
test "type coercion - variable declaration" {
    const a: u8 = 1;
    const b: u16 = a;
    _ = b;
}

test "type coercion - function call" {
    const a: u8 = 1;
    foo(a);
}

fn foo(b: u16) void {
    _ = b;
}

test "type coercion - @as builtin" {
    const a: u8 = 1;
    const b = @as(u16, a);
    _ = b;
}
```

Shell

```
$ zig test test_type_coercion.zig
1/3 test_type_coercion.test.type coercion - variable declaration...OK
2/3 test_type_coercion.test.type coercion - function call...OK
3/3 test_type_coercion.test.type coercion - @as builtin...OK
All 3 tests passed.
```

Type coercions are only allowed when it is completely unambiguous how to get from one type to another, and the transformation is guaranteed to be safe. There is one exception, which is C Pointers.

Type Coercion: Stricter Qualification

Values which have the same representation at runtime can be cast to increase the strictness of the qualifiers, no matter how nested the qualifiers are:

- `const` - non-const to const is allowed
- `volatile` - non-volatile to volatile is allowed
- `align` - bigger to smaller alignment is allowed
- error sets to supersets is allowed

These casts are no-ops at runtime since the value representation does not change.

test_no_op_casts.zig

```
test "type coercion - const qualification" {
    var a: i32 = 1;
    const b: *i32 = &a;
    foo(b);
}

fn foo(_: *const i32) void {}
```

Shell

```
$ zig test test_no_op_casts.zig
1/1 test_no_op_casts.test.type coercion - const qualification...OK
All 1 tests passed.
```

In addition, pointers coerce to const optional pointers:

test_pointer_coerce_const_optional.zig

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

test "cast *[1][*:0]const u8 to []const ?[*:0]const u8" {
    const window_name = [1][*:0]const u8{"window name"};
    const x: []const ?[*:0]const u8 = &window_name;
    try expect(mem.eql(u8, mem.span(x[0].?), "window name"));
}
```

Shell

```
$ zig test test_pointer_coerce_const_optional.zig
1/1 test_pointer_coerce_const_optional.test.cast *[1][*:0]const u8 to []const ?[*:0]const u8...
All 1 tests passed.
```

Type Coercion: Integer and Float Widening

Integers coerce to integer types which can represent every value of the old type, and likewise Floats coerce to float types which can represent every value of the old type.

test_integer_widening.zig

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

test "integer widening" {
    const a: u8 = 250;
    const b: u16 = a;
    const c: u32 = b;
    const d: u64 = c;
    const e: u64 = d;
```

```
    const f: u128 = e;
    try expect(f == a);
}

test "implicit unsigned integer to signed integer" {
    const a: u8 = 250;
    const b: i16 = a;
    try expect(b == 250);
}

test "float widening" {
    const a: f16 = 12.34;
    const b: f32 = a;
    const c: f64 = b;
    const d: f128 = c;
    try expect(d == a);
}
```

Shell

```
$ zig test test_integer_widening.zig
1/3 test_integer_widening.test.integer widening...OK
2/3 test_integer_widening.test.implicit unsigned integer to signed integer...OK
3/3 test_integer_widening.test.float widening...OK
All 3 tests passed.
```

Type Coercion: Float to Int

A compiler error is appropriate because this ambiguous expression leaves the compiler two choices about the coercion.

- Cast `54.0` to `comptime_int` resulting in `@as(comptime_int, 10)`, which is casted to `@as(f32, 10)`
- Cast `5` to `comptime_float` resulting in `@as(comptime_float, 10.8)`, which is casted to `@as(f32, 10.8)`

test_ambiguous_coercion.zig

```
// Compile time coercion of float to int
test "implicit cast to comptime_int" {
    const f: f32 = 54.0 / 5;
    _ = f;
}
```

Shell

```
$ zig test test_ambiguous_coercion.zig
doc/langref/test_ambiguous_coercion.zig:3:25: error: ambiguous coercion of division operands 'c
    const f: f32 = 54.0 / 5;
                   ~~~~~^~~
```

Type Coercion: Slices, Arrays and Pointers

test_coerce_slices_arrays_and_pointers.zig

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

// You can assign constant pointers to arrays to a slice with
// const modifier on the element type. Useful in particular for
// String literals.
test "*const [N]T to []const T" {
    const x1: []const u8 = "hello";
    const x2: []const u8 = &[5]u8{ 'h', 'e', 'l', 'l', 111 };
    try expect(std.mem.eql(u8, x1, x2));

    const y: []const f32 = &[2]f32{ 1.2, 3.4 };
    try expect(y[0] == 1.2);
}

// Likewise, it works when the destination type is an error union.
test "*const [N]T to E![]const T" {
    const x1: anyerror![]const u8 = "hello";
    const x2: anyerror![]const u8 = &[5]u8{ 'h', 'e', 'l', 'l', 111 };
    try expect(std.mem.eql(u8, try x1, try x2));

    const y: anyerror![]const f32 = &[2]f32{ 1.2, 3.4 };
    try expect((try y)[0] == 1.2);
}

// Likewise, it works when the destination type is an optional.
test "*const [N]T to ?[]const T" {
    const x1: ?[]const u8 = "hello";
    const x2: ?[]const u8 = &[5]u8{ 'h', 'e', 'l', 'l', 111 };
    try expect(std.mem.eql(u8, x1.?, x2.?));

    const y: ?[]const f32 = &[2]f32{ 1.2, 3.4 };
    try expect(y.?[0] == 1.2);
}

// In this cast, the array length becomes the slice length.
test "*[N]T to []T" {
    var buf: [5]u8 = "hello".*;
    const x: []u8 = &buf;
    try expect(std.mem.eql(u8, x, "hello"));

    const buf2 = [2]f32{ 1.2, 3.4 };
    const x2: []const f32 = &buf2;
    try expect(std.mem.eql(f32, x2, &[2]f32{ 1.2, 3.4 }));
}

// Single-item pointers to arrays can be coerced to many-item pointers.
test "*[N]T to [*]T" {
    var buf: [5]u8 = "hello".*;
    const x: [*]u8 = &buf;
    try expect(x[4] == 'o');
    // x[5] would be an uncaught out of bounds pointer dereference!
}

// Likewise, it works when the destination type is an optional.
test "*[N]T to ?[*]T" {
```

```
    var buf: [5]u8 = "hello".*;
    const x: ?[*]u8 = &buf;
    try expect(x.?[4] == 'o');
}

// Single-item pointers can be cast to len-1 single-item arrays.
test "*T to *[1]T" {
    var x: i32 = 1234;
    const y: *[1]i32 = &x;
    const z: [*]i32 = y;
    try expect(z[0] == 1234);
}
```

Shell

```
$ zig test test_coerce_slices_arrays_and_pointers.zig
1/7 test_coerce_slices_arrays_and_pointers.test.*const [N]T to []const T...OK
2/7 test_coerce_slices_arrays_and_pointers.test.*const [N]T to E![]const T...OK
3/7 test_coerce_slices_arrays_and_pointers.test.*const [N]T to ?[]const T...OK
4/7 test_coerce_slices_arrays_and_pointers.test.*[N]T to []T...OK
5/7 test_coerce_slices_arrays_and_pointers.test.*[N]T to [*]T...OK
6/7 test_coerce_slices_arrays_and_pointers.test.*[N]T to ?[*]T...OK
7/7 test_coerce_slices_arrays_and_pointers.test.*T to *[1]T...OK
All 7 tests passed.
```

See also:

- C Pointers

Type Coercion: Optionals

The payload type of Optionals, as well as null, coerce to the optional type.

test_coerce_optionals.zig

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

test "coerce to optionals" {
    const x: ?i32 = 1234;
    const y: ?i32 = null;

    try expect(x.? == 1234);
    try expect(y == null);
}
```

Shell

```
$ zig test test_coerce_optionals.zig
1/1 test_coerce_optionals.test.coerce to optionals...OK
All 1 tests passed.
```

Optionals work nested inside the Error Union Type, too:

test_coerce_optional_wrapped_error_union.zig

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

test "coerce to optionals wrapped in error union" {
    const x: anyerror!?i32 = 1234;
    const y: anyerror!?i32 = null;

    try expect((try x).? == 1234);
    try expect((try y) == null);
}
```

Shell

```
$ zig test test_coerce_optional_wrapped_error_union.zig
1/1 test_coerce_optional_wrapped_error_union.test.coerce to optionals wrapped in error union...
All 1 tests passed.
```

Type Coercion: Error Unions

The payload type of an Error Union Type as well as the Error Set Type coerce to the error union type:

test_coerce_to_error_union.zig

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

test "coercion to error unions" {
    const x: anyerror!i32 = 1234;
    const y: anyerror!i32 = error.Failure;

    try expect((try x) == 1234);
    try std.testing.expectError(error.Failure, y);
}
```

Shell

```
$ zig test test_coerce_to_error_union.zig
1/1 test_coerce_to_error_union.test.coercion to error unions...OK
All 1 tests passed.
```

Type Coercion: Compile-Time Known Numbers

When a number is comptime-known to be representable in the destination type, it may be coerced:

test_coerce_large_to_small.zig

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

test "coercing large integer type to smaller one when value is comptime-known to fit" {
    const x: u64 = 255;
    const y: u8 = x;
```

```
    try expect(y == 255);
}
```

Shell

```
$ zig test test_coerce_large_to_small.zig
1/1 test_coerce_large_to_small.test.coercing large integer type to smaller one when value is co
All 1 tests passed.
```

Type Coercion: Unions and Enums

Tagged unions can be coerced to enums, and enums can be coerced to tagged unions when they are comptime-known to be a field of the union that has only one possible value, such as void:

test_coerce_unions_enums.zig

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

const E = enum {
    one,
    two,
    three,
};

const U = union(E) {
    one: i32,
    two: f32,
    three,
};

const U2 = union(enum) {
    a: void,
    b: f32,

    fn tag(self: U2) usize {
        switch (self) {
            .a => return 1,
            .b => return 2,
        }
    }
};

test "coercion between unions and enums" {
    const u = U{ .two = 12.34 };
    const e: E = u; // coerce union to enum
    try expect(e == E.two);

    const three = E.three;
    const u_2: U = three; // coerce enum to union
    try expect(u_2 == E.three);

    const u_3: U = .three; // coerce enum literal to union
    try expect(u_3 == E.three);

    const u_4: U2 = .a; // coerce enum literal to union with inferred enum tag type.
```

```
    try expect(u_4.tag() == 1);

    // The following example is invalid.
    // error: coercion from enum '@TypeOf(.enum_literal)' to union 'test_coerce_unions_enum.U2
    //var u_5: U2 = .b;
    //try expect(u_5.tag() == 2);
}
```

Shell

```
$ zig test test_coerce_unions_enums.zig
1/1 test_coerce_unions_enums.test.coercion between unions and enums...OK
All 1 tests passed.
```

See also:

- union
- enum

Type Coercion: undefined

undefined can be coerced to any type.

Type Coercion: Tuples to Arrays

Tuples can be coerced to arrays, if all of the fields have the same type.

test_coerce_tuples_arrays.zig

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

const Tuple = struct { u8, u8 };
test "coercion from homogeneous tuple to array" {
    const tuple: Tuple = .{ 5, 6 };
    const array: [2]u8 = tuple;
    _ = array;
}
```

Shell

```
$ zig test test_coerce_tuples_arrays.zig
1/1 test_coerce_tuples_arrays.test.coercion from homogeneous tuple to array...OK
All 1 tests passed.
```

Explicit Casts

Explicit casts are performed via Builtin Functions. Some explicit casts are safe; some are not. Some explicit casts perform language-level assertions; some do not. Some explicit casts are no-ops at runtime; some are not.

- @bitCast - change type but maintain bit representation
- @alignCast - make a pointer have more alignment
- @enumFromInt - obtain an enum value based on its integer tag value
- @errorFromInt - obtain an error code based on its integer value
- @errorCast - convert to a smaller error set
- @floatCast - convert a larger float to a smaller float
- @floatFromInt - convert an integer to a float value
- @intCast - convert between integer types
- @intFromBool - convert true to 1 and false to 0
- @intFromEnum - obtain the integer tag value of an enum or tagged union
- @intFromError - obtain the integer value of an error code
- @intFromFloat - obtain the integer part of a float value
- @intFromPtr - obtain the address of a pointer
- @ptrFromInt - convert an address to a pointer
- @ptrCast - convert between pointer types
- @truncate - convert between integer types, chopping off bits

Peer Type Resolution

Peer Type Resolution occurs in these places:

- switch expressions
- if expressions
- while expressions
- for expressions
- Multiple break statements in a block
- Some binary operations

This kind of type resolution chooses a type that all peer types can coerce into. Here are some examples:

test_peer_type_resolution.zig

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

test "peer resolve int widening" {
    const a: i8 = 12;
    const b: i16 = 34;
    const c = a + b;
    try expect(c == 46);
    try expect(@TypeOf(c) == i16);
}
```

```
test "peer resolve arrays of different size to const slice" {
    try expect(mem.eql(u8, boolToStr(true), "true"));
    try expect(mem.eql(u8, boolToStr(false), "false"));
    try comptime expect(mem.eql(u8, boolToStr(true), "true"));
    try comptime expect(mem.eql(u8, boolToStr(false), "false"));
}
fn boolToStr(b: bool) []const u8 {
    return if (b) "true" else "false";
}

test "peer resolve array and const slice" {
    try testPeerResolveArrayConstSlice(true);
    try comptime testPeerResolveArrayConstSlice(true);
}
fn testPeerResolveArrayConstSlice(b: bool) !void {
    const value1 = if (b) "aoeu" else @as([]const u8, "zz");
    const value2 = if (b) @as([]const u8, "zz") else "aoeu";
    try expect(mem.eql(u8, value1, "aoeu"));
    try expect(mem.eql(u8, value2, "zz"));
}

test "peer type resolution: ?T and T" {
    try expect(peerTypeTAndOptionalT(true, false).? == 0);
    try expect(peerTypeTAndOptionalT(false, false).? == 3);
    comptime {
        try expect(peerTypeTAndOptionalT(true, false).? == 0);
        try expect(peerTypeTAndOptionalT(false, false).? == 3);
    }
}
fn peerTypeTAndOptionalT(c: bool, b: bool) ?usize {
    if (c) {
        return if (b) null else @as(usize, 0);
    }

    return @as(usize, 3);
}

test "peer type resolution: *[0]u8 and []const u8" {
    try expect(peerTypeEmptyArrayAndSlice(true, "hi").len == 0);
    try expect(peerTypeEmptyArrayAndSlice(false, "hi").len == 1);
    comptime {
        try expect(peerTypeEmptyArrayAndSlice(true, "hi").len == 0);
        try expect(peerTypeEmptyArrayAndSlice(false, "hi").len == 1);
    }
}
fn peerTypeEmptyArrayAndSlice(a: bool, slice: []const u8) []const u8 {
    if (a) {
        return &[_]u8{};
    }

    return slice[0..1];
}
test "peer type resolution: *[0]u8, []const u8, and anyerror![]u8" {
    {
        var data = "hi".*;
        const slice = data[0..];
        try expect((try peerTypeEmptyArrayAndSliceAndError(true, slice)).len == 0);
```

```
        try expect((try peerTypeEmptyArrayAndSliceAndError(false, slice)).len == 1);
    }
    comptime {
        var data = "hi".*;
        const slice = data[0..];
        try expect((try peerTypeEmptyArrayAndSliceAndError(true, slice)).len == 0);
        try expect((try peerTypeEmptyArrayAndSliceAndError(false, slice)).len == 1);
    }
}
fn peerTypeEmptyArrayAndSliceAndError(a: bool, slice: []u8) anyerror![]u8 {
    if (a) {
        return &[_]u8{};
    }

    return slice[0..1];
}

test "peer type resolution: *const T and ?*T" {
    const a: *const usize = @ptrFromInt(0x123456780);
    const b: ?*usize = @ptrFromInt(0x123456780);
    try expect(a == b);
    try expect(b == a);
}

test "peer type resolution: error union switch" {
    // The non-error and error cases are only peers if the error case is just a switch expressi
    // the pattern `if (x) {...} else |err| blk: { switch (err) {...} }` does not consider the
    // non-error and error case to be peers.
    var a: error{ A, B, C }!u32 = 0;
    _ = &a;
    const b = if (a) |x|
        x + 3
    else |err| switch (err) {
        error.A => 0,
        error.B => 1,
        error.C => null,
    };
    try expect(@TypeOf(b) == ?u32);

    // The non-error and error cases are only peers if the error case is just a switch expressi
    // the pattern `x catch |err| blk: { switch (err) {...} }` does not consider the unwrapped
    // and error case to be peers.
    const c = a catch |err| switch (err) {
        error.A => 0,
        error.B => 1,
        error.C => null,
    };
    try expect(@TypeOf(c) == ?u32);
}
```

Shell

```
$ zig test test_peer_type_resolution.zig
1/8 test_peer_type_resolution.test.peer resolve int widening...OK
2/8 test_peer_type_resolution.test.peer resolve arrays of different size to const slice...OK
3/8 test_peer_type_resolution.test.peer resolve array and const slice...OK
4/8 test_peer_type_resolution.test.peer type resolution: ?T and T...OK
```

```
5/8 test_peer_type_resolution.test.peer type resolution: *[0]u8 and []const u8...OK
6/8 test_peer_type_resolution.test.peer type resolution: *[0]u8, []const u8, and anyerror![]u8.
7/8 test_peer_type_resolution.test.peer type resolution: *const T and ?*T...OK
8/8 test_peer_type_resolution.test.peer type resolution: error union switch...OK
All 8 tests passed.
```

Zero Bit Types

For some types, @sizeOf is 0:

- void
- The Integers `u0` and `i0`.
- Arrays and Vectors with len 0, or with an element type that is a zero bit type.
- An enum with only 1 tag.
- A struct with all fields being zero bit types.
- A union with only 1 field which is a zero bit type.

These types can only ever have one possible value, and thus require 0 bits to represent. Code that makes use of these types is not included in the final generated code:

zero_bit_types.zig

```
export fn entry() void {
    var x: void = {};
    var y: void = {};
    x = y;
    y = x;
}
```

When this turns into machine code, there is no code generated in the body of `entry`, even in Debug mode. For example, on x86_64:

```
0000000000000010 <entry>:
  10:   55                      push   %rbp
  11:   48 89 e5                mov    %rsp,%rbp
  14:   5d                      pop    %rbp
  15:   c3                      retq
```

These assembly instructions do not have any code associated with the void values - they only perform the function call prologue and epilogue.

void

`void` can be useful for instantiating generic types. For example, given a `Map(Key, Value)`, one can pass `void` for the `Value` type to make it into a `Set`:

test_void_in_hashmap.zig

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

test "turn HashMap into a set with void" {
    var map = std.AutoHashMap(i32, void).init(std.testing.allocator);
    defer map.deinit();

    try map.put(1, {});
    try map.put(2, {});

    try expect(map.contains(2));
    try expect(!map.contains(3));

    _ = map.remove(2);
    try expect(!map.contains(2));
}
```

Shell

```
$ zig test test_void_in_hashmap.zig
1/1 test_void_in_hashmap.test.turn HashMap into a set with void...OK
All 1 tests passed.
```

Note that this is different from using a dummy value for the hash map value. By using `void` as the type of the value, the hash map entry type has no value field, and thus the hash map takes up less space. Further, all the code that deals with storing and loading the value is deleted, as seen above.

`void` is distinct from `anyopaque`. `void` has a known size of 0 bytes, and `anyopaque` has an unknown, but non-zero, size.

Expressions of type `void` are the only ones whose value can be ignored. For example, ignoring a non-`void` expression is a compile error:

test_expression_ignored.zig

```
test "ignoring expression value" {
    foo();
}

fn foo() i32 {
    return 1234;
}
```

Shell

```
$ zig test test_expression_ignored.zig
doc/langref/test_expression_ignored.zig:2:8: error: value of type 'i32' ignored
    foo();
    ~~~^~
doc/langref/test_expression_ignored.zig:2:8: note: all non-void values must be used
doc/langref/test_expression_ignored.zig:2:8: note: to discard the value, assign it to '_'
```

However, if the expression has type `void`, there will be no error. Expression results can be explicitly ignored by assigning them to `_`.

test_void_ignored.zig

```
test "void is ignored" {
    returnsVoid();
}

test "explicitly ignoring expression value" {
    _ = foo();
}

fn returnsVoid() void {}

fn foo() i32 {
    return 1234;
}
```

Shell

```
$ zig test test_void_ignored.zig
1/2 test_void_ignored.test.void is ignored...OK
2/2 test_void_ignored.test.explicitly ignoring expression value...OK
All 2 tests passed.
```

Result Location Semantics

During compilation, every Zig expression and sub-expression is assigned optional result location information. This information dictates what type the expression should have (its result type), and where the resulting value should be placed in memory (its result location). The information is optional in the sense that not every expression has this information: assignment to `_`, for instance, does not provide any information about the type of an expression, nor does it provide a concrete memory location to place it in.

As a motivating example, consider the statement `const x: u32 = 42;`. The type annotation here provides a result type of `u32` to the initialization expression `42`, instructing the compiler to coerce this integer (initially of type `comptime_int`) to this type. We will see more examples shortly.

This is not an implementation detail: the logic outlined above is codified into the Zig language specification, and is the primary mechanism of type inference in the language. This system is collectively referred to as "Result Location Semantics".

Result Types

Result types are propagated recursively through expressions where possible. For instance, if the expression `&e` has result type `*u32`, then `e` is given a result type of `u32`, allowing the language to perform this coercion before taking a reference.

The result type mechanism is utilized by casting builtins such as `@intCast`. Rather than taking as an argument the type to cast to, these builtins use their result type to determine this information. The result type is often known from context; where it is not, the `@as` builtin can be used to explicitly provide a result type.

We can break down the result types for each component of a simple expression as follows:

result_type_propagation.zig

```zig
const expectEqual = @import("std").testing.expectEqual;
test "result type propagates through struct initializer" {
    const S = struct { x: u32 };
    const val: u64 = 123;
    const s: S = .{ .x = @intCast(val) };
    // .{ .x = @intCast(val) }   has result type `S` due to the type annotation
    //         @intCast(val)     has result type `u32` due to the type of the field `S.x`
    //                  val      has no result type, as it is permitted to be any integer type
    try expectEqual(@as(u32, 123), s.x);
}
```

Shell

```
$ zig test result_type_propagation.zig
1/1 result_type_propagation.test.result type propagates through struct initializer...OK
All 1 tests passed.
```

This result type information is useful for the aforementioned cast builtins, as well as to avoid the construction of pre-coercion values, and to avoid the need for explicit type coercions in some cases. The following table details how some common expressions propagate result types, where `x` and `y` are arbitrary sub-expressions.

Expression	Parent Result Type	Sub-expression Result Type
`const val: T = x`	-	`x` is a `T`
`var val: T = x`	-	`x` is a `T`
`val = x`	-	`x` is a `@TypeOf(val)`
`@as(T, x)`	-	`x` is a `T`
`&x`	`*T`	`x` is a `T`
`&x`	`[]T`	`x` is some array of `T`
`f(x)`	-	`x` has the type of the first parameter of `f`
`.{x}`	`T`	`x` is a `std.meta.FieldType(T, .@"0")`

Expression	Parent Result Type	Sub-expression Result Type
`.{ .a = x }`	`T`	`x` is a `std.meta.FieldType(T, .a)`
`T{x}`	-	`x` is a `std.meta.FieldType(T, .@"0")`
`T{ .a = x }`	-	`x` is a `std.meta.FieldType(T, .a)`
`@Type(x)`	-	`x` is a `std.builtin.Type`
`@typeInfo(x)`	-	`x` is a `type`
`x << y`	-	`y` is a `std.math.Log2IntCeil(@TypeOf(x))`

Result Locations

In addition to result type information, every expression may be optionally assigned a result location: a pointer to which the value must be directly written. This system can be used to prevent intermediate copies when initializing data structures, which can be important for types which must have a fixed memory address ("pinned" types).

When compiling the simple assignment expression `x = e`, many languages would create the temporary value `e` on the stack, and then assign it to `x`, potentially performing a type coercion in the process. Zig approaches this differently. The expression `e` is given a result type matching the type of `x`, and a result location of `&x`. For many syntactic forms of `e`, this has no practical impact. However, it can have important semantic effects when working with more complex syntax forms.

For instance, if the expression `.{ .a = x, .b = y }` has a result location of `ptr`, then `x` is given a result location of `&ptr.a`, and `y` a result location of `&ptr.b`. Without this system, this expression would construct a temporary struct value entirely on the stack, and only then copy it to the destination address. In essence, Zig desugars the assignment `foo = .{ .a = x, .b = y }` to the two statements `foo.a = x; foo.b = y;`.

This can sometimes be important when assigning an aggregate value where the initialization expression depends on the previous value of the aggregate. The easiest way to demonstrate this is by attempting to swap fields of a struct or array - the following logic looks sound, but in fact is not:

result_location_interfering_with_swap.zig

```
const expect = @import("std").testing.expect;
test "attempt to swap array elements with array initializer" {
    var arr: [2]u32 = .{ 1, 2 };
```

```
    arr = .{ arr[1], arr[0] };
    // The previous line is equivalent to the following two lines:
    //   arr[0] = arr[1];
    //   arr[1] = arr[0];
    // So this fails!
    try expect(arr[0] == 2); // succeeds
    try expect(arr[1] == 1); // fails
}
```

Shell

```
$ zig test result_location_interfering_with_swap.zig
1/1 result_location_interfering_with_swap.test.attempt to swap array elements with array initia
/home/andy/src/zig/lib/std/testing.zig:580:14: 0x104896f in expect (test)
    if (!ok) return error.TestUnexpectedResult;
             ^
/home/andy/src/zig/doc/langref/result_location_interfering_with_swap.zig:10:5: 0x1048a55 in tes
    try expect(arr[1] == 1); // fails
    ^
0 passed; 0 skipped; 1 failed.
error: the following test command failed with exit code 1:
/home/andy/src/zig/.zig-cache/o/1ab5fe7528c3fb20f26dc032b4e272d9/test --seed=0x35f0babc
```

The following table details how some common expressions propagate result locations, where `x` and `y` are arbitrary sub-expressions. Note that some expressions cannot provide meaningful result locations to sub-expressions, even if they themselves have a result location.

