# To Zig from C/C++

–

a hands-on introduction into the language core

(for Zig 0.14.1)

Vadim Zavalishin

(book revision 1.1.0alpha)

The book presents a hands-on in-depth introductory course in the core features of Zig programming language for readers with a C/C++ background.

***Disclaimer:*** this book is not a language reference or the ultimate truth source for Zig language, but merely an expression of the author's knowledge and experience obtained during working with Zig. As such it reserves the possibilities of mistakes and incorrect information. Also keep in mind that Zig language is still evolving at the time of the writing of this book.

Built on June 10, 2025.

# Contents (overview)

# Introduction

*The book overview. Technical setup. Acknowledgements.*

The book is written as an introduction to the Zig language for readers who are well familiar with C/C++ programming languages. Specifically, the text assumes familiarity with the low-level programming (mostly represented by the C language or by the C language's features "imported" into C++), as well as familiarity with some of the higher-level programming concepts (primarily OOP) introduced in the C++ language.

The said assumption is exercised in the form of omitting the exact explanations of what Zig language constructs do, but rather making parallels or references to the similar constructs of the C and/or C++ languages. This allows putting more focus on some of the fine details of the Zig language.

The book is not going to cover the language absolutely 100%, and it may only briefly touch on the features of Zig's standard library. The focus is on the fundamental language features and their usage.

The text tries to follow a hands-on approach, mostly explaining by means of example. The book is based on author's knowledge and experience obtained during working with Zig versions 0.12.0.dev through 0.14.1. The Zig language described in the book corresponds to the 0.14.1 version.

## Setup

The book assumes that the reader has a functioning Zig setup. The basic steps of preparing such setup could include the following:

- Downloading a version of the Zig compiler. At the time of this writing, such versions come as archives, which can be simply unpacked into a folder. This folder then normally would be added to the system path, allowing the Zig compiler to be found by the command line.

- Having access to the Zig language reference and the Zig standard library reference, which are both coming with the compiler. At the time of this writing, the Zig language reference is available under `doc/langref.html` in the Zig compiler folder, the Zig standard library reference is launched via `zig std` command from the command line. It is highly recommended to use both as a reference.

- Installing the *Zig Language Server* (ZLS) for the used IDE. The installation details may vary per IDE.
- Optionally setting up a debugger (you may need to try different debuggers depending on your system, as Zig support in debuggers is still a work in progress).

Further, the reader is expected to have created a "scratchpad" Zig project, which can be easily edited and built. Initially (and mostly) this project is supposed to consist of a single Zig source file `main.zig`. We'll review two different options to do that: using the Zig build system and using the "manual" command line.

## Using Zig build system

One way to create a Zig project is to execute `zig init` command from a command line, while being in a folder intended for this project. This command creates some initial source files in the `src` subfolder, as well as the "build script" `build.zig` at the project's root. At the time of this writing, this command however does quite a bit more than needed for our purposes. The command prepares the source tree for both an executable and a library project. As we need only an executable, we might need to do some cleanup in both the build script and the created source files: we just want one source file `src/main.zig`, which we are going to overwrite anyway, and we want the build script to just produce an executable or to compile unit tests for our `main.zig`.

Here is a minimal example of `build.zig`, which could be used directly (at least in Zig 0.14.1):

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

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

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

    const tests = b.addTest(.{
        .root_source_file = b.path("src/main.zig"),
        .target = target,
        .optimize = optimize,
    });
    const run_tests = b.addRunArtifact(tests);
    const test_step = b.step("test", "Run unit tests");
    test_step.dependOn(&run_tests.step);
}
```

In order to build the project (once it contains a valid `src/main.zig` file), execute `zig build` from the command line from the project's root folder (containing `build.zig`). The build result will be available as `zig-out/bin/main.exe`.

In order to run the unit tests (once the source code contains those), execute `zig build test`.

> ▷ There is one problem with the tests built in this way: the test executable is created in some "random" location inside the zig cache folder, which makes it a bit cumbersome to debug. At the time of this writing the author is not aware of a way to address it, as he personally is using the command line option, which we discuss next.

> ▷ Using the Zig build system has a further drawback: the cache folder doesn't seem to be cleaned up (at the time) and needs to be regularly manually emptied or deleted. Otherwise its size can grow rather quickly, taking up the disk space. The same issue hasn't been observed by the author when using "manual" building from the command line.

## Using command line

When using the command line option, it is still helpful to structure the project folder the same way as when using the Zig build system: create `src/main.zig` and even create a valid `build.zig` file at the project's root folder, the latter will be used by the Zig Language Server (otherwise ZLS might have some difficulties resolving certain symbols).

Possible contents of `build.zig` have been presented earlier in the text, the initial contents of `src/main.zig` could be

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

(this program does nothing and immediately exits).

So we assume that the project is already prepared for the Zig build system, but now we'll take a look at how to build it "manually". From the project's root folder execute the following command:

```
zig build-exe src/main.zig
```

This will create a `main.exe` file in the project's root folder.

> ▷ Consider configuring your IDE to execute the same command (from the same folder).

> ▷ An optimized release build can be created by
>
> ```
> zig build-exe -O ReleaseFast src/main.zig
> ```

In order to build and run tests (once the source code contains any), execute the following command:

```
zig test -femit-bin=main.exe src/main.zig
```

The `-femit-bin` option will force the compiler to put the text executable as `main.exe` in the project's root folder, which you can then use for debugging the tests.[1] A further nicety compared to using Zig build system is that the names of executed tests are printed on the console, which is sometimes quite handy to make sure that all expected tests are being run indeed.

> ▷ One could learn quite a bit of details about Zig command line "manual" building options by supplying the `--verbose` option to the `zig build` command. Or execute `zig build-exe --help` for a list of options.

## Acknowledgements

The author would like to thank the Zig community and, in particular, the being active there members of the Zig core team, for their continued support of other Zig enthusiasts. The author has learned a lot about Zig language there.

The author would like to thank his current employer Urs Heckmann of Heckmann Audio GmbH for supporting the author's work on the book.

The author would also like to express deep warm thanks for Romanov I.V. for his personal support.

[1] You could also use the same option with `build-exe` specifying a different build folder than the project root.

# Chapter 1

# Basics

*Basic data types. Control flow fundamentals. Further language basics.*

## 1.1 Hello, world

Let's start with a "Hello, world" program. Type the following into `main.zig`:

```
const std = @import("std"); // Import the standard library

// The program's entry point
pub fn main() void {
    std.debug.print("Hello, world!\n", .{});
}
```

The above is a complete Zig program (compile and run it to convince yourself), roughly corresponding to the following C/C++ code:

```
#include<stdio.h> // Import the standard I/O functionality

// The program's entry point
int main() {
    printf("Hello, world!\n");
    return 0;
}
```

A few notes about the relationship between these two Zig and C/C++ programs:

- The `const std = @import("std");` line at the top of the Zig code roughly corresponds to `#include<stdio.h>`. Actually this Zig line "includes" the entire Zig standard library. There is no real way to "include only a part" of the standard library, but don't worry, Zig will ignore the unreferenced parts automatically anyway.

- The `std.debug.print` function is not exactly a counterpart of C/C++'s `printf`. The former prints to `stderr` and is actually intended to print debug messages. However a proper way to print text to the `stdout` in Zig is more involved:

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

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

  and involves handling of stream write errors.[1] For experimental or work-in-progress code `std.debug.print` is commonly used as a simpler-to-use option.

- In the Zig code you can also see a couple of single-line comments, looking pretty much like single-line comments in C/C++. *There are no multiline comments in Zig like `/* */` in C/C++.* If you want to comment out a number of Zig lines, short of commenting out each line manually, you need to use the features of your code editor. Modern code editors typically offer a way to comment out a selected code block in supported programming languages.

▷ Don't start ordinary Zig comments with three slash characters (`///`), as those have a special meaning in Zig: they denote documentation comments and are explicitly recognized by the language compiler as such.

Differently from the C/C++ version, our Zig program also didn't provide the return value. The Zig startup code will return a zero value in this case automatically. Zig also lets you specify the return value explicitly, if desired:

[1] Error handling in Zig is discussed in Section 4.7.

```
const std = @import("std"); // Import the standard library

// The program's entry point
pub fn main() u8 {
    std.debug.print("Hello, world!\n", .{});
    return 0;
}
```

Here `u8` instead of previously used `void` in the main function's header line defines the main function's return type as a 8-bit unsigned integer (`uint8_t` in C/C++ terms).

▷ There are further possible return types of `main` accommodating Zig error unions as well as `noreturn` kind of main functions. Those are outside the present introductory scope.

## 1.2 Quadratic equation, floats

The following Zig program solves a quadratic equation:

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

pub fn main() void {
    const a: f64 = 1;
    const b: f64 = -3;
    const c: f64 = 2;

    // Solve the equation a*x*x + b*x + c == 0
    const d = b * b - 4 * a * c;
    if (d >= 0) {
        const sqrt_d = std.math.sqrt(d);
        const x1 = (-b - sqrt_d) / (2 * a);
        const x2 = (-b + sqrt_d) / (2 * a);
        std.debug.print("x1 = {}, x2 = {}\n", .{ x1, x2 });
    } else {
        std.debug.print("No real solutions\n", .{});
    }
}
```

The C++ counterpart is provided below:

```
#include<cmath>
#include<print>

int main() {
    const double a = 1;
    const double b = -3;
    const double c = 2;

    // Solve the equation a*x*x + b*x + c == 0
    const auto d = b * b - 4 * a * c;
    if (d >= 0) {
        const auto sqrt_d = std::sqrt(d);
        const auto x1 = (-b - sqrt_d) / (2 * a);
        const auto x2 = (-b + sqrt_d) / (2 * a);
        std::print("x1 = {}, x2 = {}\n", x1, x2);
    } else {
        std::print("No real solutions\n");
    }

    return 0;
}
```

▷ The equation solving code in the above Zig and C++ programs is merely a textbook example. Real quadratic equation solving code would often need to take care of a number of further special and/or ill-conditioned cases.

A few notes:

- We didn't implement reading of the `a`, `b` and `c` variables from the console for the sake of focus and simplicity. Console input goes beyond the basic scope discussed here.
- The `f64` stands for *64-bit float* (`double` in C/C++ terms) and specifies the type of the variables `a`, `b` and `c`. Use `f32` if you want a 32-bit float (`float` in C/C++ terms) instead. Zig also supports `f16`, `f80` and `f128`.
- The Zig variables in this example are declared using the `const` keyword, since they are never modified after the initialization. While using the `const` keyword at the corresponding places in the C++ code is optional, *in Zig it is a must.* Conversely, Zig variables which are modified after the initialization must be declared using the `var` keyword instead.
- We didn't specify the types of the `d`, `sqrt_d`, `x1` and `x2` variables. These types are automatically inferred (to `f64`), similarly to how in C++ we have used the `auto` keyword at the respective places.

  You can verify the automatically inferred types using the `@compileLog` Zig builtin function. E.g. inserting the `@compileLog` statement after the `sqrt_d` definition line as follows:

  ```
  const d = b * b - 4 * a * c;
  if (d >= 0) {
      const sqrt_d = std.math.sqrt(d);
      @compileLog(d, sqrt_d);
  .......
  ```

  would produce the compile-time output:

  ```
  Compile Log Output:
  @as(f64, 1), @as(f64, [runtime value])
  ```

  where you can see that `d` has a (compile time-known) value of 1 of type `f64`, and `sqrt_d` has a (runtime-known) value of type `f64`.

Notice that a `@compileLog` statement also causes the compiler to issue an error and abort the compilation, but that is intentional: you are supposed to use `@compileLog` only temporary to check some compile-time information about your code.

- The `std.math.sqrt` function is loosely speaking a counterpart of C++'s `std::sqrt` overloaded function, the type of the returned value being deduced from the argument's type.
- You can see `std.debug.print` being used to print the values of the variables `x1` and `x2` in a way somewhat similar (but not identical!) to C++'s `std::print`. The pairs of braces in the format string specify where the argument values should be printed. The arguments are specified as the second argument of `std::print` which is a *tuple*[2] containing the values `x1` and `x2`.

▷ Further details of `std.debug.print`'s format string can be found in the documentation of `std.fmt.format` in the Zig standard library documentation.

## 1.3 Numeric literals

When in C/C++ we write

```
const double a = 1;
```

what is actually happening is that we use an integer literal `1` which is then being converted to the destination type `double`. The type of the integer literal `1` is `int`, and it's the same `int` type which is being used for runtime values in C/C++. On modern machines/compilers it's typically 32 bits wide.

In Zig integer literals have type `comptime_int`. This type doesn't have a defined bit width (the values can get pretty much arbitrarily large) nor does it have a defined signedness (but can accommodate both signed and unsigned values). Values of this type can only be used in compile-time expressions, but not for runtime values. When we write

```
const a: f64 = 1;
```

what is happening in Zig is that the compiler, knowing the target type `f64`, will convert the `comptime_int` constant 1 into the respective floating point type and this type will be used as the value of the `a` constant.

Similarly when in Zig we write

```
const x1 = (-b - sqrt_d) / (2 * a);
```

in the `2 * a` expression we are mixing a `comptime_int` constant with an `f64` value `a`. Here, again, the compiler will perform a similar compile-time conversion of 2 to an `f64` constant value.

▷ Actually Zig compiler will perform all computations in the previously discussed quadratic equation program until the square root computation at compile time. This also includes picking the active branch of the `if` statement as well as computing the product `2 * a`, which are also done at compile time.

[2] Tuples are properly discussed in Section 3.13.

This is done not merely as a matter of optimization but as a semantical feature of Zig language: constant expressions are generally evaluated at compile time as far as it is possible (that is, until a runtime-only construct is encountered during the evaluations).

We will return to this topic further in the book, for now it's probably much easier to mostly ignore it.

▷ If you don't specify a type in the declaration, e.g. you simply write:

```
const a = 1; // the type of "a" is comptime_int
```

the variable will have the same type as the expression in the right-hand side, in this case `comptime_int`.

▷ In large-value literals you can use underscores to separate groups of digits, e.g.

```
const one_million = 1_000_000;
```

▷ Besides base 10, Zig supports integer literals in bases 2, 8 and 16:

```
const binary = 0b1010;
const octal = 0o777;
const hex = 0xFFFF_FFFF;
const lowercase_hex = 0xffff_ffff;
const mixed_case_hex = 0xFFFF_ffff;
```

Besides integer literals, Zig also has floating point literals, which respectively have `comptime_float` type. We could have rewritten our quadratic equation solving program using float literals instead:

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

pub fn main() void {
    const a: f64 = 1.0;
    const b: f64 = -3.0;
    const c: f64 = 2.0;

    // Solve the equation a*x*x + b*x + c == 0
    const d = b * b - 4.0 * a * c;
    if (d >= 0) {
        const sqrt_d = std.math.sqrt(d);
        const x1 = (-b - sqrt_d) / (2.0 * a);
        const x2 = (-b + sqrt_d) / (2.0 * a);
        std.debug.print("x1 = {}, x2 = {}\n", .{ x1, x2 });
    } else {
        std.debug.print("No real solutions\n", .{});
    }
}
```

where we explicitly wrote trailing `.0` to enforce the literals to be floating point ones.

Using float literals in floating point computations is not a must in Zig. Similarly to C++, integer values will be often automatically promoted to floating point types in mixed computations, as we already have seen. Zig is however somewhat more restrictive, in particular when floating point and integer comptime values are involved. E.g. if we tried to write `const x: f64 = 1.0 / 2;` the language would complain about the ambiguity of an expression, Zig wouldn't be sure if we wanted to perform a floating point or an integer division. Interestingly, the same doesn't happen with types of known bit width.

## 1.4 Style and formatting

C++ has a number of formatting and naming styles/conventions. There's even certain playroom offered by configurable tools like clang-format. Zig, on the other hand, has a standard "official style".

The standard naming style conventions for Zig are described in the official Zig language reference (which is available on the language's website and is also a part of the compiler package). The Zig code examples in this book follow that convention. E.g., in the previously demonstrated quadratic equation Zig example, notice the usage of the `snake_case_convention` for the variable names.[3]

The standard formatting of Zig source code is supported by the Zig compiler (via the `zig fmt` command) and the Zig language server (ZLS). It is recommended to have the Zig auto-formatting features enabled in your code editor, which is also probably the easiest way to learn the standard formatting.

> ▷ Differently from the formatting, the naming conventions are not auto-enforced by Zig or ZLS. However ZLS might give you subtle hints of naming convention violations. At the time of this writing, ZLS has a feature which highlights certain naming convention violations by underlining the first letter of the respective identifier in gray color.

### 1.4.1 Trailing comma

In the previously shown quadratic equation solving code example the first call to `std.debug.print` is somewhat long. We might want to break it into multiple lines, but, if Zig's autoformatter is enabled in the code editor, we won't be too successful with that, as the autoformatter will bring the code back into a single line. The "official" trick here is to add a trailing comma after the last argument:

```
        std.debug.print(
            "x1 = {}, x2 = {}\n",
            .{ x1, x2 },
        );
}
```

Notice the extra comma after `.{ x1, x2 }` in the above code.

We could even go one step further and also add a trailing comma after `x2`:

```
        std.debug.print(
            "x1 = {}, x2 = {}\n",
            .{
                x1,
```

[3] Actually, only the `sqrt_d` variable allows to conclude that it's the snake case convention and not some other convention is being used.

```
                x2,
            },
        );
}
```

This would allow us to get rid of the auxiliary `x1` and `x2` variables and write the expressions directly inside the `std.debug.print` call without ending up with a too long code line:

```
        const sqrt_d = std.math.sqrt(d);
        std.debug.print(
            "x1 = {}, x2 = {}\n",
            .{
                (-b - sqrt_d) / (2 * a),
                (-b + sqrt_d) / (2 * a),
            },
        );
}
```

▷ This trailing comma approach also works in other similar situations, where comma-separated entities are involved.

## 1.5 Mutable variables, integers

As a demonstration of working with mutable variables (rather than consts) in Zig, consider the following Zig program:

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

pub fn main() void {
    var x: i32 = 1;
    var y: i32 = -3;

    // Make sure that x <= y
    // (by swapping the variable values, if necessary)
    if (x > y) {
        // There is a standard function for swapping in Zig
        // but let's swap by hand, for the sake of demonstration
        const temp = x;
        x = y;
        y = temp;
    }

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

and its C++ counterpart:

```
#include<stdio.h>

int main() {
    int x = 1;
    int y = -3;

    // Make sure that x <= y
    // (by swapping the variable values, if necessary)
    if (x > y) {
        // There is also a standard function for swapping in C++
        // but let's swap by hand, as we do in Zig code
        const auto temp = x;
        x = y;
        y = temp;
    }

    printf("x = %d, y = %d\n", x, y);
}
```

Notes:

- This time we need the variables `x` and `y` to be mutable, so we declare them using the `var` rather than `const` keyword.
- Notice the omitted "else" branch. Zig supports this possibility in a similar way to C/C++ and many other languages.
- `i32` is Zig's 32-bit signed integer type. In the C++ counterpart we have used `int` rather than `int32_t` because it's common and recommended to use the `int` in C/C++ code, unless there are special considerations dictating a particular bit size. On the other hand, Zig does not really have a "default" integer type, so we are explicitly specifying the 32-bit size as our personal "reasonable default" choice.

Differently from floating point types, where Zig only offers a handful of different bit sizes, with integer types Zig supports arbitrary bit widths from 1 through 65535, both signed and unsigned (e.g. `i7` for 7-bit signed integers, `u15` for 15-bit unsigned integers, etc). The "standard" bit widths of 8, 16, 32, 64 and 128 might be still somewhat preferred for integers due to native or close-to-native hardware support on typical platforms.

▷ Integers of "fractional-byte" bit widths still occupy an integer number of bytes (unless used in packed structs). Their size is not necessary rounded up to the very next integer, as alignment considerations are taken into account as well, possibly suggesting a larger size. E.g.

```
pub fn main() void {
    // prints 4 and 4 as size and alignment respectively
    @compileLog(@sizeOf(u21), @alignOf(u21));
}
```

The standard Zig integer types `usize` and `isize` (unsigned and signed respectively) have the bitwidths equal to the bit width of a pointer. So e.g. on a 64-bit platform they would have a bit width of 64 bits. In that sense `usize` is kind of Zig's counterpart of C/C++'s `size_t` type. Respectively `isize` is a signed version thereof (in flat memory models corresponding to C/C++'s `ptrdiff_t` type).

▷ The operations defined on Zig's primary scalar types (integer, floating-point and boolean) are approximately identical to those in C/C++. Notable differences include:

- Logical operators are denoted using `and` and `or` keywords rather than `&&` and `||`. The logical negation is denoted identically (by `!`) in both languages.
- Bitwise operators `&`, `|` and `^` share one and the same precedence, which is also higher than the precedence of the comparisons.
- There are no increment/decrement operators, e.g. you need to write `i += 1;` instead of `i++;`.
- In C++ over-/underflow is UB (undefined behavior) for signed integers and wraps around for unsigned. In Zig it's UB (in unsafe builds) or panic (in safe builds) for both signed and unsigned types. There are however special wraparound and saturating versions of addition, subtraction and multiplication. These operations are available for both signed and unsigned integer types. E.g.

```
var a: u8 = 255;
const b = a +% 1; // wraparound addition, b = 0
const c = b -| 1; // saturating subtraction, c = 0
a +|= 1; // inplace saturating addition, a = 255
a +%= 2; // inplace wraparound addition, a = 1
const d = -%a; // unary wraparound negation, d = 255
```

## 1.6 Warnings are errors

Differently from C/C++, Zig is always in the "warnings are errors" mode. Formally, Zig doesn't even have warnings, there are only errors, but typical issues which would be diagnozed as warnings in C/C++ are then diagnozed as errors in Zig.

### 1.6.1 Unused consts

If you declare a constant local variable inside a Zig function and do not use this const ever after, Zig will generate an error. E.g. the following Zig code:

```
pub fn main() void {
    const x: f64 = 1;
}
```

generates the following error:

```
main.zig:2:11: error: unused local constant
    const x: f64 = 1;
          ^
```

To quickly fix the error without really using the `x`, assign it to the '_' placeholder variable:

```
pub fn main() void {
    const x: f64 = 1;
    _ = x;
}
```

▷ This kind of fix is appropriate temporarily in code which is under constructor or modification. This fix is also automatically done by the Zig's auto formatter (possibly depending on the auto-formatter's configuration). Having such "fixes" in finished code should rather occur in explicitly intended exceptional places.

Notice that the above examples do not contain the line that imports the std. This line is unnecessary, since the example code doesn't make any use of the std. Placing the import line into the program code is nevertheless still possible:

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

pub fn main() void {
    const x: f64 = 1;
    _ = x;
}
```

Even though the `std` const is not used anywhere across the program code, the compiler doesn't complain: the requirement for consts to be used only applies to consts declared inside functions. It also doesn't really hurt: the compiler will ignore the unreferenced `std` entity.

▷ The assignment of an unused const to the '_' placeholder makes it "syntactically used" (so that the compiler doesn't complain), but does not make it "truly used" (meaning that the optimizer is still likely to discard the entire piece of code used to compute the respective value). E.g. in the following code

```
const value: f64 = someFunctionWithSideEffects();
const sqrt_value = std.math.sqrt(value);
_ = sqrt_value;
```

the call to `std.math.sqrt` is likely to be fully removed by the optimizer in the release builds. To force the optimizer to preserve the code (for whatever special reason you might have) you can use the `std.mem.doNotOptimizeAway` function:

```
const value: f64 = someFunctionWithSideEffects();
const sqrt_value = std.math.sqrt(value);
std.mem.doNotOptimizeAway(sqrt_value);
```

### 1.6.2 Unmutated vars

We have briefly mentioned that a variable which is modified after the initialization has to be declared with a `var` keyword. If, despite being declared with a `var` keyword, this variable is not modified afterwards (or is not used at all), a compile error is generated. E.g.:

```
pub fn main() void {
    var x: f64 = 1;
    _ = x;
}
```

produces the following error:

```
main.zig:2:9: error: local variable is never mutated
    var x: f64 = 1;
        ^
main.zig:2:9: note: consider using 'const'
```

Notice that our code does *use* the variable by assigning it to the '_' placeholder variable in '`_ = x;`', however this is not enough: we must modify the variable's value, not just use it.

To quickly fix the issue, assign the *address* of the variable (rather than the variable's value) to the '_' placeholder:

```
pub fn main() void {
    var x: f64 = 1;
    _ = &x;
}
```

> ▷ Same as with consts, this kind of fix is mostly appropriate for code which is being worked on.

> ▷ The assignment of the variable's address to the '_' placeholder avoids the compile error, but doesn't necessarily make the variable mutated or "used" from the optimizer's perspective. Again, `std.mem.doNotOptimizeAway` is of help here, with the strongest form being achieved by passing the address of the variable:
>
> ```
> var x: f64 = 1;
> std.mem.doNotOptimizeAway(&x);
> someFunction(x);
> ```
>
> In the above example the optimizer should not assume that the value of `x` is still equal to 1 at the time of calling `someFunction`.

### 1.6.3 Uninitialized vars

Declaring an uninitialized variable is totally acceptable in C++, but it's an error in Zig. E.g:

```
pub fn main() void {
    var x: f64;
    x = 0;
}
```

produces

```
main.zig:2:15: error: expected '=', found ';'
    var x: f64;
              ^
```

Differently from the situations with unused consts or unmutated vars, the situation with uninitialized variables is much more common. Actually every once in a while it's not really possible to provide a reasonable initial value for a variable and the variable needs to be initialized in-place (that is directly in the memory associated with the variable) after its declaration.

To prevent inadvertently uninitialized variables, Zig requires you to explicitly declare such variables as uninitialized:

```
pub fn main() void {
    var x: f64 = undefined;
    x = 0;
}
```

The above code perfectly compiles.

### 1.6.4 Identifier shadowing

If we tried to compile the following Zig code:

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

pub fn main() void {
    var i: i32 = 0;
    var i1: i32 = i + 1; // error: shadows the i1 type
    std.mem.swap(i32, &i, &i1);
}
```

the compiler would yield an error:

```
main.zig:5:9: error: name shadows primitive 'i1'
    var i1: i32 = i + 1;
        ^~
main.zig:5:9: note: consider using @"i1" to disambiguate
```

At this point we should recall that `i1` is a standard type name in Zig for a 1-bit signed integer. However, if we looked into the Zig language reference, we won't find `i1` in a list of keywords. It's not a keyword, but rather a kind of an identifier.

In C/C++ we would be able to redefine an externally defined identifier locally inside a scope:

```
#include<stdio.h>

int main() {
    // The following code will compile, unless
    // printf is defined as a macro in the particular
    // implementation of the standard library.
    const int printf = 0;
    {
        const char printf = 'a';
    }

    return printf;
}
```

In the above the `printf` identifier is redefined twice: first at the `main` function's scope and second time inside the nested code block. This kind of identifier redefinition is referred to as *identifier name shadowing.*

Zig forbids name shadowing, the rationale being that name shadowing complicates the reading of the code, and Zig puts a high value on the code readability. So this is, in particular, the case with our `i1` identifier: it shadows a standard integer type.

Similarly, one wouldn't be able to do:

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

pub fn main() void {
    const i: i32 = 1;
    {
        const i: u32 = 0; // error: shadows the outer i
        std.debug.print("inner i = {}\n", .{i});
    }
    std.debug.print("outer i = {}\n", .{i});
}
```

nor can one do

```
pub fn main() void {
    // error: shadows the function name
    const main: i32 = 0;
    std.debug.print("main = {}\n", .{main});
}
```

As a workaround in the previous case of shadowing the `i1` type name one can use `@"i1"` in place of `i1` as the identifier name. The `@" "` syntax allows to specify identifier names which would otherwise be forbidden in Zig due to naming conflicts with keywords or with standard identifiers or due to characters normally not allowed in identifiers:

```
const @"i1": i32 = 1;
const @"if": bool = true;
const @"100%": f32 = 1.0;
```

In this way one can even use non-latin UTF-8 letters in the identifiers[4]:

```
const @"qualität" = 10;
```

Notes:

- The `@"i1"` workaround doesn't work with identifiers which are defined normally in the source code (including the ones defined in the standard library). Generally Zig treats `name` and `@"name"` as one and the same identifier[5], so the workaround only works with keywords or built-in identifiers, such as `i1`. In particular, this workaround wouldn't work in the other shadowing cases that we demonstrated above.

- The "no-shadowing" rule is applied only to identifier names which may get into an actual conflict. E.g. the following code compiles:

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

  pub fn main() void {
      const print: i32 = 1;
      std.debug.print("print = {}\n", .{print});
  }
  ```

  Here `print` doesn't collide with `std.debug.print` because the latter is inside the `std.debug` namespace and needs an explicit namespace qualification.

[4]Zig treats source files as UTF-8-encoded, but forbids characters beyond the *Basic Latin* block, except in character and string literals.

[5]To the extent that Zig's autoformatter would remove the `@" "` surrounding "normal" identifier names, e.g. `@"i"` or `@"main"`.

## 1.7 Binary search, while-loops

Let's implement binary search in Zig:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    // Perform a binary search for key in data
    var low: usize = 0;
    var high: usize = data.len - 1;
    while (low + 1 < high) {
        const mid = (high + low) / 2;
        if (key <= data[mid])
            high = mid
        else
            low = mid + 1;
    }

    std.debug.assert(low <= high);
    if (low < high)
        std.debug.print("Found at {}\n", .{low})
    else
        std.debug.print("Not found\n", .{});
}
```

The C++ counterpart is:

```
#include<stdio.h>
#include<array>
#include<boost/assert.hpp>

static const int data[] = { -10, -1, 5, 12, 100 };

int main() {
    const int key = 12;

    // Perform a binary search for key in data
    size_t low = 0;
    size_t high = std::size(data) - 1;
    while (low + 1 < high) {
        auto mid = (high + low) / 2;
        if (key <= data[mid])
            high = mid;
        else
            low = mid + 1;
    }

    BOOST_VERIFY(low <= high);
    if (low < high)
```

```
        printf("Found at %zu\n", low);
    else
        printf("Not found\n");

    return 0;
}
```

Notes:

- The line `const data = [_]i32{ -10, -1, 5, 12, 100 };` initializes a 5-element array of 32-bit integers with the given values.
- In the C++ counterpart the `data` array is declared as a static variable outside of the function to ensure a comparable to Zig generated machine code. Had we declared this array simply inside the function as

  ```
  int main() {
      const int data[] = { -10, -1, 5, 12, 100 };
      const int key = 12;
      .........
  ```

  the compiler might allocate the `data` array on the stack.[6] Declaring `data` as a static variable inside the function would be better in this respect but, as it kind of implies singleton semantics, could involve explicitly running the initialization code upon the first function invocation behind a singleton guard (although the compiler might refrain from doing this for a raw array and have it as preinitialized data). So, declaring `data` as a global-lifetime object outside the function is probably the most reliable way to ensure that this object will be preinitialized before any code is executed. This is what happens here in Zig: the compiler detects that the object's value is fully known at compile time and simply creates preinitialized data out of it in the respective global data segment, rather than allocating memory on the stack.
- Differently from `data`, in the C++ version we do not bother (nor do we have to) to place the definition of the `key` constant outside of the function: since this constant has a scalar rather than array type, we do not care too much about the associated memory.
- We might have considered using `std::array` instead of raw arrays in the C++ code above, since, at least in some standard library implementations, the `std::array` contains array index range-checking assertions inside the `operator[]` implementation. Zig contains this functionality by default. Feel free to try it out by modifying the above Zig code to access array elements outside of the correct index range.
- The expression `data.len - 1` used to initialize the `high` variable is equal to the length of the data array minus 1. Zig arrays have an implicit field `len` (of `usize` type) equal to the length of the array.
- In the C++ version we have declared the `low` and `high` variables to be of the type `size_t`, which is the counterpart of Zig's `usize`. Had we been

[6] At least clang-cl 19.1.0 seems to do so in unoptimized builds.

writing this code from scratch in C++, we probably would have used `unsigned` or `int` type instead.[7]

- In the `const mid = (high + low) / 2;` line Zig performs unsigned division with truncation towards zero, same as in C/C++. With signed division Zig recommends to use the builtin functions `@divTrunc`, `@divFloor` and `@divExact` instead (the latter can also occasionally make sense with unsigned division). See Zig language reference for more info.

The `std.debug.assert` function call in the above code works *similarly* to C language's `assert` macro, but there are two important distinctions:

- The condition supplied to `std.debug.assert` is also forwarded as a hint to the optimizer, similarly to C++'s `[[assume(.....)]]` attribute. So, if the assertion condition in Zig is violated in optimized builds, it could raise undefined behavior.

- Zig has no macros and `std.debug.assert` is simply a function with a boolean argument. This means that at least the side effects of the assert's argument are preserved in the builds with disabled assertions as well (or possibly more than just the side effects is preserved, if the optimizer doesn't look "deep enough"). Lacking comparable functionality in the standard C/C++ library, in the C++ version of the code we use *boost* library's `BOOST_VERIFY` macro, which does preserve the side effects. Not that it matters in the current case, as there are no side effects, but we want to provide a formally better matching counterpart.

  In Section 1.9 we are going, among other things, to discuss a way to explicitly avoid the assertion argument side effects in Zig when the assertions are disabled, so that we can implement functionality comparable to C/C++'s `assert` macro.

## 1.8 Array types and literals

In the previously discussed code we have used the line

```
const data = [_]i32{ -10, -1, 5, 12, 100 };
```

to initialize a 5-element array of 32-bit integers with the given values. The syntactical construct `[_]i32{ -10, -1, 5, 12, 100 };` to the right of the equality sign is a Zig constant literal value, whose type is "array of 5 elements of type `i32`", where the length of the array is auto-inferred from the number of values inside the braces. The explicitly-specified-length form of this literal is `[5]i32{ -10, -1, 5, 12, 100 };` and the respective type is notated in Zig

[7] It is a controversial subject in software programming, whether array indices should use signed or unsigned types. The operation in the vicinity of the range's boundary constantly occurring with unsigned arithmetic is a source of various subtle (and not so subtle) bugs, which, in author's opinion, is an overwhelmingly strong and deciding argument to prefer signed integers by default (other things being equal). In C/C++ using signed integers for array indexing is a bit cumbersome but still quite manageable. At least the `[ ]` operation in C/C++ usually accepts signed arguments without any complaints or warnings. In Zig, however, the `[ ]` operation accepts only unsigned arguments, signed ones require an explicit cast, which makes using signed integers for array indexing not too legible.

as `[5]i32`. So writing '_' instead of the array length in the literal's "header" makes the array length inferred from the rest of the literal.

This constant literal can be used on its own outside of the initializer syntax, e.g.

```
var data: [5]i32 = undefined;
data = [_]i32{ -10, -1, 5, 12, 100 };
```

(notice that the array length needs to be specified in the first line, one cannot write `[_]i32` there, since `[5]i32` there denotes the variable's type, rather than acts as an array literal's "header").

Instead of using an array literal, an array variable can be assigned from a tuple literal[8]:

```
var data: [5]i32 = undefined;
data = .{ -10, -1, 5, 12, 100 };
```

Zig knows that the destination type is `[5]i32` and will automatically convert the tuple literal `.{ -10, -1, 5, 12, 100 }` to a value of this type. The tuple literal can also be used for array initialization, if we explicitly specify the variable type:

```
const data: [5]i32 = .{ -10, -1, 5, 12, 100 };
```

Feel free to try all these options.

▷ Compile-time-known const values can be used to specify array size, similarly to C++'s `constexprs`. E.g.

```
const size = 5;
var data: [size]i32 = undefined;
```

The size value doesn't need to have a formal `comptime_int` or `usize` type:

```
const size: u8 = 5; // this also works
var data: [size]i32 = undefined;
```

## 1.9 Safe-mode builds

In C/C++ we commonly have debug and non-debug (a.k.a. release) builds, typically differing by

- the level of optimization
- the presence and/or detail of debug info
- the enabling/disabling of the assertion functionality.

Depending on the details of the build system configuration there might be a number of different debug and release build modes.

In Zig there are 4 standard build modes:

- Debug

[8]Tuples are properly discussed in Section 3.13.

- ReleaseSafe
- ReleaseFast
- ReleaseSmall

Of these, *debug* and *release-safe* are the *safe-mode* builds, which broadly speaking means that "the assertions are enabled" in those two. This includes the assertions explicitly contained in the program source code, but also the assertions implicitly put into the code by the compiler (known as *safe-mode checks*), such as

- array bounds checks
- overflow and underflow checks during arithmetic operations, including conversions from one type to another
- zero-remainder checks in the `@divExact` builtin function
- etc.

In the previously presented binary search code there are a couple of places containing implicit safe-mode checks:

- As mentioned earlier, array element accesses (using the `[ ]` operation) in Zig are automatically bounds-checked.
- In the expression `data.len - 1` in the initializer of the `high` variable Zig performs unsigned subtraction, creating a runtime error in safe-mode builds in case of an underflow. Actually in our case the value of `data.len` is known at compile time, and so would be the underflow error. Feel free to try it by modifying the `data` declaration to

  ```
  const data = [0]i32{};
  ```

▷ Differently from C++, zero-length arrays (and zero-size entities in general) are completely valid in Zig.

Similarly to how one can find the assertions enabled/disabled state by inspecting the `NDEBUG` macro in C/C++ code, one can distinguish between safe and unsafe build modes in Zig by inspecting the `std.debug.runtime_safety` boolean constant.[9] One could thus e.g. emulate the behavior of C/C++'s `assert` macro along the lines of the following code example:

```
if(std.debug.runtime_safety)
    std.debug.assert(someFunctionWithSideEffects() == 0);
```

Furthermore, Zig knows that this constant is compile-time known, so the `if` is going to work similarly to `if constexpr` in C++. Actually, in Zig it's somewhat "stronger" than C++'s `if constexpr`: if the condition is false, the code inside the if will only undergo basic syntactic parsing, but not more, e.g. it would still compile if `someFunctionWithSideEffects` was not defined at all in unsafe builds.

[9]It is also possible to distinguish between the build modes themselves. This option will be discussed in Section 6.8.

▷ If you want to make really sure that `std.debug.runtime_safety` is a compile time-known constant and would not cause any runtime checks, you could rewrite the previous `if` as:

```
if(comptime std.debug.runtime_safety)
    std.debug.assert(someFunctionWithSideEffects() == 0);
```

The above code will fail to compile if `std.debug.runtime_safety` is not a compile time-known value.

▷ Instead of writing `std.debug.assert(false)` one can simply write `unreachable`:

```
if(!expected_condition)
    unreachable; // same as std.debug.assert(false);
```

Actually `unreachable` is how `std.debug.assert` is implemented internally (at the time of the writing). It is somewhat similar to C++'s `std::unreachable`.

So, in safe build modes the `unreachable` in Zig invokes a runtime panic, like a failing assertion does. In optimized build modes it also gives a hint to the optimizer, potentially raising undefined behavior in cases where the `unreachable` statement is reached by the runtime control flow.

## 1.10 @as() builtin function

So, numeric literals in Zig have `comptime_int` and `comptime_float` types. But what if we explicitly want a literal of, say, `i32` or `f64` type? Do we have to introduce an auxiliary constant like:

```
const i32_two: i32 = 2;
const f64_two: f64 = 2;
```

We could do either that, or we could *cast* the literal to the required destination type using the `@as()` builtin function:

```
const half: f32 = 0.5;
// Force f64 multiplication
const pi_half = half * @as(f64, 3.141592653589793);
```

In the above, the `@as()` builtin function casts the floating point literal from `comptime_float` to `f64` type.