Expression	Result Location	Sub-expression Result Locations
`const val: T = x`	-	`x` has result location `&val`
`var val: T = x`	-	`x` has result location `&val`
`val = x`	-	`x` has result location `&val`
`@as(T, x)`	`ptr`	`x` has no result location
`&x`	`ptr`	`x` has no result location
`f(x)`	`ptr`	`x` has no result location
`.{x}`	`ptr`	`x` has result location `&ptr[0]`
`.{ .a = x }`	`ptr`	`x` has result location `&ptr.a`
`T{x}`	`ptr`	`x` has no result location (typed initializers do not propagate result locations)

Expression	Result Location	Sub-expression Result Locations
`T{ .a = x }`	`ptr`	`x` has no result location (typed initializers do not propagate result locations)
`@Type(x)`	`ptr`	`x` has no result location
`@typeInfo(x)`	`ptr`	`x` has no result location
`x << y`	`ptr`	`x` and `y` do not have result locations

usingnamespace

`usingnamespace` is a declaration that mixes all the public declarations of the operand, which must be a struct, union, enum, or opaque, into the namespace:

test_usingnamespace.zig

```
test "using std namespace" {
    const S = struct {
        usingnamespace @import("std");
    };
    try S.testing.expect(true);
}
```

Shell

```
$ zig test test_usingnamespace.zig
1/1 test_usingnamespace.test.using std namespace...OK
All 1 tests passed.
```

`usingnamespace` has an important use case when organizing the public API of a file or package. For example, one might have **`c.zig`** with all of the C imports:

c.zig

```
pub usingnamespace @cImport({
    @cInclude("epoxy/gl.h");
    @cInclude("GLFW/glfw3.h");
    @cDefine("STBI_ONLY_PNG", "");
    @cDefine("STBI_NO_STDIO", "");
    @cInclude("stb_image.h");
});
```

The above example demonstrates using `pub` to qualify the `usingnamespace` additionally makes the imported declarations `pub`. This can be used to forward declarations, giving precise control over what declarations a given file exposes.

comptime

Zig places importance on the concept of whether an expression is known at compile-time. There are a few different places this concept is used, and these building blocks are used to keep the language small, readable, and powerful.

Introducing the Compile-Time Concept

Compile-Time Parameters

Compile-time parameters is how Zig implements generics. It is compile-time duck typing.

compile-time_duck_typing.zig

```
fn max(comptime T: type, a: T, b: T) T {
    return if (a > b) a else b;
}
fn gimmeTheBiggerFloat(a: f32, b: f32) f32 {
    return max(f32, a, b);
}
fn gimmeTheBiggerInteger(a: u64, b: u64) u64 {
    return max(u64, a, b);
}
```

In Zig, types are first-class citizens. They can be assigned to variables, passed as parameters to functions, and returned from functions. However, they can only be used in expressions which are known at *compile-time*, which is why the parameter `T` in the above snippet must be marked with **`comptime`**.

A **`comptime`** parameter means that:

- At the callsite, the value must be known at compile-time, or it is a compile error.
- In the function definition, the value is known at compile-time.

For example, if we were to introduce another function to the above snippet:

test_unresolved_comptime_value.zig

```
fn max(comptime T: type, a: T, b: T) T {
    return if (a > b) a else b;
}
test "try to pass a runtime type" {
    foo(false);
}
fn foo(condition: bool) void {
    const result = max(if (condition) f32 else u64, 1234, 5678);
    _ = result;
}
```

Shell

```
$ zig test test_unresolved_comptime_value.zig
doc/langref/test_unresolved_comptime_value.zig:8:28: error: unable to resolve comptime value
    const result = max(if (condition) f32 else u64, 1234, 5678);
                           ^~~~~~~~~
doc/langref/test_unresolved_comptime_value.zig:8:24: note: argument to comptime parameter must
    const result = max(if (condition) f32 else u64, 1234, 5678);
                       ^~~~~~~~~~~~~~~~~~~~~~~~~~~
doc/langref/test_unresolved_comptime_value.zig:1:8: note: parameter declared comptime here
fn max(comptime T: type, a: T, b: T) T {
       ^~~~~~~~
```

This is an error because the programmer attempted to pass a value only known at run-time to a function which expects a value known at compile-time.

Another way to get an error is if we pass a type that violates the type checker when the function is analyzed. This is what it means to have *compile-time duck typing*.

For example:

test_comptime_mismatched_type.zig

```
fn max(comptime T: type, a: T, b: T) T {
    return if (a > b) a else b;
}
test "try to compare bools" {
    _ = max(bool, true, false);
}
```

Shell

```
$ zig test test_comptime_mismatched_type.zig
doc/langref/test_comptime_mismatched_type.zig:2:18: error: operator > not allowed for type 'boo
    return if (a > b) a else b;
               ~~^~~
```

On the flip side, inside the function definition with the `comptime` parameter, the value is known at compile-time. This means that we actually could make this work for the bool type if we wanted to:

test_comptime_max_with_bool.zig

```
fn max(comptime T: type, a: T, b: T) T {
    if (T == bool) {
        return a or b;
    } else if (a > b) {
        return a;
    } else {
        return b;
    }
}
test "try to compare bools" {
```

```
    try @import("std").testing.expect(max(bool, false, true) == true);
}
```

Shell

```
$ zig test test_comptime_max_with_bool.zig
1/1 test_comptime_max_with_bool.test.try to compare bools...OK
All 1 tests passed.
```

This works because Zig implicitly inlines `if` expressions when the condition is known at compile-time, and the compiler guarantees that it will skip analysis of the branch not taken.

This means that the actual function generated for `max` in this situation looks like this:

compiler_generated_function.zig

```
fn max(a: bool, b: bool) bool {
    {
        return a or b;
    }
}
```

All the code that dealt with compile-time known values is eliminated and we are left with only the necessary run-time code to accomplish the task.

This works the same way for `switch` expressions - they are implicitly inlined when the target expression is compile-time known.

Compile-Time Variables

In Zig, the programmer can label variables as `comptime`. This guarantees to the compiler that every load and store of the variable is performed at compile-time. Any violation of this results in a compile error.

This combined with the fact that we can `inline` loops allows us to write a function which is partially evaluated at compile-time and partially at run-time.

For example:

test_comptime_evaluation.zig

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

const CmdFn = struct {
    name: []const u8,
    func: fn (i32) i32,
};

const cmd_fns = [_]CmdFn{
    CmdFn{ .name = "one", .func = one },
    CmdFn{ .name = "two", .func = two },
```

```
    CmdFn{ .name = "three", .func = three },
};
fn one(value: i32) i32 {
    return value + 1;
}
fn two(value: i32) i32 {
    return value + 2;
}
fn three(value: i32) i32 {
    return value + 3;
}

fn performFn(comptime prefix_char: u8, start_value: i32) i32 {
    var result: i32 = start_value;
    comptime var i = 0;
    inline while (i < cmd_fns.len) : (i += 1) {
        if (cmd_fns[i].name[0] == prefix_char) {
            result = cmd_fns[i].func(result);
        }
    }
    return result;
}

test "perform fn" {
    try expect(performFn('t', 1) == 6);
    try expect(performFn('o', 0) == 1);
    try expect(performFn('w', 99) == 99);
}
```

Shell

```
$ zig test test_comptime_evaluation.zig
1/1 test_comptime_evaluation.test.perform fn...OK
All 1 tests passed.
```

This example is a bit contrived, because the compile-time evaluation component is unnecessary; this code would work fine if it was all done at run-time. But it does end up generating different code. In this example, the function `performFn` is generated three different times, for the different values of `prefix_char` provided:

performFn_1

```
// From the line:
// expect(performFn('t', 1) == 6);
fn performFn(start_value: i32) i32 {
    var result: i32 = start_value;
    result = two(result);
    result = three(result);
    return result;
}
```

performFn_2

```
// From the line:
// expect(performFn('o', 0) == 1);
fn performFn(start_value: i32) i32 {
```

```
    var result: i32 = start_value;
    result = one(result);
    return result;
}
```

performFn_3

```
// From the line:
// expect(performFn('w', 99) == 99);
fn performFn(start_value: i32) i32 {
    var result: i32 = start_value;
    _ = &result;
    return result;
}
```

Note that this happens even in a debug build. This is not a way to write more optimized code, but it is a way to make sure that what *should* happen at compile-time, *does* happen at compile-time. This catches more errors and allows expressiveness that in other languages requires using macros, generated code, or a preprocessor to accomplish.

Compile-Time Expressions

In Zig, it matters whether a given expression is known at compile-time or run-time. A programmer can use a `comptime` expression to guarantee that the expression will be evaluated at compile-time. If this cannot be accomplished, the compiler will emit an error. For example:

test_comptime_call_extern_function.zig

```
extern fn exit() noreturn;

test "foo" {
    comptime {
        exit();
    }
}
```

Shell

```
$ zig test test_comptime_call_extern_function.zig
doc/langref/test_comptime_call_extern_function.zig:5:13: error: comptime call of extern functio
        exit();
        ~~~~^~
doc/langref/test_comptime_call_extern_function.zig:4:5: note: 'comptime' keyword forces comptim
    comptime {
    ^~~~~~~~
```

It doesn't make sense that a program could call `exit()` (or any other external function) at compile-time, so this is a compile error. However, a `comptime` expression does much more than sometimes cause a compile error.

Within a `comptime` expression:

- All variables are `comptime` variables.
- All `if`, `while`, `for`, and `switch` expressions are evaluated at compile-time, or emit a compile error if this is not possible.
- All `return` and `try` expressions are invalid (unless the function itself is called at compile-time).
- All code with runtime side effects or depending on runtime values emits a compile error.
- All function calls cause the compiler to interpret the function at compile-time, emitting a compile error if the function tries to do something that has global runtime side effects.

This means that a programmer can create a function which is called both at compile-time and run-time, with no modification to the function required.

Let's look at an example:

test_fibonacci_recursion.zig

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

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

test "fibonacci" {
    // test fibonacci at run-time
    try expect(fibonacci(7) == 13);

    // test fibonacci at compile-time
    try comptime expect(fibonacci(7) == 13);
}
```

Shell

```
$ zig test test_fibonacci_recursion.zig
1/1 test_fibonacci_recursion.test.fibonacci...OK
All 1 tests passed.
```

Imagine if we had forgotten the base case of the recursive function and tried to run the tests:

test_fibonacci_comptime_overflow.zig

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

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

test "fibonacci" {
    try comptime expect(fibonacci(7) == 13);
}
```

Shell

```
$ zig test test_fibonacci_comptime_overflow.zig
doc/langref/test_fibonacci_comptime_overflow.zig:5:28: error: overflow of integer type 'u32' wi
    return fibonacci(index - 1) + fibonacci(index - 2);
                     ~~~~~~^~~
doc/langref/test_fibonacci_comptime_overflow.zig:5:21: note: called from here (7 times)
    return fibonacci(index - 1) + fibonacci(index - 2);
           ~~~~~~~~~^~~~~~~~~~~
doc/langref/test_fibonacci_comptime_overflow.zig:9:34: note: called from here
    try comptime expect(fibonacci(7) == 13);
                        ~~~~~~~~~^~~
```

The compiler produces an error which is a stack trace from trying to evaluate the function at compile-time.

Luckily, we used an unsigned integer, and so when we tried to subtract 1 from 0, it triggered Illegal Behavior, which is always a compile error if the compiler knows it happened. But what would have happened if we used a signed integer?

fibonacci_comptime_infinite_recursion.zig

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

fn fibonacci(index: i32) i32 {
    //if (index < 2) return index;
    return fibonacci(index - 1) + fibonacci(index - 2);
}

test "fibonacci" {
    try comptime assert(fibonacci(7) == 13);
}
```

The compiler is supposed to notice that evaluating this function at compile-time took more than 1000 branches, and thus emits an error and gives up. If the programmer wants to increase the budget for compile-time computation, they can use a built-in function called @setEvalBranchQuota to change the default number 1000 to something else.

However, there is a design flaw in the compiler causing it to stack overflow instead of having the proper behavior here. I'm terribly sorry about that. I hope to get this resolved before the next release.

What if we fix the base case, but put the wrong value in the `expect` line?

test_fibonacci_comptime_unreachable.zig

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

fn fibonacci(index: i32) i32 {
    if (index < 2) return index;
    return fibonacci(index - 1) + fibonacci(index - 2);
```

```
}

test "fibonacci" {
    try comptime assert(fibonacci(7) == 99999);
}
```

Shell

```
$ zig test test_fibonacci_comptime_unreachable.zig
lib/std/debug.zig:522:14: error: reached unreachable code
    if (!ok) unreachable; // assertion failure
             ^~~~~~~~~~~
doc/langref/test_fibonacci_comptime_unreachable.zig:9:24: note: called from here
    try comptime assert(fibonacci(7) == 99999);
                 ~~~~~~^~~~~~~~~~~~~~~~~~~~~~
```

At container level (outside of any function), all expressions are implicitly `comptime` expressions. This means that we can use functions to initialize complex static data. For example:

test_container-level_comptime_expressions.zig

```
const first_25_primes = firstNPrimes(25);
const sum_of_first_25_primes = sum(&first_25_primes);

fn firstNPrimes(comptime n: usize) [n]i32 {
    var prime_list: [n]i32 = undefined;
    var next_index: usize = 0;
    var test_number: i32 = 2;
    while (next_index < prime_list.len) : (test_number += 1) {
        var test_prime_index: usize = 0;
        var is_prime = true;
        while (test_prime_index < next_index) : (test_prime_index += 1) {
            if (test_number % prime_list[test_prime_index] == 0) {
                is_prime = false;
                break;
            }
        }
        if (is_prime) {
            prime_list[next_index] = test_number;
            next_index += 1;
        }
    }
    return prime_list;
}

fn sum(numbers: []const i32) i32 {
    var result: i32 = 0;
    for (numbers) |x| {
        result += x;
    }
    return result;
}

test "variable values" {
    try @import("std").testing.expect(sum_of_first_25_primes == 1060);
}
```

Shell

```
$ zig test test_container-level_comptime_expressions.zig
1/1 test_container-level_comptime_expressions.test.variable values...OK
All 1 tests passed.
```

When we compile this program, Zig generates the constants with the answer pre-computed. Here are the lines from the generated LLVM IR:

```
@0 = internal unnamed_addr constant [25 x i32] [i32 2, i32 3, i32 5, i32 7, i32 11, i32 13, i32
@1 = internal unnamed_addr constant i32 1060
```

Note that we did not have to do anything special with the syntax of these functions. For example, we could call the `sum` function as is with a slice of numbers whose length and values were only known at run-time.

Generic Data Structures

Zig uses comptime capabilities to implement generic data structures without introducing any special-case syntax.

Here is an example of a generic `List` data structure.

generic_data_structure.zig

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

// The generic List data structure can be instantiated by passing in a type:
var buffer: [10]i32 = undefined;
var list = List(i32){
    .items = &buffer,
    .len = 0,
};
```

That's it. It's a function that returns an anonymous `struct`. For the purposes of error messages and debugging, Zig infers the name `"List(i32)"` from the function name and parameters invoked when creating the anonymous struct.

To explicitly give a type a name, we assign it to a constant.

anonymous_struct_name.zig

```
const Node = struct {
    next: ?*Node,
    name: []const u8,
};

var node_a = Node{
    .next = null,
    .name = "Node A",
};

var node_b = Node{
    .next = &node_a,
    .name = "Node B",
};
```

In this example, the `Node` struct refers to itself. This works because all top level declarations are order-independent. As long as the compiler can determine the size of the struct, it is free to refer to itself. In this case, `Node` refers to itself as a pointer, which has a well-defined size at compile time, so it works fine.

Case Study: print in Zig

Putting all of this together, let's see how `print` works in Zig.

print.zig

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

const a_number: i32 = 1234;
const a_string = "foobar";

pub fn main() void {
    print("here is a string: '{s}' here is a number: {}\n", .{ a_string, a_number });
}
```

Shell

```
$ zig build-exe print.zig
$ ./print
here is a string: 'foobar' here is a number: 1234
```

Let's crack open the implementation of this and see how it works:

poc_print_fn.zig

```
const Writer = struct {
    /// Calls print and then flushes the buffer.
    pub fn print(self: *Writer, comptime format: []const u8, args: anytype) anyerror!void {
        const State = enum {
            start,
            open_brace,
            close_brace,
        };

        comptime var start_index: usize = 0;
        comptime var state = State.start;
        comptime var next_arg: usize = 0;

        inline for (format, 0..) |c, i| {
            switch (state) {
                State.start => switch (c) {
                    '{' => {
                        if (start_index < i) try self.write(format[start_index..i]);
                        state = State.open_brace;
                    },
                    '}' => {
                        if (start_index < i) try self.write(format[start_index..i]);
                        state = State.close_brace;
                    },
                    else => {},
                },
                State.open_brace => switch (c) {
                    '{' => {
                        state = State.start;
                        start_index = i;
                    },
                    '}' => {
                        try self.printValue(args[next_arg]);
                        next_arg += 1;
```

```
                        state = State.start;
                        start_index = i + 1;
                    },
                    's' => {
                        continue;
                    },
                    else => @compileError("Unknown format character: " ++ [1]u8{c}),
                },
                State.close_brace => switch (c) {
                    '}' => {
                        state = State.start;
                        start_index = i;
                    },
                    else => @compileError("Single '}' encountered in format string"),
                },
            }
        }
        comptime {
            if (args.len != next_arg) {
                @compileError("Unused arguments");
            }
            if (state != State.start) {
                @compileError("Incomplete format string: " ++ format);
            }
        }
        if (start_index < format.len) {
            try self.write(format[start_index..format.len]);
        }
        try self.flush();
    }

    fn write(self: *Writer, value: []const u8) !void {
        _ = self;
        _ = value;
    }
    pub fn printValue(self: *Writer, value: anytype) !void {
        _ = self;
        _ = value;
    }
    fn flush(self: *Writer) !void {
        _ = self;
    }
};
```

This is a proof of concept implementation; the actual function in the standard library has more formatting capabilities.

Note that this is not hard-coded into the Zig compiler; this is userland code in the standard library.

When this function is analyzed from our example code above, Zig partially evaluates the function and emits a function that actually looks like this:

Emitted print Function

```
pub fn print(self: *Writer, arg0: []const u8, arg1: i32) !void {
    try self.write("here is a string: '");
    try self.printValue(arg0);
    try self.write("' here is a number: ");
    try self.printValue(arg1);
    try self.write("\n");
    try self.flush();
}
```

`printValue` is a function that takes a parameter of any type, and does different things depending on the type:

poc_printValue_fn.zig

```
const Writer = struct {
    pub fn printValue(self: *Writer, value: anytype) !void {
        switch (@typeInfo(@TypeOf(value))) {
            .int => {
                return self.writeInt(value);
            },
            .float => {
                return self.writeFloat(value);
            },
            .pointer => {
                return self.write(value);
            },
            else => {
                @compileError("Unable to print type '" ++ @typeName(@TypeOf(value)) ++ "'");
            },
        }
    }

    fn write(self: *Writer, value: []const u8) !void {
        _ = self;
        _ = value;
    }
    fn writeInt(self: *Writer, value: anytype) !void {
        _ = self;
        _ = value;
    }
    fn writeFloat(self: *Writer, value: anytype) !void {
        _ = self;
        _ = value;
    }
};
```

And now, what happens if we give too many arguments to `print`?

test_print_too_many_args.zig

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

const a_number: i32 = 1234;
const a_string = "foobar";

test "print too many arguments" {
```

```
    print("here is a string: '{s}' here is a number: {}\n", .{
        a_string,
        a_number,
        a_number,
    });
}
```

Shell

```
$ zig test test_print_too_many_args.zig
lib/std/fmt.zig:211:18: error: unused argument in 'here is a string: '{s}' here is a number: {}
                       '
            1 => @compileError("unused argument in '" ++ fmt ++ "'"),
                 ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

Zig gives programmers the tools needed to protect themselves against their own mistakes.

Zig doesn't care whether the format argument is a string literal, only that it is a compile-time known value that can be coerced to a `[]const u8`:

print_comptime-known_format.zig

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

const a_number: i32 = 1234;
const a_string = "foobar";
const fmt = "here is a string: '{s}' here is a number: {}\n";

pub fn main() void {
    print(fmt, .{ a_string, a_number });
}
```

Shell

```
$ zig build-exe print_comptime-known_format.zig
$ ./print_comptime-known_format
here is a string: 'foobar' here is a number: 1234
```

This works fine.

Zig does not special case string formatting in the compiler and instead exposes enough power to accomplish this task in userland. It does so without introducing another language on top of Zig, such as a macro language or a preprocessor language. It's Zig all the way down.

See also:

- inline while
- inline for

Assembly

For some use cases, it may be necessary to directly control the machine code generated by Zig programs, rather than relying on Zig's code generation. For these cases, one can use inline assembly. Here is an example of implementing Hello, World on x86_64 Linux using inline assembly:

inline_assembly.zig

```
pub fn main() noreturn {
    const msg = "hello world\n";
    _ = syscall3(SYS_write, STDOUT_FILENO, @intFromPtr(msg), msg.len);
    _ = syscall1(SYS_exit, 0);
    unreachable;
}

pub const SYS_write = 1;
pub const SYS_exit = 60;

pub const STDOUT_FILENO = 1;

pub fn syscall1(number: usize, arg1: usize) usize {
    return asm volatile ("syscall"
        : [ret] "={rax}" (-> usize),
        : [number] "{rax}" (number),
          [arg1] "{rdi}" (arg1),
        : "rcx", "r11"
    );
}

pub fn syscall3(number: usize, arg1: usize, arg2: usize, arg3: usize) usize {
    return asm volatile ("syscall"
        : [ret] "={rax}" (-> usize),
        : [number] "{rax}" (number),
          [arg1] "{rdi}" (arg1),
          [arg2] "{rsi}" (arg2),
          [arg3] "{rdx}" (arg3),
        : "rcx", "r11"
    );
}
```

Shell

```
$ zig build-exe inline_assembly.zig -target x86_64-linux
$ ./inline_assembly
hello world
```

Dissecting the syntax:

Assembly Syntax Explained.zig

```
pub fn syscall1(number: usize, arg1: usize) usize {
    // Inline assembly is an expression which returns a value.
    // the `asm` keyword begins the expression.
    return asm
    // `volatile` is an optional modifier that tells Zig this
    // inline assembly expression has side-effects. Without
    // `volatile`, Zig is allowed to delete the inline assembly
```

```
    // code if the result is unused.
    volatile (
    // Next is a comptime string which is the assembly code.
    // Inside this string one may use `%[ret]`, `%[number]`,
    // or `%[arg1]` where a register is expected, to specify
    // the register that Zig uses for the argument or return value,
    // if the register constraint strings are used. However in
    // the below code, this is not used. A literal `%` can be
    // obtained by escaping it with a double percent: `%%`.
    // Often multiline string syntax comes in handy here.
        \\syscall
        // Next is the output. It is possible in the future Zig will
        // support multiple outputs, depending on how
        // https://github.com/ziglang/zig/issues/215 is resolved.
        // It is allowed for there to be no outputs, in which case
        // this colon would be directly followed by the colon for the inputs.
        :
        // This specifies the name to be used in `%[ret]` syntax in
        // the above assembly string. This example does not use it,
        // but the syntax is mandatory.
          [ret]
          // Next is the output constraint string. This feature is still
          // considered unstable in Zig, and so LLVM/GCC documentation
          // must be used to understand the semantics.
          // http://releases.llvm.org/10.0.0/docs/LangRef.html#inline-asm-constraint-string
          // https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html
          // In this example, the constraint string means "the result value of
          // this inline assembly instruction is whatever is in $rax".
          "={rax}"
          // Next is either a value binding, or `->` and then a type. The
          // type is the result type of the inline assembly expression.
          // If it is a value binding, then `%[ret]` syntax would be used
          // to refer to the register bound to the value.
          (-> usize),
          // Next is the list of inputs.
          // The constraint for these inputs means, "when the assembly code is
          // executed, $rax shall have the value of `number` and $rdi shall have
          // the value of `arg1`". Any number of input parameters is allowed,
          // including none.
        : [number] "{rax}" (number),
          [arg1] "{rdi}" (arg1),
          // Next is the list of clobbers. These declare a set of registers whose
          // values will not be preserved by the execution of this assembly code.
          // These do not include output or input registers. The special clobber
          // value of "memory" means that the assembly writes to arbitrary undeclared
          // memory locations - not only the memory pointed to by a declared indirect
          // output. In this example we list $rcx and $r11 because it is known the
          // kernel syscall does not preserve these registers.
        : "rcx", "r11"
    );
}
```

For x86 and x86_64 targets, the syntax is AT&T syntax, rather than the more popular Intel syntax. This is due to technical constraints; assembly parsing is provided by LLVM and its support for Intel syntax is buggy and not well tested.

Some day Zig may have its own assembler. This would allow it to integrate more seamlessly into the language, as well as be compatible with the popular NASM syntax. This documentation section will be updated before 1.0.0 is released, with a conclusive statement about the status of AT&T vs Intel/NASM syntax.

Output Constraints

Output constraints are still considered to be unstable in Zig, and so LLVM documentation and GCC documentation must be used to understand the semantics.

Note that some breaking changes to output constraints are planned with issue #215.

Input Constraints

Input constraints are still considered to be unstable in Zig, and so LLVM documentation and GCC documentation must be used to understand the semantics.

Note that some breaking changes to input constraints are planned with issue #215.

Clobbers

Clobbers are the set of registers whose values will not be preserved by the execution of the assembly code. These do not include output or input registers. The special clobber value of `"memory"` means that the assembly causes writes to arbitrary undeclared memory locations - not only the memory pointed to by a declared indirect output.

Failure to declare the full set of clobbers for a given inline assembly expression is unchecked Illegal Behavior.

Global Assembly

When an assembly expression occurs in a container level comptime block, this is **global assembly**.

This kind of assembly has different rules than inline assembly. First, `volatile` is not valid because all global assembly is unconditionally included. Second, there are no inputs, outputs, or clobbers. All global assembly is concatenated verbatim into one long string and assembled together. There are no template substitution rules regarding `%` as there are in inline assembly expressions.

test_global_assembly.zig

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

```
comptime {
    asm (
        \\.global my_func;
        \\.type my_func, @function;
        \\my_func:
        \\  lea (%rdi,%rsi,1),%eax
        \\  retq
    );
}

extern fn my_func(a: i32, b: i32) i32;

test "global assembly" {
    try expect(my_func(12, 34) == 46);
}
```

Shell

```
$ zig test test_global_assembly.zig -target x86_64-linux
1/1 test_global_assembly.test.global assembly...OK
All 1 tests passed.
```

Atomics

TODO: @atomic rmw

TODO: builtin atomic memory ordering enum

See also:

- @atomicLoad
- @atomicStore
- @atomicRmw
- @cmpxchgWeak
- @cmpxchgStrong

Async Functions

Async functions regressed with the release of 0.11.0. Their future in the Zig language is unclear due to multiple unsolved problems:

- LLVM's lack of ability to optimize them.
- Third-party debuggers' lack of ability to debug them.
- The cancellation problem.
- Async function pointers preventing the stack size from being known.

These problems are surmountable, but it will take time. The Zig team is currently focused on other priorities.

Builtin Functions

Builtin functions are provided by the compiler and are prefixed with `@`. The **`comptime`** keyword on a parameter means that the parameter must be known at compile time.

@addrSpaceCast

```
@addrSpaceCast(ptr: anytype) anytype
```

Converts a pointer from one address space to another. The new address space is inferred based on the result type. Depending on the current target and address spaces, this cast may be a no-op, a complex operation, or illegal. If the cast is legal, then the resulting pointer points to the same memory location as the pointer operand. It is always valid to cast a pointer between the same address spaces.

@addWithOverflow

```
@addWithOverflow(a: anytype, b: anytype) struct { @TypeOf(a, b), u1 }
```

Performs `a + b` and returns a tuple with the result and a possible overflow bit.

@alignCast

```
@alignCast(ptr: anytype) anytype
```

`ptr` can be `*T`, `?*T`, or `[]T`. Changes the alignment of a pointer. The alignment to use is inferred based on the result type.

A [pointer alignment safety check] is added to the generated code to make sure the pointer is aligned as promised.

@alignOf

```
@alignOf(comptime T: type) comptime_int
```

This function returns the number of bytes that this type should be aligned to for the current target to match the C ABI. When the child type of a pointer has this alignment, the alignment can be omitted from the type.

```
const assert = @import("std").debug.assert;
comptime {
    assert(*u32 == *align(@alignOf(u32)) u32);
}
```

The result is a target-specific compile time constant. It is guaranteed to be less than or equal to @sizeOf(T).

See also:

- Alignment

@as

```
@as(comptime T: type, expression) T
```

Performs Type Coercion. This cast is allowed when the conversion is unambiguous and safe, and is the preferred way to convert between types, whenever possible.

@atomicLoad

```
@atomicLoad(comptime T: type, ptr: *const T, comptime ordering: AtomicOrder) T
```

This builtin function atomically dereferences a pointer to a `T` and returns the value.

`T` must be a pointer, a `bool`, a float, an integer or an enum.

`AtomicOrder` can be found with `@import("std").builtin.AtomicOrder`.

See also:

- @atomicStore
- @atomicRmw
- @cmpxchgWeak
- @cmpxchgStrong

@atomicRmw

```
@atomicRmw(comptime T: type, ptr: *T, comptime op: AtomicRmwOp, operand: T, comptime ordering:
```

This builtin function dereferences a pointer to a `T` and atomically modifies the value and returns the previous value.

`T` must be a pointer, a `bool`, a float, an integer or an enum.

`AtomicOrder` can be found with `@import("std").builtin.AtomicOrder`.

`AtomicRmwOp` can be found with `@import("std").builtin.AtomicRmwOp`.

See also:

- @atomicStore
- @atomicLoad
- @cmpxchgWeak
- @cmpxchgStrong

@atomicStore

```
@atomicStore(comptime T: type, ptr: *T, value: T, comptime ordering: AtomicOrder) void
```

This builtin function dereferences a pointer to a `T` and atomically stores the given value.

`T` must be a pointer, a `bool`, a float, an integer or an enum.

`AtomicOrder` can be found with `@import("std").builtin.AtomicOrder`.

See also:

- @atomicLoad
- @atomicRmw
- @cmpxchgWeak
- @cmpxchgStrong

@bitCast

```
@bitCast(value: anytype) anytype
```

Converts a value of one type to another type. The return type is the inferred result type.

Asserts that `@sizeOf(@TypeOf(value)) == @sizeOf(DestType)`.

Asserts that `@typeInfo(DestType) != .pointer`. Use `@ptrCast` or `@ptrFromInt` if you need this.

Can be used for these things for example:

- Convert `f32` to `u32` bits
- Convert `i32` to `u32` preserving twos complement

Works at compile-time if `value` is known at compile time. It's a compile error to bitcast a value of undefined layout; this means that, besides the restriction from types which possess dedicated casting builtins (enums, pointers, error sets), bare structs, error unions, slices, optionals, and any other type without a well-defined memory layout, also cannot be used in this operation.

@bitOffsetOf

```
@bitOffsetOf(comptime T: type, comptime field_name: []const u8) comptime_int
```

Returns the bit offset of a field relative to its containing struct.

For non packed structs, this will always be divisible by `8`. For packed structs, non-byte-aligned fields will share a byte offset, but they will have different bit offsets.

See also:

- @offsetOf

@bitSizeOf

```
@bitSizeOf(comptime T: type) comptime_int
```

This function returns the number of bits it takes to store `T` in memory if the type were a field in a packed struct/union. The result is a target-specific compile time constant.

This function measures the size at runtime. For types that are disallowed at runtime, such as `comptime_int` and `type`, the result is `0`.

See also:

- @sizeOf
- @typeInfo

@branchHint

```
@branchHint(hint: BranchHint) void
```

Hints to the optimizer how likely a given branch of control flow is to be reached.

`BranchHint` can be found with `@import("std").builtin.BranchHint`.

This function is only valid as the first statement in a control flow branch, or the first statement in a function.

@breakpoint

```
@breakpoint() void
```

This function inserts a platform-specific debug trap instruction which causes debuggers to break there. Unlike for `@trap()`, execution may continue after this point if the program is resumed.

This function is only valid within function scope.

See also:

- @trap

@mulAdd

```
@mulAdd(comptime T: type, a: T, b: T, c: T) T
```

Fused multiply-add, similar to `(a * b) + c`, except only rounds once, and is thus more accurate.

Supports Floats and Vectors of floats.

@byteSwap

```
@byteSwap(operand: anytype) T
```

`@TypeOf(operand)` must be an integer type or an integer vector type with bit count evenly divisible by 8.

`operand` may be an integer or vector.

Swaps the byte order of the integer. This converts a big endian integer to a little endian integer, and converts a little endian integer to a big endian integer.

Note that for the purposes of memory layout with respect to endianness, the integer type should be related to the number of bytes reported by @sizeOf bytes. This is demonstrated with `u24`. `@sizeOf(u24) == 4`, which means that a `u24` stored in memory takes 4 bytes, and those 4 bytes are what are swapped on a little vs big endian system. On the other hand, if `T` is specified to be `u24`, then only 3 bytes are reversed.

@bitReverse

```
@bitReverse(integer: anytype) T
```

`@TypeOf(anytype)` accepts any integer type or integer vector type.

Reverses the bitpattern of an integer value, including the sign bit if applicable.

For example 0b10110110 (`u8 = 182`, `i8 = -74`) becomes 0b01101101 (`u8 = 109`, `i8 = 109`).

@offsetOf

```
@offsetOf(comptime T: type, comptime field_name: []const u8) comptime_int
```

Returns the byte offset of a field relative to its containing struct.

See also:

- @bitOffsetOf

@call

```
@call(modifier: std.builtin.CallModifier, function: anytype, args: anytype) anytype
```

Calls a function, in the same way that invoking an expression with parentheses does:

test_call_builtin.zig

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

test "noinline function call" {
    try expect(@call(.auto, add, .{ 3, 9 }) == 12);
}

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

Shell

```
$ zig test test_call_builtin.zig
1/1 test_call_builtin.test.noinline function call...OK
All 1 tests passed.
```

`@call` allows more flexibility than normal function call syntax does. The `CallModifier` enum is reproduced here:

builtin.CallModifier struct.zig

```zig
pub const CallModifier = enum {
    /// Equivalent to function call syntax.
    auto,

    /// Equivalent to async keyword used with function call syntax.
    async_kw,
```

```
    /// Prevents tail call optimization. This guarantees that the return
    /// address will point to the callsite, as opposed to the callsite's
    /// callsite. If the call is otherwise required to be tail-called
    /// or inlined, a compile error is emitted instead.
    never_tail,

    /// Guarantees that the call will not be inlined. If the call is
    /// otherwise required to be inlined, a compile error is emitted instead.
    never_inline,

    /// Asserts that the function call will not suspend. This allows a
    /// non-async function to call an async function.
    no_async,

    /// Guarantees that the call will be generated with tail call optimization.
    /// If this is not possible, a compile error is emitted instead.
    always_tail,

    /// Guarantees that the call will be inlined at the callsite.
    /// If this is not possible, a compile error is emitted instead.
    always_inline,

    /// Evaluates the call at compile-time. If the call cannot be completed at
    /// compile-time, a compile error is emitted instead.
    compile_time,
};
```

@cDefine

```
@cDefine(comptime name: []const u8, value) void
```

This function can only occur inside `@cImport`.

This appends `#define $name $value` to the `@cImport` temporary buffer.

To define without a value, like this:

```
#define _GNU_SOURCE
```

Use the void value, like this:

```
@cDefine("_GNU_SOURCE", {})
```

See also:

- Import from C Header File
- @cInclude
- @cImport
- @cUndef

- void

@cImport

```
@cImport(expression) type
```

This function parses C code and imports the functions, types, variables, and compatible macro definitions into a new empty struct type, and then returns that type.

`expression` is interpreted at compile time. The builtin functions `@cInclude`, `@cDefine`, and `@cUndef` work within this expression, appending to a temporary buffer which is then parsed as C code.

Usually you should only have one `@cImport` in your entire application, because it saves the compiler from invoking clang multiple times, and prevents inline functions from being duplicated.

Reasons for having multiple `@cImport` expressions would be:

- To avoid a symbol collision, for example if foo.h and bar.h both `#define CONNECTION_COUNT`
- To analyze the C code with different preprocessor defines

See also:

- Import from C Header File
- @cInclude
- @cDefine
- @cUndef

@cInclude

```
@cInclude(comptime path: []const u8) void
```

This function can only occur inside `@cImport`.

This appends `#include <$path>\n` to the `c_import` temporary buffer.

See also:

- Import from C Header File
- @cImport
- @cDefine
- @cUndef

@clz

```
@clz(operand: anytype) anytype
```

`@TypeOf(operand)` must be an integer type or an integer vector type.

`operand` may be an integer or vector.

Counts the number of most-significant (leading in a big-endian sense) zeroes in an integer - "count leading zeroes".

The return type is an unsigned integer or vector of unsigned integers with the minimum number of bits that can represent the bit count of the integer type.

If `operand` is zero, `@clz` returns the bit width of integer type `T`.

See also:

- @ctz
- @popCount

@cmpxchgStrong

```
@cmpxchgStrong(comptime T: type, ptr: *T, expected_value: T, new_value: T, success_order: Atomi
```

This function performs a strong atomic compare-and-exchange operation, returning `null` if the current value is the given expected value. It's the equivalent of this code, except atomic:

not_atomic_cmpxchgStrong.zig

```
fn cmpxchgStrongButNotAtomic(comptime T: type, ptr: *T, expected_value: T, new_value: T) ?T {
    const old_value = ptr.*;
    if (old_value == expected_value) {
        ptr.* = new_value;
        return null;
    } else {
        return old_value;
    }
}
```

If you are using cmpxchg in a retry loop, @cmpxchgWeak is the better choice, because it can be implemented more efficiently in machine instructions.

`T` must be a pointer, a `bool`, a float, an integer or an enum.

`@typeInfo(@TypeOf(ptr)).pointer.alignment` must be `>= @sizeOf(T).`

`AtomicOrder` can be found with `@import("std").builtin.AtomicOrder`.

See also:

- @atomicStore
- @atomicLoad
- @atomicRmw
- @cmpxchgWeak

@cmpxchgWeak

```
@cmpxchgWeak(comptime T: type, ptr: *T, expected_value: T, new_value: T, success_order: AtomicO
```

This function performs a weak atomic compare-and-exchange operation, returning `null` if the current value is the given expected value. It's the equivalent of this code, except atomic:

cmpxchgWeakButNotAtomic

```
fn cmpxchgWeakButNotAtomic(comptime T: type, ptr: *T, expected_value: T, new_value: T) ?T {
    const old_value = ptr.*;
    if (old_value == expected_value and usuallyTrueButSometimesFalse()) {
        ptr.* = new_value;
        return null;
    } else {
        return old_value;
    }
}
```

If you are using cmpxchg in a retry loop, the sporadic failure will be no problem, and `cmpxchgWeak` is the better choice, because it can be implemented more efficiently in machine instructions. However if you need a stronger guarantee, use @cmpxchgStrong.

`T` must be a pointer, a `bool`, a float, an integer or an enum.

`@typeInfo(@TypeOf(ptr)).pointer.alignment` must be `>= @sizeOf(T).`

`AtomicOrder` can be found with `@import("std").builtin.AtomicOrder`.

See also:

- @atomicStore
- @atomicLoad
- @atomicRmw
- @cmpxchgStrong

@compileError

```
@compileError(comptime msg: []const u8) noreturn
```

This function, when semantically analyzed, causes a compile error with the message `msg`.

There are several ways that code avoids being semantically checked, such as using `if` or `switch` with compile time constants, and `comptime` functions.

@compileLog

```
@compileLog(...) void
```

This function prints the arguments passed to it at compile-time.

To prevent accidentally leaving compile log statements in a codebase, a compilation error is added to the build, pointing to the compile log statement. This error prevents code from being generated, but does not otherwise interfere with analysis.

This function can be used to do "printf debugging" on compile-time executing code.

test_compileLog_builtin.zig

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

const num1 = blk: {
    var val1: i32 = 99;
    @compileLog("comptime val1 = ", val1);
    val1 = val1 + 1;
    break :blk val1;
};

test "main" {
    @compileLog("comptime in main");

    print("Runtime in main, num1 = {}.\n", .{num1});
}
```

Shell

```
$ zig test test_compileLog_builtin.zig
doc/langref/test_compileLog_builtin.zig:11:5: error: found compile log statement
    @compileLog("comptime in main");
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
doc/langref/test_compileLog_builtin.zig:5:5: note: also here
    @compileLog("comptime val1 = ", val1);
    ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

Compile Log Output:
@as(*const [16:0]u8, "comptime in main")
@as(*const [16:0]u8, "comptime val1 = "), @as(i32, 99)
```

@constCast

```
@constCast(value: anytype) DestType
```

Remove `const` qualifier from a pointer.

@ctz

```
@ctz(operand: anytype) anytype
```

`@TypeOf(operand)` must be an integer type or an integer vector type.

`operand` may be an integer or vector.

Counts the number of least-significant (trailing in a big-endian sense) zeroes in an integer - "count trailing zeroes".

The return type is an unsigned integer or vector of unsigned integers with the minimum number of bits that can represent the bit count of the integer type.

If `operand` is zero, `@ctz` returns the bit width of integer type `T`.

See also:

- @clz
- @popCount

@cUndef

```
@cUndef(comptime name: []const u8) void
```

This function can only occur inside `@cImport`.

This appends `#undef $name` to the `@cImport` temporary buffer.

See also:

- Import from C Header File
- @cImport
- @cDefine
- @cInclude

@cVaArg

```
@cVaArg(operand: *std.builtin.VaList, comptime T: type) T
```

Implements the C macro `va_arg`.

See also:

- @cVaCopy
- @cVaEnd
- @cVaStart

@cVaCopy

```
@cVaCopy(src: *std.builtin.VaList) std.builtin.VaList
```

Implements the C macro `va_copy`.

See also:

- @cVaArg
- @cVaEnd
- @cVaStart

@cVaEnd

```
@cVaEnd(src: *std.builtin.VaList) void
```

Implements the C macro `va_end`.

See also:

- @cVaArg
- @cVaCopy
- @cVaStart

@cVaStart

```
@cVaStart() std.builtin.VaList
```

Implements the C macro `va_start`. Only valid inside a variadic function.

See also:

- @cVaArg
- @cVaCopy
- @cVaEnd

@divExact

```
@divExact(numerator: T, denominator: T) T
```

Exact division. Caller guarantees `denominator != 0` and `@divTrunc(numerator, denominator) * denominator == numerator`.

- `@divExact(6, 3) == 2`
- `@divExact(a, b) * b == a`

For a function that returns a possible error code, use `@import("std").math.divExact`.

See also:

- @divTrunc
- @divFloor

@divFloor

```
@divFloor(numerator: T, denominator: T) T
```

Floored division. Rounds toward negative infinity. For unsigned integers it is the same as `numerator / denominator`. Caller guarantees `denominator != 0` and `!(@typeInfo(T) == .int and T.is_signed and numerator == std.math.minInt(T) and denominator == -1)`.

- `@divFloor(-5, 3) == -2`
- `(@divFloor(a, b) * b) + @mod(a, b) == a`

For a function that returns a possible error code, use `@import("std").math.divFloor`.

See also:

- @divTrunc
- @divExact

@divTrunc

```
@divTrunc(numerator: T, denominator: T) T
```

Truncated division. Rounds toward zero. For unsigned integers it is the same as `numerator / denominator`. Caller guarantees `denominator != 0` and `!(@typeInfo(T) == .int and T.is_signed and numerator == std.math.minInt(T) and denominator == -1)`.

- `@divTrunc(-5, 3) == -1`
- `(@divTrunc(a, b) * b) + @rem(a, b) == a`

For a function that returns a possible error code, use `@import("std").math.divTrunc`.

See also:

- @divFloor
- @divExact

@embedFile

```
@embedFile(comptime path: []const u8) *const [N:0]u8
```

This function returns a compile time constant pointer to null-terminated, fixed-size array with length equal to the byte count of the file given by `path`. The contents of the array are the contents of the file. This is equivalent to a string literal with the file contents.

`path` is absolute or relative to the current file, just like `@import`.

See also:

- @import

@enumFromInt

```
@enumFromInt(integer: anytype) anytype
```

Converts an integer into an enum value. The return type is the inferred result type.

Attempting to convert an integer with no corresponding value in the enum invokes safety-checked Illegal Behavior. Note that a non-exhaustive enum has corresponding values for all integers in the enum's integer tag type: the `_` value represents all the remaining unnamed integers in the enum's tag type.

See also:

- @intFromEnum

@errorFromInt

```
@errorFromInt(value: std.meta.Int(.unsigned, @bitSizeOf(anyerror))) anyerror
```

Converts from the integer representation of an error into The Global Error Set type.

It is generally recommended to avoid this cast, as the integer representation of an error is not stable across source code changes.

Attempting to convert an integer that does not correspond to any error results in safety-checked Illegal Behavior.

See also:

- @intFromError

@errorName

```
@errorName(err: anyerror) [:0]const u8
```

This function returns the string representation of an error. The string representation of `error.OutOfMem` is `"OutOfMem"`.

If there are no calls to `@errorName` in an entire application, or all calls have a compile-time known value for `err`, then no error name table will be generated.

@errorReturnTrace

```
@errorReturnTrace() ?*builtin.StackTrace
```

If the binary is built with error return tracing, and this function is invoked in a function that calls a function with an error or error union return type, returns a stack trace object. Otherwise returns null.

@errorCast

```
@errorCast(value: anytype) anytype
```

Converts an error set or error union value from one error set to another error set. The return type is the inferred result type. Attempting to convert an error which is not in the destination error set results in safety-checked Illegal Behavior.

@export

```
@export(comptime ptr: *const anyopaque, comptime options: std.builtin.ExportOptions) void
```

Creates a symbol in the output object file which refers to the target of `ptr`.

`ptr` must point to a global variable or a comptime-known constant.

This builtin can be called from a comptime block to conditionally export symbols. When `ptr` points to a function with the C calling convention and `options.linkage` is `.Strong`, this is equivalent to the `export` keyword used on a function:

export_builtin.zig

```
comptime {
    @export(&internalName, .{ .name = "foo", .linkage = .strong });
}

fn internalName() callconv(.C) void {}
```

Shell

```
$ zig build-obj export_builtin.zig
```

This is equivalent to:

export_builtin_equivalent_code.zig

```
export fn foo() void {}
```

Shell

```
$ zig build-obj export_builtin_equivalent_code.zig
```

Note that even when using `export`, the `@"foo"` syntax for identifiers can be used to choose any string for the symbol name:

export_any_symbol_name.zig

```
export fn @"A function name that is a complete sentence."() void {}
```

Shell

```
$ zig build-obj export_any_symbol_name.zig
```

When looking at the resulting object, you can see the symbol is used verbatim:

```
00000000000001f0 T A function name that is a complete sentence.
```

See also:

- Exporting a C Library

@extern

```
@extern(T: type, comptime options: std.builtin.ExternOptions) T
```

Creates a reference to an external symbol in the output object file. T must be a pointer type.

See also:

- @export

@field

```
@field(lhs: anytype, comptime field_name: []const u8) (field)
```

Performs field access by a compile-time string. Works on both fields and declarations.

test_field_builtin.zig

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

const Point = struct {
    x: u32,
    y: u32,

    pub var z: u32 = 1;
};

test "field access by string" {
    const expect = std.testing.expect;
    var p = Point{ .x = 0, .y = 0 };

    @field(p, "x") = 4;
    @field(p, "y") = @field(p, "x") + 1;

    try expect(@field(p, "x") == 4);
    try expect(@field(p, "y") == 5);
}

test "decl access by string" {
    const expect = std.testing.expect;

    try expect(@field(Point, "z") == 1);

    @field(Point, "z") = 2;
    try expect(@field(Point, "z") == 2);
}
```

Shell

```
$ zig test test_field_builtin.zig
1/2 test_field_builtin.test.field access by string...OK
2/2 test_field_builtin.test.decl access by string...OK
All 2 tests passed.
```

@fieldParentPtr

```
@fieldParentPtr(comptime field_name: []const u8, field_ptr: *T) anytype
```

Given a pointer to a struct field, returns a pointer to the struct containing that field. The return type (and struct in question) is the inferred result type.

If `field_ptr` does not point to the `field_name` field of an instance of the result type, and the result type has ill-defined layout, invokes unchecked Illegal Behavior.

@FieldType

```
@FieldType(comptime Type: type, comptime field_name: []const u8) type
```

Given a type and the name of one of its fields, returns the type of that field.

@floatCast

```
@floatCast(value: anytype) anytype
```

Convert from one float type to another. This cast is safe, but may cause the numeric value to lose precision. The return type is the inferred result type.

@floatFromInt

```
@floatFromInt(int: anytype) anytype
```

Converts an integer to the closest floating point representation. The return type is the inferred result type. To convert the other way, use @intFromFloat. This operation is legal for all values of all integer types.

@frameAddress

```
@frameAddress() usize
```

This function returns the base pointer of the current stack frame.

The implications of this are target-specific and not consistent across all platforms. The frame address may not be available in release mode due to aggressive optimizations.

This function is only valid within function scope.

@hasDecl

```
@hasDecl(comptime Container: type, comptime name: []const u8) bool
```

Returns whether or not a [container](#) has a declaration matching `name`.

test_hasDecl_builtin.zig

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

const Foo = struct {
    nope: i32,

    pub var blah = "xxx";
    const hi = 1;
};

test "@hasDecl" {
    try expect(@hasDecl(Foo, "blah"));

    // Even though `hi` is private, @hasDecl returns true because this test is
    // in the same file scope as Foo. It would return false if Foo was declared
    // in a different file.
    try expect(@hasDecl(Foo, "hi"));

    // @hasDecl is for declarations; not fields.
    try expect(!@hasDecl(Foo, "nope"));
    try expect(!@hasDecl(Foo, "nope1234"));
}
```

Shell

```
$ zig test test_hasDecl_builtin.zig
1/1 test_hasDecl_builtin.test.@hasDecl...OK
All 1 tests passed.
```

See also:

- @hasField

@hasField

```
@hasField(comptime Container: type, comptime name: []const u8) bool
```

Returns whether the field name of a struct, union, or enum exists.

The result is a compile time constant.

It does not include functions, variables, or constants.

See also:

- @hasDecl

@import

```
@import(comptime path: []const u8) type
```

This function finds a zig file corresponding to `path` and adds it to the build, if it is not already added.

Zig source files are implicitly structs, with a name equal to the file's basename with the extension truncated. `@import` returns the struct type corresponding to the file.

Declarations which have the `pub` keyword may be referenced from a different source file than the one they are declared in.

`path` can be a relative path or it can be the name of a package. If it is a relative path, it is relative to the file that contains the `@import` function call.

The following packages are always available:

- `@import("std")` - Zig Standard Library
- `@import("builtin")` - Target-specific information The command `zig build-exe --show-builtin` outputs the source to stdout for reference.
- `@import("root")` - Root source file This is usually `src/main.zig` but depends on what file is built.

See also:

- Compile Variables
- @embedFile

@inComptime

```
@inComptime() bool
```

Returns whether the builtin was run in a `comptime` context. The result is a compile-time constant.

This can be used to provide alternative, comptime-friendly implementations of functions. It should not be used, for instance, to exclude certain functions from being evaluated at comptime.

See also:

- comptime

@intCast

```
@intCast(int: anytype) anytype
```

Converts an integer to another integer while keeping the same numerical value. The return type is the inferred result type. Attempting to convert a number which is out of range of the destination type results in safety-checked Illegal Behavior.

test_intCast_builtin.zig

```zig
test "integer cast panic" {
    var a: u16 = 0xabcd; // runtime-known
    _ = &a;
    const b: u8 = @intCast(a);
    _ = b;
}
```

Shell

```
$ zig test test_intCast_builtin.zig
1/1 test_intCast_builtin.test.integer cast panic...thread 222197 panic: integer cast truncated
/home/andy/src/zig/doc/langref/test_intCast_builtin.zig:4:19: 0x1048978 in test.integer cast pa
    const b: u8 = @intCast(a);
                  ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10eedb5 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10e734d in main (test)
        return mainTerminal();
                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10e68d2 in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e64ad in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/df39e05f4dd3b2dc748e167908afb941/test --seed=0x9bc4417f
```

To truncate the significant bits of a number out of range of the destination type, use @truncate.

If `T` is `comptime_int`, then this is semantically equivalent to Type Coercion.

@intFromBool

```
@intFromBool(value: bool) u1
```

Converts `true` to `@as(u1, 1)` and `false` to `@as(u1, 0)`.

@intFromEnum

```
@intFromEnum(enum_or_tagged_union: anytype) anytype
```

Converts an enumeration value into its integer tag type. When a tagged union is passed, the tag value is used as the enumeration value.

If there is only one possible enum value, the result is a `comptime_int` known at comptime.

See also:

- @enumFromInt

@intFromError

```
@intFromError(err: anytype) std.meta.Int(.unsigned, @bitSizeOf(anyerror))
```

Supports the following types:

- The Global Error Set
- Error Set Type
- Error Union Type

Converts an error to the integer representation of an error.

It is generally recommended to avoid this cast, as the integer representation of an error is not stable across source code changes.

See also:

- @errorFromInt

@intFromFloat

```
@intFromFloat(float: anytype) anytype
```

Converts the integer part of a floating point number to the inferred result type.

If the integer part of the floating point number cannot fit in the destination type, it invokes safety-checked Illegal Behavior.

See also:

- @floatFromInt

@intFromPtr

```
@intFromPtr(value: anytype) usize
```

Converts `value` to a `usize` which is the address of the pointer. `value` can be `*T` or `?*T`.

To convert the other way, use @ptrFromInt

@max

```
@max(...) T
```

Takes two or more arguments and returns the biggest value included (the maximum). This builtin accepts integers, floats, and vectors of either. In the latter case, the operation is performed element wise.

NaNs are handled as follows: return the biggest non-NaN value included. If all operands are NaN, return NaN.

See also:

- @min
- Vectors

@memcpy

```
@memcpy(noalias dest, noalias source) void
```

This function copies bytes from one region of memory to another.

`dest` must be a mutable slice, a mutable pointer to an array, or a mutable many-item pointer. It may have any alignment, and it may have any element type.

`source` must be a slice, a pointer to an array, or a many-item pointer. It may have any alignment, and it may have any element type.

The `source` element type must have the same in-memory representation as the `dest` element type.

Similar to for loops, at least one of `source` and `dest` must provide a length, and if two lengths are provided, they must be equal.

Finally, the two memory regions must not overlap.

@memset