The `@as()` builtin function can also be used to cast the result of any expression, not just compile-time literals. E.g.:

```
const half: f32 = 0.5;
// Cast to f64 after performing the multiplication
const pi_half = @as(f64, 3.141592653589793 * half);
```

It is however required, that the destination type can exactly accommodate the source value:

```
const pi64: f64 = 3.141592653589793;
// The next line fails, as precision is lost
// upon converting pi64 to f32
const pi32 = @as(f32, pi64);
@compileLog(pi32);
```

The above code fails for exactly the same reason as the following version of the same code:

```
const pi64: f64 = 3.141592653589793;
// The next line fails due to precision loss
const pi32: f32 = pi64;
@compileLog(pi32);
```

Similarly, the following code fails:

```
const one_thousand32: i32 = 1000;
// The next line fails since i8 cannot
// represent the source value
const one_thousand8 = @as(i8, one_thousand32);
@compileLog(one_thousand8);
```

and so does its alternative version:

```
const one_thousand32: i32 = 1000;
// The next line fails since i8 cannot
// represent the source value
const one_thousand8: i8 = one_thousand32;
@compileLog(one_thousand8);
```

The exact representability of the source value will be checked at compile time, if the source value is compile time-known, otherwise it is required that all values of the source expression's type are representable by the destination type:

```
// The next line fails, since not all i32 values
// are representable by i8
const i8_value = @as(i8, runtime_i32_value);
@compileLog(i8_value);
```

▷ Technically, the `@as()` builtin function doesn't implement type casting. Rather it provides a *destination type* for the expression. Generally, `@as(Type,expr)` works whenever `const temp: Type = expr;` works. Both constructs have pretty much the same semantics, except that the former one doesn't introduce a named variable.

## 1.11 Arithmetic casting

If the destination type cannot exactly represent the source value (or all values of the source type), or if the conversion from source to the destination type is "nontrivial", one needs to use one of the explicit casting builtin functions. E.g. if we are willing to simply discard the bits not fitting into the destination type, we could do

```
// Equivalently we could do
//   const i8_value: i8 = @truncate(runtime_i32_value);
const i8_value = @as(i8, @truncate(runtime_i32_value));
```

The above code corresponds to the C/C++ code:

```
const auto i8_value = (int8_t)runtime_i32_value;
```

The `@truncate()` builtin function in the Zig code enables the casting to the destination type, which normally would be forbidden due to the destination bitsize being too small.

The destination type for `@truncate()` is provided by the surrounding `@as()` builtin call (or by the explicitly declared type of the `i8_value` variable). This is a common pattern in Zig: the destination type for casting is provided implicitly by the "expected type of the result". This expected type could be provided in a variety of ways:

- By the `@as()` builtin function

  ```
  const i8_value = @as(i8, @truncate(runtime_i32_value));
  ```

- By the explicitly specified type of the destination variable:

  ```
  const i8_value: i8 = @truncate(runtime_i32_value);
  ```

- By the type of the function's parameter that we are passing the casting result to:

  ```
  // truncate to 8 bit
  functionExpecting8BitValue(@truncate(runtime_i32_value));
  ```

- By the declared return type of the function that we are returning from:

  ```
  pub fn main() u8 {
      .......
      return @truncate(runtime_i32_value); // truncate to u8
  }
  ```

All of these approaches provide a known *result location type*, which is then used by the casting function (such as `@truncate`) to determine the cast destination type.

▷ The following builtin functions are commonly used for arithmetic casting:

- Integer-to-integer: `@intCast` (the value is supposed to fit into the destination type), `@truncate` (the value is truncated if it doesn't fit).
- Float-to-float: `@floatCast` (precision loss is allowed, too large values are converted to infinity).
- Between integers and floats: `@intFromFloat` (truncates towards zero, result is supposed to fit into the destination type), `@floatFromInt`.

In cases where the result is not guaranteed to be representable in the destination type, runtime checks are performed in safe builds.

`@intFromFloat` always truncates towards zero. In case other rounding direction is desired, the value is supposed to be rounded within floating point representation first, using one of `@trunc`, `@floor`, `@ceil`, and `@round` builtins.

## 1.12 Linear search, for-loops

Instead of a binary search, let's run a straightforward linear one. Differently from the previous time, where we have been presenting a Zig prototype and then augmented it with a “translation to C/C++”, this time let’s start with a C++ prototype and then work from this point towards a proper Zig implementation:

```cpp
#include<stdio.h>
#include<array>

static const int data[] = { -10, -1, 5, 12, 100 };

int main() {
    const int key = 12;

    // Perform a linear search for key in data
    int found_at = -1;
    for (int i = 0; i < std::ssize(data) ; i++) {
        if (data[i] == key) {
            found_at = i;
            break;
        }
    }

    if (found_at >= 0)
        printf("Found at %d\n", found_at);
    else
        printf("Not found\n");

    return 0;
}
```

A direct translation of the above to Zig would be something like:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    // Perform a linear search for key in data.
    // NB: this is a very C-like implementation,
    // one would usually not do it like this in Zig.
    var found_at: isize = -1;
    for (0..data.len) |i| {
        if (data[i] == key) {
            found_at = @intCast(i);
            break;
        }
    }

    if (found_at >= 0)
        std.debug.print("Found at {}\n", .{found_at})
    else
        std.debug.print("Not found\n", .{});
}
```

Notes:

- The Zig loop statement `for (0..data.len) |i|` can be seen as a direct counterpart of C++'s

  ```
  for (size_t i = 0; i < std::size(data); i++)
  ```

  or, actually, more formally speaking,

  ```
  for (auto i:
      std::ranges::views::iota(size_t(0), std::size(data)))
  ```

  that is, `for (0..data.len) |i|` is, conceptually seen, a range-based for loop. Zig does only have range-based for loops.

  Note that the range bounds do not have to be compile-time-known values, runtime expressions are also allowed, e.g.

  ```
  for (1..data.len) |i| {
      // The lower bound (i-1) is a runtime value
      for ((i-1)..data.len) |j|
          .......
  }
  ```

  The range bounds can be equal, implying an empty range, but the lower bound should not be larger than the upper one.

- The variable `i` contained in the vertical-line brackets `| |` directly following the `for` loop's header works as an implicitly declared const variable inside the loop's body:

  ```
  const i = the_current_iteration_value;
  ```

  that is, the value of `i` cannot be changed inside the loop body by an assignment etc., it only changes when the loop enters a new iteration.

  Some other Zig statements (`if`, `while`, `catch`) also use or can use the same construct with implicitly declared const variables contained the vertical-line brackets `| |` directly following the statement's header. These variables are referred to as *captures*.

  The capture type cannot be explicitly specified in the program code, but is always automatically inferred. In C++ terms we could loosely say that "Zig captures are always declared with a `const auto` type" (one exception: there is also the "`auto &`" kind of declaration for captures, which is discussed in Section 2.14). Zig's integer ranges always have bounds of type `usize`, and respectively `usize` is also the type of the capture variable `i` in our example. For that reason we have chosen the type of the `found_at` variable to be `isize` in the Zig version, although `i32` would have done equally well here, without any further changes to the code.

- The `@intCast` ("integer cast") builtin function used in the assignment `found_at = @intCast(i);` works similarly to C++'s `static_cast`, the entire assignment statement loosely corresponding to

  ```
  found_at = static_cast<isize>(i);
  ```

in C++ terms (where we "imported" the `isize` type from Zig into C++), or, since we do not really have to explicitly use `static_cast` for scalar conversions,

```
found_at = (isize)i;
```

As discussed in Section 1.10, the cast's destination type in Zig is not specified explicitly, but is inferred from the context, in this case from the assignment's destination type (that is, the type of the variable `found_at`, which is `isize`).

Zig requires an explicit cast here because the destination type `isize` cannot accommodate all possible values of the source type (`usize`). The explicit cast thereby indicates the developer's awareness of the fact that some values in theory might be not convertible. In safe-mode builds inconvertible values will be detected automatically during the conversion and raise a runtime panic event, similar to a failing assertion.

▷ Sometimes you must introduce a capture, but don't need to use it. E.g. imagine we want to print "Hello, world!" 10 times. The following code would produce a compile error:

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

pub fn main() void {
    for (0..10)
        std.debug.print("Hello, world!\n", .{});
}
```

as the `for` loop requires a loop payload capture. However the following code would produce a compiler error as well:

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

pub fn main() void {
    for (0..10) |i|
        std.debug.print("Hello, world!\n", .{});
}
```

this time due to the capture being not used.

We could avoid the error by the previously discussed `_ = i;` trick, but it would be more appropriate and idiomatic in Zig to use a '_' capture instead:

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

pub fn main() void {
    for (0..10) |_|
        std.debug.print("Hello, world!\n", .{});
}
```

The just discussed Zig linear search code does a good job of mimicking the functionality of the original C++ code, but doesn't really look like code written originally in Zig, more like code which has been directly translated from C++. In a way, "it speaks a formally correct Zig language, but the phrases sound a bit unusual". Let's see how we could improve that.

## 1.13 Optionals

Instead of using negative values to represent the “not found” state, we could use Zig’s optionals. Optionals are also found in C++, but they are somewhat of a “new fancy feature in the standard library”, while in Zig they are built into the language itself, supporting their more prominent usage. Rewriting the previous Zig code in terms of Zig’s optionals gives:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    // Perform a linear search for key in data.
    // NB: the "for" loop itself is still somewhat C-like.
    var found_at: ?usize = null;
    for (0..data.len) |i| {
        if (data[i] == key) {
            found_at = i;
            break;
        }
    }

    if (found_at) |index|
        std.debug.print("Found at {}\n", .{index})
    else
        std.debug.print("Not found\n", .{});
}
```

Here is the C++ counterpart for reference:

```cpp
#include<stdio.h>
#include<array>
#include<optional>

static const int data[] = { -10, -1, 5, 12, 100 };

int main() {
    const int key = 12;

    // Perform a linear search for key in data
    std::optional<size_t> found_at;
    for (size_t i = 0; i < std::size(data) ; i++) {
        if (data[i] == key) {
            found_at = i;
            break;
        }
    }

    if (found_at)
        printf("Found at %zu\n", found_at.value());
    else
        printf("Not found\n");

    return 0;
}
```

Notes:

- The `found_at` variable's type in Zig code is an "optional `usize`", which is written as `?usize` and corresponds to `std::optional<size_t>` in C++. It is initialized to the `null` state, thereby not containing a value initially. In Zig we need to specify this `null` initial value explicitly, in C++ it's implicitly provided by the default constructor of the `std::optional`.

- The `index` variable inside the printing `if` statement is a capture. As discussed previously, captures are const variables. In this case `index` is initialized to the value contained inside the `found_at` optional. As the optional's type is `?usize`, the type of `index` is automatically inferred as `usize` and cannot be changed to any other type. This variable is available only inside the "then" branch, making sure (on the syntactical level, that is already during compile-time) that it can be accessed only if the optional really contains a value.

  Thus, the `if` statement in the result-printing code looks somewhat different between Zig and C++. Actually, if we take the C++ version and try to convert it to Zig as closely as possible to the original we would get:

  ```
  if (found_at != null)
      std.debug.print("Found at {}\n", .{found_at.?})
  else
      std.debug.print("Not found\n", .{});
  ```

  Here `if (found_at != null)` would be a direct counterpart of C++'s `if (found_at)`, and `found_at.?` would be a direct counterpart of C++'s `found_at.value()`. This construction is more error-prone: if we messed up the condition logic we could inadvertently call `found_at.value()` on a null optional, producing a runtime assertion or (potentially) a crash. The same applies to the `found_at.?` construction in Zig. So other things being equal, it's better not to use `.?` if there are safer possibilities, like the `if (found_at) |index|` construct used in our Zig code.

▷ The value of the `index` variable is thereby equal to `found_at.?`. One might formally rewrite the `if (found_at) |index|` as follows:

```
if (found_at != null) {
    const index = found_at.?;
    std.debug.print("Found at {}\n", .{index});
} else {
    std.debug.print("Not found\n", .{});
}
```

▷ At times one might be tempted to write something like

```
if (found_at) // this doesn't compile
    std.debug.print("Found at {}\n", .{found_at.?})
else
    std.debug.print("Not found\n", .{});
```

but this code doesn't compile, as the latter `if` requires a capture. If you don't want to use a capture, you must explicitly write a comparison with a `null` value: `if (found_at != null)`. Alternatively you can capture into '_':

```
if (found_at) |_| // a bit questionable
    std.debug.print("Found at {}\n", .{found_at.?})
else
    std.debug.print("Not found\n", .{});
```

but that's probably a bit questionable: why don't you explicitly compare to `null` instead or avoid the usage of the '`.?`' operation here altogether?

Likewise, you cannot write `if (!found_at)`, you need to compare to `null` instead: `if (found_at == null)`.

In other words, Zig doesn't automatically promote optionals to bools. The same is actually also the case with integers, e.g. one cannot write something like:

```
var x: i32 = 1;
if (x) x = 0; // doesn't compile
if (!x) x = 1; // doesn't compile either
```

but needs to explicitly compare to zero:

```
var x: i32 = 1;
if (x != 0) x = 0;
if (x == 0) x = 1;
```

▷ Zig has a kind of counterpart of C++ `std::optional`'s `value_or()` method in the form of the `orelse` operator. E.g. we could modify our printing code into:

```
std.debug.print("Found at {}\n", .{found_at orelse 1000});
```

which would be a counterpart of C++'s

```
printf("Found at %zu\n", found_at.value_or(1000));
```

The difference between the `orelse` and the `value_or()` is that the latter will always evaluate its argument, while the former will perform the evaluation of its right-hand side only if the optional is `null`.

The `orelse` operator is also expressible in the terms of the `if` *expression* (discussed in Section 2.3). The above example of the `orelse` operator usage could have been equivalently rewritten as:

```
std.debug.print(
    "Found at {}\n",
    .{if (found_at) |index| index else 1000},
);
```

the C++ counterpart being

```
printf("Found at %zu\n", found_at ? found_at.value() : 1000);
```

▷ You can use captures with optionals in a `while` loop header in the same way how they are used with `if`s:

```
while(optional) |inner| {
    ........
}
```

Such loop will run as long as the optional is non-empty. A practical example can be found in Section 3.2.

## 1.14 Parallel for-loops

The Zig code with introduced optionals looks "noticeably more Zig", but we could try to improve it further. Namely, we could write a Zig counterpart of C++'s range-based for loop over the data array: `for (auto item: data)`. The latter can be written in Zig as `for (&data) |item|`.

Notes:

- We have mentioned that `for (0..data.len) |i|` is a range-based for loop over the range `0..data.len`. Similarly, `for (&data) |item|` is a range-based for loop over the `data` array itself.

- Instead of writing `for (&data) |item|` one could omit the address operator (denoted by '`&`') and simply write `for (data) |item|`. Both loops would work in a similar fashion with one difference: the second loop (without the address operator) first makes a temporary local copy of the entire array and then iterates over this local copy rather than iterating over the original array. Notably, in optimized builds the optimizer might eliminate this temporary copy step, but seems to rather do it for small-size arrays. Also such elimination is not always functionally possible, if the array is being modified during the iteration.

  The author is not aware of the purpose of this language feature or use cases where its application would be justified. Therefore the author's advice (at the time of writing this text) would be to always prefix arrays with the address operator inside `for` loops in Zig.

Attempting to use a range-based for loop we would arrive at one and the same problem both in C++ and in Zig: during each iteration we now have an access to the array element, but not to the element's index anymore. Just for the sake of illustration, let's modify the code to use such loop, where we won't be able to state at which index we found the element, only whether we found it or not. The Zig version:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    // Perform a linear search for key in data.
    var found = false;
    for (&data) |item| {
        if (item == key) {
            found = true;
            break;
        }
    }

    if (found)
        std.debug.print("Found\n", .{})
    else
        std.debug.print("Not found\n", .{});
}
```

and the C++ version:

```
#include<stdio.h>
#include<array>
#include<optional>

static const int data[] = { -10, -1, 5, 12, 100 };

int main() {
    const int key = 12;

    // Perform a linear search for key in data
    bool found = false;
    for (auto item: data) {
        if (item == key) {
            found = true;
            break;
        }
    }

    if (found)
        printf("Found\n");
    else
        printf("Not found\n");

   return 0;
}
```

▷ The `found` variable in the Zig code is having type `bool`, which could have been explicitly specified as

```
var found: bool = false;
```

With omitted explicit type specification the Zig construct more corresponds to the C++ line

```
auto found = false;
```

but that one looks a bit awkward, so we wrote `bool found = false;` in the C++ code instead.

To restore the ability to identify the index of the found element, we might add a “manual index counter” to the C++ version:

```
bool found = false;
size_t index = 0;
for (auto item: data) {
    .....
    index++;
}
```

We could also do the same in Zig, but there is a better way: Zig’s `for` loop allows simultaneous iteration over two (or more) ranges, provided the ranges have one and the same length. In particular we could write our Zig loop as:

```
for (&data, 0..data.len) |item, i|
```

We could also go one step further and omit the `data.len` from the second range:

```
for (&data, 0..) |item, i|
```

In this case Zig will assume that the second range (for which we didn’t specify the upper bound and which thereby has unspecified length) has the same length as the first one. The resulting Zig code is:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    // Perform a linear search for key in data.
    var found_at: ?usize = null;
    for (&data, 0..) |item, i| {
        if (item == key) {
            found_at = i;
            break;
        }
    }

    if (found_at) |index|
        std.debug.print("Found at {}\n", .{index})
    else
        std.debug.print("Not found\n", .{});
}
```

This kind of `for` loop is pretty idiomatic in Zig. But we can make the code even “more Zig”, and we will continue with that later in Sections 2.1 and 2.7.

## 1.15 Non-unit-step loops

Until now the for loops that we have created all had a step size equal to +1. But what if we want a different step size? Zig’s for loops do not support other step sizes (or a variety of range transformations as in C++’s `std::ranges`). Therefore we have to use while loops instead in that case.

Recall that C/C++'s for loop is merely a shorthand notation for a while loop. E.g.

```
for (int i = 0 ; i < 10 ; i++) {
    .........
}
```

is roughly speaking[10] a shorthand for

```
int i = 0;
while(i < 10) {
    .........
    i++;
}
```

Similarly, in Zig we could rewrite

```
for (0..10) |i| {
    .........
}
```

as

```
var i: usize = 0;
while(i < 10) {
    .........
    i += 1; // there is no i++ in Zig
}
```

A small difference between the two is that the `i` variable will continue being visible after the `while` loop. Another difference is how a `continue` statement is going to work. In the following code the `continue` statement works as intended:

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

pub fn main() void {
    // Print all even numbers between 0 and 10
    for (0..10) |i| {
        const is_even = i % 2 == 0;
        if (!is_even)
            continue; // moves on to the next value of i
        std.debug.print("{}\n", .{i});
    }
}
```

while the following `while` loop hangs for an obvious reason as the `continue` statement skips the index increment:

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

pub fn main() void {
    // Tries to print all even numbers between 0 and 10
```

[10]The fine differences include the visibility of the `i` variable as well as the jump target for the `continue` operator. The visibility of `i` could have been formally addressed by putting an extra block around the code, but we decided against that for the sake of reduced noise.

```
    // but hangs
    var i: usize = 0;
    while (i < 10) {
        const is_even = i % 2 == 0;
        if (!is_even)
            continue; // repeats with the same value of i
        std.debug.print("{}\n", .{i});
        i += 1;
    }
}
```

Zig's while loop has a special syntactical construct to address that:

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

pub fn main() void {
    // Print all even numbers between 0 and 10
    var i: usize = 0;
    while (i < 10) : (i += 1) {
        const is_even = i % 2 == 0;
        if (!is_even)
            continue; // skip to next iteration
        std.debug.print("{}\n", .{i});
    }
}
```

(notice that thereby Zig's while loop syntax turns into something "in the middle" between C/C++'s while and for loops).

Equipped with the latter option, we can rewrite the latter implementation to directly use a step of 2:

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

pub fn main() void {
    // Print all even numbers between 0 and 10
    var i: usize = 0;
    while (i < 10) : (i += 2)
        std.debug.print("{}\n", .{i});
}
```

▷ You can have multiple "continue expressions" in a `while` loop by grouping them into a block:

```
var i: usize = 0;
var j: usize = 0;
while (i < 10) : ({
    i += 1;
    j += 2;
}) {
    std.debug.print("{} {}\n", .{ i, j });
}
```

## 1.16 Negative-step loops

Let's write a downwards loop, reversing the direction of `for (0..10)`:

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

pub fn main() void {
    // Print all numbers from 9 to 0
    var i: i32 = 9;
    while (i >= 0) : (i -= 1)
        std.debug.print("{}\n", .{i});
}
```

Let's now do a slightly more difficult task: print the elements of an array in the descending order. The difficulty is that we wouldn't be able to simply rewrite loop in the previous example using `usize`:

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

pub fn main() void {
    // Tries to print all numbers from 9 to 0
    // but panics or loops forever
    var i: usize = 9;
    while (i >= 0) : (i -= 1)
        std.debug.print("{}\n", .{i});
}
```

as the code will attempt to wrap around at 0. In safe mode builds this will cause a runtime panic due to unsigned integer underflow, unsafe build modes are likely to simply continue the loop. So we must continue using a signed index, e.g:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };

    // Prints array in descending index order,
    // panics (or worse) if data.len == 0
    var i: isize = @intCast(data.len - 1);
    while (i >= 0) : (i -= 1)
        std.debug.print(
            "data[{}] = {}\n",
            .{ i, data[@intCast(i)] },
        );
}
```

Notice how we need to use the `@intCast` twice to convert from `usize` to `isize` and back.

There is a problem with the above code: if the array size is zero, there will be an unsigned underflow on computing `data.len - 1`. We actually have to do a signed subtraction. That is, we have to convert to `isize` before subtracting 1. Trying

```
    var i: isize = @intCast(data.len) - 1;
```

doesn't work: Zig simply doesn't know anymore that we want to convert the value of `data.len` to `isize`, as we don't directly assign the `@intCast`'s result to an `isize` variable. The solution is either to introduce an auxiliary intermediate variable

```
    const ilen: isize = @intCast(data.len);
    var i: isize = ilen - 1;
```

or to use the `@as()` builtin function discussed in Section 1.10:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };

    // Print array in descending index order
    var i = @as(isize, @intCast(data.len)) - 1;
    while (i >= 0) : (i -= 1)
        std.debug.print(
            "data[{}] = {}\n",
            .{ i, data[@intCast(i)] },
        );
}
```

Notice that in the latter version we do not need to specify the type of `i` anymore: Zig will auto-infer the `isize` type for `i`, since we are subtracting 1 from an `isize` value.

If we don't want to use `isize` but prefer sticking to `usize`, we'll have to reorganize the loop. E.g.:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };

    // Print array in descending index order
    var i = data.len;
    while (i > 0) {
        i -= 1;
        std.debug.print(
            "data[{}] = {}\n",
            .{ i, data[i] },
        );
    }
}
```

The downside is that we won't be able to use `continue` to skip iterations, at least not in the usual simple manner.

## 1.17 Nested arrays and loops

The following code negates all elements of a $3 \times 3$ matrix:

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

pub fn main() void {
    const n = 3;
    var a = [n][n]i32{
        .{ 1, 2, 3 },
        .{ 4, 5, 6 },
        .{ 7, 8, 9 },
    };

    // Negate all matrix elements
    for (0..n) |i| {
        for (0..n) |j|
            a[i][j] = -a[i][j];
    }

    std.debug.print("{any}\n", .{a});
}
```

Notes:

- As in C/C++, we use an array of arrays to represent a two-dimensional array.

- We declare `const n = 3;` mostly just to demonstrate the usage of const variables in places where normally const literals would be used. Naturally, the values of such variables must be known at compile time.

- Differently from one-dimensional arrays, this time we cannot auto-infer the array dimensions from the number of the initializer elements option. Or, more precisely, we could still use it for the outer dimension, which still can be inferred automatically, but not for the inner dimension:

  ```
  var a = [_][3]i32{
      .{ 1, 2, 3 },
      .{ 4, 5, 6 },
      .{ 7, 8, 9 },
  };
  ```

- Notice the usage of tuple literals `.{ 1, 2, 3 }` etc. for the initialization of the nested arrays. As explained in a similar case earlier, those are auto converted here to the values of type `[3]i32`, which is the inner array type. Alternatively we could use array literals where we could even auto-infer the size of each literal using `[_]`:

  ```
  var a = [_][3]i32{
      [_]i32{ 1, 2, 3 },
      [_]i32{ 4, 5, 6 },
      [_]i32{ 7, 8, 9 },
  };
  ```

- The word “any” inside the print’s format string is a format specifier. Normally the `std.debug.print` function would refuse to print such relatively complicated data type as an array with an empty format specifier as in `"{}"`. But the explicit use of the `"{any}"` format specifier tells it to do that anyway.

Differently from C/C++, Zig allows breaking an outer loop, while being inside the inner loop:

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

pub fn main() void {
    const n = 3;
    const a = [n][n]i32{
        .{ 1, 2, 3 },
        .{ 4, 5, 6 },
        .{ 7, 8, 9 },
    };

    // Print all matrix elements stopping at 5
    outer: for (0..n) |i| {
        for (0..n) |j| {
            std.debug.print(
                "a[{}][{}]={}\n",
                .{ i, j, a[i][j] },
            );
            if (a[i][j] == 5)
                break :outer;
        }
    }
}
```

Here we used the `outer:` label to label the outer loop and then refer to that label in the `break` statement. Without that label, `break` would break the inner loop.

The `continue` statement also can accept the label:

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

pub fn main() void {
    const n = 3;
    const a = [n][n]i32{
        .{ 1, 2, 3 },
        .{ 4, 5, 6 },
        .{ 7, 8, 9 },
    };

    // Print all matrix elements
    // skipping the rest of the row at 5
    outer: for (0..n) |i| {
        for (0..n) |j| {
            std.debug.print(
                "a[{}][{}]={}\n",
                .{ i, j, a[i][j] },
            );
            if (a[i][j] == 5)
                continue :outer;
        }
    }
}
```

Notice how the value of `a[1][2]` is thereby not printed.

## 1.18 The no-goto theorem

Zig doesn't have a `goto` statement.

> ▷ The approach of using chained `goto`-s to handle errors in a "RAII" manner is covered by Zig's `errdefer` feature. See the discussion in Subsection 4.7.8.

> ▷ This section's title is a word pun referring to the "no-hair theorem" in astrophysics.

# Chapter 2

# Diving in

*Advanced control flow. Pointers. Functions. Further language details.*

## 2.1 Else branches of loops

At the end of Section 1.14 we have come up with an idiomatic Zig way to do a linear search in an array and we promised to continue making this code even "more Zig". One direction in which this could go is to utilize an `else` branch in the loop:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    // Perform a linear search for key in data.
    for (&data, 0..) |item, i| {
        if (item == key) {
            std.debug.print("Found at {}\n", .{i});
            break;
        }
    }
    else {
        std.debug.print("Not found\n", .{});
    }
}
```

The “else” branch is executed in case the loop terminates normally, without being interrupted by a `break` statement.

> ▷ The “else” branch feature is also available with `while` loops, following the same rule: the “else” branch is executed if and only if the loop ran completely without being interrupted by a `break` statement.

The above implementation somewhat deviates from the original idea to obtain the search result in a variable. We shall return to this original approach in Section 2.7.

## 2.2 Void block expressions

A number of constructs which are statements in C/C++ actually can be used as expressions in Zig. Under those are blocks as well as conditional and loop statements.

Blocks (series of statements enclosed in `{ }`) by default return a `void` value. Try to compile the following program:

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

pub fn main() void {
    const block_value = {
        const x = 0;
        std.debug.print("{}\n", .{x});
    };
    @compileLog(block_value);
}
```

and observe the compiler output:

```
Compile Log Output:
@as(void, {})
```

That’s right, `void` values can be copied an assigned in Zig. You could also try e.g.:

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

pub fn main() void {
    var block_value: void = undefined;
    block_value = {
        const x = 0;
        std.debug.print("{}\n", .{x});
    };
}
```

Of course such code doesn't make too much sense, but formally it works. However, occasionally, one might need `void` values with generic programming. In such case `{}` would provide a perfectly usable `void` value, as demonstrated by:

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

pub fn main() void {
    const void_value = {};
    const another_one = void_value;
    @compileLog(void_value, another_one);
}
```

▷ While Zig allows to use `void` (and other zero-size) values more or less freely within the language, some care has to be taken here. In C/C++ we might be used to using the address of an object (usually in the form of object's pointer) as the object's unique identifier: two objects (of the same type) are different if and only if their pointers are different. This no longer holds with zero-size objects, e.g. two different `void` fields of one and the same struct might be located at one and the same address.

## 2.3 Conditional expressions

The `if` statements can also be used as expressions instead. The most common use case of this is implementing the counterpart of C/C++'s ternary operator, e.g.

```
const sign: i32 = if(x >= 0) 1 else -1;
```

corresponds to C/C++'s

```
const int32_t sign = x >= 0 ? 1 : -1;
```

▷ Another formatting accepted by the Zig's formatter (at the time of writing this text) for the above code is

```
const sign: i32 =
    if (x >= 0) 1 else -1;
```

Yet another one is

```
const sign: i32 = if (x >= 0)
    1
else
    -1;
```

These formattings are not too useful for the present example, but might come in handy with longer expressions.

Similarly to the `if` statements, only one of the two branches of an `if` expression is evaluated each time. So in

```
const result: i32 = if (x >= 0)
    process1(x)
else
    process2(x);
```

only one of the two functions `process1` and `process2` would be called each time.

### Capturing if expressions

Like ordinary `if`s, capturing `if`s can also be used as expressions, e.g.:

```
const inner_value =
    if (optional_value) |inner| inner else 0;
```

▷ As mentioned earlier in Section 1.13, the same `if` can be more concisely written as

```
const inner_value = optional_value orelse 0;
```

A capturing `if` expression would be rather more appropriate e.g. in the following case:

```
const inner_value =
    if (optional_value) |inner| doSomething(inner) else 0;
```

### Omitted else branch

We have just discussed in Section 2.2 that `void` is more or less just an ordinary type in Zig, without limitations characteristic of C/C++. This means, in particular, that void values can be returned from an `if` expression. E.g.:

```
const void_value = if (condition) {
    doSomething();
} else {
    doSomethingElse();
};
```

In the above code we are having an `if` expression with both branches returning a `void` value each, as (by the discussion in Section 2.2) the respective blocks produce void values.

A particularly interesting feature of Zig language is that an omitted `else` branch is understood as "`else {}`". In particular, the following code

```
const void_value = if (condition) {
    doSomething();
};
```

is understood as

```
    const void_value = if (condition) {
        doSomething();
    } else {};
```

This might look just as a useless peculiarity, but actually it does have a number of uses in the language, as we shall see in Sections 2.7 and 5.7.

> ▷ At this point the reader might also ask, why are we talking separately of `if` statements and `if` expressions? Aren't `if` statements just a particular case of usage of `if` expressions of type `void`? They kind of are, but Zig language's grammar treats them differently, so we follow the same pattern.
>
> As we shall see further, the majority of Zig statements can be used as expressions. However not all of them. E.g. the assignment statement can not:
>
> ```
>     var x: i32 = undefined;
>     const value = (x = 0); // does not compile
> ```

**l-value results**

> ▷ Zig doesn't really "officially" have a concept of l-values and r-values, like C. We are loosely using these terms for the sake of intuitive explanation.

The conditional expressions in Zig, like in C++, can return l-values. The following Zig code:

```
    var a: i32 = 0;
    var b: i32 = 0;
    (if (condition) a else b) = 1000;
```

is a counterpart of C++'s

```
    int a = 0, b = 0;
    (c ? a : b) = 1000;
```

This feature is not restricted to conditional expressions in Zig, but is also available for other kinds of expressions: switches, loops and blocks (including the values returned by `break` statements).

## 2.4 Compile time-known conditions

We have been already mentioning earlier in the discussion of assertions and the `std.debug.runtime_safety` constant in Section 1.9, that the `if`s, whose conditions are known at compile time, compile only active branches and fully discard the inactive branches (except for basic syntactic parsing). This has an implication, that we can return two values of completely different types from the 'then' and 'else' branches of such `if`, leading to interesting effects. E.g.:

```
pub fn main() void {
    const i: i32 = 0;
    const b: bool = false;

    const comptime_condition = true;
```

```
    const value =
        if (comptime_condition) i else b;

    @compileLog(value);
}
```

Depending on the value of the `compile_time_known_condition`, the `value` variable will have either integer or boolean type.

Such effects can be sometimes useful in generic programming, and we shall discuss this topic further in Section 5.7, however otherwise such cases should be rather avoided. The main reason for this is that, unless the condition is conceptually expected to be known at compile time[1], returning two different incompatible types from the two branches is a logical error. Latest, as the condition turns from effectively-compile-time-known to a runtime-known in the process of code modification, the code will break, since a runtime `if` must produce a value of a single type, regardless of which branch is taken. E.g. the following modification of the previous code would fail to compile:

```
var runtime_condition = true;

pub fn main() void {
    const i: i32 = 0;
    const b: bool = false;

    // fails to compile
    const value =
        if (runtime_condition) i else b;

    @compileLog(value);
}
```

▷ By placing the `runtime_condition` as a global-scope `var` we avoid the "unmutated var" error, which would have occurred had we declared the `var` as a local variable. We cannot use `const` here, since in that case Zig would consider the condition to be known at compile time (as long as the `const` is initialized by a literal or another compile-time-known-value expression).

▷ The term "global scope" used above is actually a misnomer. Zig doesn't have a concept of global scope but rather of a "file scope" or "file-scope namespace". The latter will be discussed around Sections 3.8 and 3.9. For the time being for simplicity and as long as we are dealing with single-file programs in Zig, we can think of the file scope as if it was C/C++'s global scope.

As a way to mitigate the unintended effects of different-type branches, it might be helpful to explicitly specify the result type, e.g:

```
var runtime_condition = true;

pub fn main() void {
    const one: i32 = 1;
```

[1]In which case it might be a good idea to explicitly enforce it by using the `comptime` keyword inside the `if`'s condition: `if (comptime condition) ...`

```
    const zero: u8 = 0;

    // Explicitly state that we want an i32 result
    const value: i32 =
        if (runtime_condition) one else zero;

    @compileLog(value);
}
```

The explicit result type specification will not prevent potentially erroneous code from breaking upon compile-time condition becoming runtime, but it will at least make sure that the issue is local to the `value` initialization. Explicitly specified result type can also help generally to make sure that the result type is exactly the intended one.

## 2.5 Peer type resolution

We could notice that in the latter example the `i32` and `u8` types of the `one` and `zero` variables both have integer types, and furthermore that an `i32` value does accommodate the values of both types. So one could in principle expect the Zig compiler to be able to automatically deduce `i32` as the conditional expression's result type without requiring to explicitly specify the type.

At the time of writing this text the Zig compiler is indeed capable of doing this, and the conditional expression in the following code successfully compiles:

```
var runtime_condition = true;

pub fn main() void {
    const one: i32 = 1;
    const zero: u8 = 0;

    const value =
        if (runtime_condition) one else zero;

    @compileLog(value);
}
```

This feature of the Zig language is referred to as *peer type resolution* or, shortly, *PTR*. We have however seen earlier, that omitting the explicit type specification can have undesired implications in case of erroneous code. It might be advisable to decide on a case-by-case basis, whether one wants to use PTR or not.[2]

▷ Peer type resolution is also involved in "more innocently looking code", e.g.

```
pub fn main() void {
    const one: i32 = 1;
    const zero: u8 = 0;

    const value = one + zero;
```

[2]To the author's knowledge, at the time of the writing of this text there are also considerations by the Zig core team to drastically reduce the PTR functionality in the language. See Zig issue 22182.

```
    @compileLog(value);
}
```

where `value` is deduced to be of `i32` type.

▷ There are some interesting special cases in Zig, where peer type resolution (or at least a similarly looking mechanism) is involved. Specifically Zig's `@min` and `@max` builtin functions have a special resolution rule. Consider the following code:

```
pub fn main() void {
    var n: i32 = 1000;
    _ = &n; // avoid unmutated var error
    const clipped = @max(n, 0);
    @compileLog(clipped);
}
```

Contrary to what a C/C++ developer might expect, the type of `clipped` is `u31`. While originally the value of `n` would be in the range between $-2^{31}$ and $2^{31}-1$, the compiler realizes that negative values would be clipped by `@min` and thus the value of `clipped` will fit into the range between 0 and $2^{31}-1$.

Similarly, in the following code the type of `clipped` is `u7`, since the resulting value lies between 0 and 100.

```
pub fn main() void {
    var n: u32 = 1000;
    _ = &n; // avoid unmutated var error
    const clipped = @min(n, 100);
    @compileLog(clipped);
}
```

## 2.6 Nonvoid block expressions

Blocks do not have to return void values. We could return a value from a block by labeling a block and using a `break` statement with a label:

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

pub fn main() void {
    const block_value = blk: {
        const x: i32 = 0;
        break :blk x + 1;
    };
    std.debug.print("{}\n", .{block_value});
}
```

The inintialization code of the `block_value` variable in the above is pretty much equivalent to

```
    const x: i32 = 0;
    const block_value = x + 1;
```

except that the `x` variable's scope is limited to the block.

A bit more complicated situation arises in the following code:

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

var use_float = false;

pub fn main() void {
    const block_value: f32 = blk: {
        if (use_float)
            break :blk 0.25;
        break :blk 0;
    };
    std.debug.print("{}\n", .{block_value});
}
```

This time the intialization code of the `block_value` can be rather equivalently rewritten as

```
const x: i32 = 0;
const block_value: f32 = if (use_float) 0.25 else 0;
```

We thus are facing a peer type resolution situation between float and integer literals (having `comptime_float` and `comptime_int` types respectively). For this reason we rather need an explicit type specification for the `block_value` variable. This explicitly specified `f32` type is then propagated by the compiler to the respective `break` statements.

▷ Instead of specifying the type of `block_value` variable, we could also use the `@as` builtin to give both values one and the same runtime type:

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

var use_float = false;

pub fn main() void {
    const block_value = blk: {
        if (use_float)
            break :blk @as(f32, 0.25);
        break :blk @as(f32, 0);
    };
    std.debug.print("{}\n", .{block_value});
}
```

This kind of non-void-type blocks is used occasionally in Zig code. As a bit contrived example, let's imagine that we have an integer variable `n` and then we want to initialize another integer variable `f` to the factorial of `n`. So we might imagine a piece of code like:

```
const n: u32 = 5;
const f = factorial(n);
```

where we also need to additionally implement the function `factorial()`. Suppose however that we do not want to implement the factorial computation in a separate function. But we still want to retain the above code structure, where we directly initialize `f` with a factorial of `n` (remember, our example is contrived). Then we could use the following code:

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

pub fn main() void {
    const n: u32 = 5;
    const f = blk: {
        var prod: u32 = 1;
        for (1..(n + 1)) |i|
            prod *= @intCast(i);
        break :blk prod;
    };
    std.debug.print("{}\n", .{f});
}
```

## 2.7 Loop expressions

Upon a first sight loops cannot not have a return value, but in C/C++ neither had blocks. The blocks attain a return value via the usage of a (labeled) `break` statement (as discussed in Section 2.6) and then why couldn't the same be applied to loops?

Let's consider a program which would return the first negative element of an array. We could imagine something along the lines of:

```
    const data = [_]i32{ 5, 12, -10, -1, 100 };
    const first: i32 = for (&data) |item| {
        if (item < 0)
            break item;
    };
```

however, if we try to compile such code, we would obtain an error:

```
    expected type 'i32', found 'void'
```

Where does this 'void' come from? It comes from the possibility that the array doesn't contain any negative values, so the `break` statement is never executed and the loop runs to the end.

Now recall our discussion in Section 2.1 that loops can have "else" branches, and also recall what we said in Section 2.3, that omitted "else" branches imply "`else {}`". The latter applies not only to `if` statements, but also to loops. So the loop in our previous code actually reads

```
    const first: i32 = for (&data) |item| {
        if (item < 0)
            break item;
    } else {};
```

This is where the `void` type is coming from.

It should be now also clear how to fix the problem. E.g.

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

pub fn main() void {
    const data = [_]i32{ 5, 12, -10, -1, 100 };
    // Find the first negative element's value
    const first = for (&data) |item| {
        if (item < 0)
            break item;
    } else 0;

    std.debug.print("Found {}\n", .{first});
}
```

where we return 0 if no negative values are found.

▷ By now we could realize that a `break` statement without a supplied value implies a `break` statement returning a void value: `break {};`

Using the just discussed language features we could now make a further improvement of the array linear search code from the end of Section 1.14 by directly returning the value from the loop rather than manually assigning it to the `found_at` variable:

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    // Perform a linear search for key in data.
    const found_at = for (&data, 0..) |item, i| {
        if (item == key)
            break i;
    } else null;

    if (found_at) |index|
        std.debug.print("Found at {}\n", .{index})
    else
        std.debug.print("Not found\n", .{});
}
```

The `found_at` variable thereby has an inferred type `?usize`, as can be checked using a `@compileLog` statement.

▷ `while` loops can be used as expressions in a similar way.

## 2.8 Array concatenation

In C/C++ one can concatenate string literals. Zig goes a bit further and defines two special operations on arrays in general, `++` and `**`. These operations, among other things, can cover string concatenation (which will be discussed in Section 2.19), but also have other uses. E.g. they are occasionally useful in the construction of array initialization values. They are also pretty handy in certain aspects of compile-time programming.

The `++` operation concatenates two arrays together:

```
const sequence = [_]i32{ 1, 2 } ++ [_]i32{ 3, 4, 5 };
// sequence is equal to [5]i32{ 1, 2, 3, 4, 5 }
```

The `**` operation takes an array as its left-hand-side argument and a *compile-time-known* integer as its right-hand-side argument and concatenates the given number of copies of the array together. E.g.

```
const one_zero = [_]i32{0};

const five_zeros = one_zero ** 5;
// five_zeros is equal to [5]i32{ 0, 0, 0, 0, 0 }

const alternating = [_]i32{ 1, 2 } ** 3;
// alternating is equal to [6]{ 1, 2, 1, 2, 1, 2 }

const sequence = [_]i32{ 1, 2 } ++ [_]i32{0} ** 3;
// '**' has higher precedence than '++',
// therefore sequence is equal to [5]i32{ 1, 2, 0, 0, 0 }
```

▷ Don't get too tempted to use `**` to initialize large-size arrays, e.g.:

```
const array = [_]i32{0} ** 1_000_000;
```

This will put an actual array of one million zeros into the program code. It's usually better to initialize large arrays using a runtime loop.

▷ The `++` and `**` are available not only for arrays, but for any value that supports the array indexing operation [ ] and publishes a compile time-known `len` field. That includes pointers-to-arrays, slices with compile-time-known lengths as well as tuples. The produced result type will vary following the types of the arguments.

## 2.9 Single-item pointers

Consider the following Zig code:

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

pub fn main() void {
    var value: i32 = 0;
    const pointer = &value;
    pointer.* = 1;

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

The loosely corresponding C++ code is

```
#include<stdio.h>

int main() {
    int value = 0;
    const auto pointer = &value;
    *pointer = 1;

    printf("value = %d\n", value);
    return 0;
}
```

(where we used `int` type instead of `int32_t` for simplicity).

The type of `pointer` const variable in the Zig code is inferred to be `*i32` (which means “pointer to `i32`”), as can be verified by using a `@compileLog` function. We could also specify this type explicitly if we wanted to:

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

pub fn main() void {
    var value: i32 = 0;
    const pointer: *i32 = &value;
    pointer.* = 1;

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

with the C++ counterpart being

```
#include<stdio.h>

int main() {
    int value = 0;
    int *const pointer = &value;
    *pointer = 1;

    printf("value = %d\n", value);
    return 0;
}
```

If we take a pointer to a constant value, we shall find that the pointer type is now `*const i32`:

```
pub fn main() void {
    const value: i32 = 0;
    const pointer = &value;
    @compileLog(pointer);
}
```

as per `@compileLog`’s output. Thus we could do e.g. the following:

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

pub fn main() void {
    const value: i32 = 0;
    const pointer: *const i32 = &value;

    std.debug.print("value = {}\n", .{pointer.*});
}
```

the C++ counterpart being

```
#include<stdio.h>

int main() {
    const int value = 0;
    const int *const pointer = &value;

    printf("value = %d\n", value);
    return 0;
}
```

▷ In the latter C++ example the type of `value` is `const int` and the type of `pointer` is respectively `const int *const`, so the constness of the variable itself is also encoded into variable's type.

In Zig this is not the case. The type of `value` is simply `i32`. Similarly, the type of `pointer` is `*const i32`, that is the type says that it's a pointer *to a constant value*, but doesn't say anything about the pointer variable itself being const.

The kind of Zig pointers that we have introduced in this section are referred to as *single-item pointers*. They offer only a subset of functionality associated with pointers with C/C++. In particular

- Broadly speaking, single-item pointer values are created only by applying the address operation '`&`' to the "items" (unless you use type casting techniques, which would allow to convert some other pointer or integer value into a single-item pointer). You cannot modify these values further, in particular you cannot do the usual C/C++ pointer arithmetic on those: increment them, decrement them, or add or subtract an integer value to/from them.[3]

- The only way to "use" a single-item pointer value (short of assigning it to another pointer variable) is to dereference it. You cannot apply the array indexing operation `[ ]` to single-item pointers.

  Notice that the latter also makes total sense, considering that the operation `a[i]` is defined in C/C++ as `*(a+i)` and we just said that the usual pointer arithmetic is not available for single-item pointers.

- A single-item pointer value cannot be null. Actually none of the standard kinds of pointers in Zig (single-item, many-item or slice) can be null. Nullable C/C++ pointers are represented in Zig by combining pointers with Zig's optional feature as discussed in Section 3.1.

In a way, Zig's single-item pointers are closer to C++ references than to C/C++ pointers, the most prominent differences between those being:

- You can change single-item pointer variables by assignment, while C++ references cannot be changes once initialized.

- C++ references are automatically dereferenced, while Zig pointers usually (with some exceptions) need to be dereferenced explicitly.

[3]As a kind of exception to that, you can subtract one single-item pointer from another, obtaining a `usize` result.

Further kinds of Zig pointers, offering other aspects of C/C++ pointer functionality, are discussed later in the text.

▷ Like in C++, pointers to different data types are not compatible to each other:

```
pub fn main() void {
    var i: i32 = 0;
    const p = &i; // *const i32
    const p1: *const u32 = p; // does not compile
    .....
}
```

▷ As `void` is a pretty much "normal type" in Zig, single-item pointers to void (denoted as `*void` and `*const void`) are not supposed to be used for type erasure. For the latter purpose Zig introduces pointers to `anyopaque` type: `*anyopaque` and `*const anyopaque`.

▷ It is possible to apply the address operation "`&`" to "rvalues". E.g.

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

pub fn main() void {
    // Reportedly, loosely equivalent to:
    //     const time = std.time.timestamp();
    //     const ptr = &time;
    // where std.time.timestamp() returns an i64 value
    const ptr = &std.time.timestamp();

    std.debug.print("{}, {}\n", .{
        ptr.*,
        std.time.timestamp(),
    });
}
```

The author could not find an explicit documentation of this behavior, but it seems to take an address of an implicitly created const, which exists till the end of the current scope if not till the end of the entire function.

## 2.10 Functions

Now that we have introduced single-item pointers, we might be in a good position to get acquainted with Zig's functions.

Suppose we are having the following C++ function which we would like to translate into Zig:

```
bool copySign(double from, double &to) {
    bool sign_from = from >= 0;
    bool sign_to = to >= 0;

    if(sign_to == sign_from)
        return false;
    to = -to;
    return true;
}
```

Before we convert the above code to Zig, let's first convert it to C, replacing references with pointers:

```
bool copySign(double from, double *to) {
    bool sign_from = from >= 0;
    bool sign_to = *to >= 0;

    if(sign_to == sign_from)
        return false;

    *to = -*to;
    return true;
}
```

Now the latter can be converted into Zig as follows (where we also include the entire program code to try the function out):

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

fn copySign(from: f64, to: *f64) bool {
    const sign_from = from >= 0;
    const sign_to = to.* >= 0;

    if (sign_to == sign_from)
        return false;

    to.* = -to.*;
    return true;
}

pub fn main() void {
    const src: f64 = -1.0;
    var dest: f64 = 2.0;
    const changed = copySign(src, &dest);
    std.debug.print(
        "Result={}, changed={}",
        .{ dest, changed },
    );
}
```

Notes:

- It should be pretty easy to identify the semantics of the Zig's function header's elements by comparing the header to the C version.

- The arguments (`from` and `to`) declared in the Zig's function header behave as const vars inside the function body:

  ```
  const from: f64 = first_passed_value;
  const to: *f64 = second_passed_value;
  ```

This is similar to the situation when you declare the function's parameters as const in C++:

```
bool copySign(const double from, double &to) {
    // We only need to declare 'from' as const,
    // since 'to', being a reference, is already const
    .....
```

There is no way to make Zig function's arguments non-const. If you want an argument to be a mutable `var` instead, you have to create a new local variable and store a copy of the argument's value there:

```
fn copySign(from: f64, to: *f64) bool {
    var mutable_from = from;
    .......
```

▷ In case of permanently (rather than temporarily) unused function parameters, you don't need to assign those parameters to a '_' placeholder in order to avoid the "unused parameter" error, but can simply use '_' as the parameter name. E.g.

```
// Both function parameters are unused
fn func(_: i32, _: u8) void {}
```

▷ In Zig there is no function overloading for different-signature functions like there is in C++. The following does not work:

```
fn func(_: usize) void {}
fn func(_: *i32) void {} // error: duplicate function name
```

You must use different function names (as in C).

## 2.11 Pass-by-reference optimization

▷ The *pass-by-reference optimization* in Zig is planned to undergo significant changes, it is supposed to become significantly more conservative. See Zig issue 5973 and the September 2024 discussion in particular.

In the `copySign` function discussed in Section 2.10 the first argument (named `from`) looks as if it is passed by value in the Zig version, same as it was in the C++ version. That is kind of correct, but Zig language reserves the right, at the compiler's discretion, to pass such "by-value" arguments by const reference instead (using single-item pointers).[4] That is, there is a possibility, that what actually has been compiled by the Zig compiler rather corresponds to the following code:

[4]Here, by saying "passing arguments by reference" we mean the general idea of an argument passing mechanism where an address of the argument is being passed instead of the argument's value. We do not mean the specific form of this approach implemented by C++'s references. C's and Zig's argument passing via pointers are equally specific forms of this mechanism and can be also referred to as "passing by reference".

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

fn copySign(from: *const f64, to: *f64) bool {
    const sign_from = from.* >= 0;
    const sign_to = to.* >= 0;

    if (sign_to == sign_from)
        return false;

    to.* = -to.*;
    return true;
}

pub fn main() void {
    const src: f64 = -1.0;
    var dest: f64 = 2.0;
    const copied = copySign(&src, &dest);
    std.debug.print(
        "Result={}, copied={}",
        .{ dest, copied },
    );
}
```

This is referred to as *pass-by-reference optimization* or shortly *PRO*.

The compiler is expected to attempt this optimization for relatively large-size argument types, where passing an argument by value would be considered more expensive than passing the same argument by reference. So the compiler probably wouldn't do it for an `f64` argument, which is rather small, but one still shouldn't rely on that. In particular, at the time of this writing, the Zig compiler seems to apply this optimization for structs, regardless of their size.

Zig language used to recommend against trying to manually enforce this optimization (by manually passing the arguments by const reference). Instead it was recommended to always try to pass arguments by value, unless an argument is semantically meant to be passed by a reference. However at the time of this writing it seems that the whole PRO topic is being reconsidered. In the meantime, until the topic becomes resolved, it might be a good idea to manually pass larger arguments by a const reference. At any rate, one should still keep in mind that an argument passed by value, might be compiled into a passed-by-reference one.

Here are some example cases to consider with respect to the PRO topic:

- In the following code you probably do not want to pass by value in any case, as you need to know the address of the actual parameter in the function code:

  ```
  // Passing by value would be semantically incorrect here
  fn getNext(item: *const Item) *const Item {
      return if(item == &global_sentinel)
          item // don't traverse any further
      else
          item.next;
  ```

```
}
```

- As PRO gets more conservative, you might get more cautious about doing things like:

  ```
  // Maybe rather pass by const reference
  fn computeHash(huge_object: HugeObject) u32 {
      ........
  }
  ```

- Conversely, if you are not sure that the optimizer is conservative enough and might convert your passed-by-value parameter to one passed by reference, where it is not really acceptable, you still could enforce the copy of the parameter value:

  ```
  fn reallyReceiveByValue(passed_by_value: SomeType) void {
      // Using 'enforced_copy' instead of 'passed_by_value'
      // in the function's code ensures we are using a copy.
      const enforced_copy = passed_by_value;
      .......
  }
  ```

▷ The above examples are primarily offered as “food for consideration”. To get proper details about PRO and its current state one might have to check Zig language development resources, such as the issue tracker etc.

## 2.12 File-scope declarations

In our Zig example code in Section 2.10 we have placed the `copySign` function above the `main` function. This is actually unnecessary: in Zig the order of declarations at the file scope does not matter, so `copySign` also could have been placed below `main`:

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

pub fn main() void {
    const src: f64 = -1.0;
    var dest: f64 = 2.0;
    const changed = copySign(src, &dest);
    std.debug.print(
        "Result={}, changed={}",
        .{ dest, changed },
    );
}

fn copySign(from: f64, to: *f64) bool {
    const sign_from = from >= 0;
    const sign_to = to.* >= 0;

    if (sign_to == sign_from)
```

```
        return false;

    to.* = -to.*;
    return true;
}
```

No forward declaration required here (actually Zig simply doesn't have forward declarations, since they are not necessary).

The same applies to const and var declarations placed at the file scope: their initialization expressions can reference each other in any order, e.g.

```
const var1 = const2 - 1; // infers i32 type
const const2: i32 = 2;

pub fn main() void {
    // We need to reference var1 to make the var1
    // declaration analyzed by the compiler, which would
    // make const2 declaration analyzed by the compiler
    _ = var1;
}
```

This comes at a cost: the initialization expressions of file-scope consts and vars must have compile-time known values. E.g. the following doesn't compile:

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

var time = std.time.timestamp(); // doesn't compile

pub fn main() void {
    _ = time;
}
```

The rationale behind this is likely related to Zig placing high value on code readability. Zig generally tries avoid any kind of implicitly evaluated code, and runtime-initialized file scope declarations would require implicitly generated initialization code. The code would thereby become hard to read and to debug, and can also create a dependency mess, which is a rather well-known issue with initialization of global variables in C++.

Conversely, function-local consts and vars accept runtime initialization expressions, but do not allow out-of-order referencing of each other, so that their initialization has a defined order, matching their declaration order in the program text:

```
pub fn main() void {
    const a: i32 = b + 1; // doesn't compile, b is declared later
    const b: i32 = 0;
    ........
}
```

▷ By our discussion, the declaration of the `std` const could also have been placed anywhere within the file scope, e.g.

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

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

It is just somewhat common to place it at the top, similarly to the C/C++'s `#include` statements.

You can even have referencing loops (to a certain reasonable extent) formed by declarations which are cross-referencing each other. E.g.

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

fn func1(level: i32) void {
    std.debug.print("func1 {}\n", .{level});
    func2(level);
}

fn func2(level: i32) void {
    std.debug.print("func2 {}\n", .{level});
    if (level < 5)
        func1(level + 1);
}

pub fn main() void {
    func1(0);
}
```

On the other hand the following kind of cross-referencing is impossible, as it doesn't really make sense:

```
const const1: i32 = const2 + 1;
const const2: i32 = const1 + 1;

pub fn main() void {
    _ = const1;
}
```

▷ As we shall learn in Section 3.8, file scope in Zig is actually struct scope. So what we discussed in this section does apply to struct scope declarations (as well as to scopes of struct-like entities, such as enums and unions).

## 2.13 Noreturn expressions

Consider the following Zig program:

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

var runtime_condition = true;

pub fn main() void {
    const one: i32 = 1;
    const value = if (runtime_condition)
```

```
        one
    else
        return;

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

This program prints either "1" or nothing, depending on what we initialize the `runtime_condition` variable to. But what is the result type of the `if` statement? According to the material of Section 2.5, Zig should perform peer type resolution between `i32` and what? It's not like the return statement (or even, rather, the *return expression*, since it's being used inside an `if` expression) has a result type?

Actually, in Zig it does. The type is `noreturn` (indicating that the expression doesn't return a value to its "caller"). Quite naturally, this type is ignored in peer type resolution and the result of the `if` expression is simply `i32` as one can check using a `@compileLog` statement.

One could now guess what kind of other Zig statements could be also available as `noreturn`-type expressions. Those would be the statements which could be useful inside other expressions (e.g. inside `if` expressions) and which would never yield control back to the "callsite", that is to the point where the expression's value is normally received.

Most prominently, one could think of `break` and `continue` statements in this respect. E.g.:

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

pub fn main() void {
    for (0..10) |i| {
        // Peer-type resolution between
        // usize and noreturn yields usize.
        const value = if (i < 5)
            i // usize value
        else
            break; // noreturn value
        std.debug.print("value={}\n", .{value});
    }
}
```

A slightly less obvious case of the same nature is Zig's `unreachable`.

Blocks unconditionally yielding control to the outside are also `noreturn`, as demonstrated e.g. by

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

var runtime_condition = true;

pub fn main() void {
    const one: i32 = 1;
    const value = if (runtime_condition)
        one
    else {
        return;
    };

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

A slightly more complicated example of a similar nature is here:

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

var runtime_condition = true;

pub fn main() void {
    const one: i32 = 1;
    const value = if (runtime_condition)
        one
    else for (0..10) |i| {
        std.debug.print("Looping {}\n", .{i});
    } else return;

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

Another `noreturn` possibility, which might not immediately occur to one, is loops which are known to be endless:

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

var runtime_condition = true;

pub fn main() void {
    const one: i32 = 1;
    const value = if (runtime_condition)
        one
    else while (true) {
        std.debug.print("Looping...\n", .{});
    };

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

▷ The following example demonstrates a `noreturn` expression pattern which is idiomatic in Zig:

```
fn func(optional_argument: ?i32) i32 {
    const value = optional_argument orelse return 0;

    // do something with value
    ......
}
```

This pattern is also rather common with the `catch` operator, which is similar to `orelse` (the `catch` operator is introduced in Section 4.7.4).

▷ It is possible to declare functions of `noreturn` type:

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

// A void return type would be also accepted by the compiler,
// but noreturn is more correct
fn loopForever() noreturn {
    while (true) {
        std.debug.print("Looping...\n", .{});
    }
}

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

## 2.14 Capturing by reference

The Zig captures introduced in the discussions of loops and optionals in Chapter 1 are "captures by value" (where we are drawing an obvious analogy to function arguments passed by value), in C++ terms corresponding to the `const auto capture = value;` kind of declaration. Single-item pointers can be used to achieve "captures by reference", in C++ terms corresponding to the `auto &capture = value;` kind of declaration. The latter are created by placing an asterisk in front of the capture's name. E.g. we use capturing by reference in the following code to modify the value stored in the optional from 1 to 2:

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

pub fn main() void {
    var optional: ?i32 = 1;
    if (optional) |*capture|
        capture.* = 2;

    std.debug.print(
        "optional = {any}\n",
        .{optional},
    );
}
```

▷ If we wanted to do the same assignment unconditionally, we could simply

```
    optional.? = 2; // panics if optional == null
```

or

```
    optional = 2; // completely overrides the optional
```

Capturing by reference is the standard way to modify arrays in a loop. E.g. the following code initializes an array with values from −2 to 2:

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

```
pub fn main() void {
    var array: [5]i32 = undefined;
    for (&array, 0..) |*item, i|
        item.* = @as(i32, @intCast(i)) - 2;

    std.debug.print(
        "array = {any}\n",
        .{array},
    );
}
```

while the following code performs an element-by-element copying from one array into another:

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

pub fn main() void {
    const src = [_]i32{ 5, 12, -10, -1, 100 };
    var dest: [5]i32 = undefined;
    for (&dest, &src) |*d, *s|
        d.* = s.*;

    std.debug.print(
        "dest = {any}\n",
        .{dest},
    );
}
```

▷ In the latter copying example the (auto inferred) type of the `d` capture is `*i32`, while the type of the `s` capture is `*const i32`, since the latter is being captured from a const array.

Actually, we probably could have captured ‘`s`’ by value, since it’s a small-size value (32-bit integer) anyway, we captured both ‘`s`’ and ‘`d`’ by reference more for the sake of demonstration.

▷ If an array is “passed by value” into the loop (which we don’t recommend generally), capturing by reference is not available. The following code produces an error:

```
const array = [_]i32{ 5, 12, -10, -1, 100 };
for (array) |*item| // capturing by reference not possible
    _ = item.*;
```

## 2.15 Pointers to arrays

Suppose we wanted to pass an array as an argument into a function. E.g. imagine we wanted to put the linear search code developed at the end of Section 2.7 into a function.

In C/C++ we might have considered passing a pointer to the array beginning and the array size to such function. E.g.

```
#include<stdio.h>
#include<array>

int find(int key, const int *array, size_t size) {
    for (int i = 0 ; i < (int)size ; i++ ) {
        if (array[i] == key)
            return i;
    }
    return -1;
}

static const int data[] = { -10, -1, 5, 12, 100 };

int main() {
    const int key = 12;

    int found_at = find(key, data, std::size(data));

    if (found_at >= 0)
        printf("Found at %d\n", found_at);
    else
        printf("Not found\n");

    return 0;
}
```

If we tried to convert the latter code to Zig, we would have a difficulty. Translating the `const int *array` argument declaration from C/C++ into Zig as `array: *const i32` we will find out that the resulting argument is not very usable. We cannot pass an array to this argument, nor can we use the argument itself as an array Actually, there should be no wonder: after all it's a pointer to `i32`, not a pointer to an array. It's rather a specific feature of the C/C++ languages that a pointer to integer can also be used as a pointer to array of integers.

If a pointer to an integer doesn't work, why don't we go in a straightforward way and simply pass a single-item pointer to the entire array? We could do it as follows:

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

fn find(key: i32, array: *const [5]i32) ?usize {
    for (array, 0..) |item, i| {
        if (item == key)
            return i;
    }
    return null;
}

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;
    const found_at = find(key, &data);

    std.debug.print("Found at {any}\n", .{found_at});
}
```

Notes:

- We do not place the address operator in front of the `array` variable in the loop header `for (array, 0..)`, since `array` is already a pointer to an array.

- Just to simplify the code a bit, we no longer use a capturing `if` to print the value of `found_at`, but utilize an "any" formatter to print the entire optional in one go.

- Differently from the C++ prototype, our function is restricted to arrays of size 5. In order to support arrays of arbitrary sizes, (short of using generics) we have to use slices, which are discussed in Section 2.17.

▷ Single-item pointers to arrays have a number of built-in shortcuts allowing to use them in many situations as if they were simply arrays. In particular we could rewrite the `find` function's implementation as

```
fn find(key: i32, array: *const [5]i32) ?usize {
    for (0..array.len) |i| {
        if (array[i] == key)
            return i;
    }
    return null;
}
```

Notice the usage of the `[ ]` operator and the `.len` implicit field accessor, as if `array` was an array rather than a single-item pointer to an array. We can do this instead of having to explicitly dereference the array pointer each time as in the following code:

```
fn find(key: i32, array: *const [5]i32) ?usize {
    for (0..array.*.len) |i| {
        if (array.*[i] == key)
            return i;
    }
    return null;
}
```

## 2.16 Many-item pointers

We have discussed how single-item pointers implement only a subset of C/C++ pointer functionality basically needed for accessing a single item. Zig's *many-item pointers* implement the other aspect of C/C++ pointer functionality: pointing to or into an array. Using many-item pointers we could reimplement the C++ code at the beginning of Section 2.15 as follows:

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

fn find(key: i32, array: [*]const i32, len: usize) ?usize {
    for (0..len) |i| {
        if (array[i] == key)
            return i;
    }
    return null;
}

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    const found_at = find(key, &data, data.len);

    std.debug.print("Found at {any}\n", .{found_at});
}
```

Notes:

- Many-item pointers do not support single-item functionality. Specifically they cannot be dereferenced using `.*` like single-item pointers. On the other hand, one can apply the array indexing operation `[ ]` to these pointers. One can also add and subtract integers to/from such pointers.
- Similarly to single-item pointers, many-item pointers cannot be null.

▷ Many-item pointers are not too widely used in Zig, since they represent a view of unknown length (into some array). *Slices* (discussed in Section 2.17) are usually better suited for implementing a view into an array.

▷ We have seen how different aspects of a C/C++ pointer are distributed between single- and many-item Zig pointers. This is done on purpose, because usually different kinds of entities would be pointed to by the pointers of the two respective kinds. However when interfacing to C/C++ code (or maybe routinely translating from C/C++ to Zig) it might occasionally become cumbersome to handle distinctions between single- and many-item pointers. E.g. a C API function might have a pointer parameter (or return value) which might, depending on runtime conditions, refer to a single entity or to an array of entities. For such exceptional cases Zig offers a pointer type which is closer to C/C++, referred to as *C pointers* in Zig. Those combine the features of single- and many-item pointers and are denoted using `[*c]` instead of `[*]`. E.g. a C pointer to a 32-bit is denoted as `[*c]i32`.
One is not supposed to use C pointers routinely in Zig.

## 2.17 Slices

Many-item pointers are used somewhat rarely in Zig, since they don't carry around the information about the bounds of the pointed-to array memory. Instead Zig offers a special "fat pointer" type called *slice*, which is Zig's counterpart of C++'s `std::span`.

Let's return to the C/C++ implementation of the `find` function in Section 2.15 and rewrite it using `std::span`:

```
#include<stdio.h>
#include<array>
#include<span>

int find(int key, const std::span<const int> array) {
    for (int i = 0 ; i < std::ssize(array) ; i++ ) {
        if (array[i] == key)
            return i;
    }
    return -1;
}

static const int data[] = { -10, -1, 5, 12, 100 };

int main() {
    const int key = 12;

    int found_at = find(key, data);

    if (found_at >= 0)
        printf("Found at %d\n", found_at);
    else
        printf("Not found\n");

    return 0;
}
```

While `std::span` is a C++ standard library feature, slices are a fundamental feature of Zig. Using slices we can translate the above C++ code into Zig as:

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

fn find(key: i32, array: []const i32) ?usize {
    for (array, 0..) |item, i| {
        if (item == key)
            return i;
    }
    return null;
}

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    const found_at = find(key, &data);

    std.debug.print("Found at {any}\n", .{found_at});
}
```

Notes:

- The only change compared to the Zig code from Section 2.15, which was using a single-item pointer to an array, is that, instead of defining the `array` argument as a pointer to a const array of 5 integers, we now have defined the `array` argument as a *slice* of const integers. A slice of constant `i32` elements is denoted by `[]const i32` type in Zig.

- Noticing that we are passing a pointer-to-array expression `&data` for an argument expecting a slice `[]const i32`, we must conclude that pointers to arrays should probably also be assignable to slice variables. Indeed, the following code compiles:

  ```
  const array = [_]i32{ -10, -1, 5, 12, 100 };
  const slice: []const i32 = &array;
  ```

  Actually, when we supply a pointer to array (as we usually do) into a `for` loop, technically this is considered as iterating over a slice. That is

  ```
  const array = [_]i32{ -10, -1, 5, 12, 100 };
  for (&array) |item|
      .....
  ```

  is actually understood as

  ```
  const array = [_]i32{ -10, -1, 5, 12, 100 };
  const slice: []const i32 = &array;
  for (slice) |item|
      .....
  ```

- A slice value is a kind of a *fat pointer*. Technically it is a struct containing a many-item pointer and a length. A `[]const i32` slice contains a `ptr` field of type `[*]const i32` and a `len` field of type `usize`. These fields are accessible. E.g. we could rewrite the `find` function in our recent Zig example as:

  ```
  fn find(key: i32, array: []const i32) ?usize {
      for (0..array.len) |i| {
          if (array.ptr[i] == key)
              return i;
      }
      return null;
  }
  ```

  where we are explicitly using the `ptr` and `len` fields.

- The `ptr` and `len` can also be assigned to, e.g.:

  ```
  const array = [_]i32{ -10, -1, 5, 12, 100 };
  var slice: []const i32 = &array;
  slice.len = 3; // clamp the length to 3
  for (slice) |item|
      .....
  ```

It's probably better not to overuse this possibility and reserve it for exceptional cases, like e.g. if we explicitly want to modify only one of the `ptr` and the `len` fields. Normally it could be more expressive to use Zig's slicing operator, which simultaneously defines both the `ptr` and the `len` fields, and which we discuss in Section 2.18.

- Slices can be indexed with array indexing operator `[ ]` in the same way as arrays (and in the same way as Zig's many-item or C/C++'s pointers):

  ```
  fn find(key: i32, array: []const i32) ?usize {
      for (0..array.len) |i| {
          if (array[i] == key)
              return i;
      }
      return null;
  }
  ```

We just concluded that pointers to arrays can be assigned to slice variables (the "official" terminology is that "pointers to arrays *coerce* to slices"). Now let's recall that arrays can be initialized with a tuple literal, e.g.:

```
    const data: [5]i32 = .{ -10, -1, 5, 12, 100 };
```

The same kind of initialization also works with slices if we take an address of the tuple literal:

```
    const slice: []const i32 = &.{ -10, -1, 5, 12, 100 };
```

This allows to rewrite our `main` function even more concisely:

```
pub fn main() void {
    const found_at = find(12, &.{ -10, -1, 5, 12, 100 });
    std.debug.print("Found at {any}\n", .{found_at});
}
```

or, if we really wanted to,

```
pub fn main() void {
    std.debug.print(
        "Found at {any}\n",
        .{find(12, &.{ -10, -1, 5, 12, 100 })},
    );
}
```

## 2.18 Slicing

Until now we have been creating slices pointing to an entire array. It is possible to obtain slice views of only a part of an array. E.g.

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

pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    for (0..3) |left| {
```

```
        const right = left + 3;
        const slice = data[left..right];
        // The type of slice is []const i32
        std.debug.print("slice = {any}\n", .{slice});
    }
}
```

generates 3 slices of type `[]const i32`, corresponding to the index ranges `0..3`, `1..4` and `2..5`, each range being 3 indices long.

Notes:

- The "slice operation" `[m..n]` takes an index range `m..n`, this range denoting a set of indices `m`, `m+1`, ..., `n-1`. It is thereby expected that `m<=n` (where `m==n` means an empty range). This kind of range is pretty much the same as the one used in Zig's `for` loops, such as `for (1..4) |i|`. The bounds `m` and `n` thereby must be compile-time or runtime expressions of the type `usize` or coercible to `usize`.

  The slice's `ptr` field thereby points to the `m`-th element of the operation's argument array and the slice's `len` field is set to `n - m`. In particular, in the above example the `slice.ptr` field is effectively initialized to `@as([*]const i32, &data) + left` and the `slice.len` field is initialized to `right - left`.

- The slicing operation `[m..n]` with compile-time known *constant* range bounds `m` and `n` doesn't produce a slice but a single-item pointer to array.

  ```
  const data = [_]i32{ -10, -1, 5, 12, 100 };
  const slice = data[1..4];
  // slice's type is *const [3]i32
  ```

  This is the reason why we did a loop in the example code demonstrating the slicing operation: it makes sure that the bounds are not constant and the slicing operation does really produce a slice rather than a pointer to array.

  This rule is there in order to be able to make use of the fact that the length of the resulting slice is then also a compile-time constant. From the usage perspective, the result is still "kind of the same", since single-item pointers to arrays coerce to slices anyway, and since the API of the former, except for the missing `ptr` field, is essentially identical to the one of the latter.

- The slicing operation `[m..n]` can be applied anywhere, where array indexing operation `[i]` can be applied. This includes arrays, pointers to arrays, multi-item pointers and slices themselves.[5] E.g.

  ```
  const slice = data[left..right];
  const subslice = slice[0..2];
  ```

  Additionally one can apply the `[0..1]` slicing operation to single-item pointers, thereby expressing a wish to convert them to slices of size 1, but, as the indices are compile-time constants, producing a pointer to array of size 1 instead:

[5] Also C pointers are included.

```
const ptr = &@as(i32, 10); // ptr is *const i32
const slice = ptr[0..1]; // slice is *const [1]i32
```

- Omitting the right bound, as in `a[m..]`, implies `a[m..a.len]`, that is the resulting slice will run till the end of the slice operation's argument. This applies only if `a` is an array or a slice, since otherwise the expression `a.len` is not valid. E.g.

  ```
  const data = [_]i32{ -10, -1, 5, 12, 100 };
  const slice = data[2..];
  // slice contains data[2], data[3] and data[4]
  ```

  Notice that if the slice length is compile-time-known, in accordance with the rule that a slicing operation with compile-time known bounds produces pointers to arrays, we can convert such slices to pointers-to-arrays by `[0..]`. Respectively we can convert such slices to arrays by `[0..].*`:

  ```
  const data = [_]i32{ -10, -1, 5, 12, 100 };
  const slice: []const i32 = &data; // []const i32
  const ptr_to_array = slice[0..]; // *const [5]i32
  const array = slice[0..].*; // [5]i32
  ```

  In case of multi-item or C pointers, where the length is not known and respectively `a.len` is not a valid expression, `a[m..]` is still a valid operation, but this time it simply offsets the pointer, returning a new pointer:

  ```
  const ptr: [*]const i32 = &[_]i32{ -10, -1, 5, 12, 100 };
  const slicing_result = ptr[2..]; // [*]const i32
  // equivalent to const slicing_result = ptr + 2;
  ```

- The pattern `a[m..][0..n]` can be used to select a slice of length `n` starting at index `m`. Using this feature we could rewrite the example at the beginning of the current section as:

  ```
  const data = [_]i32{ -10, -1, 5, 12, 100 };
  for (0..3) |left| {
      const slice = data[left..][0..3];
      // The type of slice is *const [3]i32
      std.debug.print("slice = {any}\n", .{slice});
  }
  ```

  Notice that thereby, since the bounds of the latter slicing operation `[0..3]` are compile time-known constants, the result is now `*const [3]i32` rather than `[] const i32`.

- In safe builds the slicing operation `[m..n]`, if applied to arrays or slices, will do a bounds check. E.g.

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

  var m: usize = 1;
  var n: usize = 10;
  ```

```
pub fn main() void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const slice = data[m..n];

    std.debug.print("slice = {any}\n", .{slice});
}
```

will panic due to an out of bounds situation. Obviously, such checks are not possible with multi-item and C pointers, since those do not contain array length information.

## 2.19 Strings and sentinels

In C/C++ the `char` type was for a long time commonly used to represent 8-bit integer values, since that was the size of that type on typical platforms. Zig goes the other way around and officially specifies the `u8` type to be used for chars.

Single-character literals in Zig look kind of the same as in C/C++: `'A'`, `'\''`, `'\x41'`, etc. As with integer literals in Zig, single-character literals in Zig have `comptime_int` type and can be coerced to a known-bit-width integer type large enough to accommodate the respective value. As Zig interprets source files using UTF-8 format encoding, it is possible to specify character literals not fitting into a `u8`. Those are accepted but a larger-size integer type is needed to accommodate those.

▷ The `'\x41'` pattern is limited to 8-bit character values in Zig, for larger values use the `'\u{8A00}'` pattern instead.

Characters can be combined into strings. In C/C++ strings are represented as zero-terminated character sequences. The types associated with these constants are arrays of `char`s and pointers to `char`s:

```
const char str[] = "some text";
const char *p = str;
```

However, given a `char` array or pointer value, we cannot be sure whether it is pointing to a zero-terminated string, especially if our program also uses other approaches to store a string, e.g. by explicitly storing the string length separately, so we have to exercise some care with respect to how we treat data referenced by `char` pointers.

Zig string literals are specified as demonstrated by the following examples:

```
const str = "I am a string literal";
const special = "With special features: \" \x41 \u{8A00}";
```

The UTF-8 encoding of Zig source files is taken over into the string literals, the latter being represented via arrays of `u8`.[6] Now, with Zig's common usage of slices, explicitly storing and passing over a length of a string becomes ubiquitous. So, no zero termination is actually needed. On the other hand, as Zig places pretty high value on interoperability with C/C++, we want to be able to have strings compatible to C/C++.

[6]Zig's standard library also has functionality to handle little-endian UTF-16 (including the WTF-16 flavor) strings, contained in `std.unicode`. The library, among other things, allows compile-time conversion of Zig's UTF-8 string literals to litte-endian UTF-16.

Zig solves this by enforcing string literals to be zero-terminated, regardless of whether they are intended to be used for C/C++ interoperability or not. This sounds convenient, but “imports” the related issues from C/C++ and even creates new ones, specific to Zig:

- Whenever we are dealing with a string, we need to somehow know whether it’s zero-terminated or not. In particular we must be careful to avoid inadvertently supplying non-zero-terminated data to code which expects zero-terminated strings.

- The value of the `len` field would not be equal to the string’s length in characters (which is at least what one could expect for strings exclusively consisting of single-byte UTF-8 characters), but rather also include the terminating zero.

- Array concatenation, if applied to strings, would produce a zero byte in the middle of the result. E.g. `"Hello," ++ " world!"`, would produce `"Hello,\0 world!"`.

This kind of problems is solved in Zig by introducing separate types for zero-terminated data.

```
// The comments on the right indicate the value types.
// Notice that the string literal’s type is not array of u8,
// but a pointer to an array of u8, not unlike in C/C++.
const str = "I am a string literal"; // *const [21:0] u8
const str_data = str.*; // [21:0]u8
var n: usize = 1; // to make sure str[n..] produces a slice
_ = &n;
const slice = str[n..]; // [:0]const u8
const many_item_ptr = slice.ptr; // [*:0]const u8
```

The `:0` construct used inside the square brackets in the respective types indicates that the array or the pointed-to array is zero-terminated. In Zig terms we say that the arrays have a zero *sentinel*.