```
@memset(dest, elem) void
```

This function sets all the elements of a memory region to `elem`.

`dest` must be a mutable slice or a mutable pointer to an array. It may have any alignment, and it may have any element type.

`elem` is coerced to the element type of `dest`.

For securely zeroing out sensitive contents from memory, you should use `std.crypto.secureZero`

@min

```
@min(...) T
```

Takes two or more arguments and returns the smallest value included (the minimum). This builtin accepts integers, floats, and vectors of either. In the latter case, the operation is performed element wise.

NaNs are handled as follows: return the smallest non-NaN value included. If all operands are NaN, return NaN.

See also:

- @max
- Vectors

@wasmMemorySize

```
@wasmMemorySize(index: u32) usize
```

This function returns the size of the Wasm memory identified by `index` as an unsigned value in units of Wasm pages. Note that each Wasm page is 64KB in size.

This function is a low level intrinsic with no safety mechanisms usually useful for allocator designers targeting Wasm. So unless you are writing a new allocator from scratch, you should use something like `@import("std").heap.WasmPageAllocator`.

See also:

- @wasmMemoryGrow

@wasmMemoryGrow

```
@wasmMemoryGrow(index: u32, delta: usize) isize
```

This function increases the size of the Wasm memory identified by `index` by `delta` in units of unsigned number of Wasm pages. Note that each Wasm page is 64KB in size. On success, returns previous memory size; on failure, if the allocation fails, returns -1.

This function is a low level intrinsic with no safety mechanisms usually useful for allocator designers targeting Wasm. So unless you are writing a new allocator from scratch, you should use something like `@import("std").heap.WasmPageAllocator`.

test_wasmMemoryGrow_builtin.zig

```
const std = @import("std");
const native_arch = @import("builtin").target.cpu.arch;
const expect = std.testing.expect;

test "@wasmMemoryGrow" {
    if (native_arch != .wasm32) return error.SkipZigTest;

    const prev = @wasmMemorySize(0);
    try expect(prev == @wasmMemoryGrow(0, 1));
    try expect(prev + 1 == @wasmMemorySize(0));
}
```

Shell

```
$ zig test test_wasmMemoryGrow_builtin.zig
1/1 test_wasmMemoryGrow_builtin.test.@wasmMemoryGrow...SKIP
0 passed; 1 skipped; 0 failed.
```

See also:

- @wasmMemorySize

@mod

```
@mod(numerator: T, denominator: T) T
```

Modulus division. For unsigned integers this is the same as `numerator % denominator`. Caller guarantees `denominator > 0`, otherwise the operation will result in a Remainder Division by Zero when runtime safety checks are enabled.

- `@mod(-5, 3) == 1`
- `(@divFloor(a, b) * b) + @mod(a, b) == a`

For a function that returns an error code, see `@import("std").math.mod`.

See also:

- @rem

@mulWithOverflow

```
@mulWithOverflow(a: anytype, b: anytype) struct { @TypeOf(a, b), u1 }
```

Performs `a * b` and returns a tuple with the result and a possible overflow bit.

@panic

```
@panic(message: []const u8) noreturn
```

Invokes the panic handler function. By default the panic handler function calls the public `panic` function exposed in the root source file, or if there is not one specified, the `std.builtin.default_panic` function from `std/builtin.zig`.

Generally it is better to use `@import("std").debug.panic`. However, `@panic` can be useful for 2 scenarios:

- From library code, calling the programmer's panic function if they exposed one in the root source file.
- When mixing C and Zig code, calling the canonical panic implementation across multiple .o files.

See also:

- Panic Handler

@popCount

```
@popCount(operand: anytype) anytype
```

`@TypeOf(operand)` must be an integer type.

`operand` may be an integer or vector.

Counts the number of bits set in an integer - "population count".

The return type is an unsigned integer or vector of unsigned integers with the minimum number of bits that can represent the bit count of the integer type.

See also:

- @ctz
- @clz

@prefetch

```
@prefetch(ptr: anytype, comptime options: PrefetchOptions) void
```

This builtin tells the compiler to emit a prefetch instruction if supported by the target CPU. If the target CPU does not support the requested prefetch instruction, this builtin is a no-op. This function has no effect on the behavior of the program, only on the performance characteristics.

The `ptr` argument may be any pointer type and determines the memory address to prefetch. This function does not dereference the pointer, it is perfectly legal to pass a pointer to invalid memory to this function and no Illegal Behavior will result.

`PrefetchOptions` can be found with `@import("std").builtin.PrefetchOptions`.

@ptrCast

```
@ptrCast(value: anytype) anytype
```

Converts a pointer of one type to a pointer of another type. The return type is the inferred result type.

Optional Pointers are allowed. Casting an optional pointer which is null to a non-optional pointer invokes safety-checked Illegal Behavior.

`@ptrCast` cannot be used for:

- Removing `const` qualifier, use @constCast.
- Removing `volatile` qualifier, use @volatileCast.
- Changing pointer address space, use @addrSpaceCast.
- Increasing pointer alignment, use @alignCast.
- Casting a non-slice pointer to a slice, use slicing syntax `ptr[start..end]`.

@ptrFromInt

```
@ptrFromInt(address: usize) anytype
```

Converts an integer to a pointer. The return type is the inferred result type. To convert the other way, use @intFromPtr. Casting an address of 0 to a destination type which in not optional and does not have the `allowzero` attribute will result in a Pointer Cast Invalid Null panic when runtime safety checks are enabled.

If the destination pointer type does not allow address zero and `address` is zero, this invokes safety-checked Illegal Behavior.

@rem

```
@rem(numerator: T, denominator: T) T
```

Remainder division. For unsigned integers this is the same as `numerator % denominator`. Caller guarantees `denominator > 0`, otherwise the operation will result in a Remainder Division by Zero when runtime safety checks are enabled.

- `@rem(-5, 3) == -2`
- `(@divTrunc(a, b) * b) + @rem(a, b) == a`

For a function that returns an error code, see `@import("std").math.rem`.

See also:

- @mod

@returnAddress

```
@returnAddress() usize
```

This function returns the address of the next machine code instruction that will be executed when the current function returns.

The implications of this are target-specific and not consistent across all platforms.

This function is only valid within function scope. If the function gets inlined into a calling function, the returned address will apply to the calling function.

@select

```
@select(comptime T: type, pred: @Vector(len, bool), a: @Vector(len, T), b: @Vector(len, T)) @Ve
```

Selects values element-wise from `a` or `b` based on `pred`. If `pred[i]` is `true`, the corresponding element in the result will be `a[i]` and otherwise `b[i]`.

See also:

- Vectors

@setEvalBranchQuota

```
@setEvalBranchQuota(comptime new_quota: u32) void
```

Increase the maximum number of backwards branches that compile-time code execution can use before giving up and making a compile error.

If the `new_quota` is smaller than the default quota (`1000`) or a previously explicitly set quota, it is ignored.

Example:

test_without_setEvalBranchQuota_builtin.zig

```
test "foo" {
    comptime {
        var i = 0;
        while (i < 1001) : (i += 1) {}
    }
}
```

Shell

```
$ zig test test_without_setEvalBranchQuota_builtin.zig
doc/langref/test_without_setEvalBranchQuota_builtin.zig:4:9: error: evaluation exceeded 1000 ba
        while (i < 1001) : (i += 1) {}
        ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
doc/langref/test_without_setEvalBranchQuota_builtin.zig:4:9: note: use @setEvalBranchQuota() to
```

Now we use `@setEvalBranchQuota`:

test_setEvalBranchQuota_builtin.zig

```
test "foo" {
    comptime {
        @setEvalBranchQuota(1001);
        var i = 0;
        while (i < 1001) : (i += 1) {}
    }
}
```

Shell

```
$ zig test test_setEvalBranchQuota_builtin.zig
1/1 test_setEvalBranchQuota_builtin.test.foo...OK
All 1 tests passed.
```

See also:

- comptime

@setFloatMode

```
@setFloatMode(comptime mode: FloatMode) void
```

Changes the current scope's rules about how floating point operations are defined.

- `Strict` (default) - Floating point operations follow strict IEEE compliance.
- `Optimized` - Floating point operations may do all of the following:
  - Assume the arguments and result are not NaN. Optimizations are required to retain legal behavior over NaNs, but the value of the result is undefined.
  - Assume the arguments and result are not +/-Inf. Optimizations are required to retain legal behavior over +/-Inf, but the value of the result is undefined.
  - Treat the sign of a zero argument or result as insignificant.
  - Use the reciprocal of an argument rather than perform division.
  - Perform floating-point contraction (e.g. fusing a multiply followed by an addition into a fused multiply-add).
  - Perform algebraically equivalent transformations that may change results in floating point (e.g. reassociate).

  This is equivalent to `-ffast-math` in GCC.

The floating point mode is inherited by child scopes, and can be overridden in any scope. You can set the floating point mode in a struct or module scope by using a comptime block.

`FloatMode` can be found with `@import("std").builtin.FloatMode`.

See also:

- Floating Point Operations

@setRuntimeSafety

```
@setRuntimeSafety(comptime safety_on: bool) void
```

Sets whether runtime safety checks are enabled for the scope that contains the function call.

test_setRuntimeSafety_builtin.zig

```
test "@setRuntimeSafety" {
    // The builtin applies to the scope that it is called in. So here, integer overflow
    // will not be caught in ReleaseFast and ReleaseSmall modes:
    // var x: u8 = 255;
    // x += 1; // Unchecked Illegal Behavior in ReleaseFast/ReleaseSmall modes.
    {
        // However this block has safety enabled, so safety checks happen here,
        // even in ReleaseFast and ReleaseSmall modes.
        @setRuntimeSafety(true);
        var x: u8 = 255;
        x += 1;
```

```
        {
            // The value can be overridden at any scope. So here integer overflow
            // would not be caught in any build mode.
            @setRuntimeSafety(false);
            // var x: u8 = 255;
            // x += 1; // Unchecked Illegal Behavior in all build modes.
        }
    }
}
```

Shell

```
$ zig test test_setRuntimeSafety_builtin.zig -OReleaseFast
1/1 test_setRuntimeSafety_builtin.test.@setRuntimeSafety...thread 222792 panic: integer overflo
/home/andy/src/zig/doc/langref/test_setRuntimeSafety_builtin.zig:11:11: 0x100b1f8 in test.@setR
        x += 1;
          ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x1033308 in main (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10319ad in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10314cd in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/fb1876bcfd12bd21ad987d48fd800b31/test --seed=0x3a9b5544
```

Note: it is planned to replace `@setRuntimeSafety` with `@optimizeFor`

@shlExact

```
@shlExact(value: T, shift_amt: Log2T) T
```

Performs the left shift operation (`<<`). For unsigned integers, the result is undefined if any 1 bits are shifted out. For signed integers, the result is undefined if any bits that disagree with the resultant sign bit are shifted out.

The type of `shift_amt` is an unsigned integer with `log2(@typeInfo(T).int.bits)` bits. This is because `shift_amt >= @typeInfo(T).int.bits` triggers safety-checked Illegal Behavior.

`comptime_int` is modeled as an integer with an infinite number of bits, meaning that in such case, `@shlExact` always produces a result and cannot produce a compile error.

See also:

- @shrExact
- @shlWithOverflow

@shlWithOverflow

```
@shlWithOverflow(a: anytype, shift_amt: Log2T) struct { @TypeOf(a), u1 }
```

Performs `a << b` and returns a tuple with the result and a possible overflow bit.

The type of `shift_amt` is an unsigned integer with `log2(@typeInfo(@TypeOf(a)).int.bits)` bits. This is because `shift_amt >= @typeInfo(@TypeOf(a)).int.bits` triggers safety-checked Illegal Behavior.

See also:

- @shlExact
- @shrExact

@shrExact

```
@shrExact(value: T, shift_amt: Log2T) T
```

Performs the right shift operation (`>>`). Caller guarantees that the shift will not shift any 1 bits out.

The type of `shift_amt` is an unsigned integer with `log2(@typeInfo(T).int.bits)` bits. This is because `shift_amt >= @typeInfo(T).int.bits` triggers safety-checked Illegal Behavior.

See also:

- @shlExact
- @shlWithOverflow

@shuffle

```
@shuffle(comptime E: type, a: @Vector(a_len, E), b: @Vector(b_len, E), comptime mask: @Vector(m
```

Constructs a new vector by selecting elements from `a` and `b` based on `mask`.

Each element in `mask` selects an element from either `a` or `b`. Positive numbers select from `a` starting at 0. Negative values select from `b`, starting at `-1` and going down. It is recommended to use the `~` operator for indexes from `b` so that both indexes can start from `0` (i.e. `~@as(i32, 0)` is `-1`).

For each element of `mask`, if it or the selected value from `a` or `b` is `undefined`, then the resulting element is `undefined`.

`a_len` and `b_len` may differ in length. Out-of-bounds element indexes in `mask` result in compile errors.

If `a` or `b` is `undefined`, it is equivalent to a vector of all `undefined` with the same length as the other vector. If both vectors are `undefined`, `@shuffle` returns a vector with all elements `undefined`.

`E` must be an integer, float, pointer, or `bool`. The mask may be any vector length, and its length determines the result length.

test_shuffle_builtin.zig

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

test "vector @shuffle" {
    const a = @Vector(7, u8){ 'o', 'l', 'h', 'e', 'r', 'z', 'w' };
    const b = @Vector(4, u8){ 'w', 'd', '!', 'x' };

    // To shuffle within a single vector, pass undefined as the second argument.
    // Notice that we can re-order, duplicate, or omit elements of the input vector
    const mask1 = @Vector(5, i32){ 2, 3, 1, 1, 0 };
    const res1: @Vector(5, u8) = @shuffle(u8, a, undefined, mask1);
    try expect(std.mem.eql(u8, &@as([5]u8, res1), "hello"));

    // Combining two vectors
    const mask2 = @Vector(6, i32){ -1, 0, 4, 1, -2, -3 };
    const res2: @Vector(6, u8) = @shuffle(u8, a, b, mask2);
    try expect(std.mem.eql(u8, &@as([6]u8, res2), "world!"));
}
```

Shell

```
$ zig test test_shuffle_builtin.zig
1/1 test_shuffle_builtin.test.vector @shuffle...OK
All 1 tests passed.
```

See also:

- Vectors

@sizeOf

```
@sizeOf(comptime T: type) comptime_int
```

This function returns the number of bytes it takes to store `T` in memory. The result is a target-specific compile time constant.

This size may contain padding bytes. If there were two consecutive T in memory, the padding would be the offset in bytes between element at index 0 and the element at index 1. For integer, consider whether you want to use `@sizeOf(T)` or `@typeInfo(T).int.bits`.

This function measures the size at runtime. For types that are disallowed at runtime, such as `comptime_int` and `type`, the result is `0`.

See also:

- @bitSizeOf
- @typeInfo

@splat

```
@splat(scalar: anytype) anytype
```

Produces a vector where each element is the value `scalar`. The return type and thus the length of the vector is inferred.

test_splat_builtin.zig

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

test "vector @splat" {
    const scalar: u32 = 5;
    const result: @Vector(4, u32) = @splat(scalar);
    try expect(std.mem.eql(u32, &@as([4]u32, result), &[_]u32{ 5, 5, 5, 5 }));
}
```

Shell

```
$ zig test test_splat_builtin.zig
1/1 test_splat_builtin.test.vector @splat...OK
All 1 tests passed.
```

`scalar` must be an integer, bool, float, or pointer.

See also:

- Vectors
- @shuffle

@reduce

```
@reduce(comptime op: std.builtin.ReduceOp, value: anytype) E
```

Transforms a vector into a scalar value (of type `E`) by performing a sequential horizontal reduction of its elements using the specified operator `op`.

Not every operator is available for every vector element type:

- Every operator is available for integer vectors.
- `.And`, `.Or`, `.Xor` are additionally available for `bool` vectors,
- `.Min`, `.Max`, `.Add`, `.Mul` are additionally available for floating point vectors,

Note that `.Add` and `.Mul` reductions on integral types are wrapping; when applied on floating point types the operation associativity is preserved, unless the float mode is set to `Optimized`.

test_reduce_builtin.zig

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

test "vector @reduce" {
    const V = @Vector(4, i32);
    const value = V{ 1, -1, 1, -1 };
    const result = value > @as(V, @splat(0));
    // result is { true, false, true, false };
    try comptime expect(@TypeOf(result) == @Vector(4, bool));
    const is_all_true = @reduce(.And, result);
    try comptime expect(@TypeOf(is_all_true) == bool);
    try expect(is_all_true == false);
}
```

Shell

```
$ zig test test_reduce_builtin.zig
1/1 test_reduce_builtin.test.vector @reduce...OK
All 1 tests passed.
```

See also:

- Vectors
- @setFloatMode

@src

```
@src() std.builtin.SourceLocation
```

Returns a `SourceLocation` struct representing the function's name and location in the source code. This must be called in a function.

test_src_builtin.zig

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

test "@src" {
    try doTheTest();
}
```

```
fn doTheTest() !void {
    const src = @src();

    try expect(src.line == 9);
    try expect(src.column == 17);
    try expect(std.mem.endsWith(u8, src.fn_name, "doTheTest"));
    try expect(std.mem.endsWith(u8, src.file, "test_src_builtin.zig"));
}
```

Shell

```
$ zig test test_src_builtin.zig
1/1 test_src_builtin.test.@src...OK
All 1 tests passed.
```

@sqrt

```
@sqrt(value: anytype) @TypeOf(value)
```

Performs the square root of a floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@sin

```
@sin(value: anytype) @TypeOf(value)
```

Sine trigonometric function on a floating point number in radians. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@cos

```
@cos(value: anytype) @TypeOf(value)
```

Cosine trigonometric function on a floating point number in radians. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@tan

```
@tan(value: anytype) @TypeOf(value)
```

Tangent trigonometric function on a floating point number in radians. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@exp

```
@exp(value: anytype) @TypeOf(value)
```

Base-e exponential function on a floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@exp2

```
@exp2(value: anytype) @TypeOf(value)
```

Base-2 exponential function on a floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@log

```
@log(value: anytype) @TypeOf(value)
```

Returns the natural logarithm of a floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@log2

```
@log2(value: anytype) @TypeOf(value)
```

Returns the logarithm to the base 2 of a floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@log10

```
@log10(value: anytype) @TypeOf(value)
```

Returns the logarithm to the base 10 of a floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@abs

```
@abs(value: anytype) anytype
```

Returns the absolute value of an integer or a floating point number. Uses a dedicated hardware instruction when available. The return type is always an unsigned integer of the same bit width as the operand if the operand is an integer. Unsigned integer operands are supported. The builtin cannot overflow for signed integer operands.

Supports Floats, Integers and Vectors of floats or integers.

@floor

```
@floor(value: anytype) @TypeOf(value)
```

Returns the largest integral value not greater than the given floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@ceil

```
@ceil(value: anytype) @TypeOf(value)
```

Returns the smallest integral value not less than the given floating point number. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@trunc

```
@trunc(value: anytype) @TypeOf(value)
```

Rounds the given floating point number to an integer, towards zero. Uses a dedicated hardware instruction when available.

Supports Floats and Vectors of floats.

@round

```
@round(value: anytype) @TypeOf(value)
```

Rounds the given floating point number to the nearest integer. If two integers are equally close, rounds away from zero. Uses a dedicated hardware instruction when available.

test_round_builtin.zig

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

test "@round" {
    try expect(@round(1.4) == 1);
    try expect(@round(1.5) == 2);
    try expect(@round(-1.4) == -1);
    try expect(@round(-2.5) == -3);
}
```

Shell

```
$ zig test test_round_builtin.zig
1/1 test_round_builtin.test.@round...OK
All 1 tests passed.
```

Supports Floats and Vectors of floats.

@subWithOverflow

```
@subWithOverflow(a: anytype, b: anytype) struct { @TypeOf(a, b), u1 }
```

Performs `a - b` and returns a tuple with the result and a possible overflow bit.

@tagName

```
@tagName(value: anytype) [:0]const u8
```

Converts an enum value or union value to a string literal representing the name.

If the enum is non-exhaustive and the tag value does not map to a name, it invokes safety-checked Illegal Behavior.

@This

```
@This() type
```

Returns the innermost struct, enum, or union that this function call is inside. This can be useful for an anonymous struct that needs to refer to itself:

test_this_builtin.zig

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

test "@This()" {
    var items = [_]i32{ 1, 2, 3, 4 };
    const list = List(i32){ .items = items[0..] };
    try expect(list.length() == 4);
}

fn List(comptime T: type) type {
    return struct {
        const Self = @This();

        items: []T,

        fn length(self: Self) usize {
            return self.items.len;
        }
    };
}
```

Shell

```
$ zig test test_this_builtin.zig
1/1 test_this_builtin.test.@This()...OK
All 1 tests passed.
```

When `@This()` is used at file scope, it returns a reference to the struct that corresponds to the current file.

@trap

```
@trap() noreturn
```

This function inserts a platform-specific trap/jam instruction which can be used to exit the program abnormally. This may be implemented by explicitly emitting an invalid instruction which may cause an illegal instruction exception of some sort. Unlike for `@breakpoint()`, execution does not continue after this point.

Outside function scope, this builtin causes a compile error.

See also:

- @breakpoint

@truncate

```
@truncate(integer: anytype) anytype
```

This function truncates bits from an integer type, resulting in a smaller or same-sized integer type. The return type is the inferred result type.

This function always truncates the significant bits of the integer, regardless of endianness on the target platform.

Calling `@truncate` on a number out of range of the destination type is well defined and working code:

test_truncate_builtin.zig

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

test "integer truncation" {
    const a: u16 = 0xabcd;
    const b: u8 = @truncate(a);
    try expect(b == 0xcd);
}
```

Shell

```
$ zig test test_truncate_builtin.zig
1/1 test_truncate_builtin.test.integer truncation...OK
All 1 tests passed.
```

Use @intCast to convert numbers guaranteed to fit the destination type.

@Type

```
@Type(comptime info: std.builtin.Type) type
```

This function is the inverse of @typeInfo. It reifies type information into a `type`.

It is available for the following types:

- `type`
- `noreturn`
- `void`
- `bool`
- Integers - The maximum bit count for an integer type is `65535`.
- Floats
- Pointers

- `comptime_int`
- `comptime_float`
- `@TypeOf(undefined)`
- `@TypeOf(null)`
- Arrays
- Optionals
- Error Set Type
- Error Union Type
- Vectors
- opaque
- `anyframe`
- struct
- enum
- Enum Literals
- union
- Functions

@typeInfo

```
@typeInfo(comptime T: type) std.builtin.Type
```

Provides type reflection.

Type information of structs, unions, enums, and error sets has fields which are guaranteed to be in the same order as appearance in the source file.

Type information of structs, unions, enums, and opaques has declarations, which are also guaranteed to be in the same order as appearance in the source file.

@typeName

```
@typeName(T: type) *const [N:0]u8
```

This function returns the string representation of a type, as an array. It is equivalent to a string literal of the type name. The returned type name is fully qualified with the parent namespace included as part of the type name with a series of dots.

@TypeOf

```
@TypeOf(...) type
```

`@TypeOf` is a special builtin function that takes any (non-zero) number of expressions as parameters and returns the type of the result, using Peer Type Resolution.

The expressions are evaluated, however they are guaranteed to have no *runtime* side-effects:

test_TypeOf_builtin.zig

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

test "no runtime side effects" {
    var data: i32 = 0;
    const T = @TypeOf(foo(i32, &data));
    try comptime expect(T == i32);
    try expect(data == 0);
}

fn foo(comptime T: type, ptr: *T) T {
    ptr.* += 1;
    return ptr.*;
}
```

Shell

```
$ zig test test_TypeOf_builtin.zig
1/1 test_TypeOf_builtin.test.no runtime side effects...OK
All 1 tests passed.
```

@unionInit

```
@unionInit(comptime Union: type, comptime active_field_name: []const u8, init_expr) Union
```

This is the same thing as union initialization syntax, except that the field name is a comptime-known value rather than an identifier token.

`@unionInit` forwards its result location to `init_expr`.

@Vector

```
@Vector(len: comptime_int, Element: type) type
```

Creates Vectors.

@volatileCast

```
@volatileCast(value: anytype) DestType
```

Remove `volatile` qualifier from a pointer.

@workGroupId

```
@workGroupId(comptime dimension: u32) u32
```

Returns the index of the work group in the current kernel invocation in dimension `dimension`.

@workGroupSize

```
@workGroupSize(comptime dimension: u32) u32
```

Returns the number of work items that a work group has in dimension `dimension`.

@workItemId

```
@workItemId(comptime dimension: u32) u32
```

Returns the index of the work item in the work group in dimension `dimension`. This function returns values between `0` (inclusive) and `@workGroupSize(dimension)` (exclusive).

Build Mode

Zig has four build modes:

- Debug (default)
- ReleaseFast
- ReleaseSafe
- ReleaseSmall

To add standard build options to a **build.zig** file:

build.zig

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

pub fn build(b: *std.Build) void {
    const optimize = b.standardOptimizeOption(.{});
    const exe = b.addExecutable(.{
        .name = "example",
        .root_source_file = b.path("example.zig"),
        .optimize = optimize,
    });
    b.default_step.dependOn(&exe.step);
}
```

This causes these options to be available:

`-Doptimize=Debug`
: Optimizations off and safety on (default)

`-Doptimize=ReleaseSafe`
: Optimizations on and safety on

`-Doptimize=ReleaseFast`
: Optimizations on and safety off

`-Doptimize=ReleaseSmall`
: Size optimizations on and safety off

Debug

Shell

```
$ zig build-exe example.zig
```

- Fast compilation speed
- Safety checks enabled
- Slow runtime performance
- Large binary size
- No reproducible build requirement

ReleaseFast

Shell

```
$ zig build-exe example.zig -O ReleaseFast
```

- Fast runtime performance
- Safety checks disabled
- Slow compilation speed
- Large binary size
- Reproducible build

ReleaseSafe

Shell

```
$ zig build-exe example.zig -O ReleaseSafe
```

- Medium runtime performance
- Safety checks enabled
- Slow compilation speed
- Large binary size
- Reproducible build

ReleaseSmall

Shell

```
$ zig build-exe example.zig -O ReleaseSmall
```

- Medium runtime performance
- Safety checks disabled
- Slow compilation speed
- Small binary size
- Reproducible build

See also:

- Compile Variables
- Zig Build System
- Illegal Behavior

Single Threaded Builds

Zig has a compile option `-fsingle-threaded` which has the following effects:

- All Thread Local Variables are treated as regular Container Level Variables.
- The overhead of Async Functions becomes equivalent to function call overhead.
- The `@import("builtin").single_threaded` becomes `true` and therefore various userland APIs which read this variable become more efficient. For example `std.Mutex` becomes an empty data structure and all of its functions become no-ops.

Illegal Behavior

Many operations in Zig trigger what is known as "Illegal Behavior" (IB). If Illegal Behavior is detected at compile-time, Zig emits a compile error and refuses to continue. Otherwise, when Illegal Behavior is not caught at compile-time, it falls into one of two categories.