The sentinel information contained in the type can be used by the compiler in a variety of ways:

- Prevent inadvertent incorrect mixtures of sentinel-terminated and non-sentinel-terminated entities. Notice, however, that it is allowed to assign a sentinel-terminated array, pointer or slice to a non-sentinel-terminated array, pointer or slice, e.g.:

  ```
  const str = "I am a string literal";
  const slice: []const u8 = str;
  ```

- Have the actual string length (assuming single-byte characters) reported by the `len` field, e.g.

  ```
  const str = "abc";
  @compileLog(str.len); // outputs 3
  ```

- Correctly handle concatenations of sentinel-terminated data, e.g.

```
// equivalent to const str = "Hello, world!"
const str = "Hello," ++ " world!";
```

Besides using string literals, one can construct sentinel-terminated arrays by simply specifying the respective type. We are not limited to the `u8` type, nor do we have to use zero as the sentinel value:

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

pub fn main() void {
    const a = [_:100]i32{ 1, 2 };
    const ptr = (&a).ptr; // many-item
    const value_at_sentinel = ptr[2];
    std.debug.print("{}\n", .{value_at_sentinel});
}
```

Notice that we had to use a many-item pointer to access the sentinel, which is beyond the last index of the array.

▷ One cannot use completely arbitrary types with sentinel-terminated sequences. The compiler diagnostic message (in version 0.14.1) speaks of only scalar type values being allowed as sentinels, but that actually also includes pointers (which seem to count as scalar types at least here) as well as optionals of "scalar" types. In fact, `null` is another somewhat commonly used sentinel value:

```
const x: i32 = 0;
const y: i32 = 1;
const a = [2:null]?*const i32{ &x, &y };
```

▷ If you have a data sequence containing a sentinel value somewhere in the middle, you can obtain a sentinel-terminated slice as follows:

```
const a = [_]i32{1,2,3,0,4,5};
const slice = a[1..3:0];
// The slice will actually be a pointer to array
// *const [2:0]i32 rather than [:0]const i32,
// since the specified bounds are compile-time-known
```

The slicing operation `[m..n:0]` will perform a compile-time or a runtime check (in safe builds) to ensure that the respective memory location indeed contains the specified sentinel value.

▷ Lexicographical comparison of zero-terminated sequences can be done using the `std.mem.orderZ` function, which is kind of Zig's counterpart of C++'s `strcmp()`. Using this function might require knowledge of Zig's features discussed much further in the book, so we'll skip providing an example here.

## 2.20 Typedefs

Zig's counterpart of C/C++'s typedefs, or of C++'s `using` statement is... simply declaring a const, which is equal to the type. E.g. the following typedef:

```
typedef int32_t *int32_ptr;
// C++ only:
//    using int32_ptr = int32_t *;
```

is translated into Zig as

```
// We use a different name than in C/C++
// due to Zig's naming style convention for types.
const PtrToInt32 = *i32;
```

That's right, you declare "typedefs" in Zig the same way you declare other constant values. That's because types are treated as values in Zig. E.g. in the above example `*i32` is actually a value, the type of this value being '`type`', which is one of the builtin types in Zig:

```
// We could also explicitly specify the type
// of the PtrToInt32 constant:
const PtrToInt32: type = *i32;
```

Don't hold your hopes too high though, the type values are available only in compile-time code. We shall discuss this further in Section 5.7.

## 2.21 Function types and pointers

Similarly to C/C++, Zig allows to define pointers to functions. The respective pointer types are denoted in the same way as other pointer types in Zig, so in order to be able to denote a pointer-to-function type we first need to learn how to denote a function type itself.

Consider the C++ version of the `find` function discussed at the beginning of Section 2.15. This function's type is `int(int, const int *, size_t)`. We could also write

```
using find_func_t = int(int, const int *, size_t);
```

For the Zig version of the same function we shall take the implementation from Section 2.17 which is a better translation to Zig of the same C++ code. The type of this Zig function is `fn(i32, []const i32) ?usize` and the counterpart of the above C++ `using` statement would be as follows

```
const FindFunc = fn(i32, []const i32) ?usize;
```

▷ If we wanted to, we could also spell out the argument names in the function type;

```
const FindFunc = fn(key: i32, array: []const i32) ?usize;
```

corresponding to the C++ code

```
using find_func_t = int(int key, const int *array, size_t size);
```

It is completely optional.

We now, for the sake of exercise, will write the following program, which utilizes a single-item pointer to function as the `findWith` function's parameter:

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

fn find(key: i32, array: []const i32) ?usize {
    for (array, 0..) |item, i| {
        if (item == key)
            return i;
    }
    return null;
}

// Notice that we cannot name the findWith function's
// parameter more concisely as 'find', since the latter
// would shadow the name of the 'find' function, which is
// also accessible within the findWith function's cope.
fn findWith(findFunc: *const FindFunc) void {
    const data = [_]i32{ -10, -1, 5, 12, 100 };
    const key: i32 = 12;

    const found_at = findFunc(key, &data);

    std.debug.print("Found at {any}\n", .{found_at});
}

const FindFunc = fn (key: i32, array: []const i32) ?usize;

pub fn main() void {
    // equivalent to findWith(&find);
    findWith(find);
}
```

Notes:

- In C/C++ writing `const find_func_t *` would be redundant, as `const` in this position has no effect on pointers to functions. In Zig `const` normally needs to be used with pointers to functions, as the functions themselves are considered const objects, so upon taking an address of a function a `*const fn(....) ....` kind of value is produced.

- Similarly to C/C++, functions automatically coerce to respective pointers to function. This is why we don't have to explicitly write the address taking operator in the call to `findWith` in the `main` function, this operator is applied implicitly, since the expected argument type is a function pointer.

- You might find out that the program also compiles and works if we define the `findWith` as

  ```
  fn findWith(findFunc: FindFunc) void {
      ....
  }
  ```

  that is, instead of passing a pointer to the function, we pass the function itself. This rather defines a `findWith` as kind of a generic function, where specializations would be generated for each different value of the function argument `findFunc`. The C++ counterpart would be:

```
template<find_func_t find_func>
void find_with() {
    ....
}
```

▷ Differently from C/C++, it is not possible to rewrite the `findFunc` function call inside `fn findWith(findFunc: *FindFunc) void` as

```
const found_at = findFunc.*(key, &data);
```

which would have been a counterpart of C/C++'s

```
auto found_at = (*findFunc)(key, data, size);
```

The expression `findFunc.*` in Zig has function type, and values of this type can be used only in compile-time code. So with runtime function pointers you have to do the call operation directly on the pointers, without explicitly dereferencing them first.

# Chapter 3

# Structs

*Struct types and related concepts.*

## 3.1 Optional pointers

As a showcase for Zig's struct type usage we are going to construct a custom, essentially toy-level, single-linked list. In C++ terms we are going to represent the list nodes as

```
struct Node {
    int payload;
    Node *next; // terminated by nullptr
};
```

where we would like to use the `nullptr` value of the `next` field to denote the end of the list.

Here comes our first problem: a single-item pointer looks almost like a perfect pointer kind to represent the `next` field in Zig, however single-item pointers cannot be null. Zig addresses this by wrapping a single-item pointer into an optional:

```
var nullable_ptr: ?*i32 = null;
var nullable_const_ptr: ?*const i32 = null;
```

Notes:

- A null check thereby normally becomes a check of an optional being non-null:

  ```
  if(nullable_ptr) |ptr|
      doSomethingWith(ptr);
  ```

  or

  ```
  const non_null_ptr = nullable_ptr orelse fallback_ptr;
  ```

  or

  ```
  if(nullable_ptr != null)
      doSomethingWith(nullable_ptr.?);
  ```

  etc.

- While normally Zig optionals are represented in memory by having a separate extra internal boolean field, signifying the presence of the value, for pointer optionals Zig does a special optimization: the null value of the optional is represented as the null value of internally stored pointer. So the internal representation of an optional single- or many-item pointer in Zig matches the one of a C/C++ pointer.

- Zig has a special kind of pointer attribute allowing the pointer to assume zero value (that is to point to the zero address in memory). E.g.

  ```
  var ptr: ?*allowzero i32 = undefined;
  ```

  The zero value of such pointers is however not considered semantically null, it is just like any other value of the pointer, and such zero-value pointers are generally supposed to be dereferenceable (unlike semantically null pointers, which are not supposed to ever be dereferenced).

▷ Optionals built from `allowzero` pointers cannot employ the usual optional pointers optimization and allocate an extra hidden boolean field.

## 3.2 Single-linked list node

Now that we have introduced optional pointers, we should be capable of translating the previously introduced C++ `Node` structure to Zig:

```
const Node = struct {
    payload: i32,
    next: ?*Node,
};
```

Notes:

- the struct fields `payload` and `next` are separated by commas, rather than by semicolons as they would be in C/C++. The trailing comma after the `next` field is used optionally and controls the auto-formatting (also refer to the trailing comma discussion in Subsection 1.4.1).

- A more precise C/C++ counterpart of the Zig `Node` struct declared above is actually

  ```
  typedef struct {
      int payload;
      Node *next;
  } Node;
  ```

  since, as discussed in Section 2.20, the `const Node = .....` Zig's construct works similarly to a typedef in C/C++. Respectively, the entire Zig construct `struct { payload: i32, next: ?*Node }` is actually a value of type '`type`'. Respectively, the latter construct can also be used without being assigned to a named const first. This possibility is discussed in Section 3.14.

- Struct "typedefs" can also be placed inside other structs, enum, unions, as well as locally within function bodies (similarly to other `const` definitions). With the latter option there are severe restrictions due to strictly sequential declaration order, this is discussed further in Section 3.11.

Now let's add a couple of C++ functions for single-list node manipulation:

```
#include<stdio.h>

struct Node {
    int payload;
    Node *next = nullptr;
};

static void push(Node *&first, Node &node) {
    node.next = first;
    first = &node;
}

static void printNodes(const Node *first) {
    while(first) {
        printf("%d\n", first->payload);
        first = first->next;
    }
}

int main() {
    Node node1, node2;
    node1.payload = 1;
    node2.payload = 2;
```

```
    Node *first = nullptr;
    push(first, node1);
    push(first, node2);

    printNodes(first);
    return 0;
}
```

and here is the Zig counterpart:

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

const Node = struct {
    payload: i32,
    next: ?*Node = null,
};

fn push(first: *?*Node, node: *Node) void {
    // node.next is a shortcut for node.*.next
    node.next = first.*;
    first.* = node;
}

fn printNodes(first: ?*const Node) void {
    // 'first', being a function parameter, is a const,
    // so we need to introduce a var to loop over
    var node = first;

    // The while with a capture works similarly to
    // the conditional: if(node) |n|
    while (node) |n| : (node = n.next)
        std.debug.print("{}\n", .{n.payload});
}

pub fn main() void {
    // Two different syntax options for initialization
    // The effect of both is equivalent
    var node1 = Node{ .payload = 1 };
    var node2: Node = .{ .payload = 2 };

    var first: ?*Node = null;
    push(&first, &node1);
    push(&first, &node2);

    printNodes(first);
}
```

Notes:

- The `next` field of the `Node` struct has a default value (in both C++ and Zig versions). Differently from Zig's consts and vars, where the const's/var's type can be inferred from the initializer, the Zig struct field types are not inferred from the default values and still must be explicitly specified. In particular, we couldn't write

```
const Node = struct {
    payload: i32,
    next = @as(?*Node, null), // this doesn't work
};
```

Another limitation is that default initializer expression values must be compile time-known:

```
const Node = struct {
    // The following default initializer doesn't work
    payload: i32 = generateRandomValueAtRuntime(),
    next: ?*Node = null,
};
```

- As we know by now, in Zig variables and consts must be initialized (at least with `undefined`, although the latter should be rather used as an exception). The two code lines at the beginning of the `main` function demonstrate two different ways to initialize a struct with a literal:
  - You can do it with a `Node` struct literal `Node{ .payload = 1 }`. This literal is already of the `Node` type, therefore you don't need to specify the type of the variable.
  - Or you can use an *anonymous struct literal* `.{ .payload = 1 }`, in which case you need to explicitly specify the variable's type as `Node`. The anonymous struct literal will coerce to `Node`, provided the field names used inside the literal match the `Node` struct's definition.
- When initializing a struct or writing a struct literal in general, you must supply all fields which do not have a specified default value in the struct definition. You *can* also specify values for fields which do have defaults, if you wish, e.g.

  ```
  var node3 = Node{
      .payload = 3,
      .next = &node1,
  };
  ```

  or

  ```
  var node3: Node = .{
      .payload = 3,
      .next = &node1,
  };
  ```

- The `node.next` construct used in the body of the `push` function is a shortcut for `node.*.next`. Similarly to how single-item pointers to arrays in Zig in many cases allow usage as if they were directly arrays (specifically, you can directly apply the [ ] operation to those pointers, as well as access the `len` field), with single-item pointers to structs you can directly access the respective struct fields from the pointers.[1]

## 3.3 Methods

We would like now to transform the `push` and `printNodes` into methods. These functions would however not work too well or at all as methods of the `Node` entity and would be more appropriate as methods of a `List` entity. So let's introduce the latter. Our C++ code might start looking like follows:

```
#include<stdio.h>

struct Node {
    int payload;
    Node *next = nullptr;
};

struct List {
    Node *m_first = nullptr;

    void push(Node &node) {
        node.next = m_first;
        m_first = &node;
    }

    void printNodes() const {
        for( const Node *node = m_first;
                 node != nullptr ; node = node->next )
            printf("%d\n", node->payload);
    }
};

int main() {
    Node node1, node2;
    node1.payload = 1;
    node2.payload = 2;

    List list;
    list.push(node1);
    list.push(node2);

    list.printNodes();
    return 0;
}
```

Its Zig counterpart is:

[1]This feature can be seen as Zig's counterpart of C/C++'s `->` operator. Although one could argue that the `->` operator was primarily introduced into C to reduce the parenthesis clutter from expressions like `(*ptr).field`. Zig does not have the latter problem anyway, since pointer dereferencing is a postfix rather than prefix operator in Zig, but being able to omit '`.*`' in the middle of the respective constructs is often a nice shortcut.

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

const Node = struct {
    payload: i32,
    next: ?*Node = null,
};

const List = struct {
    // The trailing comma would be optional here, if 'first'
    // was at the very end of the struct, otherwise it's a must
    first: ?*Node = null,

    fn push(self: *List, node: *Node) void {
        node.next = self.first;
        self.first = node;
    }

    // Could have passed 'self' by const reference instead:
    //     fn printNodes(self: *const List) void
    // See the discussion of pass-by-reference optimization.
    fn printNodes(self: List) void {
        var node = self.first;
        while (node) |n| : (node = n.next)
            std.debug.print("{}\n", .{n.payload});
    }
};

pub fn main() void {
    var node1 = Node{ .payload = 1 };
    var node2: Node = .{ .payload = 2 };

    // Alternatively we could do: var list = List{};
    var list: List = .{};
    list.push(&node1); // same as List.push(&list, &node1);
    list.push(&node2); // same as List.push(&list, &node2);

    list.printNodes(); // same as List.printNodes(list);
}
```

Notes:

- In the C++ version we prefixed the field of the list with '`m_`' in order to make the method code more readable, so that we can clearly distinguish member variable (struct field) access from using local variables or function parameters. In Zig we don't need to do that: the fields are anyway accessed via the explicit `self` parameter, as, differently from C++, there is no implied `this` in Zig.

- Technically, Zig methods are just ordinary functions declared inside the struct namespace. They can also be called like ordinary functions, e.g.

  ```
  List.push(&list, &node1);
  ```

where we need to prefix the push function with the List namespace in order to make the compiler find it.

Notice that this is different from C++, where methods are having a different calling convention, which at times is rather inconvenient, e.g. you cannot store pointers to methods as pointers to functions. Zig doesn't have this problem.

Respectively, the self parameter is pretty much an ordinary parameter of a function. It doesn't have any special meaning inside the function body. It can also have a different name, we don't really have to call it "self", the latter is merely a convention.

- In order for the self parameter to function similarly to C++'s this, it must be the first parameter of the method and have one of the following types[2]: List, *const List, *List. When we write list.push(&node1) inside the main function, the Zig compiler will look for a function inside the List struct, which has the first parameter of one of the three mentioned types. It will also automatically take an address of the list, if a list pointer is expected by the function.

  If list was a pointer to Node, the compiler would automatically dereference the pointer, similarly to how it works with struct fields:

```
var list_storage: List = .{};
const list = &list_storage;
list.push(&node1); // same as list.*.push(list, &node1);
list.push(&node2); // same as list.*.push(list, &node2);
list.printNodes(); // same as list.*.printNodes(list.*);
```

- In the C++ version we have defined the methods inside the struct, but we could also have only declared them inside and then defined them outside. In Zig it's only possible to define the methods inside the struct.

- It used to be recommended in Zig to generally avoid passing the self parameter by const reference and pass it by value instead. With the pass-by-reference optimization being recently reconsidered (as discussed in Section 2.11) it looks like passing by const reference should be the new default. In the currently discussed example from the above we intentionally passed self by value in the printNodes method, this was done primarily for the sake of demonstration of details of passing the self parameter by value.

## 3.4 @This() builtin function

Explicitly writing the List type name in the declaration of the self parameter has a drawback that if we later wanted to change the name of the struct, we would have to replace it in all declarations of the self parameter as well. Instead, we could use the @This() builtin function, which returns the type (yes, types are values in Zig, as already mentioned) of the current struct that we're inside of.[3] Therefore, it would be generally more practical to write

[2]Alternatively it may be declared as anytype. See the discussion in Section 5.4.

[3]If struct definitions are nested, it returns the type of the innermost struct that we're inside of.

```
const List = struct {
    first: ?*Node = null,

    fn push(self: *@This(), node: *Node) void {
        node.next = self.first;
        self.first = node;
    }

    fn printNodes(self: @This()) void {
        var node = self.first;
        while (node) |n| : (node = n.next)
            std.debug.print("{}\n", .{n.payload});
    }
};
```

▷ Another common pattern is to introduce a `Self` "typedef", which can be done as follows:

```
const List = struct {
    const Self = @This();

    first: ?*Node = null,

    fn push(self: *Self, node: *Node) void {
        node.next = self.first;
        self.first = node;
    }

    fn printNodes(self: Self) void {
        var node = self.first;
        while (node) |n| : (node = n.next)
            std.debug.print("{}\n", .{n.payload});
    }
};
```

The author personally is not particularly fond of this latter pattern, as it is not usable in case of nested struct declarations (due to Zig's "no-identifier-shadowing" rule).

## 3.5 Memory layout

In C the struct fields appear in memory in the order of their declaration, possibly with alignment-induced padding in-between, and it goes in a similar direction in C++. Zig, on the other hand, explicitly drops any relationship between the order of field declaration and their offsets in memory. This, in particular, allows the compiler to choose the field ordering which minimizes the required padding amount.

E.g. consider the following Zig struct:

```
const S = struct {
    i: i32,
    j: u64,
    k: u8,
};
```

If the fields were in memory in the declaration order, it would have corresponded to the following C struct:

```
struct S {
    int32_t i;
    int32_t padding1; // pad to the next 8-byte boundary
    uint64_t j;
    uint8_t k;
    // pad the entire struct to the next 8-byte boundary
    uint8_t padding2a;
    uint16_t padding2b;
    uint32_t padding2c;
};
```

(where we explicitly listed the padding for the sake of illustration). Thus, 11 bytes are lost for padding. On the other hand, Zig would probably rather generate something along the lines of:

```
struct S {
    uint64_t j;
    int32_t i;
    uint8_t k;
    // pad the entire struct to the next 8-byte boundary
    uint8_t padding2a;
    uint16_t padding2b;
};
```

Here, only 3 bytes are lost for padding.

▷ If you really want to preserve the field order in memory, you can use Zig's `extern struct`:

```
const S = extern struct {
    i: i32,
    j: u64,
    k: u8,
};
```

This generates a struct layout compatible to C language struct layout (like the 11 byte-padded struct demonstrated above). Thereby extern structs are also usable for interfacing with C APIs.

## 3.6 Decls vs. fields

The term "decls" (singular: "decl") in Zig refers to all entities declared within a struct's[4] namespace.

We have seen previously how we could explicitly call the methods (or functions in general) of the `List` struct explicitly using the namespace qualification:

```
    List.push(&list, &node1);
```

[4]Or a struct-like object's, such as union or enum.

Similarly, having introduced the `Self` "typedef" inside the `List` struct, we could have referred to this typedef externally (although that wouldn't make too much sense) as `List.Self`.

Thus, `push`, `printNodes` and `Self` are all decls of the `List` struct. Note that struct fields are not considered to be within the respective struct's namespace, as the fields cannot be referenced using namespace qualification: e.g. `List.first` neither works nor makes sense, as it needs an actual list object rather than merely a namespace. So fields and decls are different elements of a struct.

▷ Apparently, decls can be seen as Zig counterparts of static members in C++. With the reservation that some of the function decls (those with the first argument of an appropriate type) can also be called as if they were non-static member functions.

▷ Zig has a special field flavor: *comptime fields*. A comptime field declaration is illustrated by the following line:

```
    comptime field_name: Type = initial_expression,
```

This feature is not really documented, which might be causing some doubts as to whether it will stay in the long run. Normally one is supposed to use a decl instead.

Comptime fields can work as consts initialized with compile-time-known values, the difference is in their access syntax:

```
const S = struct {
    comptime field: i32 = 0,
    const decl: i32 = 0;
};

pub fn main() void {
    _ = S.decl; // decl access

    // create an instance of S
    const inst = S{};
    _ = inst.field; // comptime field access
}
```

This sometimes comes as handy in generic programming, where you don't necessarily know the specific type of a variable. Comptime fields allow to concisely write `myvar.field` instead of `@TypeOf(myvar).decl`. It's even more convenient in cases of longer expressions in place of `myvar`.

You can also formally store compile time-known values into comptime fields at runtime, but it must be exactly the same value as the field's initialization value:

```
const S = struct {
    comptime field: i32 = 0,
};

pub fn main() void {
    var inst = S{};
    inst.field = 0; // store the same value
}
```

## 3.7 Construction and destruction

To further get acquainted with Zig's decls techniques, let's introduce a decl var into `List` which would be counting the total number of list objects instantiated in the program. In C++ we would have done something along the lines of:

```
struct List {
    // assume single-threaded construction/destruction,
    // so we don't need to bother about concurrent access
    static inline int total_lists = 0;

    List() { total_lists++; }
    ~List() { total_lists--; }

    Node *m_first = nullptr;
    ......
```

and that would be the only change needed to the previous code, the rest would have been achieved automatically through C/C++'s constructor and destructor mechanisms.

In Zig (which doesn't really have constructors and destructors) a little bit more work would be needed, so we present the full program code:

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

const Node = struct {
    payload: i32,
    next: ?*Node = null,
};

const List = struct {
    // Use unsigned to benefit from runtime underflow checks
    var total_count: u32 = 0;

    fn init() @This() {
        total_count += 1;
        return .{}; // all fields default-initialized
    }

    // We don't care about deinit's argument value,
    // but we formally need a self parameter in order
    // to be able to call list.deinit(). In theory, we
    // also could have passed this parameter by value
    // rather than by const reference, not that it
    // matters much in this particular case.
    fn deinit(_: *const @This()) void {
        total_count -= 1;
    }

    first: ?*Node = null,

    fn push(self: *@This(), node: *Node) void {
        node.next = self.first;
        self.first = node;
    }

    // Let's also pass the 'self' parameter by const
    // reference instead of by value, as we did before.
    fn printNodes(self: *const @This()) void {
        var node = self.first;
        while (node) |n| : (node = n.next)
            std.debug.print("{}\n", .{n.payload});
    }
};

pub fn main() void {
    var node1 = Node{ .payload = 1 };
    var node2: Node = .{ .payload = 2 };

    var list: List = .init(); // same as var list = List.init();
    defer list.deinit(); // works as "destructor scheduling"

    list.push(&node1);
    list.push(&node2);

    list.printNodes();

    // The deferred list.deinit() is executed here
}
```

Notes:

- The introduced `total_count` var in the `List` struct namespace can be accessed externally as `List.total_count`.

- We are no longer supposed to “manually” initialize list objects using struct literals like `List{}` or `.{}`, instead we must call the `init()` function of the `List` struct.

- The `init()` function of the `List` struct normally must be called using the namespace qualification, as usual: `List.init()`. However there is a special convention in Zig, called *decl initializers*, which allows to omit the namespace if the result context already expects the values of the respective struct. In our case, when we write `var list: List =`, we have thereby already specified that we are expecting a `List` value. Therefore we can omit the namespace and simply write `.init()` to the right of the equality sign.

  Similarly we could have (in principle) written the following statement: `List.printNodes(&.init());` which would have been the same as writing `List.printNodes(&List.init());`. The statement per se wouldn't have made much sense, also we wouldn't have been able to call `deinit()` for the respective list. But formally it would have worked.

- Apparently the `init()` function is a counterpart of the C++'s `List` constructor. Respectively the `deinit()` function is a counterpart of the destructor. The `defer` statement works as a kind of "destructor scheduler", scheduling the contents of the statement to be executed upon exiting the current scope (either upon reaching the end of the scope or upon exiting the scope prematurely e.g. via a `return` statement).

  Normally the `defer` statements are placed immediately following the "construction statements", such as `var list: List = .init();` in our example. The deferred statements are then executed upon leaving the scope in the reverse order, pretty much like C++ destructors would be:

  ```
  fn func() void {
      var list: List = .init();
      defer list.deinit();

      {
          var list1: List = .init();
          defer list1.deinit();

          var list2: List = .init();
          defer list2.deinit();

          // list2.deinit() is executed here
          // list1.deinit() is executed here
      }

      // list.deinit() is executed here
  }
  ```

▷ The `init` and `deinit` names are conventionally used in Zig for "construction/destruction" purposes. The details of the function signatures may vary depending on the use case. In particular, `init` doesn't have to return the "constructed" object, but might "construct the object in-place":

```
fn func(value: i32) void {
    var obj: Object = undefined;
    obj.init(value);
    defer obj.deinit();
    ..........
}
```

The `deinit()` method might take extra parameters (which would be impossible with C++ destructors) etc.

▷ The decl initializers do not have to be functions, but can also be const decls. E.g.

```
const Node = struct {
    payload: i32,
    next: ?*Node = null,

    const zero = Node{ .payload = 0 };
};
```

```
.......

pub fn main() void {
    var node0: Node = .zero;
    ......
}
```

▷ A possible pitfall with the usage of `defer` is that one might come to an idea of scheduling deferred actions conditionally, e.g.

```
fn func(value: i32) void {
    var obj: Object = undefined;
    obj.init(value);
    if (value >= 0)
        defer obj.deinit(); // doesn't work
    .........
}
```

Apparently this doesn't work as a conditional deferral. The `defer` statements implement compile-time scheduling, not runtime scheduling. At the time of this writing, the compiler won't even accept a `defer` statement immediately under an `if`. But at any rate, such statement's meaning, even if it compiled, would have been

```
    if (value >= 0) {
        defer obj.deinit();
    }
```

so the deferred statement would have been executed directly upon leaving the nested scope of the `if` statement, thereby the whole becoming simply equivalent to calling `obj.deinit();` without the whole `if`/`defer` boilerplate.

## 3.8 Multiple source files

Until now we have been always writing single-source-file Zig programs, consisting of the single `main.zig` file. Now we would like to put the `Node` and the `List` structs into a separate source file. Here is how we do this. We create a `List.zig` file located in the same folder as `main.zig`:

```
// List.zig file

const std = @import("std");

pub var total_count: u32 = 0;

pub fn init() @This() {
    total_count += 1;
    return .{};
}

pub fn deinit(_: @This()) void {
    total_count -= 1;
}

first: ?*Node = null,

pub const Node = struct {
    payload: i32,
    next: ?*Node = null,
};

pub fn push(self: *@This(), node: *Node) void {
    node.next = self.first;
    self.first = node;
}

pub fn printNodes(self: *const @This()) void {
    var node = self.first;
    while (node) |n| : (node = n.next)
        std.debug.print("{}\n", .{n.payload});
}
```

And we modify the `main.zig` to have the following contents:

```
// main.zig file

const List = @import("List.zig");

pub fn main() void {
    const Node = List.Node;
    var node1 = Node{ .payload = 1 };
    var node2: Node = .{ .payload = 2 };

    var list: List = .init();
    defer list.deinit();

    list.push(&node1);
    list.push(&node2);

    list.printNodes();
}
```

Notes:

- Note the absence of the `struct` keyword and the associated pair of the { } braces in `List.zig`: those are *implied*. That's right, *every Zig source file is treated as a struct definition.*[5] So when the compiler encounters the import builtin function call `@import("List.zig")`, the "return value" of that call is:

[5]Zig files are always treated as structs, not as any other Zig entity. E.g., despite Zig unions (discussed in Section 4.4) being very similar to Zig structs, it is not possible to have a Zig file being treated as a union. Nor is it possible to have a Zig file being treated as a generic struct (generic structs are discussed in Section 5.2).

```
struct {
    // Here comes the content of List.zig file
}
```

So the entire import statement `const List = @import("List.zig");` is treated as

```
const List = struct {
    // Here comes the content of List.zig file
};
```

- Alternatively we could have placed `List.zig` into a subfolder. We only need to modify the `@import("List.zig")` in `main.zig` to include the respective relative path, e.g. `@import("containers/List.zig")`. Generally, you can even include the parent folder segments "`..`" into the relative path, the only restriction is that you may not go outside of the root source folder of your Zig source code module.

  Note that the relative path of the `@import` call is specified relatively to the path of the file containing the `@import` call, not relatively to some globally configured "include path".

- There is one specific extra rule to the "files treated as structs" rule: by default, none of the decls from the imported file are visible to the outside code. This means that the `total_count`, `init`, `deinit`, `Node`, `push` and `printNode` decls would not have been visible by default, unless marked with a `pub` statement. Marking them as `pub` ensures their visibility outside of the file as well.

  Notice that the mentioned default visibility restriction is file-based, not entity-based. So if the `Node` struct also contained a decl, e.g.

  ```
  pub const Node = struct {
      payload: i32,
      next: ?*Node = null,

      fn disconnect(self: *@This()) void {
          self.next = null;
      }
  };
  ```

  there would be no need to mark `disconnect` as `pub` if we intended to call it from inside the very same file `List.zig` where the `Node` struct is defined. However `pub` would be required to make it accessible from outside of the `List.zig` e.g. from `main.zig`.

  In that respect the effect of `pub` is not really a counterpart of the `public` visibility in C++ code (which is entity-based, not file-based). It's rather a kind of opposite of `static` in C, the latter controlling the file-based visibility of identifiers.

▷ As in C, struct fields are always fully visible in Zig. Their visibility cannot be restricted to particular files or entities. In this respect we would like to mention again that Zig's `pub` is a counterpart opposite of C's `static`, not a counterpart of C++'s `public`.

▷ We might actually have considered to make the `total_count` not `pub` and instead provide a `pub` accessor function:

```
var total_count: u32 = 0;

pub fn totalCount() u32 {
    return total_count;
}
```

which would be callable from `main.zig` as `List.totalCount()`.

By now we should realize that the file scope discussed in Section 2.12 is simply the struct scope of the struct represented by the file. In this respect there can arise a question of uniqueness of such scopes. Indeed, suppose we import one and the same file more than once. That is, we are having multiple `@import` calls referencing one and the same file. Are such imports considered to return one and the same struct? Because if they didn't, then the variables declared at the file scope level of the respective files would exist in multiple instances, and that would be at least counterintuitive and possibly a major source of bugs. So, these different import calls are actually considered to return one and the same struct, as can be verified by

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

pub fn main() void {
    const ptr = &List.total_count;
    const ptr1 = &@import("List.zig").total_count;
    std.debug.print("{}, {}\n", .{ptr, ptr1});
}
```

Similarly, instances of the respective structs are considered to be instances of one and the same type:

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

pub fn main() void {
    const List1 = @import("List.zig");
    const list: List = .init();
    const list1: List1 = list;
    // list and list1 have one and the same type
    @compileLog(list, list1);
}
```

## 3.9 Namespacing

In C++ one can use classes as namespaces, by making the classes contain only static members. In Zig this *is* the way to create namespaces. That is, we create structs with no fields but only with decls and use such structs as namespaces.

In fact the `const std = @import("std");` statement is utilizing exactly this approach. The call `@import("std");` works pretty much identically to the call `@import("List.zig");` that we used recently, the only difference being that, while `"List.zig"` defines a relative path for the imported file, the `"std"` string references a predefined standard library path, associated with this string.

> ▷ The Zig mechanism to associate specific source file paths with other strings, similarly to `"std"`, is intended to be used with Zig *modules*, where one would associate a string with a path to the module's root source file. Thereby the `@import` statement would import a module by providing access to the module's root struct. This root struct would normally function as the module's root namespace.
>
> The associations between the module name strings and module root source file paths are specified on the build system level (or directly in the compiler's command line).

So the `const std = @import("std");` statement is loosely equivalent to something like

```
const std = struct {
    ........
    pub const debug = struct {
        .......
        pub fn print(.....) ....
        .....
    };
    ........
};
```

thereby allowing us to use, among other things, `std.debug.print`.

The `std` struct doesn't contain any fields and one is not supposed to create instances of that struct:

```
var std_instance: std = .{}; // don't attempt this
```

Instead, `std` is intended to be used as a namespace, and so is `std.debug`.

Notes:

- You can do

  ```
  const debug = std.debug;
  ```

  which would be kind of a counterpart of C++'s

  ```
  namespace debug = std::debug;
  ```

  with the difference that the former can be used both at function scope and at struct (or file) scope.

- At the time of this writing (but probably not much longer, see Zig issue 20663), you still can merge a namespace into another one (a counterpart of C++'s `using namespace`) with

  ```
  usingnamespace namespace_expression;
  ```

  where `namespace_expression` is something referring to another struct. E.g. the following would merge all pub decls from the `std` into the current namespace (the namespace that the `usingnamespace` directive is placed within):

```
// Alternative one can simply write:
//    usingnamespace @import("std");
// Both options strongly UNRECOMMENDED
const std = @import("std");
usingnamespace std;
```

If one also wants the merged in decls to become pub in the current namespace, one respectively needs to put `pub` in front of `usingnamespace`:

```
pub usingnamespace namespace_expression;
```

Namespaces are not limited to Zig modules and the standard library. It is pretty common to introduce Zig files serving as namespaces. E.g. theoretically instead of `List.zig` we could have created a `list.zig` file with contents along the lines of the following:

```
// list.zig

pub Node = struct {
    ......
};

pub List = struct {
    ......
};

pub var total_list_count: u32 = 0;
```

and respectively, in `main.zig` do things like

```
// main.zig

const list = @import("list.zig");

pub fn main() void {
    var node1 = list.Node{ .payload = 1 };
    var node2: list.Node = .{ .payload = 2 };

    var list: list.List = .init();
    .......
}
```

> ▷ The reason we switched from `PascalCase` file name `"List"` to `snake_case` file name `"list"` is that this is the Zig convention: use snake case names for namespace-only structs.

> ▷ While it is more common to use entire source files for namespaces, nothing of course prevents us from using a struct declared inside a file as a namespace:
>
> ```
> const some_namespace_struct = struct {
>     // This struct contains only decls and is thereby
>     // a namespace struct
>     .......
> };
> ```

## 3.10 Source folder trees

C/C++ programs generally consist of a number of `.c`/`.cpp` files, sharing a number of common include files but otherwise compiled independently, the shared include files serving as a kind of glue for the program's code base. In Zig (ignoring the external libraries for now) one generally compiles a single source file, the other source files being referenced via `@import` calls, as discussed earlier.

> ▷ Note that this doesn't necessarily mean that Zig recompiles the entire code base each time, there is caching of previous compilation results.

Thereby there emerges an import dependency graph, which usually contains an "essential dependencies" subgraph representable by a tree. It is only logical to try to map this tree into the source folder tree. After all, C/C++ projects typically do a similar thing. The Zig structs implicitly represented by source files would be thereby organized into a tree. There are some details/options about mapping this tree into a folder structure which we would like to discuss here.

Specifically, imagine you have a struct `A`, represented by the source file `A.zig`. Imagine further, this struct has 3 further structs `B`, `C` and `D` as child-type dependencies each represented by its respectively named source file. The question is: how do we organize these files into a folder tree?

- Do we put all 4 files into one and the same folder? If so, how do we continue organizing the tree further. E.g. if the `B` has further child-type dependencies, shall we put them into the same folder? Then, with this approach we are not going to get a folder tree at all.

- Therefore maybe we should create a subfolder named `A` next to `A.zig` and put `B.zig`, `C.zig` and `D.zig` into this subfolder. So our project folder tree might start looking like follows:

  ```
  │
  ├── main.zig
  ├── A.zig
  └── A
      ├── B.zig
      ├── C.zig
      └── D.zig
  ```

  Respectively inside `A.zig` we will have import calls `@import("A/B.zig")`, `@import("A/C.zig")` and `@import("A/D.zig")`.

  It is easy to see how this approach leads to construction of a folder tree, when further dependencies are added. This approach is commonly used in Zig.

- Occasionally one might be doing a different approach. We might create a subfolder `A` and put `A.zig`, as well as its *direct* dependencies `B.zig`, `C.zig` and `D.zig` into this subfolder:

```
├── main.zig
└── A
    ├── A.zig
    ├── B.zig
    ├── C.zig
    └── D.zig
```

Should however e.g. `B` have further dependencies, we would move it into a subfolder, e.g.:

```
├── main.zig
└── A
    ├── A.zig
    ├── B
    │   ├── B.zig
    │   └── E.zig
    ├── C.zig
    └── D.zig
```

It seems that more commonly the first approach (where we had the `A` folder next to `A.zig`) is used. The second approach (putting `A.zig` inside the `A` folder together with dependencies of `A`) might be also used just at selected places in the folder tree.

At any rate, it seems common to put the top-level dependencies (those referred to by `main.zig` or whatever the root source file name is) next to the root source file.

## 3.11 Function-local structs

Structure definition statements, such as `const List = struct { .... };` or `const Node = struct { .... };` in our previous examples don't have to be necessarily put at the file scope level or inside the scope of another struct, although those are the most common positions. Instead, those can be placed locally in function code. E.g.

```
pub fn main() void {
    const List = struct {
        .....
    };
    .....
}
```

Such declarations would be local to the code block that they are declared in.

There are however restrictions arising from the requirement that local declarations are always processed in their textual order, as it was discussed in Section 2.12. You cannot therefore reference a declaration occurring later in the source code, thereby it's e.g. impossible to have two local struct declarations `A` and `B` referencing each other. Furthermore, it is even not possible for a local struct declaration to refer to itself by name:

```
pub fn main() void {
    const Node = struct {
        payload: i32,
        next: ?*Node = null, // does not compile
    };
    _ = Node;
}
```

The problem is that the `Node` identifier is not defined until the semicolon terminating the const declaration. Only after that semicolon we could refer to `Node`.

The solution to the above could be to simply use `@This()`. The same approach also allows to endow locally declared structs with methods:

```
pub fn main() void {
    const Node = struct {
        payload: i32,
        next: ?*@This() = null,

        fn reset(self: *@This()) void {
            self.payload = 0;
        }
    };
    _ = Node;
}
```

This approach however wouldn't work if you want to have two locally defined structs `A` and `B` which cross-reference each other. In this case the solution would be to wrap the `A` and `B` structs into another locally declared struct `C`:

```
pub fn main() void {
    const C = struct {
        const A = struct { b: *B };
        const B = struct { a: *A };
    };
    var a: C.A = undefined;
    var b: C.B = .{ .a = &a };
    a = .{ .b = &b };
}
```

▷ We could have also done

```
pub fn main() void {
    const C = struct {
        const A = struct { b: *B };
        const B = struct { a: *A };
    };
    const A = C.A;
    const B = C.B;
    var a: A = undefined;
    var b: B = .{ .a = &a };
    a = .{ .b = &b };
}
```

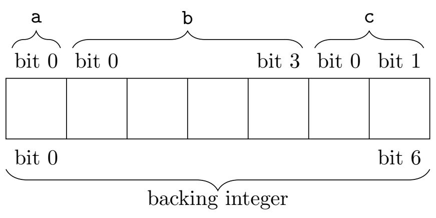
## 3.12 Packed structs

Zig's packed structs are loosely speaking a counterpart of C/C++'s bit-field feature. Rather than explicitly specifying the bit size of each field, Zig's packed structs take advantage of the fact that Zig's integer types already exactly specify the number of bits in them.

A packed struct essentially simply disables the padding between the fields, not only on the byte level, but also on the bit level, so there are no padding bytes or even padding bits between consecutive fields. Also the order of fields in memory matches their order of declaration, like with Zig's `extern struct`s. Consider the declaration:

```
const Packed = packed struct {
    a: u1,
    b: i4,
    c: u2,
};
```

This struct requires a total of 7 bits. Zig will thereby infer `u7` as a *backing integer* type for this struct. Then it will map the bit representations of the struct fields onto the bits of an `u7` integer in ascending order:



The `Packed` struct will then be represented in memory similarly to a `u7` value.

▷ Since Zig supports integers of up to $2^{16} - 1 = 65535$ bits, one can declare packed structs of up to $2^{13} - 1 = 8191$ full bytes long (plus 7 more bits).

▷ You can use `@bitCast` builtin function to reinterpret a packed struct as its backing integer and vice versa (or generally between two values of equal bitsizes):

```
const pack: Packed = .{ .a = 1, .b = -3, .c = 2 };
const backing_int: u7 = @bitCast(pack);
const signed_backing_int: i7 = @bitCast(pack);
const back: Packed = @bitCast(backing_int);
```

You can specify the backing integer explicitly for a packed struct. In this case Zig will check that the inferred total bitsize matches the one of the explicitly specified backing integer type:

```
// error: the inferred bitsize is 7 bits
const Packed = packed struct(u8) {
    a: u1,
    b: i4,
    c: u2,
};
```

```
pub fn main() void {
    _ = Packed;
}
```

▷ You may specify the backing integer type as a unsigned or signed integer:

```
// Both u7 and i7 are okay as backing integer type
const Packed = packed struct(i7) {
    a: u1,
    b: i4,
    c: u2,
};
```

▷ You can query the backing integer type, even if you didn't specify it explicitly, as

```
const BackingInteger =
    @typeInfo(Packed).@"struct".backing_integer.?;
```

## 3.13 Tuples

Differently from C++, where tuples are implemented as a standard library feature, in Zig tuples are built in. The tuples are defined as structs with omitted fields names:

```
const Tuple = struct { []const u8, i32 };
```

The above defines a 2-element tuple, the first element being a slice of bytes, the second being a 32-bit integer.

Struct vars, consts and function parameters accept (among other options) struct literals or anonymous struct literals as their initializers. Similarly, for tuples we use *tuple literals* in the respective position. The tuple literal is different from struct literal in that it doesn't specify the field names:

```
// Equivalent to:
//    const tuple = Tuple{ "string", 0 };
const tuple: Tuple = .{ "string", 0 };
```

We might realize that the type of the tuple literal value `.{ "string", 0 }` is `struct { *const [6:0]u8, comptime_int }`. During the initialization of the `tuple` const in the above, the values of the tuple's fields will be coerced to `[]const u8` and `i32` respectively.

In a way, tuples are mid-way between structs and arrays:

- Tuples are like structs whose fields do not have names. Also tuples do not have inner namespaces, that is they may not contain decls. In particular this means that tuples cannot have methods.

  Even though tuple fields do not have explicitly specified names, they get implicit names `@"0"`, `@"1"`, etc. So we could write e.g. `tuple.@"1"` to access the second file of the tuple.

- Tuples are like arrays, whose different elements may have different types. Tuples can be indexed with the `[ ]` operator and they do have a `len` field. However, due to their elements having different types, the indices used in `[ ]` must be known at compile time, since the indexing result type may vary depending on the index value. Since the value of the `len` field of tuples, like with arrays, is compile-time-known, tuples are also accommodated by the `++` and `**` operators.

  Tuples are also similar to arrays in that tuples and pointers to tuples can be iterated over with `for` statements. However since the tuple element type would be different on each iteration, the iteration has to happen in compile-time code or using an `inline for` statement (this is further discussed in Subsection 5.9.2).

The following example illustrates the previous points:

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

const Tuple = struct { []const u8, i32 };

pub fn main() void {
    const tuple: Tuple = .{ "string", 0 };
    std.debug.print(
        "{s}, {}, len={}\n",
        .{ tuple.@"0", tuple[1], tuple.len },
    );

    const concatenated = tuple ++ tuple ** 2;
    std.debug.print(
        "concatenated.len={}\n",
        .{concatenated.len},
    );
}
```

### 3.13.1 Tuple destructuring

Suppose we are calling a function, which returns a tuple:

```
    const tuple = getTuple();
```

where `getTuple()` returns a value of the previously discussed type `Tuple`. According to our previous discussion we can access individual tuple fields e.g. as

```
    const tuple = getTuple();
    const str = tuple[0]; // or tuple.@"0"
    const int = tuple[1]; // or tuple.@"1"
```

There is a shorter equivalent notation for the above:

```
    const str, const int = getTuple();
```

The latter syntax is referred to as *tuple destructuring* and is kind of equivalent to the C++ structured binding:

```
    const auto [str, integer] = getTuple();
```

Notes:

- It is also possible to use vars in the destructuring, or mix consts and vars:

  ```
  const str, var int = getTuple();
  ```

- The vars can be declared earlier:

  ```
  var int: i32 = undefined;
  const str, int = getTuple();
  ```

- If you don't need a value of one (or more) of the tuple's elements, you can use a '_' placeholder:

  ```
  const str, _ = getTuple();
  ```

## 3.14 Anonymous structs and tuples

We have already mentioned that the `struct { ..... }` construct in Zig represents a value of type '`type`'. As such, it should be usable anywhere where a type (or, more precisely, a struct type) is accepted.

We could therefore try to use the above construct to define the return type of a function:

```
fn func() struct {
    x: i32,
    y: i32,
} {
    return .{ .x = 0, .y = 0 };
}
```

Somewhat surprisingly, the above does compile and we could even use it in some way, e.g.

```
pub fn main() void {
    const a = func();
    std.debug.print("{}, {}\n", .{a.x, a.y});
}
```

This doesn't look too useful, though, as, lacking a name for the type, we cannot freely pass this struct value further.

> ▷ Similar to C/C++, two identically defined structs are still considered as different struct types in Zig:
>
> ```
> const S1 = struct { x: i32, y: i32 };
> const S2 = struct { x: i32, y: i32 };
>
> pub fn main() void {
>     const s1 = S1{ .x = 0, .y = 0 };
>
>     // The next line fails to compile,
>     // since S1 and S2 are different types
>     const s2: S2 = s1;
>     _ = s2;
> }
> ```

Notice that this is different from the situation where the struct is imported twice from one and the same file, rather that using two identical struct definitions:

```
const S1 = @import("S.zig");
const S2 = @import("S.zig");
// S1 and S2 are one and the same type
```

An anonymous definition, however, could often work quite nicely for tuples, in case a tuple is used solely as a return type for a function:

```
fn func() struct { i32, i32 } {
    return .{ 0, 0 };
}

pub fn main() void {
    const x, const y = func();
    std.debug.print("{}, {}\n", .{ x, y });
}
```

Also, differently from structs, a repeated definition of the tuple is considered as one and the same type:

```
fn func() struct { i32, i32 } {
    return .{ 0, 0 };
}

pub fn main() void {
    // Types are compatible in the next line
    const s: struct { i32, i32 } = func();
    std.debug.print("{any}\n", .{s});
}
```

This makes anonymous tuples more useful than anonymous structs.

▷ One useful application of anonymous structs in Zig is to implement closures and closure-like constructs. We'll discuss this in Section 6.15.

## 3.15 Opaque

Opaque types are structs without declared fields. E.g.:

```
// Opaque and AnotherOpaque are two different types.
// In particular, a pointer to Opaque is not compatible
// to a pointer to AnotherOpaque.
const Opaque = opaque {};
const AnotherOpaque = opaque {};
```

It's not that the underlying data has no fields, it's rather that opaque types do not publish any information about those. However they are free to publish decls, e.g.

```
pub const Opaque = opaque {
    pub const version = 1;
    pub fn reset(self: *@This()) void {
        .....
    }
};
```

As the information about the fields of opaque types is missing, the size of opaque types is not known. It is therefore not possible to create data of opaque types, but only to manipulate single-item pointers to such data. Many-item pointers to and slices of opaque data do not make sense either, as the element size of the pointed-to array would be unknown.

Opaque types are intended for implementation of facade-like interfaces, primarily across boundaries to other languages. Imagine there is a C++ library containing something like follows:

```
// ------------------------------------------
// object.h - library's public header
struct ObjectInterface {
    void setValue(int value);
    static ObjectInterace *create();
    // we omitted destroy() for brevity in this example
};

// ------------------------------------------
// object.cpp - library's private source file
#include<object.h>

class ObjectImplementation {
    int data;
    void setValue(int value) {
        this->data = value;
    }
};

void ObjectInterface::setValue(int value) {
    reinterpret_cast<ObjectImplementation *>(this)
        ->setValue(value);
}

ObjectInterace *ObjectInterface::create(int value) {
    return reinterpret_cast<ObjectInterface *>(
        new ObjectImplementation);
}
```

We could construct a Zig counterpart of `object.h` along the following lines:

```
pub const ObjectInterface = opaque {
    pub setValue(self: *@This(), value: i32) void {
        // Code that forwards the call across the language
        // boundary to C++'s ObjectInterface::setValue
    }
    pub create() *@This() {
        // Code that forwards the call across the language
        // boundary to C++'s ObjectInterface::create
    }
};
```

The details of making calls across the language boundary will be covered in Section 6.7.

# Chapter 4

# Further types

*Enums and unions. Switches. Dynamic memory. Error handling.*

## 4.1 Enums

A basic enum in Zig looks similar to the one in C/C++:

```
const Color = enum {
    red, green, blue
};
```

There are however a number of enhancements compared to C/C++.

Enums in Zig are syntactically and conceptually close to structs and it's helpful to view the former as a kind of flavor of the latter. If we consider struct fields as individual "subvariables" contained in a variable of a struct type, enum fields are rather value constants, representing the possible values of the enum. So the `red`, `green`, `blue` identifiers in the above example are considered as *enum fields* in Zig. Simultaneously, enum fields are also commonly referred to as *enum tags* in Zig, so that enum's underlying integer type is referred to as enum's *tag type*.

▷ Loosely, the choice between the "field" and "tag" terminology can be based on whether we want to highlight the struct-like aspect of an enum (in which case saying "field" might be more appropriate), or rather look at the enum as "just an enum".

Besides fields, Zig structs can contain decls, and so can enums:

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

const Color = enum {
    red,
    green,
    blue,

    // Equivalent to const default = @This().green;
    const default = .green;

    fn isDefault(self: @This()) bool {
        return self == default;
    }

    fn reset(self: *@This()) void {
        self.* = default;
    }
};

pub fn main() void {
    // Equivalent to var green = Color.green;
    // .green is a enum literal, which is then
    // coerced to the target type Color.
    var green: Color = .green;

    // Equivalent to const default = Color.default;
    // .default is a decl initializer, which is
    // looked up inside the target type Color.
    const default: Color = .default;

    std.debug.print(
        "{}, {}\n",
        .{ green.isDefault(), default },
    );
    green.reset();
    std.debug.print("{}\n", .{green});
}
```

Notice that there is no identifier shadowing in case of `green` and `default` local consts, since enum tags need a enum type as their "namespace", or at least they need to be prefixed with a dot in case of enum literals, and so also do decl initializers.

▷ Zig supports conversion of enum values to strings (returned as `[:0]const u8`):

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

const Color = enum {
    red,
    green,
    blue,

    fn print(self: @This()) void {
        std.debug.print("Value={s}\n", .{@tagName(self)});
    }
};

pub fn main() void {
    const c = Color.green;
    c.print();
}
```

Note that `@tagName` is not limited to compile-time-known argument values, but also accepts runtime values.

The inferred underlying integer type of the enum (also referred to as enum's *tag type*) can have a fractional-byte bitsize in Zig:

```
    // prints u2 as the tag type
    @compileLog(@typeInfo(Color).@"enum".tag_type);

    // alternative way to obtain the tag type, also prints u2
    @compileLog(std.meta.Tag(Color));
```

It can be overridden explicitly:

```
const Color = enum(u8) {
    red, green, blue
};
```

The integer values corresponding to enum tags, like in C/C++, can also be overridden. Unlike in C/C++, overriding these values is possible only if enum's tag type is explicitly specified:

```
const Color = enum(u8) {
    red, // auto-inferred value is 0
    green = 10,
    blue, // auto-inferred value is 11
};
```

You can convert between an enum value and its underlying integer value using `@intFromEnum` and `@enumFromInt` builtin functions:

```
@compileLog(
    @intFromEnum(Color.blue),
    @as(Color, @enumFromInt(10)),
);
```

The `@intFromEnum` function will return a value of the enum's tag type, which is already determined by the enum's type itself, while the `@enumFromInt` needs to know the target enum type to convert to. In the above example the latter is supplied using the `@as` builtin function, but more commonly it is supplied implicitly as the type of the target entity to assign the enum value to, e.g. as in

```
const color10: Color = @enumFromInt(10);
```

▷ `@enumFromInt` will check that its argument value is among the actual field values of the enum (rather than is simply fitting into the enum's underlying integer). This check, if possible, will be done at compile time, otherwise it is a runtime check in safe builds.

▷ As Zig's enum tag type inference rules are different from C/C++ (actually the latter are compiler implementation-defined), one must explicitly supply the tag type in cases of interfacing with C/C++. This will be further touched upon in Section 6.7.

## 4.2 Switches

Where there are enums, there are also switches. Switches are not restricted to usage with enums in Zig, you can also use them e.g. with integer types. However enums offer a really good showcase for Zig switches, for that reason we have been intentionally delaying the discussion of switches until now.

Consider the following example code:

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

const Color = enum {
    red,
    green,
    blue,

    fn print(self: @This()) void {
        const prt = std.debug.print;
        switch (self) {
            .red => prt("red\n", .{}),
            .green => {
                prt("green\n", .{});
                prt("(it looks nice!)\n", .{});
            },
            .blue => prt("blue\n", .{}),
        }
    }
};

pub fn main() void {
    const c = Color.green;
    c.print();
}
```

▷ The `prt` const used in the switch example code is introduced for brevity. Its initialization to a function value (rather than a pointer to a function) works because `ptr` will be a compile-time constant (like `constexpr` locals in C++).

In fact, `std.debug.print` is a generic function (similar to function templates in C++). You could use a pointer to it, but such pointer is possible only in compile-time code.

Notes:

- There should be exactly one statement (*terminated by a comma*, rather than a semicolon) per each "case" of the switch.
- Actually, the "switch cases" are referred to as *prongs* in Zig.
- The single-statement-per-prong requirement is the reason we used a braces-enclosed code block in the green color's prong.
- Unlike in C++, there is no `break` statement required at the end of each prong. However you still *can* use `break` statements in switches, this is discussed further below.
- There is no "fallthrough" at the end of the prong in case of a forgotten or intentionally omitted `break`. In necessary, you can achieve a kind of a fallthrough equivalent using switch `continue` functionality discussed further below.
- The prongs must cover all possible values of the switch's argument type (in our case, `Color`). If you omit any of the values it would be at most a warning in C/C++, but it is an error in Zig. In order to explicitly ignore some of the values you need to use the Zig's equivalent of C/C++'s `default` case, which is the `else` prong, discussed further below.
- The prong values are not restricted to literals, you can use any expressions, as long as these expressions can be evaluated at compile time:

```
fn print(self: @This()) void {
    const prt = std.debug.print;
    const swap = true;
    switch (self) {
        if (swap)
            .blue
        else
            .red => prt("red\n", .{}),
        .green => {
            prt("green\n", .{});
            prt("(it looks nice!)\n", .{});
```

```
                },
                if (swap)
                    .red
                else
                    .blue => prt("blue\n", .{}),
            }
        }
```

### 4.2.1 “Else” prong

The `else` prong is a kind of counterpart of the C/C++’s `default` branch of a switch statement:

```
fn print(self: @This()) void {
    const prt = std.debug.print;
    switch (self) {
        .red => prt("red\n", .{}),
        .green => {
            prt("green\n", .{});
            prt("(it looks nice!)\n", .{});
        },
        else => prt("some other color\n", .{}),
    }
}
```

You can further distinguish between the values within the `else` prong using a capture:

```
fn print(self: @This()) void {
    const prt = std.debug.print;
    switch (self) {
        .green => {
            prt("green\n", .{});
            prt("(it looks nice!)\n", .{});
        },
        else => |color| prt(
            "{s}\n",
            .{@tagName(color)},
        ),
    }
}
```

### 4.2.2 Multivalued prongs

In C/C++ you can combine multiple cases by listing them one by one without `break` statements in between. This makes use of the fallthrough behavior of switch branches in C/C++:

```
void print(Color color) {
    switch (color) {
        case green:
                printf("green\n");
                printf("(it looks nice!)\n");
                break;
            case red:
            case blue:
                printf("some other color\n");
                break;
        }
    }
```

In Zig we don't have switch branch fallthrough, so we use a different syntactical feature instead. We simply list the multiple values (separated by commas) within one prong:

```
    fn print(self: @This()) void {
        const prt = std.debug.print;
        switch (self) {
            .green => {
                prt("green\n", .{});
                prt("(it looks nice!)\n", .{});
            },
            .red, .blue => prt(
                "some other color\n",
                .{},
            ),
        }
    }
```

Like with the `else` prong, you can use a capture:

```
    fn print(self: @This()) void {
        const prt = std.debug.print;
        switch (self) {
            .green => {
                prt("green\n", .{});
                prt("(it looks nice!)\n", .{});
            },
            .red, .blue => |color| prt(
                "{s}\n",
                .{@tagName(color)},
            ),
        }
    }
```

You can also specify value ranges. The following example illustrates using value ranges when switching on an `i32`:

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

fn printSquared(x: i32) void {
    switch (x * x) {
        0...(10 - 1) => |x2| std.debug.print(
            "single-digit: {}\n",
            .{x2},
        ),
        else => |x2| std.debug.print(
            "multi-digit: {}\n",
            .{x2},
        ),
    }
}

pub fn main() void {
    printSquared(1);
}
```

Notice that the right bound 9 is included into the range. Also notice that the ranges employed in the switch use three dots as a separator between the range bounds rather than two dots used in the ranges used in slices and for loops. So two-dot ranges exclude the right bound, while three-dot ranges (which can be used only in switch prongs) include the right-bound.

One can also use multiple ranges and mix them with single values within one prong:

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

fn printChar(char: u8) void {
    switch (char) {
        // We do not really need the capture 'c',
        // since it's equal to 'char', but let's
        // use it for the sake of demonstration.
        'A'...'Z', 'a'...'z', '_' => |c| std.debug.print(
            "alpha: {x}\n",
            .{c},
        ),
        else => |c| std.debug.print(
            "other: {x}\n",
            .{c},
        ),
    }
}

pub fn main() void {
    printChar('x');
}
```

### 4.2.3 Breaking from a switch

You can break from a switch prong prematurely by using a `break` statement:

```
    fn print(self: @This()) void {
        const prt = std.debug.print;
        sw: switch (self) {
            .red => prt("red\n", .{}),
            .green => {
                prt("green\n", .{});
                const in_good_mood =
                    std.time.timestamp() & 1 == 0;
                if (!in_good_mood)
                    break :sw;
                prt("(it looks nice!)\n", .{});
            },
            .blue => prt("blue\n", .{}),
        }
    }
```

Notes:

- The `break` statement which is being used to break from a switch needs a label (otherwise it'll look for the innermost loop to break). This is why we labeled the switch statement in the above code.

- We enclosed the `break` statement into a runtime conditional statement, so that Zig doesn't complain about the unreachable code after the `break`. We introduced a concept of the "mood of the program code" merely in order to endow this condition with some semantics.

### 4.2.4 Continuing a switch

In C/C++ you can `continue` a loop. In Zig you can also `continue` switches. Continuing a switch is basically a goto jump to a different switch prong. E.g.

```
    fn print(self: @This()) void {
        const prt = std.debug.print;
        sw: switch(self) {
            .red => prt("red\n", .{}),
            .green => {
                prt("green\n", .{});
                prt("but how about... ", .{});
                continue :sw .red;
            },
            .blue => prt("blue\n", .{}),
        }
    }
```

Notes:

- Similarly to the `break` statement, the `continue` switch statement needs a switch label. It also needs a value. The switch `continue` statement then would effectively jump to the switch header again, passing this value to the switch, as if it was the switch's argument.

  In our example we are passing the `.red` value, so the switch will jump to the "red" prong.

- If the value supplied to the `continue` statement is compile time known, the compiler will generate an unconditional jump to the respective prong.

Otherwise runtime dispatching will be performed based on the supplied value.

- It is possible to "emulate" the "fallthrough" behavior of C++ switch cases not having a break at the end by passing the next prong's value to the `continue` statement.

  ```
  fn print(self: @This()) void {
      const prt = std.debug.print;
      sw: switch(self) {
          .red => prt("red\n", .{}),
          .green => {
              prt("green\n", .{});
              prt("but how about... ", .{});
              continue :sw .blue; // kind of fallthough
          },
          .blue => prt("blue\n", .{}),
      }
  }
  ```

  Since the value `.blue` supplied to the `continue` statement is compile-known the compiler is expected to generate a similar code to the one generated for "fallthrough" in C++.

### 4.2.5 Switch expressions

Like `if`s and loops in Zig, switches can be used as expressions. In order to do so, simply use non-void prongs:

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

const Color = enum {
    red,
    green,
    blue,

    fn shrinkWavelength(self: @This()) @This() {
        return switch (self) {
            .red => .green,
            .green => .blue,
            .blue => @panic("Already minimum wavelength"),
        };
    }
};

pub fn main() void {
    const c = Color.green;
    std.debug.print(
        "{}\n",
        .{c.shrinkWavelength()},
    );
}
```

Notes:

- The first ("red") prong is yielding the value `@This().green` as the switch's resulting value. The implied `@This()` in `@This().green` is coming from the result type of the function `shrinkWavelength`, which causes the enum literal `.green` to be coerced into `@This().green`.

  A similar thing is happening in the second prong.

- The builtin `@panic` function acts like a hard-coded failing assertion with a message. In safe builds this function never returns, but instead prints the message and exits the program. Like `std.debug.assert` it also provides a hint to the optimizer in non-safe builds.

▷ The `shrinkWavelength` function could have been implemented in a simpler way:

```
fn shrinkWavelength(self: @This()) @This() {
    return @enumFromInt(@intFromEnum(self) + 1);
}
```

This also includes panicking on out-of-range result (occurring inside `@enumFromInt`).

Here is another example:

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

const Color = enum {
    red,
    green,
    blue,

    fn toString(self: @This()) []const u8 {
        return switch (self) {
            .red => "red",
            .green => "green",
            .blue => "blue",
        };
    }
};

pub fn main() void {
    const c = Color.green;
    std.debug.print(
        "{s}\n",
        .{c.toString()},
    );
}
```

Here the 3 different string types `*const [3:0]u8`, `*const [5:0]u8` and `*const [4:0]u8` are coerced into the function's result type `[]const u8`.

▷ If the prongs have void-type values, the switch still can be formally used as expression, which has a void-type value:

```
fn print(self: @This()) void {
    const prt = std.debug.print;
```

```
        return switch (self) {
            .red => prt("red\n", .{}),
            .green => {
                prt("green\n", .{});
                prt("(it looks nice!)\n", .{});
            },
            .blue => prt("blue\n", .{}),
        };
    }
```

If you compare this implementation of the **print** function to the one we have introduced earlier, you can notice that this new one explicitly forwards the void value returned by the switch to the function's return value, since the entire switch is now the argument of the **return** statement. Also notice the semicolon after the switch, which wasn't present in the other implementation.

▷ If the prongs return values of different types while there is no specified result type, we are having another case of *peer type resolution* discussed in Section 2.5:

```
// Use var to prevent compile-time knowledge
// of what the 'value' is equal to
var value: u32 = 0;

pub fn main() void {
    const result = switch (value) {
        0 => 1,
        else => value,
    };
    @compileLog(result);
}
```

## 4.3 Non-exhaustive enums

We have mentioned that `@enumFromInt` will generally refuse to accept an integer value for which there is no corresponding tag in the enum, even if the integer value fits into the enum's underlying integer type. It is possible to bypass this safety limitation in Zig by declaring the enum as *non-exhaustive*.

A non-exhaustive enum is specified by declaring a field named '_' as the last field in the enum. This enum tag then stands for "all remaining values of the enum's underlying integer type". Respectively a non-exhaustive enum can accommodate all possible values of the underlying integer type.

Imagine we wanted to use the previously discussed `Color` enum type for binary streaming, where we also want forward compatibility to future versions of the software, possibly supporting more colors than the current version. Upon reading from a stream, the color values which are unknown to the current version will not be semantically recognized, but will still be passed over through the data structures and finally streamed back unchanged upon stream writing. We could consider declaring the `Color` enum like this:

```
const Color = enum(u8) {
    red,
    green,
    blue,
    _,
};
```

We intend to write and read the `Color` values as `u8` in binary streams. For that reason we specified the tag type explicitly as `u8`. The explicit tag type is also needed by the compiler to infer the set of values implied by the '_' tag, therefore with non-exhaustive enums the explicit tag type specification is required rather than optional, like it is with ordinary enums.

The above `Color` enum allows us to obtain the value to be written into the stream simply as `@intFromEnum(color)` (where the `color` variable has type `Color`), and conversely to convert an `u8` value read from the stream into a `Color` value by:

```
const color: Color = @intFromEnum(streamed_u8_value);
```

Due to the presence of the '_' tag there will be no runtime panic in case of `streamed_u8_value` falling outside of the set of "normal" known enum values `red`, `green` and `blue`, as the outside values will be also considered a part of the enum, represeted by the '_' tag.

The color methods for such non-exhaustive version of the `Color` enum will need to be adapted. E.g. the `toString` method could look like follows:

```
    fn toString(self: @This()) []const u8 {
        return switch (self) {
            .red => "red",
            .green => "green",
            .blue => "blue",
            _ => "<unknown>",
        };
    }
```

Notes:

- The '_' prong of the switch handles all the values implied by the '_' tag of the enum. This prong is only available for non-exhaustive enums.
- Similarly to the `else` prong, the '_' prong may use a capture to distinguish between different values accommodated under the '_' tag:

```
fn printColor(color: Color) void {
    const prt = std.debug.print;
    switch (color) {
        .red => prt("red\n", .{}),
        .green => prt("green\n", .{}),
        .blue => prt("blue\n", .{}),
        _ => |unknown_color| prt(
            "<unknown:{}>\n",
            .{unknown_color},
        ),
    }
}
```

- The difference between the `else` and the '_' prongs is that the '_' prong denotes the "remaining" values of the enum type, while the `else` prong denotes the "remaining" values of the switch. Consider the following modification of the previous `printColor` function:

  ```
  fn printColor(color: Color) void {
      const prt = std.debug.print;
      switch (color) {
          .red => prt("red\n", .{}),
          .green => prt("green\n", .{}),
          else => |unknown_color| prt(
              "<unknown:{}>\n",
              .{unknown_color},
          ),
      }
  }
  ```

  Here we omitted the `.blue` prong plus we used an `else` prong instead of a '_' prong. The effect is that now the blue color value will fall under the `else` prong and will be respectively printed as "unknown". Had we used the '_' prong here instead, the compiler would have complained about the missing prong for the `.blue` value.

▷ At the time of the writing it is not possible to simultaneously use the `else` and the '_' prongs in a switch. The Zig issue 12250 might address it in future.

## 4.4 Unions

A basic union is declared in Zig like follows:

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

const Value = union {
    int: i32,
    float: f32,
    bool: bool,
};

pub fn main() void {
    // Equivalently we could have written:
    //     const value = Value{
    //         .float = 1.5,
    //     };
    const value: Value = .{
        .float = 1.5,
    };
    std.debug.print("{}\n", .{value.float});
}
```

The corresponding C++ code:

```
#include<stdio.h>

union Value {
    int m_int;
    float m_float;
    bool m_bool;
};

int main() {
    const Value value = {
        .m_float = 1.5
    };
    printf("%f\n", value.m_float);
    return 0;
}
```

Notes:

- So, unions are basically treated by Zig as structs and follow pretty much the same syntax, except that unions may have only one of the fields active at the time. In the above example we have specified the `.float` field in the initialization of the `value` local const. It would not have been possible to specify more than one field simultaneously there.

- In safe builds Zig is tracking the union field activity state in an internal part of the union data. This state is being checked during union field accesses, causing a runtime panic upon an attempt to access an inactive field. By default this state is only used by the safety-checking code generated by the compiler and cannot be explicitly accessed from Zig program's code. It is however possible to make this state explicitly accessible by using the tagged unions feature discussed in Section 4.5.

- Note that the active field safety checking mechanism will prevent code like the following from running:

  ```
  pub fn main() void {
      var value: Value = .{
          .float = 1.5,
      };
      value.int = 1; // panics due to access to inactive field
      std.debug.print("{}\n", .{value.int});
  }
  ```

  In order to change the active field one needs to assign the entire union, like in the initialization:

  ```
  pub fn main() void {
      var value: Value = .{
          .float = 1.5,
      };
      value = .{ .int = 1 }; // this works
      std.debug.print("{}\n", .{value.int});
  }
  ```

- Since unions, like enums, are special kinds of structs in Zig, besides fields they may contain decls (including methods and other functions). E.g.:

  ```
  const Value = union {
      int: i32,
      float: f32,
      bool: bool,

      // Can be used as decl initializer
      const default = @This(){ .int = 0 };
  };
  ```

- We didn't need to declare the `bool` field of the `Value` union as `@"bool"`, because the namespace of the fields of a union doesn't conflict with the namespace of global types. There is no syntactic construct, where there would be an ambiguity as to which of the two `bool`s is meant. E.g.

  ```
  pub fn main() void {
      const value: Value = .{
          // Obviously, here we mean the 'bool' field
          .bool = false,
      };
      // Obviously, here we mean the bool type
      const bool_value: bool = true;
      std.debug.print(
          "{} {}\n",
          .{
              value.bool, // Obviously we mean the field
              bool_value,
          },
      );
  }
  ```

▷ Like with Zig's **`struct`** types, the memory layout of Zig unions is not compatible to C ABI. In particular, the hidden activity state is not a part of the memory layout of C unions. Like with Zig structs, C-compatible unions can be declared in Zig using an **`extern`** keyword:

```
// This union is C-compatible
const Value = extern union {
    int32: i32,
    float: f32,
    double: f64,
};
```

Notice that, since C-compatible unions do not contain the hidden activity state, they cannot cause runtime panic upon inactive field accesses.

▷ Similarly to packed structs, Zig also has packed unions. These unions can have fractional-byte bitsizes and are intended for usage inside packed structs:

```
// This union has a size of 4 bits (if used in a packed struct)
const PackableValue = packed union {
```

```
    int: i4,
    uint: u4,
    bool: bool,
};

const Packed = packed struct(u8) {
    value: PackableValue,
    padding: u4,
};
```

## 4.5 Tagged unions

Tagged union declarations take an enum type and establish a 1:1 matching (based on the field names) between the fields of the supplied enum and the fields declared in the union:

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

// We'll use this struct to demonstrate some shortcuts
const Empty = struct {};

const Type = enum {
    int,
    float,
    bool,
    void,
    empty,
};

const Value = union(Type) {
    // For each field of the 'Type' enum we must
    // declare a union field with the same name.
    int: i32,
    float: f32,
    bool: bool,
    void: void,
    empty: Empty,
    // Field and type names do not collide. So we can
    // declare fields 'bool' and 'void' without having
    // to enclose them into @"".

    fn print(self: @This()) void {
        const prt = std.debug.print;

        // We can switch on the value of 'self' as if
        // its type was 'Type'. But we also can use
        // captures (if we want to) to access the payloads.
        switch (self) {
            .int => |value| prt("int:{}\n", .{value}),
            .float => |value| prt("float:{}\n", .{value}),
            .bool => |value| prt("bool:{}\n", .{value}),
            .void => prt("void\n", .{}),
            .empty => prt("empty\n", .{}),
        }
    }
};

pub fn main() void {
    const value: Value = .{
        .float = 1.5,
    };
    value.print();
}
```

Notes:

- With tagged unions the field activity state becomes an explicit part of the program logic. This state is stored inside the tagged union's data (regardless of safe/non-safe build modes) and is represented by a value of type `Type` (or whatever enum type was supplied to the union upon its declaration).

  The field activity state can be accessed by coercing the tagged union value to the respective enum type. E.g.:

  ```
  // The name of the 'type' parameter will collide
  // with Zig's standard type ‘type’, so we need to
  // enclose it in @"".
  fn isNumeric(@"type": Type) bool {
      return switch (@"type") {
          .int, .float => true,
          .bool, .void, .empty => false,
      };
  }

  pub fn main() void {
      const value: Value = .{ .float = 1.5 };
      std.debug.print("{}\n", .{isNumeric(value)});
  }
  ```

- The coercion also works in another direction to quite an extent. This could be somewhat surprising, as a tagged union value, besides the activity state also needs the active tag's payload. So the limitation is that the “reverse direction” coercion is possible whenever the payload's value is not needed:

  ```
  pub fn main() void {
      var value: Value = undefined;

      // Coercion from Type allowed, no need for payload
      // for void (or also for structs with no members)
      value = Type.void;
  ```

```
    value = Type.empty;

    // This is the same as the above
    value = .void;
    value = .empty;

    // We don’t really have to use coercion from Type
    // with ’void’ or ’empty’, it was only an option:
    value = .{ .void = {} };
    value = .{ .empty = .{} };

    // The ’Value.void’ literal actually has type ’Type’
    // and means ’Type.void’, as can be checked by calling
    // @compileLog(Value.void).
    // So the line below still implies a coercion.
    value = Value.void;

    // Coercion doesn’t work here, as a payload is needed:
    //    value = Type.float;

    // But this is fine, @tagName expects an enum value
    // and Value.float actually means Type.float
    std.debug.print("{s}\n", .{@tagName(Value.float)});
}
```

- Tagged union value can be compared to its enum tag type:

  ```
  pub fn main() void {
      const value: Value = .{
          .float = 1.5,
      };

      // Same as: if (@as(Type, value) == Type.float)
      if (value == .float)
          std.debug.print("Is a float\n", .{});
  }
  ```

  This is similar to what was happening in the switch inside the `Value.print` function in the main example.

- Peer type resolution between tagged union and its enum tag type produces the former:

  ```
  // use a var to prevent compile-time checking
  var condition = true;

  pub fn main() void {
      const value = if (condition)
          Type.void // will be coerced to Value
      else
          Value{ .float = 1.5 };
  ```

```
    @compileLog(value); // value's type is Value
}
```

This allows to rewrite the above code as:

```
pub fn main() void {
    const value = if (condition)
        .void
    else
        Value{ .float = 1.5 };
    @compileLog(value); // value's type is Value
}
```

- You can use capturing by reference to modify the stored values directly. E.g. we could add the following method to the `Value` union:

  ```
  fn reset(self: *@This()) void {
      switch (self.*) {
          .int => |*value| value.* = 0,
          .float => |*value| value.* = 0,
          .bool => |*value| value.* = false,
          .void, .empty => {}, // do nothing
      }
  }
  ```

- Actually, you don't have to declare the enum "manually". Instead you can have it automatically constructed from the union's field names:

  ```
  const Empty = struct {};

  // The enum tag type will be created implicitly
  const Value = union(enum) {
      int: i32,
      float: f32,
      bool: bool,
      void: void,
      empty: Empty,
  };

  pub fn main() void {
      var value: Value = undefined;

      // The line below can also be written as:
      //     value = Value.void;
      // where the type of Value.void is the
      // implicitly created enum type.
      value = .void;
  }
  ```

  The enum type associated with such union (the union's *tag type*) can be obtained via:

```
const Type = @typeInfo(Value).@"union".tag_type.?;
```

or

```
const Type = std.meta.Tag(Value);
```

▷ The same way can also be used to obtain the tag type of a union whose declaration explicitly specifies the enum type (like the earlier presented version of the `Value` union).

## 4.6 Dynamic memory

In C/C++ it is possible to use multiple allocators, but commonly a single global memory allocator is implicitly used. Zig promotes a convention of explicit usage of allocators. The following example demonstrates a basic memory-allocating setup:

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

const SomeStruct = struct {
    x: i32 = 0,
    y: i32 = 0,
};

pub fn main() void {
    // std.heap.DebugAllocator(.{}) defines a
    // default-configured debug allocator type
    // (the .{} in parentheses is configuration options
    // initializer). 'dbga' is an object of that type,
    // which is default-initialized by "= .{}".
    // The same line could also have been written as:
    // var dbga = std.heap.DebugAllocator(.{}){};
    var dbga: std.heap.DebugAllocator(.{}) = .{};

    // The deinit() call, among other things, checks for
    // memory leaks and returns .ok if none are found.
    // Remember that std.debug.assert is more like
    // BOOST_VERIFY, so the deinit() call will also be
    // done in non-safe build modes.
    // NB: as the assert condition is also used as the
    // optimizer hint, in real code it might be safer
    // to call @panic() instead of using an assert.
    defer std.debug.assert(dbga.deinit() == .ok);

    // Obtain the abstract allocator interface, which
    // we can then use to allocate/deallocate memory.
    // The type of 'alloc' is 'std.mem.Allocator',
    // this is the standard interface type used across
    // the entire "std" library.
    const alloc = dbga.allocator();
    // Allocate memory for a value of SomeStruct type.
    // 'some_struct' receives a value of '*SomeStruct'
    // type. Notice that the pointer is not optional,
    // hence it is never null.
    const some_struct = alloc.create(SomeStruct) catch {
        std.debug.print("Allocation failure\n", .{});
        return;
    };

    // Schedule memory release
    defer alloc.destroy(some_struct);

    // The allocated memory is not initialized,
    // we have to do it manually. The following
    // line initializes x and y fields to their
    // default values.
    some_struct.* = .{};

    // If some_struct required deinitialization,
    // this would be a place to schedule it:
    //    defer some_struct.deinit();

    // Pretend we are doing something useful with
    // the allocated struct.
    std.debug.print(
        "x={}, y={}\n",
        .{ some_struct.x, some_struct.y },
    );

    // Here all scheduled deinitializations would
    // be executed:
    //    // if scheduled: some_struct.deinit();
    //    alloc.destroy(some_struct);
    //    std.debug.assert(dbga.deinit() == .ok);
}
```

Notes:

- Generally, in Zig you have to explicitly construct the allocator object, where the `std` library offers a selection of allocator types (you can provide your own ones as well, of course). In the above example we are constructing an allocator of `std.heap.DebugAllocator(.{})` type, which is a kind of default allocator type choice during program development, unless specific considerations suggest other choices. The `.{}` in parentheses is the type's configuration argument (`DebugAllocator` is not a type, but, using C++ terminology, a type template, parameterized by this configuration argument).

- In C/C++ the standard allocator will be deinitialized by the startup code. As Zig's startup has no knowledge of the allocator used by us (nor does the allocator object exist upon returning from `main`), it is our job to deinitialize the allocator.

For brevity, we used the assertion for checking the heap state returned by the `deinit`. As the assertion also gives a hint to the optimizer, a dirty heap state could theoretically (although rather unlikely) cause further undefined behavior after this point. It might be a bit safer, in production code, to write something like:

```
defer {
    if (dbga.deinit() != .ok and std.debug.runtime_safety)
        @panic("Heap dirty upon shutdown");
}
```

On the other hand, in production we might choose to use a different allocator altogether (in particular, see Subsection 4.6.1).

- The allocator objects themselves provide functionality specific to the given allocator type (usually related to the allocator's configuration). The actual allocation is supposed to occur through a standardized interface of `std.mem.Allocator` type, which thereby hides the differences between different allocators. This interface internally contains a pointer to a virtual function table and a type-erased pointer to the allocator and thereby may be freely copied around and/or passed by value to other functions. E.g.