Some Illegal Behavior is *safety-checked*: this means that the compiler will insert "safety checks" anywhere that the Illegal Behavior may occur at runtime, to determine whether it is about to happen. If it is, the safety check "fails", which triggers a panic.

All other Illegal Behavior is *unchecked*, meaning the compiler is unable to insert safety checks for it. If Unchecked Illegal Behavior is invoked at runtime, anything can happen: usually that will be some kind of crash, but the optimizer is free to make Unchecked Illegal Behavior do anything, such as calling arbitrary functions or clobbering arbitrary data. This is similar to the concept of "undefined behavior" in some other languages. Note that Unchecked Illegal Behavior still always results in a compile error if evaluated at comptime, because the Zig compiler is able to perform more sophisticated checks at compile-time than at runtime.

Most Illegal Behavior is safety-checked. However, to facilitate optimizations, safety checks are disabled by default in the ReleaseFast and ReleaseSmall optimization modes. Safety checks can also be enabled or disabled on a per-block basis, overriding the default for the current optimization mode, using @setRuntimeSafety. When safety checks are disabled, Safety-Checked Illegal Behavior behaves like Unchecked Illegal Behavior; that is, any behavior may result from invoking it.

When a safety check fails, Zig's default panic handler crashes with a stack trace, like this:

test_illegal_behavior.zig

```
test "safety check" {
    unreachable;
}
```

Shell

```
$ zig test test_illegal_behavior.zig
1/1 test_illegal_behavior.test.safety check...thread 210560 panic: reached unreachable code
/home/andy/src/zig/doc/langref/test_illegal_behavior.zig:2:5: 0x1048948 in test.safety check (t
    unreachable;
    ^
/home/andy/src/zig/lib/compiler/test_runner.zig:214:25: 0x10eed75 in mainTerminal (test)
        if (test_fn.func()) |_| {
                        ^
/home/andy/src/zig/lib/compiler/test_runner.zig:62:28: 0x10e730d in main (test)
        return mainTerminal();
                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10e6892 in posixCallMainAndExit (test)
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e646d in _start (test)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
error: the following test command crashed:
/home/andy/src/zig/.zig-cache/o/ad5fcf79fd9ccd7e0493595fe81df80a/test --seed=0x4cb0f272
```

Reaching Unreachable Code

At compile-time:

test_comptime_reaching_unreachable.zig

```
comptime {
    assert(false);
}
fn assert(ok: bool) void {
```

```
    if (!ok) unreachable; // assertion failure
}
```

Shell

```
$ zig test test_comptime_reaching_unreachable.zig
doc/langref/test_comptime_reaching_unreachable.zig:5:14: error: reached unreachable code
    if (!ok) unreachable; // assertion failure
             ^~~~~~~~~~~
doc/langref/test_comptime_reaching_unreachable.zig:2:11: note: called from here
    assert(false);
    ~~~~~~^~~~~~~
```

At runtime:

runtime_reaching_unreachable.zig

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

pub fn main() void {
    std.debug.assert(false);
}
```

Shell

```
$ zig build-exe runtime_reaching_unreachable.zig
$ ./runtime_reaching_unreachable
thread 217396 panic: reached unreachable code
/home/andy/src/zig/lib/std/debug.zig:522:14: 0x1048a9d in assert (runtime_reaching_unreachable)
    if (!ok) unreachable; // assertion failure
             ^
/home/andy/src/zig/doc/langref/runtime_reaching_unreachable.zig:4:21: 0x10de72a in main (runtim
    std.debug.assert(false);
                    ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de232 in posixCallMainAndExit (runtime_reachir
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10dde0d in _start (runtime_reaching_unreachable)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Index out of Bounds

At compile-time:

test_comptime_index_out_of_bounds.zig

```
comptime {
    const array: [5]u8 = "hello".*;
    const garbage = array[5];
    _ = garbage;
}
```

Shell

```
$ zig test test_comptime_index_out_of_bounds.zig
doc/langref/test_comptime_index_out_of_bounds.zig:3:27: error: index 5 outside array of length
    const garbage = array[5];
                          ^
```

At runtime:

runtime_index_out_of_bounds.zig

```
pub fn main() void {
    const x = foo("hello");
    _ = x;
}

fn foo(x: []const u8) u8 {
    return x[5];
}
```

Shell

```
$ zig build-exe runtime_index_out_of_bounds.zig
$ ./runtime_index_out_of_bounds
thread 219134 panic: index out of bounds: index 5, len 5
/home/andy/src/zig/doc/langref/runtime_index_out_of_bounds.zig:7:13: 0x10df001 in foo (runtime_
    return x[5];
            ^
/home/andy/src/zig/doc/langref/runtime_index_out_of_bounds.zig:2:18: 0x10de766 in main (runtime
    const x = foo("hello");
                 ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de262 in posixCallMainAndExit (runtime_index_o
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10dde3d in _start (runtime_index_out_of_bounds)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Cast Negative Number to Unsigned Integer

At compile-time:

test_comptime_invalid_cast.zig

```
comptime {
    const value: i32 = -1;
    const unsigned: u32 = @intCast(value);
    _ = unsigned;
}
```

Shell

```
$ zig test test_comptime_invalid_cast.zig
doc/langref/test_comptime_invalid_cast.zig:3:36: error: type 'u32' cannot represent integer val
```

```
    const unsigned: u32 = @intCast(value);
                               ^~~~~
```

At runtime:

runtime_invalid_cast.zig

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

pub fn main() void {
    var value: i32 = -1; // runtime-known
    _ = &value;
    const unsigned: u32 = @intCast(value);
    std.debug.print("value: {}\n", .{unsigned});
}
```

Shell

```
$ zig build-exe runtime_invalid_cast.zig
$ ./runtime_invalid_cast
thread 212398 panic: attempt to cast negative value to unsigned integer
/home/andy/src/zig/doc/langref/runtime_invalid_cast.zig:6:27: 0x10de866 in main (runtime_invali
    const unsigned: u32 = @intCast(value);
                      ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de352 in posixCallMainAndExit (runtime_invalic
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf2d in _start (runtime_invalid_cast)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

To obtain the maximum value of an unsigned integer, use `std.math.maxInt`.

Cast Truncates Data

At compile-time:

test_comptime_invalid_cast_truncate.zig

```
comptime {
    const spartan_count: u16 = 300;
    const byte: u8 = @intCast(spartan_count);
    _ = byte;
}
```

Shell

```
$ zig test test_comptime_invalid_cast_truncate.zig
doc/langref/test_comptime_invalid_cast_truncate.zig:3:31: error: type 'u8' cannot represent int
    const byte: u8 = @intCast(spartan_count);
                              ^~~~~~~~~~~~~
```

At runtime:

runtime_invalid_cast_truncate.zig

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

pub fn main() void {
    var spartan_count: u16 = 300; // runtime-known
    _ = &spartan_count;
    const byte: u8 = @intCast(spartan_count);
    std.debug.print("value: {}\n", .{byte});
}
```

Shell

```
$ zig build-exe runtime_invalid_cast_truncate.zig
$ ./runtime_invalid_cast_truncate
thread 212601 panic: integer cast truncated bits
/home/andy/src/zig/doc/langref/runtime_invalid_cast_truncate.zig:6:22: 0x10de8f8 in main (runti
    const byte: u8 = @intCast(spartan_count);
                     ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de3e2 in posixCallMainAndExit (runtime_invalid
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddfbd in _start (runtime_invalid_cast_truncate)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

To truncate bits, use @truncate.

Integer Overflow

Default Operations

The following operators can cause integer overflow:

- `+` (addition)
- `-` (subtraction)
- `-` (negation)
- `*` (multiplication)
- `/` (division)
- @divTrunc (division)
- @divFloor (division)
- @divExact (division)

Example with addition at compile-time:

test_comptime_overflow.zig

```
comptime {
    var byte: u8 = 255;
    byte += 1;
}
```

Shell

```
$ zig test test_comptime_overflow.zig
doc/langref/test_comptime_overflow.zig:3:10: error: overflow of integer type 'u8' with value '2
    byte += 1;
    ~~~~~^~~~
```

At runtime:

runtime_overflow.zig

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

pub fn main() void {
    var byte: u8 = 255;
    byte += 1;
    std.debug.print("value: {}\n", .{byte});
}
```

Shell

```
$ zig build-exe runtime_overflow.zig
$ ./runtime_overflow
thread 217657 panic: integer overflow
/home/andy/src/zig/doc/langref/runtime_overflow.zig:5:10: 0x10de8f9 in main (runtime_overflow)
    byte += 1;
         ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de3e2 in posixCallMainAndExit (runtime_overflo
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddfbd in _start (runtime_overflow)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Standard Library Math Functions

These functions provided by the standard library return possible errors.

- `@import("std").math.add`
- `@import("std").math.sub`
- `@import("std").math.mul`
- `@import("std").math.divTrunc`
- `@import("std").math.divFloor`
- `@import("std").math.divExact`

- `@import("std").math.shl`

Example of catching an overflow for addition:

math_add.zig

```
const math = @import("std").math;
const print = @import("std").debug.print;
pub fn main() !void {
    var byte: u8 = 255;

    byte = if (math.add(u8, byte, 1)) |result| result else |err| {
        print("unable to add one: {s}\n", .{@errorName(err)});
        return err;
    };

    print("result: {}\n", .{byte});
}
```

Shell

```
$ zig build-exe math_add.zig
$ ./math_add
unable to add one: Overflow
error: Overflow
/home/andy/src/zig/lib/std/math.zig:565:21: 0x10dea25 in add__anon_24023 (math_add)
    if (ov[1] != 0) return error.Overflow;
                    ^
/home/andy/src/zig/doc/langref/math_add.zig:8:9: 0x10de9bb in main (math_add)
        return err;
        ^
```

Builtin Overflow Functions

These builtins return a tuple containing whether there was an overflow (as a `u1`) and the possibly overflowed bits of the operation:

- @addWithOverflow
- @subWithOverflow
- @mulWithOverflow
- @shlWithOverflow

Example of @addWithOverflow:

addWithOverflow_builtin.zig

```
const print = @import("std").debug.print;
pub fn main() void {
    const byte: u8 = 255;

    const ov = @addWithOverflow(byte, 10);
    if (ov[1] != 0) {
        print("overflowed result: {}\n", .{ov[0]});
```

```
    } else {
        print("result: {}\n", .{ov[0]});
    }
}
```

Shell

```
$ zig build-exe addWithOverflow_builtin.zig
$ ./addWithOverflow_builtin
overflowed result: 9
```

Wrapping Operations

These operations have guaranteed wraparound semantics.

- `+%` (wraparound addition)
- `-%` (wraparound subtraction)
- `-%` (wraparound negation)
- `*%` (wraparound multiplication)

test_wraparound_semantics.zig

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

test "wraparound addition and subtraction" {
    const x: i32 = maxInt(i32);
    const min_val = x +% 1;
    try expect(min_val == minInt(i32));
    const max_val = min_val -% 1;
    try expect(max_val == maxInt(i32));
}
```

Shell

```
$ zig test test_wraparound_semantics.zig
1/1 test_wraparound_semantics.test.wraparound addition and subtraction...OK
All 1 tests passed.
```

Exact Left Shift Overflow

At compile-time:

test_comptime_shlExact_overwlow.zig

```
comptime {
    const x = @shlExact(@as(u8, 0b01010101), 2);
    _ = x;
}
```

Shell

```
$ zig test test_comptime_shlExact_overwlow.zig
doc/langref/test_comptime_shlExact_overwlow.zig:2:15: error: operation caused overflow
    const x = @shlExact(@as(u8, 0b01010101), 2);
              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

At runtime:

runtime_shlExact_overflow.zig

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

pub fn main() void {
    var x: u8 = 0b01010101; // runtime-known
    _ = &x;
    const y = @shlExact(x, 2);
    std.debug.print("value: {}\n", .{y});
}
```

Shell

```
$ zig build-exe runtime_shlExact_overflow.zig
$ ./runtime_shlExact_overflow
thread 222268 panic: left shift overflowed bits
/home/andy/src/zig/doc/langref/runtime_shlExact_overflow.zig:6:5: 0x10de981 in main (runtime_sh
    const y = @shlExact(x, 2);
    ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de442 in posixCallMainAndExit (runtime_shlExac
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10de01d in _start (runtime_shlExact_overflow)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Exact Right Shift Overflow

At compile-time:

test_comptime_shrExact_overflow.zig

```
comptime {
    const x = @shrExact(@as(u8, 0b10101010), 2);
    _ = x;
}
```

Shell

```
$ zig test test_comptime_shrExact_overflow.zig
doc/langref/test_comptime_shrExact_overflow.zig:2:15: error: exact shift shifted out 1 bits
    const x = @shrExact(@as(u8, 0b10101010), 2);
              ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

At runtime:

runtime_shrExact_overflow.zig

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

pub fn main() void {
    var x: u8 = 0b10101010; // runtime-known
    _ = &x;
    const y = @shrExact(x, 2);
    std.debug.print("value: {}\n", .{y});
}
```

Shell

```
$ zig build-exe runtime_shrExact_overflow.zig
$ ./runtime_shrExact_overflow
thread 215275 panic: right shift overflowed bits
/home/andy/src/zig/doc/langref/runtime_shrExact_overflow.zig:6:5: 0x10de97d in main (runtime_sh
    const y = @shrExact(x, 2);
    ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de442 in posixCallMainAndExit (runtime_shrExac
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10de01d in _start (runtime_shrExact_overflow)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Division by Zero

At compile-time:

test_comptime_division_by_zero.zig

```
comptime {
    const a: i32 = 1;
    const b: i32 = 0;
    const c = a / b;
    _ = c;
}
```

Shell

```
$ zig test test_comptime_division_by_zero.zig
doc/langref/test_comptime_division_by_zero.zig:4:19: error: division by zero here causes undefi
    const c = a / b;
                  ^
```

At runtime:

runtime_division_by_zero.zig

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

```
pub fn main() void {
    var a: u32 = 1;
    var b: u32 = 0;
    _ = .{ &a, &b };
    const c = a / b;
    std.debug.print("value: {}\n", .{c});
}
```

Shell

```
$ zig build-exe runtime_division_by_zero.zig
$ ./runtime_division_by_zero
thread 213940 panic: division by zero
/home/andy/src/zig/doc/langref/runtime_division_by_zero.zig:7:17: 0x10de89a in main (runtime_di
    const c = a / b;
                ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de352 in posixCallMainAndExit (runtime_divisio
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf2d in _start (runtime_division_by_zero)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Remainder Division by Zero

At compile-time:

test_comptime_remainder_division_by_zero.zig

```
comptime {
    const a: i32 = 10;
    const b: i32 = 0;
    const c = a % b;
    _ = c;
}
```

Shell

```
$ zig test test_comptime_remainder_division_by_zero.zig
doc/langref/test_comptime_remainder_division_by_zero.zig:4:19: error: division by zero here cau
    const c = a % b;
                  ^
```

At runtime:

runtime_remainder_division_by_zero.zig

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

pub fn main() void {
    var a: u32 = 10;
    var b: u32 = 0;
```

```
    _ = .{ &a, &b };
    const c = a % b;
    std.debug.print("value: {}\n", .{c});
}
```

Shell

```
$ zig build-exe runtime_remainder_division_by_zero.zig
$ ./runtime_remainder_division_by_zero
thread 209979 panic: division by zero
/home/andy/src/zig/doc/langref/runtime_remainder_division_by_zero.zig:7:17: 0x10de89a in main (
    const c = a % b;
                ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de352 in posixCallMainAndExit (runtime_remainc
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf2d in _start (runtime_remainder_division_by_
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Exact Division Remainder

At compile-time:

test_comptime_divExact_remainder.zig

```
comptime {
    const a: u32 = 10;
    const b: u32 = 3;
    const c = @divExact(a, b);
    _ = c;
}
```

Shell

```
$ zig test test_comptime_divExact_remainder.zig
doc/langref/test_comptime_divExact_remainder.zig:4:15: error: exact division produced remainder
    const c = @divExact(a, b);
              ^~~~~~~~~~~~~~~
```

At runtime:

runtime_divExact_remainder.zig

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

pub fn main() void {
    var a: u32 = 10;
    var b: u32 = 3;
    _ = .{ &a, &b };
    const c = @divExact(a, b);
    std.debug.print("value: {}\n", .{c});
}
```

Shell

```
$ zig build-exe runtime_divExact_remainder.zig
$ ./runtime_divExact_remainder
thread 212495 panic: exact division produced remainder
/home/andy/src/zig/doc/langref/runtime_divExact_remainder.zig:7:15: 0x10de8bb in main (runtime_
    const c = @divExact(a, b);
              ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de352 in posixCallMainAndExit (runtime_divExac
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf2d in _start (runtime_divExact_remainder)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Attempt to Unwrap Null

At compile-time:

test_comptime_unwrap_null.zig

```
comptime {
    const optional_number: ?i32 = null;
    const number = optional_number.?;
    _ = number;
}
```

Shell

```
$ zig test test_comptime_unwrap_null.zig
doc/langref/test_comptime_unwrap_null.zig:3:35: error: unable to unwrap null
    const number = optional_number.?;
                   ~~~~~~~~~~~~~~~^~
```

At runtime:

runtime_unwrap_null.zig

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

pub fn main() void {
    var optional_number: ?i32 = null;
    _ = &optional_number;
    const number = optional_number.?;
    std.debug.print("value: {}\n", .{number});
}
```

Shell

```
$ zig build-exe runtime_unwrap_null.zig
$ ./runtime_unwrap_null
thread 212451 panic: attempt to use null value
/home/andy/src/zig/doc/langref/runtime_unwrap_null.zig:6:35: 0x10de886 in main (runtime_unwrap_
    const number = optional_number.?;
```

```
                                  ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de352 in posixCallMainAndExit (runtime_unwrap_
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf2d in _start (runtime_unwrap_null)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

One way to avoid this crash is to test for null instead of assuming non-null, with the `if` expression:

testing_null_with_if.zig

```
const print = @import("std").debug.print;
pub fn main() void {
    const optional_number: ?i32 = null;

    if (optional_number) |number| {
        print("got number: {}\n", .{number});
    } else {
        print("it's null\n", .{});
    }
}
```

Shell

```
$ zig build-exe testing_null_with_if.zig
$ ./testing_null_with_if
it's null
```

See also:

- Optionals

Attempt to Unwrap Error

At compile-time:

test_comptime_unwrap_error.zig

```
comptime {
    const number = getNumberOrFail() catch unreachable;
    _ = number;
}

fn getNumberOrFail() !i32 {
    return error.UnableToReturnNumber;
}
```

Shell

```
$ zig test test_comptime_unwrap_error.zig
doc/langref/test_comptime_unwrap_error.zig:2:44: error: caught unexpected error 'UnableToReturn
```

```
    const number = getNumberOrFail() catch unreachable;
                                           ^~~~~~~~~~~
doc/langref/test_comptime_unwrap_error.zig:7:18: note: error returned here
    return error.UnableToReturnNumber;
                 ^~~~~~~~~~~~~~~~~~~~
```

At runtime:

runtime_unwrap_error.zig

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

pub fn main() void {
    const number = getNumberOrFail() catch unreachable;
    std.debug.print("value: {}\n", .{number});
}

fn getNumberOrFail() !i32 {
    return error.UnableToReturnNumber;
}
```

Shell

```
$ zig build-exe runtime_unwrap_error.zig
$ ./runtime_unwrap_error
thread 214943 panic: attempt to unwrap error: UnableToReturnNumber
/home/andy/src/zig/doc/langref/runtime_unwrap_error.zig:9:5: 0x10df16f in getNumberOrFail (runt
    return error.UnableToReturnNumber;
    ^
/home/andy/src/zig/doc/langref/runtime_unwrap_error.zig:4:44: 0x10de901 in main (runtime_unwrap
    const number = getNumberOrFail() catch unreachable;
                                           ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de3c2 in posixCallMainAndExit (runtime_unwrap_
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf9d in _start (runtime_unwrap_error)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

One way to avoid this crash is to test for an error instead of assuming a successful result, with the `if` expression:

testing_error_with_if.zig

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

pub fn main() void {
    const result = getNumberOrFail();

    if (result) |number| {
        print("got number: {}\n", .{number});
    } else |err| {
        print("got error: {s}\n", .{@errorName(err)});
```

```
    }
}

fn getNumberOrFail() !i32 {
    return error.UnableToReturnNumber;
}
```

Shell

```
$ zig build-exe testing_error_with_if.zig
$ ./testing_error_with_if
got error: UnableToReturnNumber
```

See also:

- Errors

Invalid Error Code

At compile-time:

test_comptime_invalid_error_code.zig

```
comptime {
    const err = error.AnError;
    const number = @intFromError(err) + 10;
    const invalid_err = @errorFromInt(number);
    _ = invalid_err;
}
```

Shell

```
$ zig test test_comptime_invalid_error_code.zig
doc/langref/test_comptime_invalid_error_code.zig:4:39: error: integer value '11' represents no
    const invalid_err = @errorFromInt(number);
                                      ^~~~~~
```

At runtime:

runtime_invalid_error_code.zig

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

pub fn main() void {
    const err = error.AnError;
    var number = @intFromError(err) + 500;
    _ = &number;
    const invalid_err = @errorFromInt(number);
    std.debug.print("value: {}\n", .{invalid_err});
}
```

Shell

```
$ zig build-exe runtime_invalid_error_code.zig
$ ./runtime_invalid_error_code
```

```
thread 222506 panic: invalid error code
/home/andy/src/zig/doc/langref/runtime_invalid_error_code.zig:7:5: 0x10de916 in main (runtime_i
    const invalid_err = @errorFromInt(number);
    ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de3b2 in posixCallMainAndExit (runtime_invalic
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf8d in _start (runtime_invalid_error_code)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Invalid Enum Cast

At compile-time:

test_comptime_invalid_enum_cast.zig

```zig
const Foo = enum {
    a,
    b,
    c,
};
comptime {
    const a: u2 = 3;
    const b: Foo = @enumFromInt(a);
    _ = b;
}
```

Shell

```
$ zig test test_comptime_invalid_enum_cast.zig
doc/langref/test_comptime_invalid_enum_cast.zig:8:20: error: enum 'test_comptime_invalid_enum_c
    const b: Foo = @enumFromInt(a);
                   ^~~~~~~~~~~~~~~
doc/langref/test_comptime_invalid_enum_cast.zig:1:13: note: enum declared here
const Foo = enum {
            ^~~~
```

At runtime:

runtime_invalid_enum_cast.zig

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

const Foo = enum {
    a,
    b,
    c,
};

pub fn main() void {
    var a: u2 = 3;
    _ = &a;
```

```
    const b: Foo = @enumFromInt(a);
    std.debug.print("value: {s}\n", .{@tagName(b)});
}
```

Shell

```
$ zig build-exe runtime_invalid_enum_cast.zig
$ ./runtime_invalid_enum_cast
thread 220434 panic: invalid enum value
/home/andy/src/zig/doc/langref/runtime_invalid_enum_cast.zig:12:20: 0x10de8da in main (runtime_
    const b: Foo = @enumFromInt(a);
                   ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de3c2 in posixCallMainAndExit (runtime_invalic
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddf9d in _start (runtime_invalid_enum_cast)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Invalid Error Set Cast

At compile-time:

test_comptime_invalid_error_set_cast.zig

```
const Set1 = error{
    A,
    B,
};
const Set2 = error{
    A,
    C,
};
comptime {
    _ = @as(Set2, @errorCast(Set1.B));
}
```

Shell

```
$ zig test test_comptime_invalid_error_set_cast.zig
doc/langref/test_comptime_invalid_error_set_cast.zig:10:19: error: 'error.B' not a member of e1
    _ = @as(Set2, @errorCast(Set1.B));
                  ^~~~~~~~~~~~~~~~~~
```

At runtime:

runtime_invalid_error_set_cast.zig

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

const Set1 = error{
    A,
    B,
```

```
};
const Set2 = error{
    A,
    C,
};
pub fn main() void {
    foo(Set1.B);
}
fn foo(set1: Set1) void {
    const x: Set2 = @errorCast(set1);
    std.debug.print("value: {}\n", .{x});
}
```

Shell

```
$ zig build-exe runtime_invalid_error_set_cast.zig
$ ./runtime_invalid_error_set_cast
thread 217676 panic: invalid error code
/home/andy/src/zig/doc/langref/runtime_invalid_error_set_cast.zig:15:21: 0x10df1cd in foo (runt
    const x: Set2 = @errorCast(set1);
                    ^
/home/andy/src/zig/doc/langref/runtime_invalid_error_set_cast.zig:12:8: 0x10de8fc in main (runt
    foo(Set1.B);
       ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de3f2 in posixCallMainAndExit (runtime_invalic
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10ddfcd in _start (runtime_invalid_error_set_cast
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Incorrect Pointer Alignment

At compile-time:

test_comptime_incorrect_pointer_alignment.zig

```
comptime {
    const ptr: *align(1) i32 = @ptrFromInt(0x1);
    const aligned: *align(4) i32 = @alignCast(ptr);
    _ = aligned;
}
```

Shell

```
$ zig test test_comptime_incorrect_pointer_alignment.zig
doc/langref/test_comptime_incorrect_pointer_alignment.zig:3:47: error: pointer address 0x1 is r
    const aligned: *align(4) i32 = @alignCast(ptr);
                                              ^~~
```

At runtime:

runtime_incorrect_pointer_alignment.zig

```
const mem = @import("std").mem;
pub fn main() !void {
    var array align(4) = [_]u32{ 0x11111111, 0x11111111 };
    const bytes = mem.sliceAsBytes(array[0..]);
    if (foo(bytes) != 0x11111111) return error.Wrong;
}
fn foo(bytes: []u8) u32 {
    const slice4 = bytes[1..5];
    const int_slice = mem.bytesAsSlice(u32, @as([]align(4) u8, @alignCast(slice4)));
    return int_slice[0];
}
```

Shell

```
$ zig build-exe runtime_incorrect_pointer_alignment.zig
$ ./runtime_incorrect_pointer_alignment
thread 222554 panic: incorrect alignment
/home/andy/src/zig/doc/langref/runtime_incorrect_pointer_alignment.zig:9:64: 0x10de7c2 in foo (
    const int_slice = mem.bytesAsSlice(u32, @as([]align(4) u8, @alignCast(slice4)));
                                                               ^
/home/andy/src/zig/doc/langref/runtime_incorrect_pointer_alignment.zig:5:12: 0x10de6bf in main
    if (foo(bytes) != 0x11111111) return error.Wrong;
           ^
/home/andy/src/zig/lib/std/start.zig:656:37: 0x10de5aa in posixCallMainAndExit (runtime_incorre
            const result = root.main() catch |err| {
                                    ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10de15d in _start (runtime_incorrect_pointer_alig
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Wrong Union Field Access

At compile-time:

test_comptime_wrong_union_field_access.zig

```
comptime {
    var f = Foo{ .int = 42 };
    f.float = 12.34;
}

const Foo = union {
    float: f32,
    int: u32,
};
```

Shell

```
$ zig test test_comptime_wrong_union_field_access.zig
doc/langref/test_comptime_wrong_union_field_access.zig:3:6: error: access of union field 'float
    f.float = 12.34;
    ~^~~~~~
doc/langref/test_comptime_wrong_union_field_access.zig:6:13: note: union declared here
```

```
const Foo = union {
            ^~~~~
```

At runtime:

runtime_wrong_union_field_access.zig

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

const Foo = union {
    float: f32,
    int: u32,
};

pub fn main() void {
    var f = Foo{ .int = 42 };
    bar(&f);
}

fn bar(f: *Foo) void {
    f.float = 12.34;
    std.debug.print("value: {}\n", .{f.float});
}
```

Shell

```
$ zig build-exe runtime_wrong_union_field_access.zig
$ ./runtime_wrong_union_field_access
thread 215740 panic: access of union field 'float' while field 'int' is active
/home/andy/src/zig/doc/langref/runtime_wrong_union_field_access.zig:14:6: 0x10e49e8 in bar (rur
    f.float = 12.34;
     ^
/home/andy/src/zig/doc/langref/runtime_wrong_union_field_access.zig:10:8: 0x10e413c in main (ru
    bar(&f);
       ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10e3c32 in posixCallMainAndExit (runtime_wrong_u
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10e380d in _start (runtime_wrong_union_field_acce
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

This safety is not available for `extern` or `packed` unions.

To change the active field of a union, assign the entire union, like this:

change_active_union_field.zig

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

const Foo = union {
    float: f32,
    int: u32,
```

```
};

pub fn main() void {
    var f = Foo{ .int = 42 };
    bar(&f);
}

fn bar(f: *Foo) void {
    f.* = Foo{ .float = 12.34 };
    std.debug.print("value: {}\n", .{f.float});
}
```

Shell

```
$ zig build-exe change_active_union_field.zig
$ ./change_active_union_field
value: 1.234e1
```

To change the active field of a union when a meaningful value for the field is not known, use undefined, like this:

undefined_active_union_field.zig

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

const Foo = union {
    float: f32,
    int: u32,
};

pub fn main() void {
    var f = Foo{ .int = 42 };
    f = Foo{ .float = undefined };
    bar(&f);
    std.debug.print("value: {}\n", .{f.float});
}

fn bar(f: *Foo) void {
    f.float = 12.34;
}
```

Shell

```
$ zig build-exe undefined_active_union_field.zig
$ ./undefined_active_union_field
value: 1.234e1
```

See also:

- union
- extern union

Out of Bounds Float to Integer Cast

This happens when casting a float to an integer where the float has a value outside the integer type's range.

At compile-time:

test_comptime_out_of_bounds_float_to_integer_cast.zig

```
comptime {
    const float: f32 = 4294967296;
    const int: i32 = @intFromFloat(float);
    _ = int;
}
```

Shell

```
$ zig test test_comptime_out_of_bounds_float_to_integer_cast.zig
doc/langref/test_comptime_out_of_bounds_float_to_integer_cast.zig:3:36: error: float value '429
    const int: i32 = @intFromFloat(float);
                                   ^~~~~
```

At runtime:

runtime_out_of_bounds_float_to_integer_cast.zig

```
pub fn main() void {
    var float: f32 = 4294967296; // runtime-known
    _ = &float;
    const int: i32 = @intFromFloat(float);
    _ = int;
}
```

Shell

```
$ zig build-exe runtime_out_of_bounds_float_to_integer_cast.zig
$ ./runtime_out_of_bounds_float_to_integer_cast
thread 215812 panic: integer part of floating point value out of bounds
/home/andy/src/zig/doc/langref/runtime_out_of_bounds_float_to_integer_cast.zig:4:22: 0x10de7a9
    const int: i32 = @intFromFloat(float);
                     ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de252 in posixCallMainAndExit (runtime_out_of_
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10dde2d in _start (runtime_out_of_bounds_float_to
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Pointer Cast Invalid Null

This happens when casting a pointer with the address 0 to a pointer which may not have the address 0. For example, C Pointers, Optional Pointers, and allowzero pointers allow address zero, but normal Pointers do not.

At compile-time:

test_comptime_invalid_null_pointer_cast.zig

```
comptime {
    const opt_ptr: ?*i32 = null;
    const ptr: *i32 = @ptrCast(opt_ptr);
    _ = ptr;
}
```

Shell

```
$ zig test test_comptime_invalid_null_pointer_cast.zig
doc/langref/test_comptime_invalid_null_pointer_cast.zig:3:32: error: null pointer casted to typ
    const ptr: *i32 = @ptrCast(opt_ptr);
                               ^~~~~~~
```

At runtime:

runtime_invalid_null_pointer_cast.zig

```
pub fn main() void {
    var opt_ptr: ?*i32 = null;
    _ = &opt_ptr;
    const ptr: *i32 = @ptrCast(opt_ptr);
    _ = ptr;
}
```

Shell

```
$ zig build-exe runtime_invalid_null_pointer_cast.zig
$ ./runtime_invalid_null_pointer_cast
thread 214922 panic: cast causes pointer to be null
/home/andy/src/zig/doc/langref/runtime_invalid_null_pointer_cast.zig:4:23: 0x10de75c in main (r
    const ptr: *i32 = @ptrCast(opt_ptr);
                      ^
/home/andy/src/zig/lib/std/start.zig:647:22: 0x10de232 in posixCallMainAndExit (runtime_invalic
            root.main();
                     ^
/home/andy/src/zig/lib/std/start.zig:271:5: 0x10dde0d in _start (runtime_invalid_null_pointer_c
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
(process terminated by signal)
```

Memory

The Zig language performs no memory management on behalf of the programmer. This is why Zig has no runtime, and why Zig code works seamlessly in so many environments, including real-time software, operating system kernels, embedded devices, and low latency servers. As a consequence, Zig programmers must always be able to answer the question:

Where are the bytes?

Like Zig, the C programming language has manual memory management. However, unlike Zig, C has a default allocator - `malloc`, `realloc`, and `free`. When linking against libc, Zig exposes this allocator with `std.heap.c_allocator`. However, by convention, there is no default allocator in Zig. Instead, functions which need to allocate accept an `Allocator` parameter. Likewise, data structures such as `std.ArrayList` accept an `Allocator` parameter in their initialization functions:

test_allocator.zig

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

test "using an allocator" {
    var buffer: [100]u8 = undefined;
    var fba = std.heap.FixedBufferAllocator.init(&buffer);
    const allocator = fba.allocator();
    const result = try concat(allocator, "foo", "bar");
    try expect(std.mem.eql(u8, "foobar", result));
}

fn concat(allocator: Allocator, a: []const u8, b: []const u8) ![]u8 {
    const result = try allocator.alloc(u8, a.len + b.len);
    @memcpy(result[0..a.len], a);
    @memcpy(result[a.len..], b);
    return result;
}
```

Shell

```
$ zig test test_allocator.zig
1/1 test_allocator.test.using an allocator...OK
All 1 tests passed.
```

In the above example, 100 bytes of stack memory are used to initialize a `FixedBufferAllocator`, which is then passed to a function. As a convenience there is a global `FixedBufferAllocator` available for quick tests at `std.testing.allocator`, which will also perform basic leak detection.

Zig has a general purpose allocator available to be imported with `std.heap.GeneralPurposeAllocator`. However, it is still recommended to follow the Choosing an Allocator guide.

Choosing an Allocator

What allocator to use depends on a number of factors. Here is a flow chart to help you decide:

1. Are you making a library? In this case, best to accept an `Allocator` as a parameter and allow your library's users to decide what allocator to use.
2. Are you linking libc? In this case, `std.heap.c_allocator` is likely the right choice, at least for

your main allocator.

3. Need to use the same allocator in multiple threads? Use one of your choice wrapped around `std.heap.ThreadSafeAllocator`
4. Is the maximum number of bytes that you will need bounded by a number known at comptime? In this case, use `std.heap.FixedBufferAllocator`.
5. Is your program a command line application which runs from start to end without any fundamental cyclical pattern (such as a video game main loop, or a web server request handler), such that it would make sense to free everything at once at the end? In this case, it is recommended to follow this pattern:

   cli_allocation.zig

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

pub fn main() !void {
    var arena = std.heap.ArenaAllocator.init(std.heap.page_allocator);
    defer arena.deinit();

    const allocator = arena.allocator();

    const ptr = try allocator.create(i32);
    std.debug.print("ptr={*}\n", .{ptr});
}
```

   Shell

```
$ zig build-exe cli_allocation.zig
$ ./cli_allocation
ptr=i32@7f23d195f010
```

   When using this kind of allocator, there is no need to free anything manually. Everything gets freed at once with the call to `arena.deinit()`.
6. Are the allocations part of a cyclical pattern such as a video game main loop, or a web server request handler? If the allocations can all be freed at once, at the end of the cycle, for example once the video game frame has been fully rendered, or the web server request has been served, then `std.heap.ArenaAllocator` is a great candidate. As demonstrated in the previous bullet point, this allows you to free entire arenas at once. Note also that if an upper bound of memory can be established, then `std.heap.FixedBufferAllocator` can be used as a further optimization.
7. Are you writing a test, and you want to make sure `error.OutOfMemory` is handled correctly? In this case, use `std.testing.FailingAllocator`.
8. Are you writing a test? In this case, use `std.testing.allocator`.
9. Finally, if none of the above apply, you need a general purpose allocator. Zig's general purpose allocator is available as a function that takes a comptime struct of configuration options and returns a type. Generally, you will set up one `std.heap.GeneralPurposeAllocator` in your main function, and then pass it or sub-allocators around to various parts of your application.

10. You can also consider Implementing an Allocator.

Where are the bytes?

String literals such as `"hello"` are in the global constant data section. This is why it is an error to pass a string literal to a mutable slice, like this:

test_string_literal_to_slice.zig

```
fn foo(s: []u8) void {
    _ = s;
}

test "string literal to mutable slice" {
    foo("hello");
}
```

Shell

```
$ zig test test_string_literal_to_slice.zig
doc/langref/test_string_literal_to_slice.zig:6:9: error: expected type '[]u8', found '*const [5
    foo("hello");
        ^~~~~~~
doc/langref/test_string_literal_to_slice.zig:6:9: note: cast discards const qualifier
doc/langref/test_string_literal_to_slice.zig:1:11: note: parameter type declared here
fn foo(s: []u8) void {
          ^~~~
```

However if you make the slice constant, then it works:

test_string_literal_to_const_slice.zig

```
fn foo(s: []const u8) void {
    _ = s;
}

test "string literal to constant slice" {
    foo("hello");
}
```

Shell

```
$ zig test test_string_literal_to_const_slice.zig
1/1 test_string_literal_to_const_slice.test.string literal to constant slice...OK
All 1 tests passed.
```

Just like string literals, `const` declarations, when the value is known at comptime, are stored in the global constant data section. Also Compile Time Variables are stored in the global constant data section.

`var` declarations inside functions are stored in the function's stack frame. Once a function returns, any Pointers to variables in the function's stack frame become invalid references, and dereferencing them becomes unchecked Illegal Behavior.

`var` declarations at the top level or in struct declarations are stored in the global data section.

The location of memory allocated with `allocator.alloc` or `allocator.create` is determined by the allocator's implementation.

TODO: thread local variables

Implementing an Allocator

Zig programmers can implement their own allocators by fulfilling the Allocator interface. In order to do this one must read carefully the documentation comments in std/mem.zig and then supply a `allocFn` and a `resizeFn`.

There are many example allocators to look at for inspiration. Look at std/heap.zig and `std.heap.GeneralPurposeAllocator`.

Heap Allocation Failure

Many programming languages choose to handle the possibility of heap allocation failure by unconditionally crashing. By convention, Zig programmers do not consider this to be a satisfactory solution. Instead, `error.OutOfMemory` represents heap allocation failure, and Zig libraries return this error code whenever heap allocation failure prevented an operation from completing successfully.

Some have argued that because some operating systems such as Linux have memory overcommit enabled by default, it is pointless to handle heap allocation failure. There are many problems with this reasoning:

- Only some operating systems have an overcommit feature.
  - Linux has it enabled by default, but it is configurable.
  - Windows does not overcommit.
  - Embedded systems do not have overcommit.
  - Hobby operating systems may or may not have overcommit.
- For real-time systems, not only is there no overcommit, but typically the maximum amount of memory per application is determined ahead of time.
- When writing a library, one of the main goals is code reuse. By making code handle allocation failure correctly, a library becomes eligible to be reused in more contexts.
- Although some software has grown to depend on overcommit being enabled, its existence is the source of countless user experience disasters. When a system with overcommit enabled, such as Linux on default settings, comes close to memory exhaustion, the system

locks up and becomes unusable. At this point, the OOM Killer selects an application to kill based on heuristics. This non-deterministic decision often results in an important process being killed, and often fails to return the system back to working order.

Recursion

Recursion is a fundamental tool in modeling software. However it has an often-overlooked problem: unbounded memory allocation.

Recursion is an area of active experimentation in Zig and so the documentation here is not final. You can read a summary of recursion status in the 0.3.0 release notes.

The short summary is that currently recursion works normally as you would expect. Although Zig code is not yet protected from stack overflow, it is planned that a future version of Zig will provide such protection, with some degree of cooperation from Zig code required.

Lifetime and Ownership

It is the Zig programmer's responsibility to ensure that a pointer is not accessed when the memory pointed to is no longer available. Note that a slice is a form of pointer, in that it references other memory.

In order to prevent bugs, there are some helpful conventions to follow when dealing with pointers. In general, when a function returns a pointer, the documentation for the function should explain who "owns" the pointer. This concept helps the programmer decide when it is appropriate, if ever, to free the pointer.

For example, the function's documentation may say "caller owns the returned memory", in which case the code that calls the function must have a plan for when to free that memory. Probably in this situation, the function will accept an `Allocator` parameter.

Sometimes the lifetime of a pointer may be more complicated. For example, the `std.ArrayList(T).items` slice has a lifetime that remains valid until the next time the list is resized, such as by appending new elements.

The API documentation for functions and data structures should take great care to explain the ownership and lifetime semantics of pointers. Ownership determines whose responsibility it is to free the memory referenced by the pointer, and lifetime determines the point at which the memory becomes inaccessible (lest Illegal Behavior occur).

Compile Variables

Compile variables are accessible by importing the `"builtin"` package, which the compiler makes available to every Zig source file. It contains compile-time constants such as the current target, endianness, and release mode.

compile_variables.zig

```
const builtin = @import("builtin");
const separator = if (builtin.os.tag == .windows) '\\' else '/';
```

Example of what is imported with `@import("builtin")`:

@import("builtin")

```
const std = @import("std");
/// Zig version. When writing code that supports multiple versions of Zig, prefer
/// feature detection (i.e. with `@hasDecl` or `@hasField`) over version checks.
pub const zig_version = std.SemanticVersion.parse(zig_version_string) catch unreachable;
pub const zig_version_string = "0.14.0-dev.3451+d8d2aa9af";
pub const zig_backend = std.builtin.CompilerBackend.stage2_llvm;

pub const output_mode: std.builtin.OutputMode = .Exe;
pub const link_mode: std.builtin.LinkMode = .static;
pub const unwind_tables: std.builtin.UnwindTables = .@"async";
pub const is_test = false;
pub const single_threaded = false;
pub const abi: std.Target.Abi = .gnu;
pub const cpu: std.Target.Cpu = .{
    .arch = .x86_64,
    .model = &std.Target.x86.cpu.znver4,
    .features = std.Target.x86.featureSet(&.{
        .@"64bit",
        .adx,
        .aes,
        .allow_light_256_bit,
        .avx,
        .avx2,
        .avx512bf16,
        .avx512bitalg,
        .avx512bw,
        .avx512cd,
        .avx512dq,
        .avx512f,
        .avx512ifma,
        .avx512vbmi,
        .avx512vbmi2,
        .avx512vl,
        .avx512vnni,
        .avx512vpopcntdq,
        .bmi,
        .bmi2,
        .branchfusion,
        .clflushopt,
        .clwb,
        .clzero,
        .cmov,
        .crc32,
```

```
        .cx16,
        .cx8,
        .evex512,
        .f16c,
        .fast_15bytenop,
        .fast_bextr,
        .fast_dpwssd,
        .fast_imm16,
        .fast_lzcnt,
        .fast_movbe,
        .fast_scalar_fsqrt,
        .fast_scalar_shift_masks,
        .fast_variable_perlane_shuffle,
        .fast_vector_fsqrt,
        .fma,
        .fsgsbase,
        .fsrm,
        .fxsr,
        .gfni,
        .idivq_to_divl,
        .invpcid,
        .lzcnt,
        .macrofusion,
        .mmx,
        .movbe,
        .mwaitx,
        .nopl,
        .pclmul,
        .pku,
        .popcnt,
        .prfchw,
        .rdpid,
        .rdpru,
        .rdrnd,
        .rdseed,
        .sahf,
        .sbb_dep_breaking,
        .sha,
        .shstk,
        .slow_shld,
        .smap,
        .smep,
        .sse,
        .sse2,
        .sse3,
        .sse4_1,
        .sse4_2,
        .sse4a,
        .ssse3,
        .vaes,
        .vpclmulqdq,
        .vzeroupper,
        .wbnoinvd,
        .x87,
        .xsave,
        .xsavec,
        .xsaveopt,
```

```
        .xsaves,
    }),
};
pub const os: std.Target.Os = .{
    .tag = .linux,
    .version_range = .{ .linux = .{
        .range = .{
            .min = .{
                .major = 6,
                .minor = 13,
                .patch = 2,
            },
            .max = .{
                .major = 6,
                .minor = 13,
                .patch = 2,
            },
        },
        .glibc = .{
            .major = 2,
            .minor = 39,
            .patch = 0,
        },
        .android = 14,
    }},
};
pub const target: std.Target = .{
    .cpu = cpu,
    .os = os,
    .abi = abi,
    .ofmt = object_format,
    .dynamic_linker = .init("/nix/store/nqb2ns2d1lahnd5ncwmn6k84qfd7vx2k-glibc-2.40-36/lib/ld-l
};
pub const object_format: std.Target.ObjectFormat = .elf;
pub const mode: std.builtin.OptimizeMode = .Debug;
pub const link_libc = false;
pub const link_libcpp = false;
pub const have_error_return_tracing = true;
pub const valgrind_support = true;
pub const sanitize_thread = false;
pub const fuzz = false;
pub const position_independent_code = false;
pub const position_independent_executable = false;
pub const strip_debug_info = false;
pub const code_model: std.builtin.CodeModel = .default;
pub const omit_frame_pointer = false;
```

See also:

- Build Mode

Compilation Model

A Zig compilation is separated into *modules*. Each module is a collection of Zig source files, one of which is the module's *root source file*. Each module can *depend* on any number of other modules, forming a directed graph (dependency loops between modules are allowed). If module A depends on module B, then any Zig source file in module A can import the *root source file* of module B using `@import` with the module's name. In essence, a module acts as an alias to import a Zig source file (which might exist in a completely separate part of the filesystem).

A simple Zig program compiled with `zig build-exe` has two key modules: the one containing your code, known as the "main" or "root" module, and the standard library. Your module *depends on* the standard library module under the name "std", which is what allows you to write `@import("std")`! In fact, every single module in a Zig compilation — including the standard library itself — implicitly depends on the standard library module under the name "std".

The "root module" (the one provided by you in the `zig build-exe` example) has a special property. Like the standard library, it is implicitly made available to all modules (including itself), this time under the name "root". So, `@import("root")` will always be equivalent to `@import` of your "main" source file (often, but not necessarily, named `main.zig`).

Source File Structs

Every Zig source file is implicitly a `struct` declaration; you can imagine that the file's contents are literally surrounded by `struct { ... }`. This means that as well as declarations, the top level of a file is permitted to contain fields:

TopLevelFields.zig

```
//! Because this file contains fields, it is a type which is intended to be instantiated, and s
//! is named in TitleCase instead of snake_case by convention.

foo: u32,
bar: u64,

/// `@This()` can be used to refer to this struct type. In files with fields, it is quite comm
/// name the type here, so it can be easily referenced by other declarations in this file.
const TopLevelFields = @This();

pub fn init(val: u32) TopLevelFields {
    return .{
        .foo = val,
        .bar = val * 10,
    };
}
```

Such files can be instantiated just like any other `struct` type. A file's "root struct type" can be referred to within that file using [@This](#This).

File and Declaration Discovery

Zig places importance on the concept of whether any piece of code is *semantically analyzed*; in essence, whether the compiler "looks at" it. What code is analyzed is based on what files and declarations are "discovered" from a certain point. This process of "discovery" is based on a simple set of recursive rules:

- If a call to `@import` is analyzed, the file being imported is analyzed.
- If a type (including a file) is analyzed, all `comptime`, `usingnamespace`, and `export` declarations within it are analyzed.
- If a type (including a file) is analyzed, and the compilation is for a test, and the module the type is within is the root module of the compilation, then all `test` declarations within it are also analyzed.
- If a reference to a named declaration (i.e. a usage of it) is analyzed, the declaration being referenced is analyzed. Declarations are order-independent, so this reference may be above or below the declaration being referenced, or even in another file entirely.

That's it! Those rules define how Zig files and declarations are discovered. All that remains is to understand where this process *starts*.

The answer to that is the root of the standard library: every Zig compilation begins by analyzing the file `lib/std/std.zig`. This file contains a `comptime` declaration which imports `lib/std/start.zig`, and that file in turn uses `@import("root")` to reference the "root module"; so, the file you provide as your main module's root source file is effectively also a root, because the standard library will always reference it.

It is often desirable to make sure that certain declarations — particularly `test` or `export` declarations — are discovered. Based on the above rules, a common strategy for this is to use `@import` within a `comptime` or `test` block:

force_file_discovery.zig

```
comptime {
    // This will ensure that the file 'api.zig' is always discovered (as long as this file is c
    // It is useful if 'api.zig' contains important exported declarations.
    _ = @import("api.zig");

    // We could also have a file which contains declarations we only want to export depending c
    // condition. In that case, we can use an `if` statement here:
    if (builtin.os.tag == .windows) {
        _ = @import("windows_api.zig");
    }
}

test {
    // This will ensure that the file 'tests.zig' is always discovered (as long as this file is
    // if this compilation is a test. It is useful if 'tests.zig' contains tests we want to ens
    _ = @import("tests.zig");

    // We could also have a file which contains tests we only want to run depending on a compti
```

```
    // In that case, we can use an `if` statement here:
    if (builtin.os.tag == .windows) {
        _ = @import("windows_tests.zig");
    }
}

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

Special Root Declarations

Because the root module's root source file is always accessible using `@import("root")`, is is sometimes used by libraries — including the Zig Standard Library — as a place for the program to expose some "global" information to that library. The Zig Standard Library will look for several declarations in this file.

Entry Point

When building an executable, the most important thing to be looked up in this file is the program's *entry point*. Most commonly, this is a function named `main`, which `std.start` will call just after performing important initialization work.

Alternatively, the presence of a declaration named `_start` (for instance, `pub const _start = {};`) will disable the default `std.start` logic, allowing your root source file to export a low-level entry point as needed.

entry_point.zig

```
/// `std.start` imports this file using `@import("root")`, and uses this declaration as the pro
/// user-provided entry point. It can return any of the following types:
/// * `void`
/// * `E!void`, for any error set `E`
/// * `u8`
/// * `E!u8`, for any error set `E`
/// Returning a `void` value from this function will exit with code 0.
/// Returning a `u8` value from this function will exit with the given status code.
/// Returning an error value from this function will print an Error Return Trace and exit with
pub fn main() void {
    std.debug.print("Hello, World!\n", .{});
}

// If uncommented, this declaration would suppress the usual std.start logic, causing
// the `main` declaration above to be ignored.
//pub const _start = {};

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

Shell

```
$ zig build-exe entry_point.zig
$ ./entry_point
Hello, World!
```

If the Zig compilation links libc, the `main` function can optionally be an **`export fn`** which matches the signature of the C `main` function:

libc_export_entry_point.zig

```
pub export fn main(argc: c_int, argv: [*]const [*:0]const u8) c_int {
    const args = argv[0..@intCast(argc)];
    std.debug.print("Hello! argv[0] is '{s}'\n", .{args[0]});
    return 0;
}

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

Shell

```
$ zig build-exe libc_export_entry_point.zig -lc
$ ./libc_export_entry_point
Hello! argv[0] is './libc_export_entry_point'
```

`std.start` may also use other entry point declarations in certain situations, such as `wWinMain` or `EfiMain`. Refer to the `lib/std/start.zig` logic for details of these declarations.

Standard Library Options

The standard library also looks for a declaration in the root module's root source file named `std_options`. If present, this declaration is expected to be a struct of type `std.Options`, and allows the program to customize some standard library functionality, such as the `std.log` implementation.

std_options.zig

```
/// The presence of this declaration allows the program to override certain behaviors of the s
/// For a full list of available options, see the documentation for `std.Options`.
pub const std_options: std.Options = .{
    // By default, in safe build modes, the standard library will attach a segfault handler to
    // print a helpful stack trace if a segmentation fault occurs. Here, we can disable this, c
    // it in unsafe build modes.
    .enable_segfault_handler = true,
    // This is the logging function used by `std.log`.
    .logFn = myLogFn,
};

fn myLogFn(
    comptime level: std.log.Level,
    comptime scope: @Type(.enum_literal),
    comptime format: []const u8,
    args: anytype,
) void {
    // We could do anything we want here!
    // ...but actually, let's just call the default implementation.
    std.log.defaultLog(level, scope, format, args);
}
```

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

Panic Handler

The Zig Standard Library looks for a declaration named `panic` in the root module's root source file. If present, it is expected to be a namespace (container type) with declarations providing different panic handlers.

See `std.debug.simple_panic` for a basic implementation of this namespace.

Overriding how the panic handler actually outputs messages, but keeping the formatted safety panics which are enabled by default, can be easily achieved with `std.debug.FullPanic`:

panic_handler.zig

```
pub fn main() void {
    @setRuntimeSafety(true);
    var x: u8 = 255;
    // Let's overflow this integer!
    x += 1;
}

pub const panic = std.debug.FullPanic(myPanic);

fn myPanic(msg: []const u8, first_trace_addr: ?usize) noreturn {
    _ = first_trace_addr;
    std.debug.print("Panic! {s}\n", .{msg});
    std.process.exit(1);
}

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

Shell

```
$ zig build-exe panic_handler.zig
$ ./panic_handler
Panic! integer overflow
```

Zig Build System

The Zig Build System provides a cross-platform, dependency-free way to declare the logic required to build a project. With this system, the logic to build a project is written in a build.zig file, using the Zig Build System API to declare and configure build artifacts and other tasks.

Some examples of tasks the build system can help with:

- Performing tasks in parallel and caching the results.
- Depending on other projects.
- Providing a package for other projects to depend on.

- Creating build artifacts by executing the Zig compiler. This includes building Zig source code as well as C and C++ source code.
- Capturing user-configured options and using those options to configure the build.
- Surfacing build configuration as comptime values by providing a file that can be imported by Zig code.
- Caching build artifacts to avoid unnecessarily repeating steps.
- Executing build artifacts or system-installed tools.
- Running tests and verifying the output of executing a build artifact matches the expected value.
- Running `zig fmt` on a codebase or a subset of it.
- Custom tasks.