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

const SomeStruct = struct {
    x: i32 = 0,
    y: i32 = 0,
};

fn useAllocator(alloc: std.mem.Allocator) void {
    // 'catch return' causes the function to silently
    // return upon allocation failure
    const some_struct =
        alloc.create(SomeStruct) catch return;
    defer alloc.destroy(some_struct);

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

pub fn main() void {
    var dbga: std.heap.DebugAllocator(.{}) = .{};
    defer std.debug.assert(dbga.deinit() == .ok);

    const alloc = dbga.allocator();
    useAllocator(alloc);
}
```

- The Zig line calling `alloc.create()` is roughly comparable to C code:

  ```
  struct SomeStruct *some_struct =
      malloc(sizeof(struct SomeStruct));
  if(!some_struct) {
      printf("Allocation failure\n");
      return;
  }
  ```

  with the reservation that scheduled deinitializations (if any) would need to be manually invoked before returning, as C has no `defer`.

  Respectively the scheduled call `alloc.destroy()` is roughly comparable to C's:

  ```
  free(some_struct);
  ```

- The `catch` operator is part of Zig's error-handling mechanism and shall be discussed in Section 4.7.

### 4.6.1 SmpAllocator

Introduced in Zig 0.14.0, the `SmpAllocator` is intended to be used in optimized unsafe builds instead of `DebugAllocator` (the latter was formerly known as `GeneralPurposeAllocator` and was intended to be used in all types of builds, this has changed in 0.14.0).

Differently from `DebugAllocator`, of which, at least in theory, one could create multiple instances, the `SmpAllocator` is supposed to exist in a single instance. This instance is already provided by the `std` library. More precisely, the `SmpAllocator`'s interface (of `std.mem.Allocator` type) is directly provided, so the `main` function from our previous example becomes simply

```
pub fn main() !void {
    useAllocator(std.heap.smp_allocator);
}
```

That is we do not need to create the allocator, or obtain the allocator's interface, this interface is directly available as `std.heap.smp_allocator`.

In order to automatically switch between `DebugAllocator` and `SmpAllocator` one could e.g. use code along the following lines:

```
var dbga: std.heap.DebugAllocator(.{}) = .{};
const use_dbga = std.debug.runtime_safety;

fn initAlloc() std.mem.Allocator {
    return if (use_dbga)
        dbga.allocator()
    else
        std.heap.smp_allocator;
}
```

```
fn deinitAlloc() void {
    if (use_dbga)
        std.debug.assert(dbga.deinit() == .ok);
}

pub fn main() !void {
    const alloc = initAlloc();
    defer deinitAlloc();

    useAllocator(alloc);
}
```

Notice that in unsafe builds the `dbga` variable will not be referenced (as `use_dbga` condition will be compile time-known to be false) and is likely not to be generated by the compiler at all.

### 4.6.2 Further allocators

A number of further allocators is defined in `std.heap` and `std.testing`. To highlight a few:

- `std.heap.ArenaAllocator` – this one doesn't release the memory until the entire allocator is reset, which allows for faster operation of the allocator.
- `std.heap.FixedBufferAllocator` – this one operates within a provided fixed buffer, thereby not causing any real heap allocations whatsoever (but one might relatively easily run out of the buffer, resulting in allocation failures).
- `std.testing.allocator` – this one is supposed to be used in tests (see Section 6.2).

## 4.7 Error handling

In the example code demonstrated in Section 4.6 we have used the `catch` operator to handle allocation errors, but we haven't explained how it works. Now it is time to discuss this.

▷ As we shall see, the `try` and `catch` operators used in Zig's error handling are quite different from the same operators in C++. The former do not handle exceptions. There are no exceptions (and the associated hidden control transfer propagation) in Zig. The control transfer in Zig, including the error handling cases, is occurring explicitly.

### 4.7.1 Error unions

The `alloc.create()` function in the example code from Section 4.6 doesn't actually return a pointer value of the type `*SomeStruct`, instead it returns an *error union* containing `*SomeStruct` value. In C terms this error union can be loosely thought of as the following struct:

```
struct AllocationResult {
    struct SomeStruct *allocation;
    uint16_t error_code;
};
```

The `catch` operator performs the action of *unwrapping* the error union. It checks the error code, and if the code indicates no error, it returns the actual value contained in the union. Otherwise it executes the code following the catch statement. Thereby the combination of `alloc.create()` and `catch` in our example code from Section 4.6 can be loosely written in C terms as

```
struct AllocationResult alloc_result = alloc_some_struct();
struct SomeStruct *some_struct;
if (alloc_result.error_code == NO_ERROR) {
    some_struct = alloc_result.allocation;
} else {
    printf("Allocation failure\n");
    return;
}
```

In C++ terms Zig's error unions can be compared to `std::expected`:

```
std::expected<SomeStruct *, Error> alloc_result =
    allocate<SomeStruct>();
SomeStruct *some_struct;
if (alloc_result) {
    some_struct = *alloc_result;
} else {
    printf("Allocation failure\n");
    return;
}
```

One could also loosely try to express Zig's error unions in terms of Zig's unions:

```
const AllocationResult = union(Error) {
    no_error: *SomeStruct,
    _: void,
    // The '_' field name above is not really accepted by Zig, but
    // we use it to loosely express the idea that all other union
    // fields contain void payloads. Intuitively, this would be a
    // "non-exhaustive union" (if there was such thing in Zig).
}
```

(where `Error` would be an enum of possible errors). This is probably the reason why this construct is referred to as error union.

Now let's look into further details of working with error unions. Before we do that, we need to look at error values and error sets.

### 4.7.2 Error sets

An *error set* could be loosely thought of as an enum, which is declared with an intention of representing error values.

Imagine we implemented a non-blocking allocator, which, if being in use by another thread, would immediately return. This allocator would possibly want to return the following errors:

```
const NonblockingAllocError = error{
    OutOfMemory,
    AllocatorInUse,
};
```

So in the above code excerpt we have declared an *error set* which contains two error values: `OutOfMemory` and `AllocatorInUse`.

The respective error values *can* be produced as demonstrated by the following example:

```
// This dummy implementation returns only an error,
// the actual implementation would have to return
// an error union.
fn nonBlockingAlloc(size: usize) NonblockingAllocError {
    _ = size;
    return NonblockingAllocError.OutOfMemory;
}
```

It is also possible to write `return error.OutOfMemory;` with the same effect, but in many cases it might be a good idea to explicitly qualify the error with the respective error set, whenever possible/makes sense. In our above example this qualification is not so much necessary, since the compiler will check that the returned error is contained in the returned type anyway:

```
fn nonBlockingAlloc(size: usize) NonblockingAllocError {
    _ = size;
    // The compiler will complain if error.OutOfMemory
    // is not contained in NonblockingAllocError declaration.
    return error.OutOfMemory;
}
```

▷ The error sets, while looking similarly to enums, have an essential difference, which we shall discuss in Subsections 4.7.5 and 4.7.6. This will also include the reason for `NonblockingAllocError.OutOfMemory` and `error.OutOfMemory` being equivalent.

### 4.7.3 Producing error unions

Let's now turn the previously introduced `NonblockingAllocError` into an error union, which we then could return from an allocation function:

```
fn nonBlockingAlloc(
    size: usize,
) NonblockingAllocError![]u8 {
    const locked = lockAllocator();
    if (!locked)
        return NonblockingAllocError.AllocatorInUse;
    defer unlockAllocator();
    if (allocMemorySlice(size)) |slice|
        return slice;
    // For the sake of demonstration, let's use the
    // 'error.' prefix instead of 'NonblockingAllocError.'
    return error.OutOfMemory;
}
```

Notes:

- We didn't attempt to implement the Zig's standard `std.mem.Allocator` interface for our allocator, but it's not just because that would've required noticeably more code. Actually it would have been impossible as `std.mem.Allocator` interface can return only one kind of error ("out of memory"), and that wouldn't have covered our current example case.

  Also, for the sake of brevity. our implementation sketch implements a kind of global allocator function, while Zig typically (although not always) expects allocators to be implemented as structs which potentially could be instantiated multiple times.

- The return type `NonblockingAllocError![]u8` is an *error union*. A value of this type either contains a value of type `[]u8` or a value of type `NonblockingAllocError`. The approximate details of memory representation of this error union have been touched before, but to repeat and to summarize, this value consists of a `[]u8` and an integer (typically `u16`) containing the error code. One of the error code values (which is implicitly generated by the compiler and is not explicitly available as a part of the error set) means "no error", which technically allows the program code to tell, whether the error union contains a "normal value" or an error.

  Respectively the values of both `[]u8` and `NonblockingAllocError` coerce to the return type of the function, meaning that we can return values of both of these types from inside the function's code.

### 4.7.4 Unwrapping of error unions

The error unions are similar to Zig's optionals in that they either carry the "payload" or they carry a "fallback value", the latter being `null` in case of optionals and an error value in case of error unions. The unwrapping of error unions is also done similarly to the one of optionals:

```
pub fn main() void {
    const SomeStruct = struct {
        ........
    };
    if (nonBlockingAlloc(@sizeOf(SomeStruct))) |bytes| {
        const some_struct =
            std.mem.bytesAsValue(SomeStruct, bytes);
        some_struct.* = .{ .x = 0, .y = 0 };
        .........
    } else |err| {
        std.debug.print("Failed: {s}\n", .{@errorName(err)});
    }
}
```

Notes:

- You must provide the "else" branch to an `if` which unwraps an error union. Zig doesn't make it easy to simply discard an error value without making some effort to handle it.

- You must capture the error value in the "else" branch. If you have no intention of using the detailed error information you can provide the '_' placeholder as the capture:

  ```
  } else |_| {
      std.debug.print("Failed\n", .{});
  }
  ```

  otherwise it's common to switch on the error value:

  ```
  } else |err| switch (err) {
      error.OutOfMemory => std.debug.print(
          "Out of memory\n",
          .{},
      ),
      error.AllocatorInUse => std.debug.print(
          "AllocatorInUse\n",
          .{},
      ),
  }
  ```

- Similarly to how unwrapping of optionals has the `orelse` shortcut operator, unwrapping of error unions has a `catch` shortcut operator. The following construct:

  ```
  if(A) |payload| payload else |err| B(err)
  ```

  (where `if` can be used as a statement or as an expression) has a shorter notation:

  ```
  A catch |err| B(err)
  ```

  Further, in cases where the error value is not used:

  ```
  if(A) |payload| payload else |_| B
  ```

  there is an even shorter version:

  ```
  A catch B
  ```

  Thus, the `catch` operator used in the example code in the beginning of Section 4.6 could be equivalently rewritten as follows:

  ```
  const some_struct = if (alloc.create(SomeStruct)) |ptr|
      ptr
  else |_| {
      std.debug.print("Allocation failure\n", .{});
      return;
  };
  ```

The `catch` operator is thereby similar to the `value_or` method of C++'s `std::expected`, with the difference that the latter always evaluates its argument, while `catch` does so only if the error union contains an error.

- Similarly to the following idiomatic `noreturn` pattern used with optionals:

  ```
  const value = optional_valued_expr orelse return;
  ```

  there is a counterpart for error unions:

  ```
  const value = error_union_valued_expr catch return;
  ```

  which is *somewhat* commonly used with function calls returning error unions:

  ```
  const value = func() catch return;
  ```

  Notice that thereby we do not really handle the error, this construct is more of a quick safety guard and might be more appropriate for work-in-progress code or for really unexpected errors. A more appropriate error-handling construct is:

  ```
  const value = func() catch |err| return err;
  ```

  The latter has its own shortcut form, which is absolutely idiomatic in Zig:

  ```
  const value = try func();
  ```

  In case of void payload we have

  ```
  func() catch |err| return err;
  ```

  and its shortcut equivalent

  ```
  try func();
  ```

  respectively.[1]

### 4.7.5 Error set merging

We mentioned that error sets only look similar to enums, but do have an essential difference. Let's now look at that difference.

As we should remember, the namespaces of explicitly declared enums are strictly separated. That is, given two different enum types `A` and `B`, two identically named fields of these enums (e.g. `A.some_field` and `B.some_field`) would be totally unrelated.[2] On the contrary, the namespaces of error sets overlap as soon as an identically-named error appears in both. E.g. imagine we also implemented a "fragmentation-aware" allocator, which, upon returning an error, distinguishes between being truly out of memory, or simply being not able to allocate due to high fragmentation in the preallocated memory:

[1]Obviously, in the latter snippets, `func()` is just used as an example of an expression.

[2]The only exception to that is that an anonymous enum literal `.field` can be coerced to both, but such enum literal actually doesn't represent a runtime value prior to coercion.

```
const FragmentationAwareAllocError = error{
    OutOfMemory,
    FragmentationTooHigh,
};
```

Now the `OutOfMemory` member of `FragmentationAwareAllocError` and the `OutOfMemory` member of the previously declared `NonblockingAllocError` error set would actually represent *one and the same error value*. That's right, *error values are global across the entire Zig program*. Identically named error values are representing one and the same value, even when coming from different error set declarations.

> ▷ As enums do have their backing integer type (either explicitly specified or inferred from the values contained in the enum), so do errors in Zig. This topic is further addressed in Subsection 4.7.6, for now we shall assume that the backing integer has `u16` type (which is what Zig uses by default for errors).
>
> Differently from enums, it is not possible to assign specific integer values to errors:
>
> ```
> const FragmentationAwareAllocError = error{
>     OutOfMemory,
>     FragmentationTooHigh = 10, // is not allowed and doesn't compile
> };
> ```
>
> This actually looks like a good idea, otherwise one could potentially specify different values for the same error name, or conversely reuse the same value for different errors (included into different sets). Those particularly could come from different independent 3rd-party components, thereby rendering the program uncompilable and unfixable without modifying the respective 3rd-party code.

Suppose we now want to implement an allocator function, which decides at runtime, which of the two allocators (the non-blocking and fragmentation-aware ones) should be used:

```
pub fn allocate(size: usize) AllocError![]u8 {
    return switch (active_allocator) {
        .non_blocking => nonBlockingAlloc(size),
        .fragmentation_aware => fragmentationAwareAlloc(size),
    };
}
```

What is the actual error set `AllocError` used in the return value type of the above function?

Apparently, the above function can return one of the three following error values: `OutOfMemory`, `AllocatorInUse`, `FragmentationTooHigh`, where the `OutOfMemory` value is shared between the returned error sets of the two underlying allocators. How do we define `AllocError`?

Of course, we could define it as

```
const AllocError = error{
    OutOfMemory,
    AllocatorInUse,
    FragmentationTooHigh,
};
```

(remember, identically-named error values represent identical errors), but that doesn't look too nice.

Zig provides an operation to *merge* the error sets:

```
const AllocError = NonblockingAllocError ||
    FragmentationAwareAllocError;
```

▷ Yes, that's right, the '`||`' operation in Zig is not the "logical or" (the latter is denoted simply as '`or`'). Notice that the "logical not" in Zig is denoted as '`!`', same as in C/C++. It's just that the '`!`' operator in Zig is also used to construct error unions.

Actually we do not even need to introduce a name for the merged type, we could simply do

```
pub fn allocate(size: usize) (NonblockingAllocError ||
    FragmentationAwareAllocError)![]u8 {
    return switch (active_allocator) {
        .non_blocking => nonBlockingAlloc(size),
        .fragmentation_aware => fragmentationAwareAlloc(size),
    };
}
```

## 4.7.6 Global error set

So, we have already mentioned that error sets in Zig can be merged and that identically named errors denote one and the same error regardless of the set they are used in. Therefore one can imagine an error set which is a merge of all error sets used in a given Zig program. Furthermore, this set would also contain the values of the form `error.SomeErrorName` used in the program but not explicitly included in any of the sets. Essentially we are talking of a set of all error values used in a Zig program. This error set is referred to as the *global error set* of the Zig program.

The global error set type can be explicitly referred to as the predefined built-in `anyerror` type. You should rather rarely need to use this type, since it provides no knowledge about which error subset is expected at a given position:

```
// Not a too good idea to use anyerror as
// the error type, as it's too unspecific.
pub fn myFunction() anyerror!void {
    .......
    if (.......)
        return error.SomeError;
}
```

▷ We mentioned that, similarly to enums, errors do have their backing integer type. Actually all error sets share the same backing integer type, which is the backing integer type of the global error set. Now, in order to automatically infer the size of the backing integer for the global error set, Zig compiler would need to count the number of different error name identifiers used across the entire program. At the time of this writing the Zig compiler doesn't do that, but assumes by default that `u16` would suffice and uses that type as a backing integer for errors. It is possible to specify a different type as the compiler command line option.

### 4.7.7 Inferred error sets

Instead of specifying the error set part of the error union explicitly, we could ask the Zig compiler to infer the error set for the returned error union, which can be done by omitting the error set in front of the '`!`' sign:

```
pub fn allocate(size: usize) ![]u8 {
    return switch (active_allocator) {
        .non_blocking => nonBlockingAlloc(size),
        .fragmentation_aware => fragmentationAwareAlloc(size),
    };
}
```

The compiler would then collect all possible error values returned by the function and construct an (unnamed) error set of all these values. Effectively we would obtain the following function header:

```
pub fn allocate(size: usize) error{
    AllocatorInUse,
    OutOfMemory,
    FragmentationTooHigh,
}![]u8 {
    return switch (active_allocator) {
        .non_blocking => nonBlockingAlloc(size),
        .fragmentation_aware => fragmentationAwareAlloc(size),
    };
}
```

▷ The auto-inferred error set feature is available only for return types of functions and has some limitations. E.g. at the time of this writing it has difficulties inferring along the recursion loops.

▷ A somewhat common pattern in Zig, especially in prototyping, is to use the auto-inferred error set feature with the `main` function. E.g.

```
pub fn main() !void {
    var dbga: std.heap.DebugAllocator(.{}) = .{};
    defer std.debug.assert(dbga.deinit() == .ok);
    const alloc = dbga.allocator();

    const some_struct = try alloc.create(SomeStruct);
    .......
}
```

(notice the usage of the `try` operator). Upon `main` returning an error, the program will print the error information to the console and exit with a nonzero exit code.

### 4.7.8 errdefer

Imagine we are using a GUI framework, accessible via the `gui` namespace in our Zig code. Using this framework we wish to create a window, create a device context for this window, and ultimately make the window visible. Something along the lines of:

```
const Window = struct {
    window: gui.WindowHandle,
    context: gui.DeviceContext,
};

fn createWindow(initial_rect: gui.Rect) Window {
    const window = gui.createWindow(initial_rect);
    const context = gui.createDeviceContext(window);
    window.setVisible(true);
    return .{
        .window = window,
        .context = context,
    };
}
```

So far so good, but now imagine that both window creation and device context creation may fail, and, strictly speaking, so can even the `setVisible` function. This means that all of those should actually return error unions and we have to handle the respective errors. So we could try something along the lines of:

```
fn createWindow(initial_rect: gui.Rect) !Window {
    const window = try gui.createWindow(initial_rect);
    const context = try gui.createDeviceContext(window);
    try window.setVisible(true);
    return .{
        .window = window,
        .context = context,
    };
}
```

But there is a problem. Suppose `createDeviceContext` fails. Then we actually have to destroy the created window before returning from the function. Furthermore, if `setVisible` fails, we have to destroy both the created window and the device context. We might briefly have an idea to try using `defer` only to realize that it doesn't really work:

```
fn createWindow(initial_rect: gui.Rect) !Window {
    const window = try gui.createWindow(initial_rect);
    defer window.destroy(); // doesn't really work

    const context = try gui.createDeviceContext(window);
    defer context.destroy(); // doesn't really work

    try window.setVisible(true);
    return .{
        .window = window,
        .context = context,
    };
}
```

Indeed, now the destruction is going to happen unconditionally, while we only need it to happen if we the function fails. For such cases Zig offers the `errdefer` statement:

```
fn createWindow(initial_rect: gui.Rect) !Window {
    const window = try gui.createWindow(initial_rect);
    errdefer window.destroy();

    const context = try gui.createDeviceContext(window);
    errdefer context.destroy();

    try window.setVisible(true);
    return .{
        .window = window,
        .context = context,
    };
}
```

Now the deferred statements will be executed only if our function returns an error.

▷ The closest C++ counterpart to the above, barring some code duplication or somewhat tricky RAII constructs, is probably the following code:

```
struct Window {
    gui::WindowHandle window;
    gui::DeviceContext context;
};

// For simplicity let's not deal with detailed error codes
// and just use an std::optional.
std::optional<Window> createWindow(gui::Rect initial_rect) {
    std::optional<gui::WindowHandle> window =
        gui::createWindow(initial_rect);
    if (!window)
        return {};

    std::optional<gui::DeviceContext> context =
        gui::createDeviceContext(*window);
    if (!context) {
    DestroyWindow:
        window->destroy();
        return {};
    }

    if (!window->setVisible(true)) {
        context->destroy();
        goto DestroyWindow;
    }
    return Window{ .window = *window, .context = *context };
}
```

### 4.7.9 Error payload techniques

When returning an error it is often useful to also have extended error information, whose type might also vary depending on the error. E.g. if we're returning errors from a parser, we could have the following errors with the respective extended information data:

- `error.SyntaxError`. This one might want to additionally report the specific syntax (or lexical) issue, the file, the line and the column where the error was detected.
- `error.FileReadError`. This one might want to report the same data, except the syntax/lexical part.
- `error.OutOfMemory`. This one probably doesn't have any extended information data.

However error values in Zig are essentially enums (internally represented by integers), they cannot carry any extra payload. So what could we do about it? We could think of the following two main approaches:

- Abandon the usage of Zig's error unions and use custom data types instead. This has a strong drawback of not being able to use Zig's builtin error-handling features, such as `catch`, `try` and `errdefer`.
- Return the "error payload" together with the error union.

Now which options do we have to organize the latter approach?

- If we're having only one type of error payload, e.g. we only report syntax errors from a parser, we could create a `SyntaxErrorDetails` struct containing the syntax error details and pass it as an additional parameter to the parsing call:

```
// Notice that this name doesn't collide
// with error.SyntaxError as errors use
// their own separate namespace.
const SyntaxError = enum { ........ };

const SyntaxErrorDetails = struct {
    err: SyntaxError,
    file: []const u8, // mind the string lifetime!
    line: u32,

    fn print(self: *const @This()) void {
        ........
    }
};

fn callParser(parser: *Parser) void {
    var err_details: SyntaxErrorDetails = undefined;
    parser.parse(&err_details) catch |err| switch (err) {
        // Use a switch to make sure we do not miss
        // any errors
        error.SyntaxError => err_details.print(),
    };
}
```

- Initializing the `err_details` to `undefined` might feel a little bit unsafe against a possibility that we forget to return the details in some cases in the parser code. We could add extra safety by using an optional initialized to `null`:

  ```
  fn callParser(parser: *Parser) void {
      var err_details: ?SyntaxErrorDetails = null;
      parser.parse(&err_details) catch |err| switch (err) {
          error.SyntaxError => {
              if (err_details) |dtl|
                  dtl.print()
              else
                  std.debug.print(
                      "Unknown syntax error\n",
                      .{},
                  );
          },
      };
  }
  ```

- If there is more than one error with a payload, we could create an error payload union instead. Whether this union should be directly passed via a pointer or wrapped into an optional is subject to the considerations discussed above. The following example demonstrates a union directly passed via a pointer:

  ```
  fn callParser(parser: *Parser) void {
      var err_details: ParseErrorDetails = undefined;

      parser.parse(&err_details) catch |err| switch (err) {
          error.SyntaxError => err_details.syntax.print(),
          error.FileReadError => err_details.file.print(),
          error.OutOfMemory => std.debug.print(
              "Out of memory\n",
              .{},
          ),
      };
  }
  ```

- We could also return error payload via a tuple, where we'd need a helper variable for the tuple value, or we could use tuple destructuring:

  ```
  fn callParser(parser: *Parser) void {
      const parse_result, const err_details =
          parser.parse();

      // parse_result's type is ParseError!void,
      // so we don't need to assign the following
      // expression's result to anything
      parse_result catch |err| switch (err) {
          error.SyntaxError => err_details.syntax.print(),
  ```

```
            error.FileReadError => err_details.file.print(),
            error.OutOfMemory => std.debug.print(
                "Out of memory\n",
                .{},
            ),
        };
    }
```

This solution might require more verbosity in the implementation of the `parse()` function. Since the latter function now returns a tuple rather than an error union, we won't be able to use `try` statements inside that function to propagate internally received errors to the caller. But error propagation might need some extra care anyway, and this is what we shall discuss in Subsection 4.7.10

### 4.7.10 Error propagation considerations

The `try` operator in Zig, especially if combined with inferred error sets, offers a convenient way to propagate the returned error further up the call stack, without being too much concerned of what kind of errors might be arriving at the respective points. One thereby may end up with a situation similar to C++ exceptions, where the exceptions can easily become transparently propagated throughout the call stack, until they are finally intercepted by a matching `catch` statement (or they are not, and the program silently terminates). Something like

```
fn funcB() !void {
    ........
    try funcA();
    ........
}

fn funcC() !void {
    ........
    try funcB();
    ........
}

fn main() !void {
    ........
    try funcC();
    ........
}
```

The situation of "uncaught exceptions" is pretty unlikely in Zig, as there is no implicit exception propagation. However there could be more subtle issues. Recall that the error identifiers in Zig share a global namespace across the entire program, where identical identifiers do not collide, but are simply considered as one and the same identifier. This might result in "error identifier overloading", where one and the same error identifier is used in different parts of the program with somewhat or totally different semantics. Now imagine that this error is being propagated from one part of the program (using one kind of semantics for a given identifier) to a different part of the program (using different semantics). This could easily lead to an error being misinterpreted. E.g. consider the following code spread across several different files, one of them coming from a 3rd-party library:

```
// ---------------------------------------------------
// Decryptor.zig - located in a 3rd party library

......

pub const DecryptionError = error{
    BadChecksum,
};

// ---------------------------------------------------
// DocumentManager.zig - part of our own sources

..........

pub const LoadError = error{
    FileReadError,
} ||
    Decryptor.DecryptionError ||
    std.mem.Allocator.Error;

pub fn loadDocument(
    self: *@This(),
    file_path: []const u8,
) LoadError!void {
    const data = self.loadRawDocument(file_path) catch
        return error.FileReadError;
    errdefer self.destroyDocumentData(data);

    if (self.decryptor.isEncrypted(data))
        try self.decryptor.decryptData(data);

    try self.addLoadedDocument(file_path, data);
}

// ---------------------------------------------------
// FileMenu.zig - part of our own sources

..........

pub fn onFileOpen(self: *@This()) void {
    const file_path = self.runFileDialog() catch return;

    self.document_manager.loadDocument(
        file_path,
    ) catch |err| switch (err) {
        error.FileReadError => self.showError(
            "Error reading file {s}",
            .{file_path},
        ),
        error.BadChecksum => self.showError(
            "Error decrypting file {s}",
            .{file_path},
        ),
        error.OutOfMemory => self.showError(
            "Out of memory",
            .{},
        ),
    };
}
```

So far it looks good, but now imagine that the next version of the program is using a newer version of the decryption library. Suppose that this newer version of the decryption library supports a new encryption format, which needs a dictionary loaded from a preinstalled location. This dictionary is then loaded on-demand during the encryption/decryption whenever the new encryption mode is in use. Therefore the author of the decryption library modified the `DecryptionError` as follows:

```
// --------------------------------------------------
// Decryptor.zig

......

pub const DecryptionError = error{
    BadChecksum,
    FileReadError, // while loading dictionary
};
```

Now, arguably `FileReadError` might be not the best name for this second error, the library author could have e.g. used `DictionaryReadError` instead, but imperfect naming is not uncommon in software development. Anyway, this example is given purely for the sake of demonstration.

What is going to happen now is that a dictionary loading error will be forwarded by the `try decryptData()` located in the `loadDocument` code to the latter function's return value and received in `onFileOpen`, thus triggering the `FileReadError` branch of the switch. But the switch is assuming that the `FileReadError` value indicates an error of reading the file at `file_path` and will display the respective message, which will be completely misleading, as the respective file is actually in place, and the actual file which could not be read is the dictionary file (probably due to a corrupt installation).

The situation might be further aggravated if we e.g. are using the error payload techniques discussed in Subsection 4.7.9. E.g. imagine that the error value `FileReadError` produced inside the `loadDocument` function is augmented with an error payload:

```
pub const LoadErrorDetails = union {
    file_read_error: RawLoadingError,
    other: void,
};

pub fn loadDocument(
    self: *@This(),
    file_path: []const u8,
    err_details: *LoadErrorDetails,
) LoadError!void {
    const data = self.loadRawDocument(file_path) catch |err| {
        err_details.* = err;
        return error.FileReadError;
    };

    ..........
}
```

The caller (such as `onFileOpen`) will be accessing the `err_details` field whenever it receives `error.FileReadError`. If the latter was produced directly by the `loadDocument` function, everything is fine. However if it was forwarded from the decryptor, we are going to be accessing a wrong union field (whereas we actually should be accessing the '`other`' field of the union, if any), causing runtime panic or possibly a crash.

Apparently, one way to avoid such situations could be to use the `try` operator with caution, if not avoiding it altogether in many cases.

# Chapter 5

# Generics and metaprogramming

*Generic functions, structs, etc. Compile-time techniques.*

## 5.1 Generic functions

We have mentioned a few times that types are essentially one of the possible kinds of values in Zig. We shall further dive into this topic in Section 5.7, for now let's consider one further usage case of Zig types.

Suppose we are having the following function template in C++:

```
template<class T>
T max(T a, T b) {
    return a > b ? a : b;
}
```

There are different ways how we can express the same in Zig, the most straightforward one being:

```
fn max(T: type, a: T, b: T) T {
    return if(a > b) a else b;
}
```

The first parameter of the function is ‘`T`’ which is a parameter of type ‘`type`’. The values of the ‘`type`’ type are Zig types. So the ‘`T`’ parameter can take `u8`, `i32`, `void`, `SomeStruct` etc. as its value. Respectively the way to call this function would be e.g.:

```
    const a: i32 = 1;
    const b: i16 = 2;
    const m = max(i32, a, b);
```

corresponding to the C++ code:

```
    const int32_t a = 1;
    const int16_t b = 2;
    auto m = max<int32_t>(a, b);
```

Notes:

- The function `max` thereby becomes a generic function. A different runtime function is generated by the compiler for each different passed value of the ‘`T`’ parameter.

- ‘`T`’ must be the first parameter of the function in order to allow the following parameters of the function (and the function’s return type) to refer to ‘`T`’. It is not possible to refer to a type which is declared further down in the function’s parameters order:

  ```
  fn max(
      a: T, // error: T unknown
      b: T,
      T: type,
  ) T {
      return if(a > b) a else b;
  }
  ```

- You can also refer to ‘`T`’ in the function body:

  ```
  fn max(T: type, a: T, b: T) T {
      const result: T = if(a > b) a else b;
      return result;
  }
  ```

- You can build type expressions containing ‘`T`’ in the same way how you build type expressions involving builtin types, e.g. `*const T`, `[]T` etc.:

```
fn makePtrConst(T: type, ptr: *T) *const T {
    // 'ptr' will be converted to '*const T'
    // upon returning from the function
    return ptr;
}

pub fn main() void {
    var i: i32 = 0;
    const p = &i; // has '*i32' type
    const p1 = makePtrConst(i32, p);

    // Prints '*const i32' as the type of 'p1'
    @compileLog(p1);
}
```

- You can have more than one type parameter, e.g.:

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

fn negate(T: type, x: T, Result: type) Result {
    return -@as(Result, x);
}

pub fn main() void {
    const a: i16 = 1;

    // The type of n is i32
    const n = negate(i16, a, i32);

    std.debug.print(
        "n=@as({}, {})\n",
        .{ @TypeOf(n), n },
    );
}
```

- C++'s non-type function template parameters cannot be specified in Zig in exactly the same way as type parameters. Indeed, consider the following C++ code:

```
template<class T, size_t N>
void zeroInit(std::array<T,N> &a) {
    for (auto &item: a)
        item = 0;
}
```

If we translated it into Zig in a naive way:

```
// The following doesn't compile
fn zeroInit(T: type, n: usize, a: *[n]T) void {
    for (a) |*item|
        item.* = 0;
}
```

we would find that it doesn't compile (once our code actually uses this function, so that the function code is really complied, not merely parsed), because '`n`' is considered as a usual function parameter passed at runtime, not as a "template parameter".

The solution is to prefix '`n`' with a `comptime` keyword:

```
// This does compile
fn zeroInit(T: type, comptime n: usize, a: *[n]T) void {
    for (a) |*item|
        item.* = 0;
}
```

This usage of `comptime` keyword is further discussed in Section 5.8.

▷ Differently from C++'s function templates, Zig cannot infer "template parameters" of generic functions at the callsite. We wouldn't be able to call the `negate` function defined earlier as

```
const a: i16 = 1;
const n = negate(a, i32); // error: missing 1st argument
```

Likewise, we wouldn't be able to do it with the earlier defined `zeroInit` function:

```
var a: [10]i32 = undefined;
zeroInit(&a); // error: missing 1st and 2nd arguments
```

It is still possible to define the `negate` and `zeroInit` functions so that they can be called like in the examples immediately above, automatically inferring the "template parameters". This can be done using the `anytype` feature, discussed in Section 5.4.

▷ In C++ sometimes one "manually instantiates" a function template without calling it, for the purpose of taking a pointer to the instantiated version of the template. E.g. using the earlier defined `zeroInit` C++ function template one could do:

```
// The following line will automatically instantiate
// zeroInit<int,10> and obtain a pointer to it
void (*zeroInit10Ints)(std::array<int,10> &) = zeroInit;
```

In Zig it is not possible to "manually instantiate" a generic function without calling it. As a workaround one could define the respective function as a method of a generic struct. Generic structs are about to be discussed in Section 5.2.

## 5.2 Generic structs

In Section 5.1 we have been passing types as arguments of generic functions. Likewise, we can return types from generic functions:

```
fn PtrTo(T: type) type {
    return *T;
}

pub fn main() void {
    const p: PtrTo(i32) = undefined;
    // In @compileLog's output you can see
    // that the type of 'p' is '*i32'
    @compileLog(p);
}
```

▷ By the standard convention, function names are `camelCase` in Zig, except functions returning types, which are `PascalCase`, like type constants.

We shall see a more generic example of type-returning functions in Section 5.7. For now we shall use this possibility to make generic structs:

```
pub fn Vector2D(T: type) type {
    return struct {
        x: T,
        y: T,

        pub fn add(
            self: @This(),
            other: @This(),
        ) @This() {
            return .{
                .x = self.x + other.x,
                .y = self.y + other.y,
            };
        }

        pub fn scaleBy(
            self: @This(),
            factor: T,
        ) @This() {
            return .{
                .x = self.x * factor,
                .y = self.y * factor,
            };
        }
    };
}

pub fn main() void {
    // Alternatively:
    //    const v = Vector2D(i32){ .x = 0, .y = 0 };
    const v: Vector2D(i32) = .{ .x = 0, .y = 0 };

    @compileLog(v);
}
```

Notes:

- The function `Vector2D()` in the above code is used to generate 2D vector types. The argument '`T`' defines the type of vector's coordinates, so in the above example we are creating a vector with coordinates of `i32` type.

- The C++ counterpart of the `Vector2D()` function would be the struct template:

```
template<class T> struct Vector2D {
    T x;
    T y;

    Vector2D add(Vector2D other) const {
        return {
            .x = x + other.x,
            .y = y + other.y,
        };
    }

    Vector2D scaleBy(T factor) const {
        return {
            .x = x * factor,
            .y = y * factor,
        };
    }
};
```

- The return value supplied to the `return` statement of the `Vector2D()` function is an (initially) anonymous struct[1]. This struct will be returned as the function's result and becomes a named struct upon return. The full name of the returned struct type may vary across different compiler versions, its actual form could be found out using the `@typeName()` builtin function, but that name is mostly not needed, we should simply refer to the returned type as `Vector2D(i32)` (or whatever argument we wanted to use in place of `i32`).

  Invoking the `Vector2D()` function multiple times with one and the same argument doesn't generate new struct types, it is considered as one and the same type (the Zig compiler caches the invocation result or does something like that):

```
pub fn main() void {
    const v = Vector2D(i32){ .x = 1, .y = 2 };
    const v1: Vector2D(i32) = .{ .x = 3, .y = 4 };

    // The following line successfully compiles,
    // v and v1 have one and the same type.
    const v2 = v.add(v1);

    std.debug.print("{any}\n", .{v2});
}
```

- "Non-type template parameters" for Zig structs can be specified in the same way:

[1]Anonymous structs were discussed in Section 3.14

```
pub fn Array(T: type, n: usize) type {
    return struct {
        data: [n]T,
    };
}

pub fn main() void {
    const a: Array(i32,3) = .{
        .data = .{1,2,3},
    };

    @compileLog(a);
}
```

Differently from generic functions, with generic Zig structs you don't need to explicitly qualify the integer parameter '`n`' as `comptime`, this happens implicitly.[2]

- Similarly to how you cannot have an entire file representing a union or an enum, you cannot have an entire file representing a generic struct. Files in Zig correspond only to ordinary structs, not to anything else, in particular not to functions, which are used in Zig to define generic structs. Respectively, Zig generics will be always put into some (possibly local) namespace. Thus, if you want to dedicate an entire Zig file to a generic struct, you need to use this file as a namespace for that generic struct:

```
// ---------------------------------
// generic.zig

pub fn Generic(T: type) type {
    return struct {
        field: T,
        .......
    };
}

// ---------------------------------
// main.zig

const Generic = @import("generic.zig").Generic;

pub fn main() void {
    var v: Generic(i32) = undefined;
    .......
}
```

[2]The reason you don't need explicit `comptime` here is that functions returning `type` are automatically comptime and so are all their parameters. Further discussion of this is found in Subection 5.8.1.

## 5.3 Generic unions and enums

You can create generic unions in the same way how you can create generic structs:

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

pub fn ValueOrPtr(T: type) type {
    return union(enum) {
        value: T,
        ptr: *const T,

        pub fn getValue(self: @This()) T {
            return switch (self) {
                .value => |v| v,
                .ptr => |p| p.*,
            };
        }
    };
}

pub fn main() void {
    const i: i32 = 1;
    const u: ValueOrPtr(i32) = .{ .ptr = &i };

    std.debug.print("{}\n", .{u.getValue()});
}
```

You can also do the same with enums:

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

pub fn EnumFrom(T: type, start: T) type {
    return enum(T) {
        first = start,
        second,
    };
}

pub fn main() void {
    const EnumStartingAt10 = EnumFrom(i32, 10);
    const e: EnumStartingAt10 = .second;

    // Prints 11
    std.debug.print("{}\n", .{@intFromEnum(e)});
}
```

## 5.4 anytype

Recall the `negate` function introduced closer to the end of Section 5.1:

```
fn negate(T: type, x: T, Result: type) Result {
    return -@as(Result, x);
}
```

At the end of that section we said that it's not possible to have the type of its 'x' argument automatically inferred like it is possible in C++:

```
    const a: i16 = 1;
    const n = negate(a, i32); // error: missing 1st argument
```

More precisely, this is not possible using the means discussed in Section 5.1. However, if we really wanted the `negate` function to be usable like that, we could have defined it using the `anytype` feature:

```
fn negate(x: anytype, Result: type) Result {
    return -@as(Result, x);
}
```

A corresponding C++ implementation would be something like

```
template<class Result>
Result negate(auto x) {
    return -static_cast<Result>(x);
}
```

(barring the integer casting rule differences between C++ and Zig). So `anytype` arguments accept any kind of value.

▷ Similarly, we can rewrite the `zeroInit` function (also introduced close to the end of Section 5.1) as:

```
fn zeroInit(a: anytype) void {
    for (a) |*item|
        item.* = 0;
}
```

approximately corresponding to the C++ implementation:

```
void zeroInitialize(auto a) {
    for (auto &item: *a)
        item = 0;
}
```

Notes:

- By using an `anytype` parameter a function becomes a generic one, same like in C++ by using an `auto` parameter a function becomes a function template.

- You cannot declare consts and vars of type `anytype`, the latter is not really a type, but a keyword indicating a generic function parameter:

  ```
      const a: i32 = 1;
      const b: anytype = a; // does not compile
  ```

- Likewise, you cannot declare the function's return parameter to be `anytype`:

```
// This function header does not compile
fn func() anytype {
    return @as(i32, 1);
}
```

Thus there is no direct counterpart in Zig for C++'s `auto` return type of a function. The return type must be specified explicitly in Zig and the way to achive that with generic functions at times can get really involved. We shall discuss this further throughout the text.

- Being not a type, `anytype` cannot be used in type expressions. Thus, e.g. you cannot force a generic function to accept only pointers by writing its header as

```
// This function header does not compile
fn usePointer(ptr: *anytype) void {
    ..........
}
```

You have to resort to other means to achieve things like that, those will be discussed further in the text.

- Declaring the '`self`' argument as `anytype` will make the '`self`' argument assume either `*@This()` or `*const @This()` type (but never `@This()`), depending of whether the actual passed object is const or non-const:

```
const Struct = struct {
    i: i32 = 0,
    fn access(self: anytype) void {
        _ = self;
    }
};

pub fn main() void {
    const c = Struct{};
    var v = Struct{};
    c.access(); // 'self' is '*const Struct'
    v.access(); // 'self' is '*Struct'
}
```

This feature is somewhat similar to C++23's "deducing this" feature in that it allows to provide shared implementations for const and non-const versions of the same method. To be really able to use this feature we need to combine it with a few other language features, so a more realistic usage example will be provided in Section 5.7.

## 5.5 Duck typing

From the previous discussion we could conclude that in Zig we are essentially having two ways of specifying generic function parameters:

- You can pass a type explicitly as a function argument and restrict your function's parameter types with the help of this explicitly passed type, e.g. the following function accepts only pointers as its second argument:

  ```
  fn usePointer(T: type, ptr: *T) *const T {
      ..........
  }
  ```

- You can declare the argument as `anytype`, which means the argument will accept values of any type:

  ```
  // This function accepts any argument, not only pointers
  fn usePointer(ptr: anytype) void {
      ..........
  }
  ```

As the first option is pretty similar to C++'s templates (except the missing auto-inference of "template parameters"), it shouldn't raise too many questions, and we shall now concentrate on the second option.

A common guideline for working with `anytype` parameters in Zig is the *duck typing* approach. This approach is roughly described by the rule: "if it looks like a duck, walks like a duck and quacks like a duck, then it must be a duck".[3] With respect to the above `usePointer` function with the `anytype` parameter this would mean that "if it can be used as a pointer, it is probably a pointer". So, roughly speaking, we just don't bother with trying to explicitly check, whether the passed parameter is really a pointer, and simply go on with using it. E.g.

```
fn usePointer(ptr: anytype) void {
    std.debug.print("{}\n", .{ptr.*});
}
```

Notice the expression `ptr.*` in the above code. Apparently, if `ptr` was anything but a pointer, this expression would have raised a compile error. Thus, we don't really need to check if `ptr` is really a pointer.

▷ The duck typing approach might kind of fail with pointers, e.g. if a pointer is solely used to access structure fields, such as in:

```
fn usePtrToStruct(ptr: anytype) void {
    const value = ptr.common_field;
    .....
}
```

since the above code will also compile with structs passed by value (provided they contain the expected `common_field` field). In cases where accepting passed by value arguments is undesired, one simple way to enforce the pionter type could be:

```
fn usePtrToStruct(ptr: anytype) void {
    const value = ptr.*.common_field;
    .....
}
```

Another way could be

[3]There are different wording variations of this rule, expressing one and the same idea.

```
fn usePtrToStruct(ptr: anytype) void {
    const really_a_pointer = &ptr.*;
    // Use 'really_a_pointer' instead of 'ptr'
    // in the following code
    .....
}
```

or

```
fn usePtrToStruct(ptr: anytype) void {
    // The next line would fail to compile
    // if 'ptr' is not a pointer
    _ = &ptr.*;
    .....
}
```

etc.

## 5.6 @TypeOf() builtin function

Suppose, for the sake of exercise, we wanted to write a generic function `derefPtr` which takes a pointer argument `ptr` and returns the dereferenced value `ptr.*`. It would be easy to write such function using an explicit type argument:

```
fn derefPtr(T: type, ptr: *T) T {
    return ptr.*;
}
```

But how can we do the same using an `anytype` argument? As we already know, Zig doesn't allow to use `anytype` in the function's return value position:

```
// Can't do that:
fn derefPtr(ptr: anytype) anytype {
    return ptr.*;
}
```

The `@TypeOf()` builtin function to the rescue:

```
fn derefPtr(ptr: anytype) @TypeOf(ptr.*) {
    return ptr.*;
}
```

Notes:

- The `@TypeOf()` function can also be used elsewhere in the function's header (in positions following the `ptr` argument) or in the function's body:

  ```
  fn storeToPtr(ptr: anytype, value: @TypeOf(ptr.*)) void {
      // For the sake of demonstration let's
      // introduce a temp var
      var temp: @TypeOf(ptr.*) = undefined;
      temp = ptr.*;
      ptr.* = temp;
  }
  ```

- In principle, `@TypeOf()` can accept any expression as its argument, e.g.

  ```
  fn usePtr(ptr: anytype) void {
      var temp: @TypeOf(myFunc(ptr).some_field) = undefined;
      .......
  }
  ```

  In that sense it is similar to C++'s `decltype`. There are however a number of important differences:

  – Differently from C++, Zig doesn't have a concept of *unevaluated context*. This means that the expressions inside `@TypeOf()` will, generally speaking, produce code, although one could expect that code to be removed by the optimizer, if there are no side effects. Nevertheless, in particular the expressions, which can be evaluated at compile time, will be evaluated at compile time, possibly causing compile errors, even though otherwise the expression looks "innocent". E.g. the following code

  ```
  const formal_ptr: *i32 = undefined;
  var value: @TypeOf(formal_ptr.*) = undefined;
  ```

  produces an error:

  ```
  main.zig:9:34: error: cannot dereference undefined value
      var value: @TypeOf(formal_ptr.*) = undefined;
                         ~~~~~~~~~~~~
  ```

  Even though the above code dereferences the pointer only formally, Zig tries to dereference the pointer "for real" during compile time, since the expression can be evaluated at compile time. If you need to do things like that, you may have to resort to the reflection features of the Zig language, which are discussed in Section 5.10.

  – Differently from C++, Zig doesn't consider `const` and `volatile` qualifiers to be a part of a variable's type:

  ```
  // The type of 'a' is 'i32', not 'const i32'
  const a: i32 = 0;
  // Prints 'i32', not 'const i32'
  @compileLog(@TypeOf(a));
  ```

  You also cannot declare struct fields to be `const` or `volatile`, these qualifiers are essentially reserved for pointers, e.g. `*const i32`.

  Also Zig doesn't have C++'s references, respectively `@TypeOf()` cannot make distinction between "rvalues" and "lvalues".

  – The `@TypeOf()` function also accepts multiple arguments. In this case the returned type is the result of the *peer type resolution*[4] among the argument types:

  ```
  const a: i32 = 0;
  const b: u16 = 0;
  @compileLog(@TypeOf(a, b)); // prints 'i32'
  ```

[4] Peer type resolution is discussed in Section 2.5.

▷ Functions return types, which require really elaborate derivations, are usually implemented using helper functions. These functions are constructed similarly to the ones used to define generic types and will be covered in Section 5.7.

## 5.7 Type manipulation

We have mentioned that types are actually values in Zig and can be manipulated similarly to other values, with the restriction that type manipulation must be done at compile time. We have seen in Section 5.1 how types are being passed as values to function arguments of type '`type`' and we have further seen some type expression examples like `*const T`, where '`T`' would be a function parameter.

This kind of type manipulation in Zig is not restricted to function arguments. Actually types can be manipulated pretty much as any other kind of value, provided it occurs at compile time (one cannot manipulate type values in Zig at runtime). With the introduction of the `@TypeOf()` builtin function we obtain additional freedom in type manipulation, so let's take a look at the following example:

```
const Struct = struct {
    i: i32 = 0,

    fn access(
        self: anytype,
    ) switch (@TypeOf(self)) {
        *@TypeOf(self.*) => *i32,
        *const @TypeOf(self.*) => *const i32,
        else => @compileError(
            "Unsupported argument type " ++
                @typeName(@TypeOf(self)),
        ),
    } {
        return &self.i;
    }
};

pub fn main() void {
    const c = Struct{};
    var v = Struct{};
    // Type of 'ic' is '*const i32'
    const ic = c.access();
    // Type of 'iv' is '*i32'
    const iv = v.access();
    @compileLog(ic, iv);
}
```

The example is somewhat contrived, but demonstrates how one could write a shared method for const and non-const objects.

- The `access` method returns a pointer to the '`i`' field of the object, but the returned pointer type varies along with the type of the '`self`' pointer.

Recall that an `anytype` ‘`self`’ parameter is a const or a non-const pointer to `@This()`. Respectively the switch in the function’s return type position will determine the return type based on the type of ‘`self`’.

- The switch expression in the `access` function’s return type might look a bit too cumbersome and unreadable. It is common in Zig to introduce auxiliary functions to compute the function’s return type:

```
const Struct = struct {
    i: i32 = 0,

    fn access(
        self: anytype,
    ) AccessResult(@TypeOf(self)) {
        return &self.i;
    }

    fn AccessResult(SelfPtr: type) type {
        return switch (SelfPtr) {
            *@This() => *i32,
            *const @This() => *const i32,
            else => @compileError(
                "Unsupported argument type " ++
                    @typeName(SelfPtr),
            ),
        };
    }
};
```

  The `AccessResult` function is somewhat similar to the functions we have been using to implement generic structs in Section 5.2 in that this function accepts type as its argument and returns another type as its result. It’s name is `PascalCase`, since this function returns a type.

▷ In implementing the `AccessResult` function we have been somewhat fortunate to know the expected result of dereferencing a value of `SelfPtr` type. Indeed, since `SelfPtr` is obtained by calling `@TypeOf(self)` in the `access()` function’s header, we expect `SelfPtr` to be either equal to `*@This()` or to `*const @This()`.

However suppose we are in a situation where `SelfPtr` (or a similar argument of a similar function) can be pointing to a type, which is not known in advance. How could we provide the expressions for the switch prongs in `AccessResult`?

We might come to the idea of doing something like:

```
fn AccessResult(SelfPtr: type) type {
    const dummy_ptr: SelfPtr = undefined;
    const Self = @TypeOf(dummy_ptr.*); // fails
    return switch (SelfPtr) {
        *Self => *i32,
        *const Self => *const i32,
        else => @compileError(
            "Unsupported argument type " ++
                @typeName(SelfPtr),
        ),
        };
    }
```

Unfortunately it fails in the line `const Self = @TypeOf(dummy_ptr.*);`. The problem is that, as mentioned before, unlike C++, Zig doesn't have a concept of unevaluated context, and therefore tries to evaluate the expression `dummy_ptr.*` (at compile time), failing upon trying to access an uninitialized pointer. To work around this problem we need to resort to more advanced type-manipulation techniques discussed in Section 5.10.

Another relatively common example of compile-time type manipulation is provided by the following code:

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

const Server = struct {
    num_clients: NumClients = if (track_clients) 0,

    const NumClients =
        if (track_clients) usize else void;
    const track_clients = std.debug.runtime_safety;

    pub fn addClient(self: *@This()) void {
        if (track_clients)
            self.num_clients += 1;
    }

    pub fn removeClient(self: *@This()) void {
        if (track_clients)
            self.num_clients -= 1;
    }

    pub fn init() @This() {
        return .{};
    }

    pub fn deinit(self: *@This()) void {
        if (track_clients)
            std.debug.assert(self.num_clients == 0);
    }
};

pub fn main() void {
    var s = Server.init();
    defer s.deinit();
    s.addClient();
    s.removeClient();
}
```

Notes:

- The `num_clients` counter is a "debug-mode counter variable" (more precisely, "safe-mode counter variable"), which is effectively available only in safe builds, otherwise its type is void, as per definition of `NumClients` type.

- As we should recall from the earlier discussion of the `if` operator and the omitted "else" branch, the default initialization expression of the `num_clients` field is equivalent to:

  ```
  num_clients: NumClients = if (track_clients) 0 else {},
  ```

  Thereby the initialization expression's type matches the `NumClients` type, which is `usize` in safe builds and `void` in unsafe builds.

- All '`if (track_clients)`' conditions are evaluated at compile time. Thus `addClient` and `removeClient` functions unconditionally increment/decrement the counter in safe builds and do nothing in unsafe builds. The assertion in the `deinit` function is evaluated only in safe builds.

## 5.8 Comptime

We have been occasionally discussing the situations where Zig would decide to evaluate certain expressions at compile time. Some of these decisions would not be a mere optimization but have syntactical implications, allowing or disallowing certain usages. E.g. if we wanted to use a value to define the number of elements in an array, this value doesn't have to be a literal, expressions are allowed, but their values must be compile-time known in both Zig and C/C++.

Both C++ and Zig are somewhat conservative in trying to evaluate things at compile time. E.g. a function would normally be evaluated at runtime, unless it is being explicitly told to evaluate at compile time. C++ offers the `constexpr`, `consteval` and `constinit` keywords to enforce that certain functions or variable initializations are (or may be upon demand) evaluated at compile time. In comparison, Zig offers a way to unconditionally enforce compile-time evaluation on the expression, function parameter and variable (rather than variable initialization) levels, in each case using the `comptime` keyword.

▷ Due to the existence of the `comptime` keyword it is common in Zig to refer to compile time as *comptime*.

▷ Note that comptime code is usually/often interleaved with runtime code. A function evaluated at runtime may contain a few lines or subexpressions evaluated at comptime:

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

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

    // Comptime-evaluated code
    const safety = if (std.debug.runtime_safety)
        "on"
    else
        "off";
```

```
    // Runtime call to print
    std.debug.print(
        "Runtime safety: {s}\n",
        .{safety},
    );
}
```

On the other hand, if an entire function is called at comptime, then its entire body will be comptime-evaluated.

### 5.8.1 Comptime expressions

Zig *may* decide to evaluate a given expression (or a part thereof) at compile time. It is not exactly specified, when this should happen, but the language tries to be “reasonable”. Here are a few (possibly not 100% precise) rules, applicable at the time of this writing:

- An expression containing only comptime-known values would be usually evaluated at comptime:

  ```
      // All following initialization expressions
      // are evaluated at comptime, so the respective
      // consts are comptime-known
      const a: i32 = 1;
      const b = a + 1;
      const c = if (a > 0) b else 0;
      const d = switch (c) {
          0, 1 => false,
          else => true,
      };
  ```

  Some exceptions are mentioned below.

- A function call will not be evaluated at comptime, unless explicitly told so, or unless the function cannot be called at runtime (because it returns a type):

  ```
  fn addOne(x: i32) i32 {
      return x + 1;
  }

  fn MakePtr(T: type) type {
      return *T;
  }

  pub fn main() void {
      // Initialized using a function call,
      // hence 'two' is runtime-known
      const two = addOne(1);

      // Initialized using a function call,
      // but the function returns a type,
  ```

```
    // hence 'Ptr' is comptime-known
    const Ptr = MakePtr(i32);

    @compileLog(two, Ptr);
}
```

- Comptime-known pointers to comptime-known data will be dereferenced at comptime:

```
pub fn main() void {
    const comptime_value: i32 = 1;
    const ptr = &comptime_value;
    const dereferenced_ptr = ptr.*;

    var runtime_value: i32 = 2;
    const ptr2 = &runtime_value;
    const dereferenced_ptr2 = ptr2.*;

    @compileLog(
        ptr, // comptime
        dereferenced_ptr, // comptime
        ptr2, // runtime
        dereferenced_ptr2, // runtime
    );
}
```

- An `if` or `switch` operator whose body cannot be evaluated at comptime (or is decided against being evaluated at comptime), would still consider evaluating its "condition expression" at comptime[5]:

```
fn func() i32 {
    return 2;
}

pub fn main() void {
    const comptime_value: i32 = 1;

    // 'a' is runtime-known, since func()
    // will be evaluated at runtime,
    // but the if's branch decision is done
    // at comptime, the 'else' branch is even
    // not considered.
    const a: i32 = if (comptime_value == 1)
        func()
    else {};

    @compileLog(a);
}
```

[5] Such `if` operator would behave similarly to `if constexpr` in C++, except that Zig does less syntactical checks in the inactive branch.

> Notice that in the above code the “else” branch is essentially ignored (otherwise we’d have had a compile error upon trying to coerce the void value returned by the “else” branch to the `i32` type of the variable ‘`a`’).

In cases where we would like to enforce the comptime evaluation of an expression (those primarily would be the cases involving function calls), or where we just want to make sure that an expression will be evaluated at comptime, we could prefix the expression with the `comptime` keyword, thereby forcing the comptime evaluation:

```
fn addOne(x: i32) i32 {
    return x + 1;
}

pub fn main() void {
    // Force 'two' to be comptime-known,
    // despite using a function call
    const two = comptime addOne(1);

    @compileLog(two);
}
```

Similarly, we can force e.g. an `if` to be decided at comptime as follows:

```
fn func() i32 {
    return 2;
}

fn addOne(x: i32) i32 {
    return x + 1;
}

pub fn main() void {
    const a: i32 = if (comptime addOne(0) == 1)
        func()
    else {};

    @compileLog(a);
}
```

▷ Apparently, in cases of forced comptime evaluation, the respective expression must be capable of comptime evaluation to begin with:

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

fn printMessage() void {
    std.debug.print("Some text\n", .{});
}

pub fn main() void {
    // Fails, since printMessage() cannot be evaluated at
    // comptime due to the call to print function, which
    // can only be done at runtime, since it calls into the
    // OS API.
    comptime printMessage();
}
```

The above example is also probably the reason, why Zig doesn't attempt to evaluate function calls at comptime by default: even if the called function is comptime-compatible at some moment, it might stop being so at another moment. If that function is located far away from the callsite (possibly in a 3rd-party library), this could create a rather undesired "far-distance dependency" on the comptime-compatibility of the said function.

### 5.8.2 Comptime parameters

We have already encountered the usage of `comptime` with function parameters in order to turn integer parameters into "non-type template parameters" of generic functions. This is effectively the function of the `comptime` prefix with function parameters: it turns a runtime parameter into a "template parameter" of a generic function (the function thereby becomes a generic one). Such parameter then can be used in comptime-only positions:

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

fn makeArray(
    comptime n: usize,
    initial_value: i32, // runtime parameter
) [n]i32 {
    return [1]i32{initial_value} ** n;
}

pub fn main() void {
    const a = makeArray(3, 10);
    std.debug.print("{any}\n", .{a});
}
```

▷ For functions called at comptime there is no real distinction between generic and ordinary functions, since the entire function is evaluated at comptime anyway.
Notice, however, that in the above example the `makeArray` function is called at runtime, since the call itself occurs in runtime code and is not prefixed with `comptime`. The function is thereby being treated as a generic one and is instantiated for `n==3`. Thus the above Zig code is roughly equivalent to the following C++ code:

```
#include<array>

template<size_t N>
std::array<int32_t,N> makeArray(int32_t initial_value) {
    std::array<int32_t,N> a;
    a.fill(initial_value);
    return a;
}

int main() {
    auto a = makeArray<3>(10);
    // Print array contents here:
```

```
    .......
    return 0;
}
```

▷ The reason we didn't have to prefix the type parameters of generic functions with `comptime` is because type values can be passed only at comptime anyway, so Zig treats them as comptime by default. For the same reason you don't need to prefix any parameters of type-returning functions with `comptime`, those functions can be evaluated only at comptime anyway.

### 5.8.3 Comptime variables

Imagine you are having a function which mixes runtime and comptime code, and imagine you want to use a comptime-only loop inside this function. A useful loop would normally need to modify some memory, but any writable variables would normally qualify as runtime ones:

```
pub fn main() void {
    var n: usize = 0;

    // comptime doesn't work due to
    // access to runtime variable 'n'
    comptime for (0..4) |i| {
        n += i;
    };

    var a: [n]i32 = [1]i32{0} ** n;
    ......
}
```

For this purpose Zig introduces `comptime` variables:

```
pub fn main() void {
    comptime var n: usize = 0;

    // fine now, since n is comptime
    comptime for (0..4) |i| {
        n += i;
    };

    var a: [n]i32 = [1]i32{0} ** n;
    ......
}
```

Comptime variables can only be accessed from comptime code. Any access to a comptime variable forces the respective place in the code to be comptime-evaluated. If it is not possible to evaluate the respective place in the code at comptime, the compilation fails:

```
fn func() usize {
    comptime var n: usize = 0;
    n += 1; // fine, access is forced to comptime

    for (0..4) |i| {
        // as the loop is not prefixed with comptime,
        // the loop body is purely runtime code,
        // hence the next line **fails**
        n += i;
    }

    // fine, 'a' is a comptime-known constant
    const a = n;

    // fine, 'n + a' will be evaluated at comptime
    // and then returned at runtime as if it was
    // a literal value
    return n + a;
}

pub fn main() void {
    _ = func(); // force call at runtime
}
```

### 5.8.4 Comptime blocks

Comptime blocks (but not other kinds of comptime expressions) can occur directly at the decls level (that is at the file/struct/union/enum scope):

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

const str = makeString();
const N: i32 = 10;

fn makeString() [:0]const u8 {
    return "abcdefg";
}

comptime {
    // Explicitly reference the 'str' entity,
    // forcing it to be compiled, thereby
    // invoking makeString() at comptime
    _ = str;

    // Check a static precondition
    std.debug.assert(N > 0);
}

pub fn main() void {}
```

Notes:

- The comptime block will be evaluated during compile time.
- The `_ = str;` line references the (otherwise unreferenced) `str` entity, thereby causing the `makeString()` to be invoked at comptime (recall from Section 2.12 that decl-level consts are initialized at comptime). It is easy to check that otherwise `makeString()` is not going to be invoked. E.g. we could try returning a zero value (instead of a string) from `makeString()`, while keeping the return type of `makeString()` unchanged. This will cause the compilation to fail, unless we comment the `_ = str;` line out, in which case `makeString()` is no longer being compiled and doesn't cause a compilation error.
- The assertion statement inside the comptime block works similarly to C++'s `static_assert`.

### 5.8.5 Comptime memory

The standard Zig allocators (defined in `std.heap`) are intended for runtime usage. What if we need some "dynamic" memory at comptime?

At the time of the writing there is no true comptime allocator in Zig. However there is a fallback: *you may return pointers to local constants from comptime functions*. The comptime in Zig utilizes some kind of garbage collection scheme, so the values do not disappear by merely going out of scope. Rather they disappear when they stop being reachable or something like that. So, while at runtime returning a pointer to a local constant would imply a dangling pointer, at comptime this is just a way to "allocate memory".

If the memory thus "allocated" is referenced from runtime code, it becomes available at runtime in the form of preinitialized constant memory. This preinitialized memory contains all values which are still reachable "by the end of comptime". E.g. consider the following program:

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

const Node = union(enum) {
    inner: struct {
        left: *const Node,
        right: *const Node,
    },
    leaf: i32,
};

fn makeBinaryTree(depth: usize, start: i32) *const Node {
    if (!@inComptime())
        @compileError("This function is comptime-only");

    if (depth == 0)
        return &.{ .leaf = start };

    const branch_depth = depth - 1;
    const branch_size: i32 = 1 << branch_depth;
    return &.{ .inner = .{
        .left = makeBinaryTree(branch_depth, start),
        .right = makeBinaryTree(
            branch_depth,
            start + branch_size,
        ),
    } };
}

fn printBinaryTree(root: *const Node) void {
    switch (root.*) {
        .leaf => |value| std.debug.print(
            "{}\n",
            .{value},
        ),
        .inner => |node| {
            printBinaryTree(node.left);
            printBinaryTree(node.right);
        },
    }
}

// Decl-level constants are always initialized
// at comptime, no comptime keyword needed
const tree = makeBinaryTree(3, 0);

pub fn main() void {
    printBinaryTree(tree);
}
```

The `makeBinaryTree` function will be invoked at comptime and recursively build a binary tree structure, which is stored in the file-scope constant '`tree`'. This tree is subsequently printed at runtime by calling `printBinaryTree`.

> ▷ In earlier versions of Zig it was also possible to return const pointers to local variables from comptime functions. The respective variables would be auto-converted to consts upon leaving the scope. Reportedly, the mechanism had a number of issues, and, as a solution, it became forbidden to return pointers to variables.
>
> As a workaround for that limitation, one can copy the variable to a local constant and return a pointer to this constant, as demonstrated by the following (contrived) example:

```
fn returnVar() *const i32 {
    var sum: i32 = 0;
    for (0..4) |i|
        sum += i;

    const result = sum;
    return &result;
}

const ret = returnVar();

pub fn main() void {
    @compileLog(ret.*);
}
```

Unfortunately, this makes it impossible to build cyclic graph structures with pointers at comptime, which are later available at runtime.

### 5.8.6 Concatenation at comptime

The array concatenation operations '++' and '**' become particularly useful in manipulating slices at comptime, since the lengths of the slices become automatically and trivially comptime-known, and since the memory associated with local vars and consts in comptime code is managed in a garbage-collected style.

Effectively it becomes possible to concatenate slices pretty much naively as if they were strings, especially slices of u8, which do represent strings:

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

// Comptime-only function
// constructing the string "0123456789".
fn allDecimalDigits() [10:0]u8 {
    var result: [:0]const u8 = "";

    for ('0'..('9' + 1)) |code| {
        const char: u8 = @intCast(code);
        // We need to convert the char into an array
        // so that we can apply concatenation.
        result = result ++ (&char)[0..1];

        // The concatednated string, to which 'result'
        // is pointing by now, is about to go out of
        // scope, but it doesn't matter, since we are
        // in comptime.
    }

    // It would probably have been simplest and most
    // convenient to have the function return a slice,
    // but just for the sake of the exercise let's
    // demonstrate that we also can return an array.
    return result[0..].*;
}

pub fn main() void {
    // The next line wouldn't compile without comptime,
    // More specifically, allDecimalDigits() cannot be
    // used in runtime code due to its comptime-only
    // slice-manipulation code.
    const s = comptime allDecimalDigits();

    // Use the comptime string 's' in a runtime call.
    std.debug.print("{s}\n", .{s});
}
```

▷ The above techniques are not limited to slices of `u8` or null-terminated slices and can be applied to slices/arrays of pretty much any type.

## 5.9 Inlining

Inlining in Zig differs from the one in C/C++ and it's related to the comptime mechanisms of Zig, therefore we have postponed its discussion until after the introduction of comptime.

### 5.9.1 Function inlining

Function inlining works and looks pretty similar to C/C++, but still with some differences:

- Unlike in C++, a function with an `inline` specifier is guaranteed to produce inlined code, pretty much like in C.
- Inlining not only requests inlined code generation, it also does (to an extent) semantical inlining, thus looking a little bit more like a macro expansion. Specifically, the actual parameter values which are known to be comptime at the callsite will result in the formal function parameters being treated as comptime-known as well:

  ```
  pub inline fn func(c: bool) i32 {
      return if (c) 0 else {};
  }

  var runtime_condition = true;

  pub fn main() void {
      // okay, the 'then' branch is selected
      // at comptime
      @compileLog(func(true));

      // fails, as we return void value from the 'else'
      // branch, which doesn't match the function's
      // return type
      @compileLog(func(runtime_condition));
  }
  ```

▷ The "inline" calling convention can be used to manipulate the inlining based on comptime-known conditions:

```
const inlined_mode: bool = ......;

pub fn func(
    c: bool,
) callconv(if (inlined_mode)
    .@"inline"
```

```
else
    .auto) i32 {
    return if (c) 1 else 0;
}
```

or, using an auxiliary variable for the calling convention:

```
const inlined_mode: bool = ......;
const conv: std.builtin.CallingConvention =
    if (inlined_mode) .@"inline" else .auto;

pub fn func(
    c: bool,
) callconv(conv) i32 {
    return if (c) 1 else 0;
}
```

This mechanism does the same kind of inlining as specified by the '`inline`' keyword.

▷ Use '`noinline`' to prevent function's code inlining by the compiler:

```
noinline fn pleaseDontInlineMe() void {
    std.debug.print("I am not inlined\n", .{});
}
```

Note, that the compiler will not automatically endow a function, which is not specified as inline (or which doesn't specify the inline calling convention), with inlining semantics, such as propagating the comptime properties of the parameters. Only the generated code can be inlined automatically in optimized builds, but no automatic semantics change is done. The '`noinline`' keyword merely disables that inlining optimization.

### 5.9.2 Loop inlining

The `inline` keyword in front of a loop causes loop unrolling. The unrolling itself is (obviously) done at comptime, but the contents of the loop body is not forced to comptime by the `inline` keyword, which is the main difference between `inline` and `comptime` usage in front of a loop.

This feature is not so much there as a means to enforce loop unrolling for the optimization means (although it can also be used in that quality), but rather due to its semantical effects, not unlike Zig's function inlining. In inline loops each loop iteration goes separately through code generation. This, in particular, allows usage of iteration-dependent types:

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

const Data = struct {
    c: u8 = 1,
    i: i32 = 2,
    u: usize = 3,

    // Zero the fields with the given names
    fn zeroFields(
        self: *@This(),
        comptime fields: []const []const u8,
    ) void {
        inline for (fields) |field|
            @field(self, field) = 0;
    }
};

pub fn main() void {
    var d: Data = .{};
    d.zeroFields(&.{ "c", "i" });
    std.debug.print("{any}\n", .{d});
}
```

Notes:

- `@field(a,"name")` is the same as `a.name`, where `name` is a field or a decl name. That is, `@field()` covers the applications of the dot operator for both field and decl accesses. Respectively '`a`' needs to be a struct/union/enum instance or a pointer to struct/union/enum instance or a struct/union/enum type:

  ```
  pub fn main() void {
      const S = struct {
          i: i32 = 0,
          const n = 10;
          fn reset(self: *@This()) void {
              self.i = 0;
          }
      };
      var v = S{};
      @field(v, "i") = 1; // same as: v.i = 1;
      @field(&v, "i") = 1; // same as: (&v).i = 1;
      _ = @field(S, "n"); // same as: _ = S.n;
      @field(S, "reset")(&v); // same as: S.reset(&v)
  }
  ```

  The difference between `@field(a,"name")` and `a.name` is that the former allows to construct the field name at comptime (in the form of comptime-known string).

  Also `@field` doesn't support the method call syntax `a.method()`:

  ```
      // trying instead of v.reset() but this doesn't work,
      // have to do @field(S, "reset")(&v) instead
      @field(v, "reset")();
  ```

- The type of the accessed field[6] thereby varies across the loop iterations and it is not possible to generate a single runtime loop body for all iterations. Thus, the loop wouldn't compile without the `inline` keyword (or without forcing the entire code to execute at comptime).

[6]The type of the accessed field can be explicitly obtained (in the discussed example) as `@FieldType(@This(), field)`.

- The iteration range (‘`fields`’ in this case) must be comptime-known to allow comptime loop unrolling. This also makes the iteration capture ‘`field`’ comptime-known within each respective unrolled iteration. This allows it to be passed to the `@field` builtin function, which requires its second argument to be comptime-known.
- The function `zeroFields` has `fields` as a comptime parameter, but `self` is not a comptime parameter, therefore the function is still invoked at runtime (different runtime functions being generated for different values of the `fields` parameter).

▷ Inline while loops can be used in a similar fashion. Notice that, since the iteration conditions of inline loops must be comptime-known, we would probably need to use a comptime var in such case. In the following, somewhat artificial example, we have rewritten the previously defined `zeroFields()` function using an inline while loop, which terminates upon encountering an empty field name:

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

const Data = struct {
    c: u8 = 1,
    i: i32 = 2,
    u: usize = 3,

    // The list of fields must be terminated
    // with an empty string.
    fn zeroFields(
        self: *@This(),
        comptime fields: [*]const []const u8,
    ) void {
        comptime var field = fields;
        inline while (field[0].len > 0) : (field += 1)
            @field(self, field[0]) = 0;
    }
};

pub fn main() void {
    var d: Data = .{};
    d.zeroFields(&.{ "c", "i", "" });
    std.debug.print("{any}\n", .{d});
}
```

### 5.9.3 Switch prong inlining

One can also inline multi-valued switch prongs, which causes the prong to be “unrolled”, that is a separate single-valued prong will be generated for each value included into the original multi-valued prong. Thereby it becomes similar to loop unrolling, where one can in particular have values of different types in different unrolled prongs:

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

const Field = enum { a, b, c, d, e, f };
const Struct = struct {
    a: void,
    b: []const u8,
    c: f32,
    d: f64,
    e: i32,
    f: u64,

    fn resetField(self: *@This(), field: Field) void {
        switch (field) {
            .a => {},
            .b => self.b = "",
            inline .c, .d => |f| @field(self, @tagName(f)) = -1e10,
            inline else => |f| @field(self, @tagName(f)) = 0,
        }
    }
};

pub fn main() void {
    var s: Struct = undefined;
    s.resetField(.d);
    std.debug.print("{}\n", .{s.d});
}
```

The compiler will unroll the inline prongs in the switch in the `resetField` function, effectively producing the following code:

```
    switch (field) {
        .a => {},
        .b => self.b = "",
        // unrolled from 'inline .c, d.'
        .c => @field(self, @tagName(Field.c)) = -1e10,
        .d => @field(self, @tagName(Field.d)) = -1e10,
        // unrolled from 'inline else'
        .e => @field(self, @tagName(Field.e)) = 0,
        .f => @field(self, @tagName(Field.f)) = 0,
    }
```

▷ The '`inline else`' switch prong is a particularly powerful feature, allowing, among other things, to effectively implement "C-style polymorphism", which we discuss in Section 6.13.1.

## 5.10 Reflection and reification

Reflection and reification are key elements of Zig's metaprogramming features. Reflection allows taking Zig types apart and reification allows putting them back together. Let's start with reflection.

Given a Zig's type '`T`' we can obtain (at comptime) information about the details of this type by calling `@typeInfo(T)`. E.g.

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

fn dumpStructInfo(T: type) void {
    // The type of 'info' is std.builtin.Type
    // which is a tagged union
    const info = @typeInfo(T);

    // We expect 'T' to be a struct type,
    // so we directly access the 'struct'
    // field of the 'info' union
    const struct_info = info.@"struct";

    // 'fields' is a slice, so we can iterate over it
    // using a for loop
    const fields = struct_info.fields;

    // We need to use inline for, since we must iterate
    // at comptime, but the print function call must be
    // done at runtime
    inline for (fields) |*field|
        std.debug.print("{s}\n", .{field.name});
}

pub fn main() void {
    const S = struct { x: i32, y: i32 };
    dumpStructInfo(S);
}
```

Notes:

- The `dumpStructInfo` function is called at runtime, but it is a generic function, which is instantiated (in the `main` function) for the struct ‘`S`’.
- Despite being called at runtime, the entire body of `dumpStructInfo` is evaluated at comptime, except the `print` calls. The generated code for the `dumpStructInfo(S)` function is thereby effectively containing a sequence of `print` calls (from the unrolled loop):

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

We can use the `std.builtin.Type` to construct new types from scratch or by modifying the info of existing types. The following code demonstrates the latter option:

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

// field_name must be of [:0]const u8 type,
// since std.builtin.Type.StructField.name
// requires a zero-terminated string
fn AddField(
    Struct: type,
```

```
    field_name: [:0]const u8,
    FieldType: type,
    default_value: ?FieldType,
) type {
    var struct_info = @typeInfo(Struct).@"struct";

    const added_field = std.builtin.Type.StructField{
        .name = field_name,
        .type = FieldType,
        .default_value_ptr = if (default_value) |*v|
            v
        else
            null,
        .is_comptime = false,
        .alignment = 0, // '0' means 'use default alignment'
    };

    // Add the field to the existing ones
    struct_info.fields = struct_info.fields ++ .{added_field};

    // Create and return new type
    return @Type(.{ .@"struct" = struct_info });
}

pub fn main() void {
    const S = struct { x: i32 = 0, y: i32 = 0 };
    const S1 = AddField(S, "z", f64, 1.2);
    const s1: S1 = .{};
    @compileLog(s1);
}
```

Notes:

- The function `AddField` takes a struct type and adds one more field to that type.
- The `@Type()` function is a kind of “inverse” of the `@typeInfo()` function, the former takes a type info struct and creates a new type based on that info. The process of converting the type info into a type is referred to as *reification*.

▷ While type info returned by `@typeInfo()` also contains information about *pub* decls contained in a struct, union, or enum, the `@Type()` function doesn't support decls creation (it will fail if the supplied info contains any decls).

If you want to construct a custom type which contains decls then (short of auto-generating Zig code in a separate compilation step) you'll have to wrap a reified custom type into a generic struct, the latter containing the decls that you wish it to have. E.g. we could augment the previous code as:

```
fn WithGetter(T: type, extra_field_name: []const u8) type {
    const ExtraFieldType = @FieldType(T, extra_field_name);

    return struct {
        data: T = .{},

        pub fn getExtraField(
            self: *const @This(),
        ) ExtraFieldType {
            return @field(self.data, extra_field_name);
        }
    };
}

pub fn main() void {
    const S = struct {
        x: i32 = 0,
        y: i32 = 0,
    };
    const S1 = AddField(S, "z", f64, 1.2);
    const WrappedS1 = WithGetter(S1, "z");
    const s1: WrappedS1 = .{};
    @compileLog(comptime s1.getExtraField());
}
```

Apparently the exact set of added decls and their names have to be determined in advance, one can only manipulate the details of the types of the added decls, like we're manipulating the return type of `getExtraField` function in the above example. Conditional addition of decls used to be possible by clever application of the `usingnamespace` feature, but as the latter is about to be deprecated, it doesn't seem possible anymore to conditionally construct a set of decls in a Zig type.

As one further example of using reflection and reification in Zig, consider the following (contrived) code, which retrieves a pointer to a struct's field, retaining the const and volatile properties of the original pointer:

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

fn retrieveFieldPtr(
    struct_ptr: anytype,
    comptime field_name: []const u8,
) RetrievedFieldPtr(@TypeOf(struct_ptr), field_name) {
    return &@field(struct_ptr, field_name);
}

fn RetrievedFieldPtr(
    StructPtr: type,
    comptime field_name: []const u8,
) type {
    var ptr_info = @typeInfo(StructPtr).pointer;

    // make sure it's a single-item pointer
    std.debug.assert(ptr_info.size == .one);

    // retrieve the field type
    const Struct = ptr_info.child;
    const FieldType = @FieldType(Struct, field_name);
    // overwrite the pointed-to type
    ptr_info.child = FieldType;

    // set to default alignment (indicated by 0)
    // as a quick-and-dirty solution
    ptr_info.alignment = 0;

    // construct ptr type with ovewritten type
    return @Type(.{ .pointer = ptr_info });
}

pub fn main() void {
    const S = struct {
        x: i32 = 0,
    };
    const c = S{};
    var v = S{};
    const ptr_c = retrieveFieldPtr(&c, "x");
    const ptr_v = retrieveFieldPtr(&v, "x");
    @compileLog(ptr_c, ptr_v);
}
```

# Chapter 6

# Bits and pieces

*Further language features. Patterns and techniques. C/C++/Zig construct correspondence table.*

In this chapter we are going to cover topics which didn’t fit into the previous chapters. We’ll begin by covering some of the remaining features of the language. After that we’ll demonstrate some ideas and patterns which are commonly used or which could be used in Zig.[1] We conclude by a table loosely translating a number of common C/C++ constructs to Zig.

[1]We would like to point out that it’s not necessarily suggested to use the presented ideas in exactly the presented form, the code examples are rather there to illustrate the overall approach, the implementation details may vary.

## 6.1 Modules

Zig program consists of one or more modules. A Zig module is a source file tree with a root file. This root file can be given an alias name, which serves as a module name. This module name can then be given to an `@import()` call instead of a file path. Zig compiler (in the normal compilation mode) then will resolve that `@import()` call to importing the root file of the respective module instead.

Consider the following project structure:

```
├── src
│   ├── main.zig
│   └── ...
└── my_module
        ├── root.zig
        └── ...
```

Our main module is contained in the `src` folder and has `main.zig` as its root file (this is what we supply to the compiler). We also have the `my_module` folder containing an extra module for our program. We therefore define `my_module` as a module name alias for `my_module/root.zig` file. This can be done in `build.zig`:

```
// extra module
const my_module = b.createModule(.{
    .root_source_file = b.path("my_module/root.zig"),
    .target = target,
    .optimize = optimize,
});

// main module
const main = b.addExecutable(.{
    .name = "main",
    .root_source_file = b.path("src/main.zig"),
    .target = target,
    .optimize = optimize,
});
// add the extra module as "my_module"
main.root_module.addImport("my_module", my_module);
b.installArtifact(main);
```

or in the command line:

```
zig build-exe --dep my_module -Mroot=src/main.zig
    -Mmy_module=my_module/root.zig
```

Then the builtin function call `@import("my_module")` will work (at least semantically) pretty much identically to `@import("my_module/root.zig")` in normal builds. There will be however a difference in test builds: as we shall learn in Section 6.2, tests are compiled only for one module at a time.

▷ The `std` library is therefore effectively simply a module which is automatically imported (under the `"std"` name) by the Zig compiler.

▷ There is one more difference between importing a module and importing a file: you cannot import files outside of the folder containing the root file of a module. E.g.