To use the build system, run `zig build --help` to see a command-line usage help menu. This will include project-specific options that were declared in the build.zig script.

For the time being, the build system documentation is hosted externally: Build System Documentation

C

Although Zig is independent of C, and, unlike most other languages, does not depend on libc, Zig acknowledges the importance of interacting with existing C code.

There are a few ways that Zig facilitates C interop.

C Type Primitives

These have guaranteed C ABI compatibility and can be used like any other type.

- `c_char`
- `c_short`
- `c_ushort`
- `c_int`
- `c_uint`
- `c_long`
- `c_ulong`
- `c_longlong`
- `c_ulonglong`
- `c_longdouble`

To interop with the C `void` type, use `anyopaque`.

See also:

- Primitive Types

Import from C Header File

The `@cImport` builtin function can be used to directly import symbols from `.h` files:

cImport_builtin.zig

```
const c = @cImport({
    // See https://github.com/ziglang/zig/issues/515
    @cDefine("_NO_CRT_STDIO_INLINE", "1");
    @cInclude("stdio.h");
});
pub fn main() void {
    _ = c.printf("hello\n");
}
```

Shell

```
$ zig build-exe cImport_builtin.zig -lc
$ ./cImport_builtin
hello
```

The `@cImport` function takes an expression as a parameter. This expression is evaluated at compile-time and is used to control preprocessor directives and include multiple `.h` files:

@cImport Expression

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

const c = @cImport({
    @cDefine("NDEBUG", builtin.mode == .ReleaseFast);
    if (something) {
        @cDefine("_GNU_SOURCE", {});
    }
    @cInclude("stdlib.h");
    if (something) {
        @cUndef("_GNU_SOURCE");
    }
    @cInclude("soundio.h");
});
```

See also:

- @cImport
- @cInclude
- @cDefine
- @cUndef
- @import

C Translation CLI

Zig's C translation capability is available as a CLI tool via `zig translate-c`. It requires a single filename as an argument. It may also take a set of optional flags that are forwarded to clang. It writes the translated file to stdout.

Command line flags

- `-I`: Specify a search directory for include files. May be used multiple times. Equivalent to clang's `-I` flag. The current directory is *not* included by default; use `-I.` to include it.
- `-D`: Define a preprocessor macro. Equivalent to clang's `-D` flag.
- `-cflags [flags] --`: Pass arbitrary additional command line flags to clang. Note: the list of flags must end with `--`
- `-target`: The target triple for the translated Zig code. If no target is specified, the current host target will be used.

Using -target and -cflags

Important! When translating C code with `zig translate-c`, you **must** use the same `-target` triple that you will use when compiling the translated code. In addition, you **must** ensure that the `-cflags` used, if any, match the cflags used by code on the target system. Using the incorrect `-target` or `-cflags` could result in clang or Zig parse failures, or subtle ABI incompatibilities when linking with C code.

varytarget.h

```
long FOO = __LONG_MAX__;
```

Shell

```
$ zig translate-c -target thumb-freestanding-gnueabihf varytarget.h|grep FOO
pub export var FOO: c_long = 2147483647;
$ zig translate-c -target x86_64-macos-gnu varytarget.h|grep FOO
pub export var FOO: c_long = 9223372036854775807;
```

varycflags.h

```
enum FOO { BAR };
int do_something(enum FOO foo);
```

Shell

```
$ zig translate-c varycflags.h|grep -B1 do_something
pub const enum_FOO = c_uint;
pub extern fn do_something(foo: enum_FOO) c_int;
$ zig translate-c -cflags -fshort-enums -- varycflags.h|grep -B1 do_something
pub const enum_FOO = u8;
pub extern fn do_something(foo: enum_FOO) c_int;
```

@cImport vs translate-c

`@cImport` and `zig translate-c` use the same underlying C translation functionality, so on a technical level they are equivalent. In practice, `@cImport` is useful as a way to quickly and easily access numeric constants, typedefs, and record types without needing any extra setup. If you need to pass cflags to clang, or if you would like to edit the translated code, it is recommended to use `zig translate-c` and save the results to a file. Common reasons for editing the generated code include: changing `anytype` parameters in function-like macros to more specific types; changing `[*c]T` pointers to `[*]T` or `*T` pointers for improved type safety; and enabling or disabling runtime safety within specific functions.

See also:

- Targets
- C Type Primitives
- Pointers
- C Pointers
- Import from C Header File
- @cInclude
- @cImport
- @setRuntimeSafety

C Translation Caching

The C translation feature (whether used via `zig translate-c` or `@cImport`) integrates with the Zig caching system. Subsequent runs with the same source file, target, and cflags will use the cache instead of repeatedly translating the same code.

To see where the cached files are stored when compiling code that uses `@cImport`, use the `--verbose-cimport` flag:

verbose_cimport_flag.zig

```
const c = @cImport({
    @cDefine("_NO_CRT_STDIO_INLINE", "1");
    @cInclude("stdio.h");
});
pub fn main() void {
    _ = c;
}
```

Shell

```
$ zig build-exe verbose_cimport_flag.zig -lc --verbose-cimport
info(compilation): C import source: /home/andy/src/zig/.zig-cache/o/d5beb8c0b08d3a28e44bf4f197b
info(compilation): C import .d file: /home/andy/src/zig/.zig-cache/o/d5beb8c0b08d3a28e44bf4f197
$ ./verbose_cimport_flag
```

`cimport.h` contains the file to translate (constructed from calls to `@cInclude`, `@cDefine`, and `@cUndef`), `cimport.h.d` is the list of file dependencies, and `cimport.zig` contains the translated output.

See also:

- Import from C Header File
- C Translation CLI
- @cInclude
- @cImport

Translation failures

Some C constructs cannot be translated to Zig - for example, *goto*, structs with bitfields, and token-pasting macros. Zig employs *demotion* to allow translation to continue in the face of non-translatable entities.

Demotion comes in three varieties - opaque, *extern*, and `@compileError`. C structs and unions that cannot be translated correctly will be translated as `opaque{}`. Functions that contain opaque types or code constructs that cannot be translated will be demoted to `extern` declarations. Thus, non-translatable types can still be used as pointers, and non-translatable functions can be called so long as the linker is aware of the compiled function.

`@compileError` is used when top-level definitions (global variables, function prototypes, macros) cannot be translated or demoted. Since Zig uses lazy analysis for top-level declarations, untranslatable entities will not cause a compile error in your code unless you actually use them.

See also:

- opaque
- extern
- @compileError

C Macros

C Translation makes a best-effort attempt to translate function-like macros into equivalent Zig functions. Since C macros operate at the level of lexical tokens, not all C macros can be translated to Zig. Macros that cannot be translated will be demoted to `@compileError`. Note that C code which *uses* macros will be translated without any additional issues (since Zig operates on the pre-processed source with macros expanded). It is merely the macros themselves which may not be translatable to Zig.

Consider the following example:

macro.c

```
#define MAKELOCAL(NAME, INIT) int NAME = INIT
int foo(void) {
   MAKELOCAL(a, 1);
   MAKELOCAL(b, 2);
   return a + b;
}
```

Shell

```
$ zig translate-c macro.c > macro.zig
```

macro.zig

```
pub export fn foo() c_int {
    var a: c_int = 1;
    _ = &a;
    var b: c_int = 2;
    _ = &b;
    return a + b;
}
pub const MAKELOCAL = @compileError("unable to translate C expr: unexpected token .Equal"); //
```

Note that `foo` was translated correctly despite using a non-translatable macro. `MAKELOCAL` was demoted to `@compileError` since it cannot be expressed as a Zig function; this simply means that you cannot directly use `MAKELOCAL` from Zig.

See also:

- @compileError

C Pointers

This type is to be avoided whenever possible. The only valid reason for using a C pointer is in auto-generated code from translating C code.

When importing C header files, it is ambiguous whether pointers should be translated as single-item pointers (`*T`) or many-item pointers (`[*]T`). C pointers are a compromise so that Zig code can utilize translated header files directly.

`[*c]T` - C pointer.

- Supports all the syntax of the other two pointer types (`*T`) and (`[*]T`).

- Coerces to other pointer types, as well as Optional Pointers. When a C pointer is coerced to a non-optional pointer, safety-checked Illegal Behavior occurs if the address is 0.
- Allows address 0. On non-freestanding targets, dereferencing address 0 is safety-checked Illegal Behavior. Optional C pointers introduce another bit to keep track of null, just like `?usize`. Note that creating an optional C pointer is unnecessary as one can use normal Optional Pointers.
- Supports Type Coercion to and from integers.
- Supports comparison with integers.
- Does not support Zig-only pointer attributes such as alignment. Use normal Pointers please!

When a C pointer is pointing to a single struct (not an array), dereference the C pointer to access the struct's fields or member data. That syntax looks like this:

`ptr_to_struct.*.struct_member`

This is comparable to doing `->` in C.

When a C pointer is pointing to an array of structs, the syntax reverts to this:

`ptr_to_struct_array[index].struct_member`

C Variadic Functions

Zig supports extern variadic functions.

test_variadic_function.zig

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

pub extern "c" fn printf(format: [*:0]const u8, ...) c_int;

test "variadic function" {
    try testing.expect(printf("Hello, world!\n") == 14);
    try testing.expect(@typeInfo(@TypeOf(printf)).@"fn".is_var_args);
}
```

Shell

```
$ zig test test_variadic_function.zig -lc
1/1 test_variadic_function.test.variadic function...OK
All 1 tests passed.
Hello, world!
```

Variadic functions can be implemented using @cVaStart, @cVaEnd, @cVaArg and @cVaCopy.

test_defining_variadic_function.zig

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

fn add(count: c_int, ...) callconv(.C) c_int {
    var ap = @cVaStart();
    defer @cVaEnd(&ap);
    var i: usize = 0;
    var sum: c_int = 0;
    while (i < count) : (i += 1) {
        sum += @cVaArg(&ap, c_int);
    }
    return sum;
}

test "defining a variadic function" {
    if (builtin.cpu.arch == .aarch64 and builtin.os.tag != .macos) {
        // https://github.com/ziglang/zig/issues/14096
        return error.SkipZigTest;
    }
    if (builtin.cpu.arch == .x86_64 and builtin.os.tag == .windows) {
        // https://github.com/ziglang/zig/issues/16961
        return error.SkipZigTest;
    }

    try std.testing.expectEqual(@as(c_int, 0), add(0));
    try std.testing.expectEqual(@as(c_int, 1), add(1, @as(c_int, 1)));
    try std.testing.expectEqual(@as(c_int, 3), add(2, @as(c_int, 1), @as(c_int, 2)));
}
```

Shell

```
$ zig test test_defining_variadic_function.zig
1/1 test_defining_variadic_function.test.defining a variadic function...OK
All 1 tests passed.
```

Exporting a C Library

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:

Shell

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

To make a shared library:

Shell

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

Here is an example with the Zig Build System:

test.c

```
// This header is generated by zig from mathtest.zig
#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_c.zig

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

pub fn build(b: *std.Build) void {
    const lib = b.addSharedLibrary(.{
        .name = "mathtest",
        .root_source_file = b.path("mathtest.zig"),
        .version = .{ .major = 1, .minor = 0, .patch = 0 },
    });
    const exe = b.addExecutable(.{
        .name = "test",
    });
    exe.addCSourceFile(.{ .file = b.path("test.c"), .flags = &.{"-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);
}
```

Shell

```
$ zig build test
1379
```

See also:

- export

Mixing Object Files

You can mix Zig object files with any other object files that respect the C ABI. Example:

base64.zig

```
const base64 = @import("std").base64;

export fn decode_base_64(
    dest_ptr: [*]u8,
    dest_len: usize,
    source_ptr: [*]const u8,
    source_len: usize,
) usize {
    const src = source_ptr[0..source_len];
    const dest = dest_ptr[0..dest_len];
    const base64_decoder = base64.standard.Decoder;
    const decoded_size = base64_decoder.calcSizeForSlice(src) catch unreachable;
    base64_decoder.decode(dest[0..decoded_size], src) catch unreachable;
    return decoded_size;
}
```

test.c

```
// This header is generated by zig from base64.zig
#include "base64.h"

#include <string.h>
#include <stdio.h>

int main(int argc, char **argv) {
    const char *encoded = "YWxsIHlvdXIgYmFzZSBhcmUgYmVsb25nIHRvIHVz";
    char buf[200];

    size_t len = decode_base_64(buf, 200, encoded, strlen(encoded));
    buf[len] = 0;
    puts(buf);

    return 0;
}
```

build_object.zig

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

pub fn build(b: *std.Build) void {
    const obj = b.addObject(.{
        .name = "base64",
        .root_source_file = b.path("base64.zig"),
    });

    const exe = b.addExecutable(.{
        .name = "test",
    });
    exe.addCSourceFile(.{ .file = b.path("test.c"), .flags = &.{"-std=c99"} });
    exe.addObject(obj);
    exe.linkSystemLibrary("c");
    b.installArtifact(exe);
}
```

Shell

```
$ zig build
$ ./zig-out/bin/test
all your base are belong to us
```

See also:

- Targets
- Zig Build System

WebAssembly

Zig supports building for WebAssembly out of the box.

Freestanding

For host environments like the web browser and nodejs, build as an executable using the freestanding OS target. Here's an example of running Zig code compiled to WebAssembly with nodejs.

math.zig

```
extern fn print(i32) void;

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

Shell

```
$ zig build-exe math.zig -target wasm32-freestanding -fno-entry --export=add
```

test.js

```
const fs = require('fs');
const source = fs.readFileSync("./math.wasm");
const typedArray = new Uint8Array(source);

WebAssembly.instantiate(typedArray, {
  env: {
    print: (result) => { console.log(`The result is ${result}`); }
  }}).then(result => {
  const add = result.instance.exports.add;
  add(1, 2);
});
```

Shell

```
$ node test.js
The result is 3
```

WASI

Zig's support for WebAssembly System Interface (WASI) is under active development. Example of using the standard library and reading command line arguments:

wasi_args.zig

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

pub fn main() !void {
    var general_purpose_allocator: std.heap.GeneralPurposeAllocator(.{}) = .init;
    const gpa = general_purpose_allocator.allocator();
    const args = try std.process.argsAlloc(gpa);
    defer std.process.argsFree(gpa, args);

    for (args, 0..) |arg, i| {
        std.debug.print("{}: {s}\n", .{ i, arg });
    }
}
```

Shell

```
$ zig build-exe wasi_args.zig -target wasm32-wasi
```

Shell

```
$ wasmtime wasi_args.wasm 123 hello
0: wasi_args.wasm
1: 123
2: hello
```

A more interesting example would be extracting the list of preopens from the runtime. This is now supported in the standard library via `std.fs.wasi.Preopens`:

wasi_preopens.zig

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

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

    var arena_instance = std.heap.ArenaAllocator.init(gpa);
    defer arena_instance.deinit();
    const arena = arena_instance.allocator();

    const preopens = try fs.wasi.preopensAlloc(arena);

    for (preopens.names, 0..) |preopen, i| {
        std.debug.print("{}: {s}\n", .{ i, preopen });
    }
}
```

Shell

```
$ zig build-exe wasi_preopens.zig -target wasm32-wasi
```

Shell

```
$ wasmtime --dir=. wasi_preopens.wasm
0: stdin
1: stdout
2: stderr
3: .
```

Targets

Target refers to the computer that will be used to run an executable. It is composed of the CPU architecture, the set of enabled CPU features, operating system, minimum and maximum operating system version, ABI, and ABI version.

Zig is a general-purpose programming language which means that it is designed to generate optimal code for a large set of targets. The command `zig targets` provides information about all of the targets the compiler is aware of.

When no target option is provided to the compiler, the default choice is to target the **host computer**, meaning that the resulting executable will be *unsuitable for copying to a different computer*. In order to copy an executable to another computer, the compiler needs to know about the target requirements via the `-target` option.

The Zig Standard Library (`@import("std")`) has cross-platform abstractions, making the same source code viable on many targets. Some code is more portable than other code. In general, Zig code is extremely portable compared to other programming languages.

Each platform requires its own implementations to make Zig's cross-platform abstractions work. These implementations are at various degrees of completion. Each tagged release of the compiler comes with release notes that provide the full support table for each target.

Style Guide

These coding conventions are not enforced by the compiler, but they are shipped in this documentation along with the compiler in order to provide a point of reference, should anyone wish to point to an authority on agreed upon Zig coding style.

Avoid Redundancy in Names

Avoid these words in type names:

- Value

- Data
- Context
- Manager
- utils, misc, or somebody's initials

Everything is a value, all types are data, everything is context, all logic manages state. Nothing is communicated by using a word that applies to all types.

Temptation to use "utilities", "miscellaneous", or somebody's initials is a failure to categorize, or more commonly, overcategorization. Such declarations can live at the root of a module that needs them with no namespace needed.

Avoid Redundant Names in Fully-Qualified Namespaces

Every declaration is assigned a **fully qualified namespace** by the compiler, creating a tree structure. Choose names based on the fully-qualified namespace, and avoid redundant name segments.

redundant_fqn.zig

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

pub const json = struct {
    pub const JsonValue = union(enum) {
        number: f64,
        boolean: bool,
        // ...
    };
};

pub fn main() void {
    std.debug.print("{s}\n", .{@typeName(json.JsonValue)});
}
```

Shell

```
$ zig build-exe redundant_fqn.zig
$ ./redundant_fqn
redundant_fqn.json.JsonValue
```

In this example, "json" is repeated in the fully-qualified namespace. The solution is to delete `Json` from `JsonValue`. In this example we have an empty struct named `json` but remember that files also act as part of the fully-qualified namespace.

This example is an exception to the rule specified in Avoid Redundancy in Names. The meaning of the type has been reduced to its core: it is a json value. The name cannot be any more specific without being incorrect.

Whitespace

- 4 space indentation
- Open braces on same line, unless you need to wrap.
- If a list of things is longer than 2, put each item on its own line and exercise the ability to put an extra comma at the end.
- Line length: aim for 100; use common sense.

Names

Roughly speaking: `camelCaseFunctionName`, `TitleCaseTypeName`, `snake_case_variable_name`. More precisely:

- If `x` is a `type` then `x` should be `TitleCase`, unless it is a `struct` with 0 fields and is never meant to be instantiated, in which case it is considered to be a "namespace" and uses `snake_case`.
- If `x` is callable, and `x`'s return type is `type`, then `x` should be `TitleCase`.
- If `x` is otherwise callable, then `x` should be `camelCase`.
- Otherwise, `x` should be `snake_case`.

Acronyms, initialisms, proper nouns, or any other word that has capitalization rules in written English are subject to naming conventions just like any other word. Even acronyms that are only 2 letters long are subject to these conventions.

File names fall into two categories: types and namespaces. If the file (implicitly a struct) has top level fields, it should be named like any other struct with fields using **`TitleCase`**. Otherwise, it should use **`snake_case`**. Directory names should be **`snake_case`**.

These are general rules of thumb; if it makes sense to do something different, do what makes sense. For example, if there is an established convention such as `ENOENT`, follow the established convention.

Examples

style_example.zig

```zig
const namespace_name = @import("dir_name/file_name.zig");
const TypeName = @import("dir_name/TypeName.zig");
var global_var: i32 = undefined;
const const_name = 42;
const primitive_type_alias = f32;
const string_alias = []u8;

const StructName = struct {
    field: i32,
```

```
};
const StructAlias = StructName;

fn functionName(param_name: TypeName) void {
    var functionPointer = functionName;
    functionPointer();
    functionPointer = otherFunction;
    functionPointer();
}
const functionAlias = functionName;

fn ListTemplateFunction(comptime ChildType: type, comptime fixed_size: usize) type {
    return List(ChildType, fixed_size);
}

fn ShortList(comptime T: type, comptime n: usize) type {
    return struct {
        field_name: [n]T,
        fn methodName() void {}
    };
}

// The word XML loses its casing when used in Zig identifiers.
const xml_document =
    \\<?xml version="1.0" encoding="UTF-8"?>
    \\<document>
    \\</document>
;
const XmlParser = struct {
    field: i32,
};

// The initials BE (Big Endian) are just another word in Zig identifier names.
fn readU32Be() u32 {}
```

See the Zig Standard Library for more examples.

Doc Comment Guidance

- Omit any information that is redundant based on the name of the thing being documented.
- Duplicating information onto multiple similar functions is encouraged because it helps IDEs and other tools provide better help text.
- Use the word **assume** to indicate invariants that cause *unchecked* Illegal Behavior when violated.
- Use the word **assert** to indicate invariants that cause *safety-checked* Illegal Behavior when violated.

Source Encoding

Zig source code is encoded in UTF-8. An invalid UTF-8 byte sequence results in a compile error.

Throughout all zig source code (including in comments), some code points are never allowed:

- Ascii control characters, except for U+000a (LF), U+000d (CR), and U+0009 (HT): U+0000 - U+0008, U+000b - U+000c, U+000e - U+0001f, U+007f.
- Non-Ascii Unicode line endings: U+0085 (NEL), U+2028 (LS), U+2029 (PS).

LF (byte value 0x0a, code point U+000a, `'\n'`) is the line terminator in Zig source code. This byte value terminates every line of zig source code except the last line of the file. It is recommended that non-empty source files end with an empty line, which means the last byte would be 0x0a (LF).

Each LF may be immediately preceded by a single CR (byte value 0x0d, code point U+000d, `'\r'`) to form a Windows style line ending, but this is discouraged. Note that in multiline strings, CRLF sequences will be encoded as LF when compiled into a zig program. A CR in any other context is not allowed.

HT hard tabs (byte value 0x09, code point U+0009, `'\t'`) are interchangeable with SP spaces (byte value 0x20, code point U+0020, `' '`) as a token separator, but use of hard tabs is discouraged. See Grammar.

For compatibility with other tools, the compiler ignores a UTF-8-encoded byte order mark (U+FEFF) if it is the first Unicode code point in the source text. A byte order mark is not allowed anywhere else in the source.

Note that running `zig fmt` on a source file will implement all recommendations mentioned here.

Note that a tool reading Zig source code can make assumptions if the source code is assumed to be correct Zig code. For example, when identifying the ends of lines, a tool can use a naive search such as `/\n/`, or an advanced search such as `/\r\n?|[\n\u0085\u2028\u2029]/`, and in either case line endings will be correctly identified. For another example, when identifying the whitespace before the first token on a line, a tool can either use a naive search such as `/[ \t]/`, or an advanced search such as `/\s/`, and in either case whitespace will be correctly identified.

Keyword Reference

Keyword	Description
`addrspace`	The `addrspace` keyword. TODO add documentation for addrspace

Keyword	Description
`align`	`align` can be used to specify the alignment of a pointer. It can also be used after a variable or function declaration to specify the alignment of pointers to that variable or function. See also Alignment
`allowzero`	The pointer attribute `allowzero` allows a pointer to have address zero. See also allowzero
`and`	The boolean operator `and`. See also Operators
`anyframe`	`anyframe` can be used as a type for variables which hold pointers to function frames. See also Async Functions
`anytype`	Function parameters can be declared with `anytype` in place of the type. The type will be inferred where the function is called. See also Function Parameter Type Inference
`asm`	`asm` begins an inline assembly expression. This allows for directly controlling the machine code generated on compilation. See also Assembly
`async`	`async` can be used before a function call to get a pointer to the function's frame when it suspends. See also Async Functions

Keyword	Description
`await`	`await` can be used to suspend the current function until the frame provided after the `await` completes. `await` copies the value returned from the target function's frame to the caller. See also Async Functions
`break`	`break` can be used with a block label to return a value from the block. It can also be used to exit a loop before iteration completes naturally. See also Blocks, while, for
`callconv`	`callconv` can be used to specify the calling convention in a function type. See also Functions
`catch`	`catch` can be used to evaluate an expression if the expression before it evaluates to an error. The expression after the `catch` can optionally capture the error value. See also catch, Operators
`comptime`	`comptime` before a declaration can be used to label variables or function parameters as known at compile time. It can also be used to guarantee an expression is run at compile time. See also comptime
`const`	`const` declares a variable that can not be modified. Used as a pointer attribute, it denotes the value referenced by the pointer cannot be modified. See also Variables
`continue`	`continue` can be used in a loop to jump back to the beginning of the loop.

Keyword	Description
	See also while, for
`defer`	`defer` will execute an expression when control flow leaves the current block. See also defer
`else`	`else` can be used to provide an alternate branch for `if`, `switch`, `while`, and `for` expressions. If used after an if expression, the else branch will be executed if the test value returns false, null, or an error. If used within a switch expression, the else branch will be executed if the test value matches no other cases. If used after a loop expression, the else branch will be executed if the loop finishes without breaking. See also if, switch, while, for
`enum`	`enum` defines an enum type. See also enum
`errdefer`	`errdefer` will execute an expression when control flow leaves the current block if the function returns an error, the errdefer expression can capture the unwrapped value. See also errdefer
`error`	`error` defines an error type. See also Errors

Keyword	Description
`export`	`export` makes a function or variable externally visible in the generated object file. Exported functions default to the C calling convention. See also Functions
`extern`	`extern` can be used to declare a function or variable that will be resolved at link time, when linking statically or at runtime, when linking dynamically. See also Functions
`fn`	`fn` declares a function. See also Functions
`for`	A `for` expression can be used to iterate over the elements of a slice, array, or tuple. See also for
`if`	An `if` expression can test boolean expressions, optional values, or error unions. For optional values or error unions, the if expression can capture the unwrapped value. See also if
`inline`	`inline` can be used to label a loop expression such that it will be unrolled at compile time. It can also be used to force a function to be inlined at all call sites. See also inline while, inline for, Functions
`linksection`	The `linksection` keyword can be used to specify what section the function or global variable will be put into (e.g. `.text`).

Keyword	Description
`noalias`	The `noalias` keyword. TODO add documentation for noalias
`noinline`	`noinline` disallows function to be inlined in all call sites. See also Functions
`nosuspend`	The `nosuspend` keyword can be used in front of a block, statement or expression, to mark a scope where no suspension points are reached. In particular, inside a `nosuspend` scope: Using the `suspend` keyword results in a compile error. Using `await` on a function frame which hasn't completed yet results in safety-checked Illegal Behavior. Calling an async function may result in safety-checked Illegal Behavior, because it's equivalent to `await async some_async_fn()`, which contains an `await`. Code inside a `nosuspend` scope does not cause the enclosing function to become an async function. See also Async Functions
`opaque`	`opaque` defines an opaque type. See also opaque
`or`	The boolean operator `or`. See also Operators
`orelse`	`orelse` can be used to evaluate an expression if the expression before it evaluates to null.

Keyword	Description
	See also Optionals, Operators
`packed`	The `packed` keyword before a struct definition changes the struct's in-memory layout to the guaranteed `packed` layout. See also packed struct
`pub`	The `pub` in front of a top level declaration makes the declaration available to reference from a different file than the one it is declared in. See also import
`resume`	`resume` will continue execution of a function frame after the point the function was suspended.
`return`	`return` exits a function with a value. See also Functions
`struct`	`struct` defines a struct. See also struct
`suspend`	`suspend` will cause control flow to return to the call site or resumer of the function. `suspend` can also be used before a block within a function, to allow the function access to its frame before control flow returns to the call site.
`switch`	A `switch` expression can be used to test values of a common type. `switch` cases can capture field values of a Tagged union. See also switch
`test`	The `test` keyword can be used to denote a top-level block of code used to make sure behavior meets expectations.

Keyword	Description
	See also Zig Test
`threadlocal`	`threadlocal` can be used to specify a variable as thread-local. See also Thread Local Variables
`try`	`try` evaluates an error union expression. If it is an error, it returns from the current function with the same error. Otherwise, the expression results in the unwrapped value. See also try
`union`	`union` defines a union. See also union
`unreachable`	`unreachable` can be used to assert that control flow will never happen upon a particular location. Depending on the build mode, `unreachable` may emit a panic. Emits a panic in `Debug` and `ReleaseSafe` mode, or when using `zig test`. Does not emit a panic in `ReleaseFast` and `ReleaseSmall` mode. See also unreachable
`usingnamespace`	`usingnamespace` is a top-level declaration that imports all the public declarations of the operand, which must be a struct, union, or enum, into the current scope. See also usingnamespace

Keyword	Description
`var`	`var` declares a variable that may be modified. See also Variables
`volatile`	`volatile` can be used to denote loads or stores of a pointer have side effects. It can also modify an inline assembly expression to denote it has side effects. See also volatile, Assembly
`while`	A `while` expression can be used to repeatedly test a boolean, optional, or error union expression, and cease looping when that expression evaluates to false, null, or an error, respectively. See also while

Appendix

Containers

A *container* in Zig is any syntactical construct that acts as a namespace to hold variable and function declarations. Containers are also type definitions which can be instantiated. Structs, enums, unions, opaques, and even Zig source files themselves are containers.

Although containers (except Zig source files) use curly braces to surround their definition, they should not be confused with blocks or functions. Containers do not contain statements.

Grammar

grammar.y

```
Root <- skip container_doc_comment? ContainerMembers eof

# *** Top level ***
ContainerMembers <- ContainerDeclaration* (ContainerField COMMA)* (ContainerField / ContainerDe

ContainerDeclaration <- TestDecl / ComptimeDecl / doc_comment? KEYWORD_pub? Decl

TestDecl <- KEYWORD_test (STRINGLITERALSINGLE / IDENTIFIER)? Block
```

```
ComptimeDecl <- KEYWORD_comptime Block

Decl
    <- (KEYWORD_export / KEYWORD_extern STRINGLITERALSINGLE? / KEYWORD_inline / KEYWORD_noinlir
     / (KEYWORD_export / KEYWORD_extern STRINGLITERALSINGLE?)? KEYWORD_threadlocal? GlobalVarDe
     / KEYWORD_usingnamespace Expr SEMICOLON

FnProto <- KEYWORD_fn IDENTIFIER? LPAREN ParamDeclList RPAREN ByteAlign? AddrSpace? LinkSectior

VarDeclProto <- (KEYWORD_const / KEYWORD_var) IDENTIFIER (COLON TypeExpr)? ByteAlign? AddrSpace

GlobalVarDecl <- VarDeclProto (EQUAL Expr)? SEMICOLON

ContainerField <- doc_comment? KEYWORD_comptime? !KEYWORD_fn (IDENTIFIER COLON)? TypeExpr ByteA

# *** Block Level ***
Statement
    <- KEYWORD_comptime ComptimeStatement
     / KEYWORD_nosuspend BlockExprStatement
     / KEYWORD_suspend BlockExprStatement
     / KEYWORD_defer BlockExprStatement
     / KEYWORD_errdefer Payload? BlockExprStatement
     / IfStatement
     / LabeledStatement
     / SwitchExpr
     / VarDeclExprStatement

ComptimeStatement
    <- BlockExpr
     / VarDeclExprStatement

IfStatement
    <- IfPrefix BlockExpr ( KEYWORD_else Payload? Statement )?
     / IfPrefix AssignExpr ( SEMICOLON / KEYWORD_else Payload? Statement )

LabeledStatement <- BlockLabel? (Block / LoopStatement)

LoopStatement <- KEYWORD_inline? (ForStatement / WhileStatement)

ForStatement
    <- ForPrefix BlockExpr ( KEYWORD_else Statement )?
     / ForPrefix AssignExpr ( SEMICOLON / KEYWORD_else Statement )

WhileStatement
    <- WhilePrefix BlockExpr ( KEYWORD_else Payload? Statement )?
     / WhilePrefix AssignExpr ( SEMICOLON / KEYWORD_else Payload? Statement )

BlockExprStatement
    <- BlockExpr
     / AssignExpr SEMICOLON

BlockExpr <- BlockLabel? Block

# An expression, assignment, or any destructure, as a statement.
VarDeclExprStatement
    <- VarDeclProto (COMMA (VarDeclProto / Expr))* EQUAL Expr SEMICOLON
     / Expr (AssignOp Expr / (COMMA (VarDeclProto / Expr))+ EQUAL Expr)? SEMICOLON
```

```
# *** Expression Level ***

# An assignment or a destructure whose LHS are all lvalue expressions.
AssignExpr <- Expr (AssignOp Expr / (COMMA Expr)+ EQUAL Expr)?

SingleAssignExpr <- Expr (AssignOp Expr)?

Expr <- BoolOrExpr

BoolOrExpr <- BoolAndExpr (KEYWORD_or BoolAndExpr)*

BoolAndExpr <- CompareExpr (KEYWORD_and CompareExpr)*

CompareExpr <- BitwiseExpr (CompareOp BitwiseExpr)?

BitwiseExpr <- BitShiftExpr (BitwiseOp BitShiftExpr)*

BitShiftExpr <- AdditionExpr (BitShiftOp AdditionExpr)*

AdditionExpr <- MultiplyExpr (AdditionOp MultiplyExpr)*

MultiplyExpr <- PrefixExpr (MultiplyOp PrefixExpr)*

PrefixExpr <- PrefixOp* PrimaryExpr

PrimaryExpr
    <- AsmExpr
     / IfExpr
     / KEYWORD_break BreakLabel? Expr?
     / KEYWORD_comptime Expr
     / KEYWORD_nosuspend Expr
     / KEYWORD_continue BreakLabel?
     / KEYWORD_resume Expr
     / KEYWORD_return Expr?
     / BlockLabel? LoopExpr
     / Block
     / CurlySuffixExpr

IfExpr <- IfPrefix Expr (KEYWORD_else Payload? Expr)?

Block <- LBRACE Statement* RBRACE

LoopExpr <- KEYWORD_inline? (ForExpr / WhileExpr)

ForExpr <- ForPrefix Expr (KEYWORD_else Expr)?

WhileExpr <- WhilePrefix Expr (KEYWORD_else Payload? Expr)?

CurlySuffixExpr <- TypeExpr InitList?

InitList
    <- LBRACE FieldInit (COMMA FieldInit)* COMMA? RBRACE
     / LBRACE Expr (COMMA Expr)* COMMA? RBRACE
     / LBRACE RBRACE

TypeExpr <- PrefixTypeOp* ErrorUnionExpr
```

```
ErrorUnionExpr <- SuffixExpr (EXCLAMATIONMARK TypeExpr)?

SuffixExpr
    <- KEYWORD_async PrimaryTypeExpr SuffixOp* FnCallArguments
     / PrimaryTypeExpr (SuffixOp / FnCallArguments)*

PrimaryTypeExpr
    <- BUILTINIDENTIFIER FnCallArguments
     / CHAR_LITERAL
     / ContainerDecl
     / DOT IDENTIFIER
     / DOT InitList
     / ErrorSetDecl
     / FLOAT
     / FnProto
     / GroupedExpr
     / LabeledTypeExpr
     / IDENTIFIER
     / IfTypeExpr
     / INTEGER
     / KEYWORD_comptime TypeExpr
     / KEYWORD_error DOT IDENTIFIER
     / KEYWORD_anyframe
     / KEYWORD_unreachable
     / STRINGLITERAL
     / SwitchExpr

ContainerDecl <- (KEYWORD_extern / KEYWORD_packed)? ContainerDeclAuto

ErrorSetDecl <- KEYWORD_error LBRACE IdentifierList RBRACE

GroupedExpr <- LPAREN Expr RPAREN

IfTypeExpr <- IfPrefix TypeExpr (KEYWORD_else Payload? TypeExpr)?

LabeledTypeExpr
    <- BlockLabel Block
     / BlockLabel? LoopTypeExpr

LoopTypeExpr <- KEYWORD_inline? (ForTypeExpr / WhileTypeExpr)

ForTypeExpr <- ForPrefix TypeExpr (KEYWORD_else TypeExpr)?

WhileTypeExpr <- WhilePrefix TypeExpr (KEYWORD_else Payload? TypeExpr)?

SwitchExpr <- KEYWORD_switch LPAREN Expr RPAREN LBRACE SwitchProngList RBRACE

# *** Assembly ***
AsmExpr <- KEYWORD_asm KEYWORD_volatile? LPAREN Expr AsmOutput? RPAREN

AsmOutput <- COLON AsmOutputList AsmInput?

AsmOutputItem <- LBRACKET IDENTIFIER RBRACKET STRINGLITERAL LPAREN (MINUSRARROW TypeExpr / IDEN

AsmInput <- COLON AsmInputList AsmClobbers?
```

```
AsmInputItem <- LBRACKET IDENTIFIER RBRACKET STRINGLITERAL LPAREN Expr RPAREN

AsmClobbers <- COLON StringList

# *** Helper grammar ***
BreakLabel <- COLON IDENTIFIER

BlockLabel <- IDENTIFIER COLON

FieldInit <- DOT IDENTIFIER EQUAL Expr

WhileContinueExpr <- COLON LPAREN AssignExpr RPAREN

LinkSection <- KEYWORD_linksection LPAREN Expr RPAREN

AddrSpace <- KEYWORD_addrspace LPAREN Expr RPAREN

# Fn specific
CallConv <- KEYWORD_callconv LPAREN Expr RPAREN

ParamDecl
    <- doc_comment? (KEYWORD_noalias / KEYWORD_comptime)? (IDENTIFIER COLON)? ParamType
     / DOT3

ParamType
    <- KEYWORD_anytype
     / TypeExpr

# Control flow prefixes
IfPrefix <- KEYWORD_if LPAREN Expr RPAREN PtrPayload?

WhilePrefix <- KEYWORD_while LPAREN Expr RPAREN PtrPayload? WhileContinueExpr?

ForPrefix <- KEYWORD_for LPAREN ForArgumentsList RPAREN PtrListPayload

# Payloads
Payload <- PIPE IDENTIFIER PIPE

PtrPayload <- PIPE ASTERISK? IDENTIFIER PIPE

PtrIndexPayload <- PIPE ASTERISK? IDENTIFIER (COMMA IDENTIFIER)? PIPE

PtrListPayload <- PIPE ASTERISK? IDENTIFIER (COMMA ASTERISK? IDENTIFIER)* COMMA? PIPE

# Switch specific
SwitchProng <- KEYWORD_inline? SwitchCase EQUALRARROW PtrIndexPayload? SingleAssignExpr

SwitchCase
    <- SwitchItem (COMMA SwitchItem)* COMMA?
     / KEYWORD_else

SwitchItem <- Expr (DOT3 Expr)?

# For specific
ForArgumentsList <- ForItem (COMMA ForItem)* COMMA?

ForItem <- Expr (DOT2 Expr?)?
```

```
# Operators
AssignOp
    <- ASTERISKEQUAL
     / ASTERISKPIPEEQUAL
     / SLASHEQUAL
     / PERCENTEQUAL
     / PLUSEQUAL
     / PLUSPIPEEQUAL
     / MINUSEQUAL
     / MINUSPIPEEQUAL
     / LARROW2EQUAL
     / LARROW2PIPEEQUAL
     / RARROW2EQUAL
     / AMPERSANDEQUAL
     / CARETEQUAL
     / PIPEEQUAL
     / ASTERISKPERCENTEQUAL
     / PLUSPERCENTEQUAL
     / MINUSPERCENTEQUAL
     / EQUAL

CompareOp
    <- EQUALEQUAL
     / EXCLAMATIONMARKEQUAL
     / LARROW
     / RARROW
     / LARROWEQUAL
     / RARROWEQUAL

BitwiseOp
    <- AMPERSAND
     / CARET
     / PIPE
     / KEYWORD_orelse
     / KEYWORD_catch Payload?

BitShiftOp
    <- LARROW2
     / RARROW2
     / LARROW2PIPE

AdditionOp
    <- PLUS
     / MINUS
     / PLUS2
     / PLUSPERCENT
     / MINUSPERCENT
     / PLUSPIPE
     / MINUSPIPE

MultiplyOp
    <- PIPE2
     / ASTERISK
     / SLASH
     / PERCENT
     / ASTERISK2
```

```
     / ASTERISKPERCENT
     / ASTERISKPIPE

PrefixOp
    <- EXCLAMATIONMARK
     / MINUS
     / TILDE
     / MINUSPERCENT
     / AMPERSAND
     / KEYWORD_try
     / KEYWORD_await

PrefixTypeOp
    <- QUESTIONMARK
     / KEYWORD_anyframe MINUSRARROW
     / SliceTypeStart (ByteAlign / AddrSpace / KEYWORD_const / KEYWORD_volatile / KEYWORD_allow
     / PtrTypeStart (AddrSpace / KEYWORD_align LPAREN Expr (COLON Expr COLON Expr)? RPAREN / KE
     / ArrayTypeStart

SuffixOp
    <- LBRACKET Expr (DOT2 (Expr? (COLON Expr)?)?)? RBRACKET
     / DOT IDENTIFIER
     / DOTASTERISK
     / DOTQUESTIONMARK

FnCallArguments <- LPAREN ExprList RPAREN

# Ptr specific
SliceTypeStart <- LBRACKET (COLON Expr)? RBRACKET

PtrTypeStart
    <- ASTERISK
     / ASTERISK2
     / LBRACKET ASTERISK (LETTERC / COLON Expr)? RBRACKET

ArrayTypeStart <- LBRACKET Expr (COLON Expr)? RBRACKET

# ContainerDecl specific
ContainerDeclAuto <- ContainerDeclType LBRACE container_doc_comment? ContainerMembers RBRACE

ContainerDeclType
    <- KEYWORD_struct (LPAREN Expr RPAREN)?
     / KEYWORD_opaque
     / KEYWORD_enum (LPAREN Expr RPAREN)?
     / KEYWORD_union (LPAREN (KEYWORD_enum (LPAREN Expr RPAREN)? / Expr) RPAREN)?

# Alignment
ByteAlign <- KEYWORD_align LPAREN Expr RPAREN

# Lists
IdentifierList <- (doc_comment? IDENTIFIER COMMA)* (doc_comment? IDENTIFIER)?

SwitchProngList <- (SwitchProng COMMA)* SwitchProng?

AsmOutputList <- (AsmOutputItem COMMA)* AsmOutputItem?

AsmInputList <- (AsmInputItem COMMA)* AsmInputItem?
```

```
StringList <- (STRINGLITERAL COMMA)* STRINGLITERAL?

ParamDeclList <- (ParamDecl COMMA)* ParamDecl?

ExprList <- (Expr COMMA)* Expr?

# *** Tokens ***
eof <- !.
bin <- [01]
bin_ <- '_'? bin
oct <- [0-7]
oct_ <- '_'? oct
hex <- [0-9a-fA-F]
hex_ <- '_'? hex
dec <- [0-9]
dec_ <- '_'? dec

bin_int <- bin bin_*
oct_int <- oct oct_*
dec_int <- dec dec_*
hex_int <- hex hex_*

ox80_oxBF <- [\200-\277]
oxF4 <- '\364'
ox80_ox8F <- [\200-\217]
oxF1_oxF3 <- [\361-\363]
oxF0 <- '\360'
ox90_0xBF <- [\220-\277]
oxEE_oxEF <- [\356-\357]
oxED <- '\355'
ox80_ox9F <- [\200-\237]
oxE1_oxEC <- [\341-\354]
oxE0 <- '\340'
oxA0_oxBF <- [\240-\277]
oxC2_oxDF <- [\302-\337]

# From https://lemire.me/blog/2018/05/09/how-quickly-can-you-check-that-a-string-is-valid-unico
# First Byte      Second Byte     Third Byte      Fourth Byte
# [0x00,0x7F]
# [0xC2,0xDF]     [0x80,0xBF]
#    0xE0         [0xA0,0xBF]     [0x80,0xBF]
# [0xE1,0xEC]     [0x80,0xBF]     [0x80,0xBF]
#    0xED         [0x80,0x9F]     [0x80,0xBF]
# [0xEE,0xEF]     [0x80,0xBF]     [0x80,0xBF]
#    0xF0         [0x90,0xBF]     [0x80,0xBF]     [0x80,0xBF]
# [0xF1,0xF3]     [0x80,0xBF]     [0x80,0xBF]     [0x80,0xBF]
#    0xF4         [0x80,0x8F]     [0x80,0xBF]     [0x80,0xBF]

mb_utf8_literal <-
       oxF4      ox80_ox8F ox80_oxBF ox80_oxBF
     / oxF1_oxF3 ox80_oxBF ox80_oxBF ox80_oxBF
     / oxF0      ox90_0xBF ox80_oxBF ox80_oxBF
     / oxEE_oxEF ox80_oxBF ox80_oxBF
     / oxED      ox80_ox9F ox80_oxBF
     / oxE1_oxEC ox80_oxBF ox80_oxBF
     / oxE0      oxA0_oxBF ox80_oxBF
```

```
     / oxC2_oxDF ox80_oxBF

ascii_char_not_nl_slash_squote <- [\000-\011\013-\046\050-\133\135-\177]

char_escape
    <- "\\x" hex hex
     / "\\u{" hex+ "}"
     / "\\" [nr\\t'"]
char_char
    <- mb_utf8_literal
     / char_escape
     / ascii_char_not_nl_slash_squote

string_char
    <- char_escape
     / [^\\"\n]

container_doc_comment <- ('//!' [^\n]* [ \n]* skip)+
doc_comment <- ('///' [^\n]* [ \n]* skip)+
line_comment <- '//' ![!/][^\n]* / '////' [^\n]*
line_string <- ("\\\\" [^\n]* [ \n]*)+
skip <- ([ \n] / line_comment)*

CHAR_LITERAL <- "'" char_char "'" skip
FLOAT
    <- "0x" hex_int "." hex_int ([pP] [-+]? dec_int)? skip
     /      dec_int "." dec_int ([eE] [-+]? dec_int)? skip
     / "0x" hex_int [pP] [-+]? dec_int skip
     /      dec_int [eE] [-+]? dec_int skip
INTEGER
    <- "0b" bin_int skip
     / "0o" oct_int skip
     / "0x" hex_int skip
     /      dec_int   skip
STRINGLITERALSINGLE <- "\"" string_char* "\"" skip
STRINGLITERAL
    <- STRINGLITERALSINGLE
     / (line_string                 skip)+
IDENTIFIER
    <- !keyword [A-Za-z_] [A-Za-z0-9_]* skip
     / "@" STRINGLITERALSINGLE
BUILTINIDENTIFIER <- "@"[A-Za-z_][A-Za-z0-9_]* skip


AMPERSAND            <- '&'      ![=]      skip
AMPERSANDEQUAL       <- '&='               skip
ASTERISK             <- '*'      ![*%=|]   skip
ASTERISK2            <- '**'               skip
ASTERISKEQUAL        <- '*='               skip
ASTERISKPERCENT      <- '*%'     ![=]      skip
ASTERISKPERCENTEQUAL <- '*%='              skip
ASTERISKPIPE         <- '*|'     ![=]      skip
ASTERISKPIPEEQUAL    <- '*|='              skip
CARET                <- '^'      ![=]      skip
CARETEQUAL           <- '^='               skip
COLON                <- ':'                skip
COMMA                <- ','                skip
```

```
DOT                  <- '.'      ![*.?]   skip
DOT2                 <- '..'     ![.]     skip
DOT3                 <- '...'             skip
DOTASTERISK          <- '.*'              skip
DOTQUESTIONMARK      <- '.?'              skip
EQUAL                <- '='      ![>=]    skip
EQUALEQUAL           <- '=='              skip
EQUALRARROW          <- '=>'              skip
EXCLAMATIONMARK      <- '!'      ![=]     skip
EXCLAMATIONMARKEQUAL <- '!='              skip
LARROW               <- '<'      ![<=]    skip
LARROW2              <- '<<'     ![=|]    skip
LARROW2EQUAL         <- '<<='             skip
LARROW2PIPE          <- '<<|'    ![=]     skip
LARROW2PIPEEQUAL     <- '<<|='            skip
LARROWEQUAL          <- '<='              skip
LBRACE               <- '{'               skip
LBRACKET             <- '['               skip
LPAREN               <- '('               skip
MINUS                <- '-'      ![%=>|]  skip
MINUSEQUAL           <- '-='              skip
MINUSPERCENT         <- '-%'     ![=]     skip
MINUSPERCENTEQUAL    <- '-%='             skip
MINUSPIPE            <- '-|'     ![=]     skip
MINUSPIPEEQUAL       <- '-|='             skip
MINUSRARROW          <- '->'              skip
PERCENT              <- '%'      ![=]     skip
PERCENTEQUAL         <- '%='              skip
PIPE                 <- '|'      ![|=]    skip
PIPE2                <- '||'              skip
PIPEEQUAL            <- '|='              skip
PLUS                 <- '+'      ![%+=|]  skip
PLUS2                <- '++'              skip
PLUSEQUAL            <- '+='              skip
PLUSPERCENT          <- '+%'     ![=]     skip
PLUSPERCENTEQUAL     <- '+%='             skip
PLUSPIPE             <- '+|'     ![=]     skip
PLUSPIPEEQUAL        <- '+|='             skip
LETTERC              <- 'c'               skip
QUESTIONMARK         <- '?'               skip
RARROW               <- '>'      ![>=]    skip
RARROW2              <- '>>'     ![=]     skip
RARROW2EQUAL         <- '>>='             skip
RARROWEQUAL          <- '>='              skip
RBRACE               <- '}'               skip
RBRACKET             <- ']'               skip
RPAREN               <- ')'               skip
SEMICOLON            <- ';'               skip
SLASH                <- '/'      ![=]     skip
SLASHEQUAL           <- '/='              skip
TILDE                <- '~'               skip

end_of_word <- ![a-zA-Z0-9_] skip
KEYWORD_addrspace   <- 'addrspace'   end_of_word
KEYWORD_align       <- 'align'       end_of_word
KEYWORD_allowzero   <- 'allowzero'   end_of_word
KEYWORD_and         <- 'and'         end_of_word
```

```
KEYWORD_anyframe    <- 'anyframe'    end_of_word
KEYWORD_anytype     <- 'anytype'     end_of_word
KEYWORD_asm         <- 'asm'         end_of_word
KEYWORD_async       <- 'async'       end_of_word
KEYWORD_await       <- 'await'       end_of_word
KEYWORD_break       <- 'break'       end_of_word
KEYWORD_callconv    <- 'callconv'    end_of_word
KEYWORD_catch       <- 'catch'       end_of_word
KEYWORD_comptime    <- 'comptime'    end_of_word
KEYWORD_const       <- 'const'       end_of_word
KEYWORD_continue    <- 'continue'    end_of_word
KEYWORD_defer       <- 'defer'       end_of_word
KEYWORD_else        <- 'else'        end_of_word
KEYWORD_enum        <- 'enum'        end_of_word
KEYWORD_errdefer    <- 'errdefer'    end_of_word
KEYWORD_error       <- 'error'       end_of_word
KEYWORD_export      <- 'export'      end_of_word
KEYWORD_extern      <- 'extern'      end_of_word
KEYWORD_fn          <- 'fn'          end_of_word
KEYWORD_for         <- 'for'         end_of_word
KEYWORD_if          <- 'if'          end_of_word
KEYWORD_inline      <- 'inline'      end_of_word
KEYWORD_noalias     <- 'noalias'     end_of_word
KEYWORD_nosuspend   <- 'nosuspend'   end_of_word
KEYWORD_noinline    <- 'noinline'    end_of_word
KEYWORD_opaque      <- 'opaque'      end_of_word
KEYWORD_or          <- 'or'          end_of_word
KEYWORD_orelse      <- 'orelse'      end_of_word
KEYWORD_packed      <- 'packed'      end_of_word
KEYWORD_pub         <- 'pub'         end_of_word
KEYWORD_resume      <- 'resume'      end_of_word
KEYWORD_return      <- 'return'      end_of_word
KEYWORD_linksection <- 'linksection' end_of_word
KEYWORD_struct      <- 'struct'      end_of_word
KEYWORD_suspend     <- 'suspend'     end_of_word
KEYWORD_switch      <- 'switch'      end_of_word
KEYWORD_test        <- 'test'        end_of_word
KEYWORD_threadlocal <- 'threadlocal' end_of_word
KEYWORD_try         <- 'try'         end_of_word
KEYWORD_union       <- 'union'       end_of_word
KEYWORD_unreachable <- 'unreachable' end_of_word
KEYWORD_usingnamespace <- 'usingnamespace' end_of_word
KEYWORD_var         <- 'var'         end_of_word
KEYWORD_volatile    <- 'volatile'    end_of_word
KEYWORD_while       <- 'while'       end_of_word

keyword <- KEYWORD_addrspace / KEYWORD_align / KEYWORD_allowzero / KEYWORD_and
         / KEYWORD_anyframe / KEYWORD_anytype / KEYWORD_asm / KEYWORD_async
         / KEYWORD_await / KEYWORD_break / KEYWORD_callconv / KEYWORD_catch
         / KEYWORD_comptime / KEYWORD_const / KEYWORD_continue / KEYWORD_defer
         / KEYWORD_else / KEYWORD_enum / KEYWORD_errdefer / KEYWORD_error / KEYWORD_export
         / KEYWORD_extern / KEYWORD_fn / KEYWORD_for / KEYWORD_if
         / KEYWORD_inline / KEYWORD_noalias / KEYWORD_nosuspend / KEYWORD_noinline
         / KEYWORD_opaque / KEYWORD_or / KEYWORD_orelse / KEYWORD_packed
         / KEYWORD_pub / KEYWORD_resume / KEYWORD_return / KEYWORD_linksection
         / KEYWORD_struct / KEYWORD_suspend / KEYWORD_switch / KEYWORD_test
         / KEYWORD_threadlocal / KEYWORD_try / KEYWORD_union / KEYWORD_unreachable
```

```
    / KEYWORD_usingnamespace / KEYWORD_var / KEYWORD_volatile / KEYWORD_while
```

Zen

- Communicate intent precisely.
- Edge cases matter.
- Favor reading code over writing code.
- Only one obvious way to do things.
- Runtime crashes are better than bugs.
- Compile errors are better than runtime crashes.
- Incremental improvements.
- Avoid local maximums.
- Reduce the amount one must remember.
- Focus on code rather than style.
- Resource allocation may fail; resource deallocation must succeed.
- Memory is a resource.
- Together we serve the users.