```
// --------------------------------------------------
// src/main.zig - this is the root file of a module

// Not possible: imported.zig is outside
// the module's folder
const imported = @import("../imported.zig");
```

Notice that you generally *can* import files outside of the folder of the current file, the limitation is only that you may not go outside the module root file's folder:

```
// ------------------------------------------------------
// src/subfolder1/subfolder2/src1.zig - this file is deep
// in the module's tree hierarchy

// Okay, the imported file's path is "src/subfolder1/import.zig",
// this is still within the "src" folder,
// which is the module root file's folder.
const imported = @import("../imported.zig");
```

## 6.2 Tests

Zig has a testing framework integrated into the language. The tests can (and generally should) be written directly in the same source files where the normal program code resides:

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

pub fn Storage(T: type) type {
    return struct {
        value: ?T = null,

        pub fn store(self: *@This(), value: T) !void {
            if (self.value != null)
                return error.AlreadyFull;
            self.value = value;
        }

        pub fn retrieve(self: *@This()) !T {
            if (self.value) |value| {
                self.value = null;
                return value;
            }
            return error.IsEmpty;
        }
    };
}
```

```
test "Store/retrieve" {
    var s = Storage(i32){};
    try std.testing.expectEqual({}, s.store(2));
    try std.testing.expectEqual(2, s.retrieve());
    try std.testing.expectEqual(error.IsEmpty, s.retrieve());
}

test "Double store" {
    var s = Storage(i32){};
    try std.testing.expectEqual({}, s.store(2));
    try std.testing.expectEqual(error.AlreadyFull, s.store(1));
}
```

Notes:

- The above source should be compiled in test mode (otherwise it will not compile/run the tests and will also complain that there is no `main` function, which we didn't bother to provide).
- The `expectEqual` function returns an error value if the comparison fails. This error value is then returned from the test by the `try` operator, causing the entire test to fail. Otherwise (no error returned by `expectEqual`) the test proceeds further, succeeding upon reaching the end of the test body.
- It is possible to programmatically disable the test by returning the special error value `error.SkipZigTest`:

  ```
  test "Double store" {
      return error.SkipZigTest; // skip this test for now
      var s = Storage(i32){};
      try std.testing.expectEqual({}, s.store(2));
      try std.testing
          .expectEqual(error.AlreadyFull, s.store(1));
  }
  ```

- Compiling a Zig module in test mode compiles and runs tests only for this module (but not for modules referenced by the compiled module). This is different from normal Zig compilation, where the referenced modules (specifically, their referenced parts) are also included into the compilation.
- Not all tests contained in a Zig module are guaranteed to be included into the test compilation. Similarly to normal compilation, Zig follows the referenced entities, starting from the program's root and compiles only the tests for these referenced entities. Once in a while this mechanism seems to work not exactly as one would intuitively expect, skipping some tests which one would rather want/expect to be included. For such cases Zig `std` library offers a possibility to "add further references" to the dependency tree. This is normally done by placing the following comptime block into one or more of the source files, until all necessary dependencies are followed by the compiler:

  ```
  comptime {
      std.testing.refAllDecls(@This());
  }
  ```

The above block adds all pub decls of the current source file as the test dependencies, so that the tests for those dependencies should be compiled as well.[2]

- For tests involving memory allocation/deallocation, `std.testing` offers a special test allocator, available as `std.testing.allocator`. This allocator checks that all memory has been deallocated by the end of each test:

  ```
  test "Allocation leak" {
      const alloc = std.testing.allocator;
      const p = try alloc.create(i32);
      //defer alloc.destroy(p);

      // With alloc.destroy(p) commented out,
      // the test will fail due to a memory leak
      _ = p;
  }
  ```

## 6.3 Alignment

Until now we have been pretty much ignoring the alignment topic. One of the reasons is that in C/C++ the alignment is rather in the background of the language, most of the time we are dealing with default-aligned data. This is similar in Zig, we are still dealing with default-aligned data most of the time, nevertheless the alignment topic is present in Zig somewhat more prominently.

It starts with Zig pointers having their associated alignment. Zig pointer types specified in their usual form, e.g. `*const i32` assume the default alignment of the data they are pointing to. E.g. `*const i32` and `*i32` assume 4-byte aligned data. This can be overridden by specifying the alignment explicitly:

```
    const UnalignedI32 = *align(1) i32;
    const OveralignedI32 = *align(8) i32;
```

Pointers are not supposed to take values violating their declared alignment and the language has a number of embedded runtime checks in safe build modes to ensure that.

There are also compile-time checks, preventing assignment of a lower-aligned pointer to a higher-aligned pointer:

```
pub fn main() void {
    var i: i32 = 0;
    const p: *align(1) const i32 = &i;

    // The next line fails: cannot assign
    // to a higher-aligned pointer
    const p1: *const i32 = p;
    _ = p1;
}
```

[2] Besides `std.testing.refAllDecls` there is also `std.testing.refAllDeclsRecursive`. The latter is normally considered to be excessive and is not that much recommended to be used.

If we are sure that the source pointer is sufficiently aligned (like in our above example), we could tell so to the Zig compiler by using the `@alignCast` builtin:

```
pub fn main() void {
    var i: i32 = 0;
    const p: *align(1) const i32 = &i;

    // This compiles
    const p1: *const i32 = @alignCast(p);
    _ = p1;
}
```

▷ The "natural" or default alignment of a Zig type can be found out using the `@alignOf` builtin:

```
pub fn main() void {
    const n = @alignOf(u32);
    @compileLog(n); // prints 4
}
```

▷ The size of Zig types takes their "natural" alignment into account:

```
pub fn main() void {
    const size = @sizeOf(u24);
    const alignment = @alignOf(u24);
    @compileLog(size, alignment); // prints 4, 4
}
```

▷ It is possible to take pointers to fields of packed structs in Zig. These pointers have special alignment properties:

```
pub fn main() void {
    const Packed = packed struct {
        a: u2 = 2,
        b: u3 = 3,
        c: u15 = 1,
    };
    const s = Packed{};
    const pb = &s.b;

    // The next line prints '*align(4:2:3) const u3'
    // as the type of pb
    @compileLog(pb);
}
```

The alignment of such pointers has the form:

```
    align(host_alignment:bit_offset:host_size)
```

where `host_alignment` is the alignment of the "host" packed struct in bytes (4 in our case), the `bit_offset` is the bit offset of the pointed to field from the "host beginning" (2 in our case), and the `host_size` is the host size in bytes (3 in our case).

Therefore such pointers are not compatible to ordinary pointers and are mostly not compatible to each other.

Further details about alignment control in Zig can be taken from Zig documentation.

## 6.4 Pointer casting

Casting in Zig normally uses the result location type to determine the casting destiniation type, as discussed in Section 1.11. Pointer casting is no different: it needs result location type provided by the `@as()` builtin, by the type of the destination variable, etc.

The primary cast operation between pointer types is `@ptrCast()`, it casts from one pointer type to another:

```
pub fn main() void {
    var i: i32 = 0x12345678;
    const p = &i; // p is '*i32'
    const p8: *u8 = @ptrCast(p);

    // Prints 78 on low-endian platforms
    std.debug.print("{x}\n", .{p8.*});
}
```

▷ Reportedly, Zig doesn't use TBAA (type-based aliasing analysis), therefore one can use pointer casting more freely to reinterpret memory. There is also the `@bitCast()` builtin, which is a counterpart of C++20's `std::bit_cast<>`.

Similarly to how in C++ one has to use `const_cast<>` to cast away constness, in Zig one has to use `@constCast()`:

```
    var i: i32 = 0x12345678;
    const p: *const i32 = &i;
    const pc: *i32 = @constCast(p);
    const p8: *u8 = @constCast(@ptrCast(p));
```

▷ In C++ `const_cast<>` can also cast away volatility. In Zig one needs to separately use `@volatileCast()`.

As Zig pointers also contain the alignment attribute, one may need to cast the alignment. Somewhat surprisingly, this is often required even if the pointers do not override the default alignment:

```
    var a = [4]u8{ 0, 1, 2, 3 };
    const p = &a[0]; // p is '*u8'

    // The following line won't compile without @alignCast(),
    // since default alignment of '*u8' is 1, while default
    // alignment of '*u32' is 4.
    const p32: *u32 = @alignCast(@ptrCast(p));
```

The same situation occurs with casting from `anyopaque` pointers (which is the Zig's counterpart of C/C++'s type-erased void pointers), as `anyopaque` pointers also have default alignment of 1:

```
    var a = [4]u8{ 0, 1, 2, 3 };
    const p: *anyopaque = &a;

    // The next line needs an @alignCast()
    const p32: *u32 = @alignCast(@ptrCast(p));
```

▷ Converting *to* an `anyopaque` pointer doesn't need `@ptrCast()` (pointers to all types coerce to `anyopaque`), nor does it need `@alignCast()` (pointers to `anyopaque` have the lowest possible alignment of 1). `@constCast()` might be however needed if a const pointer is converted to a non-const `anyopaque` pointer.

## 6.5 Address manipulation

The builtin pointer arithmetic in Zig is somewhat restrictive. E.g. negative pointer differences are apparently not supposed to be computed. At the time of the writing, the type of the pointer difference result is `usize` and casting a negative difference to `isize` doesn't produce a correct value:

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

pub fn main() void {
    var a: [10]i32 = undefined;
    const p = &a[0];
    const p1 = &a[5];
    const n: isize = @intCast(p - p1);

    // Prints 3ffffffffffffffb on a 64-bit system,
    // this doesn't look like a correct answer
    std.debug.print("{x}\n", .{n});
}
```

We could however implement our own signed pointer difference by working directly with addresses stored inside the pointers:

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

fn ptrDiff(T: type, p1: *const T, p2: *const T) isize {
    // @intFromPtr returns usize, so we need to cast to isize
    const addr1: isize = @intCast(@intFromPtr(p1));
    const addr2: isize = @intCast(@intFromPtr(p2));

    // Builtin difference p1 - p2 also doesn't check
    // for exact division, let's do that here
    return @divExact(addr1 - addr2, @sizeOf(T));
}

pub fn main() void {
    var a: [10]i32 = undefined;
    const p = &a[0];
    const p1 = &a[5];
    const n = ptrDiff(i32, p, p1);

    // Prints -5, as expected
    std.debug.print("{x}\n", .{n});
}
```

Similarly, builtin addition of an integer to a pointer expects a `usize`. We could implement addition of a signed integer to a pointer:

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

fn addToPtr(T: type, p: [*]const T, offs: isize) [*]const T {
    const addr: isize = @intCast(@intFromPtr(p));
    const addr1 = addr + offs * @sizeOf(T);
    return @ptrFromInt(@as(usize, @intCast(addr1)));
}

pub fn main() void {
    var a: [10]i32 = undefined;
    for (&a, 0..) |*item, i|
        item.* = @intCast(i);

    const p: [*]i32 = a[5..];
    const offs: isize = -3;
    const p1 = addToPtr(i32, p, offs);

    // Prints 2, as expected
    std.debug.print("{x}\n", .{p1[0]});
}
```

## 6.6 SIMD features

We are not going to get into an in-depth discussion of Zig's built-in SIMD features, but merely give a short example:

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

pub fn main() void {
    const num_channels =
        std.simd.suggestVectorLength(f32) orelse {
            std.debug.print("Unknown SIMD platform\n", .{});
            return;
        };
    const SimdF32 = @Vector(num_channels, f32);

    // Implement unipolar sawtooth phase drivers
    // in SIMD channels
    var phases: SimdF32 = undefined;
    var increments: SimdF32 = undefined;
    for (0..num_channels) |i| {
        const t: f32 = @floatFromInt(i);
        phases[i] = 0.1 * t;
        increments[i] = 0.4 - 0.02 * t;
    }

    // Run 10 iterations
    for (0..10) |_| {
        const one: SimdF32 = @splat(1.0);
        phases += increments;
        const phases_wrapped = phases - one;
        const cmp = phases > one;
        if (@reduce(.Or, cmp))
            phases = @select(f32, cmp, phases_wrapped, phases);
        std.debug.print("{any}\n", .{phases});
    }
}
```

## 6.7 Interfacing to other languages

Zig contains a number of features for interfacing with code written in C. Respectively, interfacing with other languages generally goes to the extent to which the other language in question supports C ABI. For the sake of demonstration, here we are going to consider interfacing with C++.

Consider the following program consisting of a Zig source file (containing the main entry point) and a C++ source file:

```
// --------------------------
// main.zig

const std = @import("std");

const Data = extern struct {
    i: c_int,
    f: f64,
};

extern fn fillData(*Data) callconv(.c) void;

export fn provideDefaultInt() callconv(.c) c_int {
    return 10;
}

pub fn main() void {
    var data: Data = undefined;
    fillData(&data);
    std.debug.print("{any}\n", .{data});
}

// --------------------------
// cpp_part.cpp

struct Data {
    int i;
    double f;
};

extern "C" int provideDefaultInt();

extern "C" void fillData(Data *data) {
    data->i = provideDefaultInt();
    data->f = 0.5;
}
```

Notes:

- The `Data` struct is used both in Zig and in C++ parts of the code. Respectively it is declared twice. In the Zig code we have to use `extern struct` rathern than simply `struct` to ensure memory-layout compatibility with C.
- We have used the `c_int` integer type in the `Data` struct, because this type's size corresponds to the size of C/C++'s `int` type. Had we used `i32` instead, we might have problems on some platforms where C/C++'s `int` type is not 32-bit.
- The `fillData` function is declared as `extern` in Zig code, because it is imported from C++ code. Conversely, the `provideDefaultInt` function is declared using the `export` keyword in the Zig code, because it is made available to the C++ part.
- Both `fillData` and `provideDefaultInt` specify the C calling convention in Zig code to ensure the ABI compatibility with C. In C++ code these functions are declared as `extern "C"` for the same reason.

▷ The `c_int` type is reserved exactly for the purposes of C compatibility in Zig and is not really supposed to be used otherwise. A number of further types (e.g. `c_uint`, `c_long`, etc.) are provided in Zig for compatibility with C/C++'s integer types. `float` is `f32` and `double` is `f64`.

Enum size in C/C++ is implementation-defined (unless specified explicitly), one needs to read the documentation of the respective C/C++ compiler or specific ABI in order to declare a compatible enum in Zig (by explicitly specifying the enum's tag type in Zig source).

▷ It is possible to endow the extern function declarations with a `pub` keyword, making them available outside of the Zig file where they are declared:

```
// --------------------------
// imports.zig

pub const Data = extern struct {
    i: c_int,
    f: f64,
};
```

```
pub extern fn fillData(*Data) callconv(.c) void;

// --------------------------
// main.zig

const std = @import("std");
const imports = @import("imports.zig");

export fn provideDefaultInt() callconv(.c) c_int {
    return 10;
}

pub fn main() void {
    var data: imports.Data = undefined;
    imports.fillData(&data);
    std.debug.print("{any}\n", .{data});
}
```

▷ When importing symbols from a library, we might need to specify the library name after the `extern` keyword:

```
const std = @import("std");
const os = std.os.windows;

extern "user32" fn MessageBoxA(
    hwnd: ?os.HWND,
    text: os.LPCSTR,
    caption: os.LPCSTR,
    mb_type: os.UINT,
) callconv(.winapi) c_int;

pub fn main() void {
    _ = MessageBoxA(null, "text", "caption", 0);
}
```

## 6.8 Access to “compiler defines”

Besides the `"std"` module routinely available in Zig sources, there is also the `"builtin"` module, containing the “compiler defines”, providing information about the current build configuration, platform, etc.

The full list of symbols can be retrieved by `zig build-exe --show-builtin`, here we are going to demonstrate how to obtain the information about the current build mode:

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

pub fn main() void {
    const prt = std.debug.print;
    switch(builtin.mode) {
        .Debug => prt("Debug", .{}),
        .ReleaseSafe => prt("ReleaseSafe", .{}),
        .ReleaseFast => prt("ReleaseFast", .{}),
        .ReleaseSmall => prt("ReleaseSmall", .{}),
    }
}
```

## 6.9 Data embedding

Zig has a nice feature allowing to import binary data into the data segment of the Zig program, which is done by the `@embedFile()` builtin function:

```
const data = @embedFile("file.dat");
```

`@embedFile()` is similar to `@import()`, but does not parse the file's contents, but rather reads the file's contents as-is in the binary form.

The return type of `@embedFile()` is `[:0]const u8`. The zero sentinel is automatically added to the file contents for the cases where we want to treat the file contents as a zero-terminated string.

## 6.10 Iterators

C++'s standard library introduces an iterator pattern, where iterators normally iterate within the range between the values returned by `begin()` and `end()`. This pattern is also supported by the C++'s range-based for loop.

Zig also introduces an iterator pattern, but it looks quite different from the one of C++. The following example illustrates a Zig-style iterator implementation for a single-linked list:

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

pub fn List(T: type) type {
    return struct {
        first: ?*Node = null,

        const Node = struct {
            data: T,
            next: ?*Node = null,
        };

        const Iterator = struct {
            next_node: ?*Node,
            pub fn next(self: *@This()) ?*Node {
                if (self.next_node) |next_node| {
                    self.next_node = next_node.next;
                    return next_node;
                } else {
                    return null;
                }
            }
        };

        pub fn iterate(self: *const @This()) Iterator {
            return .{ .next_node = self.first };
        }
    };
}

pub fn main() void {
    // Construct the list manually, in order to keep
    // the code example concise
    const List32 = List(i32);
    var n1 = List32.Node{ .data = 1 };
    var n0 = List32.Node{ .data = 0, .next = &n1 };
    const list = List32{ .first = &n0 };

    // The following is the standard Zig iteration pattern:
    var it = list.iterate();
    while (it.next()) |node|
        std.debug.print("{}\n", .{node.data});
}
```

Notes:

- So, the Zig iteration pattern is defined by the `iterate()` method of the container, which returns an iterator, and the `next()` method of the iterator. Notice that the `next()` method actually performs two tasks: it returns the iterated value and advances the iterator. Thus, the first call to `next()` returns the first element of the list, the second call to `next()` returns the second element of the list, and the third call to `next()` returns `null`, indicating the end of the iteration.

- Thus, differently from C++ iterators, which contain a kind of pointer to the "current value", Zig iterators store (a kind of pointer to the) "next value". During each iteration of the `while` loop in the above example it is no longer possible to retrieve the "current value" from the iterator, as the iterator has been already advanced to the "next value".

- This "storing the next value" approach of Zig iterators has been found to result in somewhat suboptimal code generation (see Zig issue 20254). Also, in more complicated scenarios it might be highly convenient to be able to access the "current value" from an iterator. Both might be an argument to consider using C++-like iteration pattern at times in Zig, even though it doesn't seem to be officially encouraged:

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

  pub fn List(T: type) type {
      return struct {
          first: ?*Node = null,
  ```

```
            const Node = struct {
                data: T,
                next: ?*Node = null,
            };

            const CppIterator = struct {
                current_node: ?*Node,
                pub fn current(self: *const @This()) ?*Node {
                    return self.current_node;
                }
                pub fn next(self: *@This()) void {
                    // panics (in safe modes) upon attempt
                    // to iterate past the end
                    self.current_node = self.current_node.?.next;
                }
            };

            pub fn cppIterate(self: *const @This()) CppIterator {
                return .{ .current_node = self.first };
            }
        };
    }

    pub fn main() void {
        const List32 = List(i32);
        var n1 = List32.Node{ .data = 1 };
        var n0 = List32.Node{ .data = 0, .next = &n1 };
        const list = List32{ .first = &n0 };

        // C++-style iteration
        var it = list.cppIterate();
        while (it.current()) |node| : (it.next())
            std.debug.print("{}\n", .{node.data});
    }
```

## 6.11 Inheritance

Zig, being not really an object-oriented language, doesn't explicitly support the concept of inheritance. One can emulate inheritance to an extent by embedding the "base class" as a member into a "derived class":

```
const Shape = struct {
    // shape has x,y position
    x: f32,
    y: f32,
};

const Rect = struct {
    shape: Shape, // the "base" subobject
    width: f32,
    height: f32,

    // A counterpart of C++'s
    //    static_cast<Rect *>(shape)
    fn fromShape(shape: *Shape) *@This() {
        return @alignCast(@fieldParentPtr(
            "shape",
            shape,
        ));
    }
};
```

Notes:

- Placing the `Shape` as a subobject into the `Rect` loosely corresponds how C++ implements inheritance memory-wise.

- One can convert from a pointer to `Rect` to a pointer to `Shape` as:

  ```
  const ptr_to_shape = &ptr_to_rect.shape;
  ```

  The inverse conversion is done by

  ```
  const ptr_to_rect: *Rect =
      @alignCast(@fieldParentPtr("shape", ptr_to_shape));
  const const_ptr_to_rect: *const Rect =
      @alignCast(
          @fieldParentPtr("shape", const_ptr_to_shape),
      );
  ```

  or by calling the `Rect.fromShape` method which we added for the demonstration purposes:

  ```
  const ptr_to_rect = Rect.fromShape(ptr_to_shape);
  ```

- The `@fieldParentPtr()` builtin function is a kind of "inverse dot operation". It converts from a pointer to a struct's field to the pointer to the struct itself. The function's parameters are the field name (as a comptime-known string) and the pointer to the field. The third parameter, which is the type of the struct, is obtained implicitly from the expected type of the result. E.g. in our above examples the result type is a pointer to `Rect` which is either declared as the destination variable's type, or as the function's return type.

- The `@alignCast()` around `@fieldParentPtr()` is strictly speaking redundant in the current example, but generally it seems to be a good idea to always place `@alignCast()` around `@fieldParentPtr()`. The reason is that `@fieldParentPtr()` takes the alignment of the field pointer type over to the alignment of the returned struct pointer type, but the struct might actually have a larger alignment (e.g. if the field contains only 32-bit values, the field's alignment would be 4-byte, but suppose the struct contains a 64-bit value, so the struct's alignment is 8-byte). In the latter case Zig would produce a compile error, as we'd be assigning a lower-aligned pointer to a higher-aligned pointer. Using `@alignCast()` tells the Zig compiler that we are sure that the resulting pointer will be actually sufficiently aligned.

As the contents of structs and their fields can change during program's development, and as we typically do not have any specific expectations towards the alignments of structs (but rather expect them to simply have whatever alignments are reasonable), we probably shouldn't assume that a struct field's type natural alignment is as large as the alignment of the entire struct's type. But by omitting the `@alignCast()` around the `@fieldParentPtr()` we would effectively be making such assumption. Avoiding this assumption is the rationale behind the idea to surround `@fieldParentPtr()` with an `@alignCast()` by default (other things being equal).

## 6.12 Inherited methods

The inheritance approach described in Section 6.11 doesn't truly inherit from the base class. Yes, the base members are contained inside the derived "class", but they are not available as members of the derived class, one needs to explicitly access the base field in order to get to base members.

The same applies to methods: if the "base class" contains any methods, those are not available as methods of the "derived class". With a single-step inheritance, as in our example in Section 6.11, this might be not a big deal. But suppose we have a multiple levels-deep inheritance. It might quickly become illegible to access the upper "base classes":

```
obj.base.base.base.method();
```

The `usingnamespace` feature of Zig language allowed the implementation of a trick, where one could "append" the methods from the "base class" to the "derived class", and do so on each level of inheritance. With `usingnamespace` about to be deprecated we have to resort to other means. Short of resorting to code auto-generation, the two main available approaches seem to be the following:

- Implement a helper utility function, which will search through the inheritance hierarchy for a method with a given name and call it:

  ```
  const result = callInherited(
      &obj,
      "methodName",
      .{ arg1, arg2, arg3 },
  );
  ```

  where the function itself would be defined along the following lines:

  ```
  pub fn callInherited(
      obj_ptr: anytype,
      comptime method_name: []const u8,
  ```

```
    args_tuple: anytype,
) CallInheritedResultType(
    @TypeOf(obj_ptr.*),
    method_name,
) {
    // findBase() function would find the base pointer
    // given the method name
    const base_ptr = findBase(obj_ptr, method_name);

    const BaseType = @TypeOf(base_ptr.*);
    const method = @field(BaseType, method_name);

    // We cannot call the method using the () operator,
    // as the arguments are represented by a tuple,
    // but we can use the @call() builtin
    @call(.auto, method, .{base_ptr} ++ args_tuple);
}
```

The structs in the inheritance hierarchy would respectively need to follow a certain convention. E.g. each struct would use the name ‘`base`’ for its respective base field. The `callInherited` function would use the Zig’s reflection features to traverse the type infos of the respective structs, looking for the method with the specified name and calling it.

The drawback is apparently that a call to `callInherited` doesn’t look as nicely as a call to a struct’s method.

- Implement a function, searching through the struct hierarchy for a specific base type and returning a pointer to the respective base. In this case we’d need to know, to which particular “base class” the method that we’re calling belongs:

  ```
      const result = findBase(&obj, BaseType)
          .method(arg1, arg2, arg3);
  ```

  As a convenience feature, we could manually “inject” this function into each of the “classes” in the hierarchy:

  ```
  const Object = struct {
      pub const as = findBase;

      // rest of the "class" definition
  };
  ```

  in which case we would be able to call `findBase` in a nicer way:

  ```
      const result = obj.as(BaseType).method(arg1, arg2, arg3);
  ```

  The main drawback of this approach is that one always needs to specify the name of the “class” containing the respective method.

Possibly some other approaches already exist or will be developed with time.

## 6.13 Polymorphism

So far with have talked of inheritance without covering polymorphism. Let's now address the latter. As Zig doesn't have any built-in polypmorphism features, polymorphism will need to be implemented manually. The upside is that we're not restricted to a single kind of polymorphism.

### 6.13.1 C-style

A possibly well-forgotten kind of polymorphism is switch-based polymorphism, which used to be commonly found (and maybe still is) in C code. In its classical form we introduce an enum of all possible "derived types" and then switch on this enum, handling each "derived type" in a separate branch of the switch.

In C this approach requires quite an amount of boilerplate, writing out each branch of the switch. In Zig this is much easier, thanks to the `inline else` feature:

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

const A = struct {
    i: i32 = 0,
    fn print(self: *const @This()) void {
        std.debug.print("{}\n", .{self.i});
    }
};

const B = struct {
    f: f32 = 1.2,
    fn print(self: *const @This()) void {
        std.debug.print("{}\n", .{self.f});
    }
};

const C = struct {
    s: []const u8 = "str",
    fn print(self: *const @This()) void {
        std.debug.print("{s}\n", .{self.s});
    }
};

const U = union(enum) {
    a: A,
    b: B,
    c: C,

    fn print(self: *const @This()) void {
        switch (self.*) {
            inline else => |*v| v.print(),
        }
    }
    fn init(comptime as: std.meta.Tag(@This())) @This() {
        return @unionInit(@This(), @tagName(as), .{});
    }
};

pub fn main() void {
    const u = U.init(.b);
    u.print();
}
```

Notes:

- Notice the “inline else” switch in the `print` method of the ‘`U`’ union. The inline else prong covers all possible cases.
- We introduced the `U.init` method primarily to showcase the usage of the `@unionInit` builtin, which is often pretty handy in union metaprogramming. Otherwise, we could have simply initialized the ‘`u`’ variable in our example as:

  ```
  const u = U{ .b = .{} };
  ```

In the above example we have used a union to implement a polymorphic data type. This has a drawback that such union values always occupy the maximum possible memory, independently of the actual contained value. With `A`, `B` and `C` types being small this is not much of an issue, but it could become one with larger types. In such case, instead of using a union, we might decide to utilize the inheritance approach, but still rely on the inline else switch technique for the polymorphism:

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

const main_scope = @This();

const Base = struct {
    // By Zig style convention we should use
    // snake case for the enum values, but we
    // want to have enum values named identically
    // to the "derived classes", so we make an
    // exception to the naming rule.
    const Derived = enum { A, B, C };

    actual: Derived,

    fn init(d: Derived) @This() {
        return .{ .actual = d };
    }

    fn print(self: *const @This()) void {
        switch (self.actual) {
            inline else => |actual| {
```

```
                // Pick up the struct type (in the main scope)
                // with the same name as the active enum tag
                const Actual = @field(
                    main_scope,
                    @tagName(actual),
                );
                const derived: *const Actual = @alignCast(
                    @fieldParentPtr("base", self),
                );
                derived.print();
            },
        }
    }
};

const A = struct {
    base: Base = .init(.A),
    i: i32 = 0,
    fn print(self: *const @This()) void {
        std.debug.print("{}\n", .{self.i});
    }
};

const B = struct {
    base: Base = .init(.B),
    f: f32 = 1.2,
    fn print(self: *const @This()) void {
        std.debug.print("{}\n", .{self.f});
    }
};

const C = struct {
    base: Base = .init(.C),
    s: []const u8 = "str",
    fn print(self: *const @This()) void {
        std.debug.print("{s}\n", .{self.s});
    }
};

pub fn main() void {
    const b = B{};
    const base = &b.base;
    base.print();
}
```

The switch approach is likely to produce code with a jump table (or with a few conditional jumps if the number of "derived classes" is small), which might run faster than indirect calls used in virtual function table-based implementations. Also, in switch-based approach the knowledge of all "derived classes" is explicitly present at each callsite (in the form of the enum), which provides extra opportunities to instantiate various pieces of generic code (like generically implementing extra "virtual methods" outside of the "derived classes", which requires techniques like "visitor pattern" in C++).

### 6.13.2 C++-style

Instead of using switches to implement polymorphism, we could implement virtual function tables, like in C++:

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

const main_scope = @This();

const Base = struct {
    const Vftbl = struct {
        print: *const fn (*const Base) void,

        fn init(T: type) @This() {
            return .{
                .print = T.print,
            };
        }
    };

    vftbl: *const Vftbl,

    fn init(Derived: type) @This() {
        return .{ .vftbl = comptime &.init(Derived) };
    }

    fn print(self: *const @This()) void {
        self.vftbl.print(self);
    }
};

const A = struct {
    base: Base = .init(@This()),
    i: i32 = 0,
    fn print(base: *const Base) void {
        const self: *const @This() = @alignCast(
            @fieldParentPtr("base", base),
        );
        std.debug.print("{}\n", .{self.i});
    }
};

const B = struct {
    base: Base = .init(@This()),
    f: f32 = 1.2,
    fn print(base: *const Base) void {
        const self: *const @This() = @alignCast(
            @fieldParentPtr("base", base),
        );
        std.debug.print("{}\n", .{self.f});
    }
};

const C = struct {
    base: Base = .init(@This()),
    s: []const u8 = "str",
    fn print(base: *const Base) void {
        const self: *const @This() = @alignCast(
            @fieldParentPtr("base", base),
        );
        std.debug.print("{s}\n", .{self.s});
    }
};

pub fn main() void {
    const b = B{};
    const base = &b.base;
    base.print();
}
```

Notes:

- Normally `Base.init()` is supposed to be used in the default initializers of the `base` fields of the "derived classes" and therefore would be called at comptime. However, just in case it's used in a different way, we added a `comptime` keyword in the initializer for the `.vftbl` field inside `Base.init()`. This will make sure that, even if `Base.init()` is called at runtime, the virtual function table as well as its address will be both generated as comptime values, and therefore it will be safe to store the address in the `.vftbl` field (no dangling pointer occurs).

### 6.13.3 Type-erased interfaces

The C++-style virtual function table-based polymorphism is kind of optimal memory-wise (only one extra pointer is needed), but it could give a hard-time to the optimizer, in case the information about the actual "derived type" is available at compile time. The problem is that the pointer to the virtual function table is stored among other members of the struct and the values of those members might be undergoing some changes, while the virtual function table pointer stays constant. So, it would be nice to tell the compiler that the virtual function table pointer is constant but we cannot do it: we either have to declare the entire struct as const (but we cannot do it, as other members need to change their values), or we do not (in which case the optimizer might miss the knowledge about the specific values stored in the virtual function table).

As a solution one could separate the virtual function table pointer from the "class" and store it inside an "interface fat pointer". Such fat pointer would contain the virtual function table pointer and a type-erased pointer to the object itself. This fat pointer can often be declared constant, making it much easier for the optimizer to reason about:

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

const main_scope = @This();

const Interface = struct {
    const Vftbl = struct {
        print: *const fn (*const anyopaque) void,

        fn init(T: type) @This() {
            return .{
                .print = T.print,
            };
        }
    };

    vftbl: *const Vftbl,
    obj: *anyopaque,

    fn init(derived_ptr: anytype) @This() {
        const Derived = @TypeOf(derived_ptr.*);
        return .{
            .vftbl = comptime &.init(Derived),
            .obj = derived_ptr,
        };
    }

    fn print(self: *const @This()) void {
        self.vftbl.print(self.obj);
    }
};

const A = struct {
    i: i32 = 0,
    fn print(obj: *const anyopaque) void {
        const self: *const @This() =
            @alignCast(@ptrCast(obj));
        std.debug.print("{}\n", .{self.i});
    }
    fn interface(self: *@This()) Interface {
        return .init(self);
    }
};

const B = struct {
    f: f32 = 1.2,
    fn print(obj: *const anyopaque) void {
        const self: *const @This() =
            @alignCast(@ptrCast(obj));
        std.debug.print("{}\n", .{self.f});
    }
    fn interface(self: *@This()) Interface {
        return .init(self);
    }
};

const C = struct {
    s: []const u8 = "str",
    fn print(obj: *const anyopaque) void {
        const self: *const @This() =
            @alignCast(@ptrCast(obj));
        std.debug.print("{s}\n", .{self.s});
    }
    fn interface(self: *@This()) Interface {
        return .init(self);
    }
};

pub fn main() void {
    var b = B{};
    const int = b.interface();
    int.print();
}
```

Notes:

- Pointers to `anyopaque` are Zig's counterpart to C/C++'s pointers to `void`. In Zig, `void` is used to denote a type containing no data, while type-erased pointers are supposed to use `anyopaque`.
- Pointers to `anyopaque`, being type-erased, have the minimum possible alignment, which is 1. For this reason it is usually necessary to wrap a `@ptrCast()` from `anyopaque` into an additional `@alignCast()`.
- The "interface" kind of polymorphism is used in Zig's standard library allocators to implement a uniform type-abstract API.

## 6.14 Pointers to members

Zig doesn't have C++'s pointers-to-members feature, but we could emulate it using field offsets:

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

pub fn PtrToField(Host: type, Field: type) type {
    if (@typeInfo(Host) != .@"struct")
        @compileError(@typeName(Host) ++ " is not a struct");

    return struct {
        field_offset: usize,

        pub fn init(comptime field_name: []const u8) @This() {
            const ActualField = @FieldType(Host, field_name);

            if (ActualField != Field)
                @compileError("Expected field of type " ++
                    @typeName(Field) ++
                    ", but field '" ++ field_name ++
                    "' of " ++ @typeName(Host) ++
                    " has type " ++ @typeName(ActualField));

            return .{
                .field_offset = @offsetOf(Host, field_name),
            };
        }

        pub fn apply(self: @This(), host: *Host) *Field {
            return @ptrFromInt(
                @intFromPtr(host) + self.field_offset,
            );
        }
    };
}

pub fn main() void {
    const S = struct {
        a: i32 = 0,
        b: i32 = 1,
    };
    const fptr: PtrToField(S, i32) = .init("b");

    var v = S{};
    const p = fptr.apply(&v);

    // Prints 1 (the value of the "b" field)
    std.debug.print("{}\n", .{p.*});
}
```

Notes:

- `@offsetOf()` is a counterpart of C/C++'s `offsetof()` macro.

- A more elaborate implementation could support both const and non-const pointers to host in a single `apply()` method. Alternatively, one could introduce a separate `applyConst()` method, accepting a const pointer to `Host` and returning a const pointer `Field`.

## 6.15 Closures

In C++ closures can be conveniently implemented by using lambdas. Zig doesn't have lambdas, so a bit more boilerplate would be required. Let's see how it could look.

In the following example we apply a closure to all array elements in turn. This is obviously over the top, as writing a simple loop over the array would have suffices. But we do it just in order to illustrate closure techniques, which would be applicable to other, more complicated scenarios.

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

// A single-item pointer to array would
// also be okay for the 'slice' parameter
fn forAllSliceElements(
    slice: anytype,
    functor: anytype,
) void {
    for (slice) |*item|
        functor.apply(item);
}

pub fn main() void {
    const Setter = struct {
        value: i32,
        pub fn apply(
            self: *const @This(),
            item: *i32,
        ) void {
            item.* = self.value;
        }
    };

    var a: [10]i32 = undefined;
    forAllSliceElements(&a, &Setter{ .value = 1 });

    std.debug.print("{any}\n", .{a});
}
```

Notes:

- We could pass the functor by value or by reference. In the above example we chose to pass it by reference (notice the '&' in front of the functor argument).

- In principle we do not need to give a name to the functor struct and could write smply:

  ```
  pub fn main() void {
      var a: [10]i32 = undefined;

      forAllSliceElements(&a, &struct {
  ```

```
        value: i32,
        pub fn apply(
            self: *const @This(),
            item: *i32,
        ) void {
            item.* = self.value;
        }
    }{ .value = 1 });

    std.debug.print("{any}\n", .{a});
}
```

- In the above example and its latter variation we have been passing the functor by const reference. If we need the functor to be modifiable, we'd need to explicitly create an auxiliary variable, e.g.:

```
pub fn main() void {
    var a: [10]i32 = undefined;
    var functor = struct {
        value: i32,
        pub fn apply(
            self: *@This(),
            item: *i32,
        ) void {
            item.* = self.value;
            self.value += 1;
        }
    }{ .value = 1 };

    forAllSliceElements(&a, &functor);

    std.debug.print("{any}\n", .{a});
}
```

  Notice the `self.value += 1;` increment that we added to the `apply` function.

## 6.16 Private data

Zig doesn't have a concept of private/protected/public data and methods. The closest feature is the `pub` modifier for the decls, but that one is file- rather than entity-based, so it's more like an anti-counterpart of C's `static` visibility modifier.

Whether struct's data fields are supposed to be accessed "from the outside" is a design decision which cannot be enforced using Zig's language features. Therefore one generally has to resort to documentation or self-documenting features of the code. E.g. one could consider the following options:

- Explicitly document struct fields making it clear, which fields are and which are not a part of struct's public API.

- Use a naming convention, which makes it clear which fields are “public” and which are “private”. E.g. prefix all private fields with an underscore.
- Pack all “private” fields into a single field named ‘`impl`’ or ‘`priv`’.
- If one really wants to prevent access to the “private fields”, hide the actual data implementation behind a facade, and make the actual data types inaccessible from the outside of that facade.[3]

## 6.17 C/C++/Zig construct correspondence

The following table describes the *approximate* correspondence of a selected subset of C/C++ constructs to Zig constructs. The selection focuses on C/C++ constructs which do not have a 100% or 99% identical counterpart in Zig, but still do have Zig counterparts or close parallels. Some of the table entries may be providing only approximate translations, since the language constructs do not necessarily correspond 1:1 to each other.

| **C/C++ construct** | **Zig construct** |
|---|---|
| `*p` | `p.*` |
| `p->field` | `p.field` |
| `i++;` | `i += 1;` |
| `i--;` | `i -= 1;` |
| `a \|\| b` | `a or b` |
| `a && b` | `a and b` |
| `i / j` | `@divTrunc(i, j)` (signed)<br>`i / j` (unsigned) |
| `(a, b)` | `blk: {`<br>`    a;`<br>`    break :blk b;`<br>`}` |
| `c ? a : b` | `if (c) a else b` |
| `if(c) a; else b;` | `if (c) a else b;` |

[3]One still might be able to access the data types by importing the respective source files, manually specifying these source files’ paths. But such thing is probably rather unlikely to happen inadvertently.

```
for (auto a: array)                  for (&array) |a|

for (auto &a: array)                 for (&array) |*a|

for (size_t n = a; n < b ; n++)      for (a..b) |n|

                                     var i: i32 = a;
for (int i = a; n < b ; i += c)      while (i < b) : (i += c)

                                     while (true) {
                                         .....
do { ..... } while(c);                   if (!c) break;
                                     }

switch (n) {                         switch (n) {
    case 0: doA(); break;                0 => doA(),
    case 1: doB(); break;                1 => doB(),
    default: doC(); break;               else => doC(),
}                                    }

auto m = func(n);
switch (m) {                         switch (func(n)) {
    case 0: doA(); break;                0 => doA(),
    case 1: doB(); break;                1 => doB(),
    default: doC(m); break;              else => |m| doC(m),
}                                    }

static void f(int n) {               fn f(n: i32) void {

void f(int n) {                      pub fn f(n: i32) void {

void *                               *anyopaque

const void *                         *const anyopaque

size_t                               usize

ptrdiff_t                            isize

sizeof(T)                            @sizeOf(T)

offsetof(T,field)                    @offsetOf(T, "field")

unsigned char                        u8
```

| | |
|---|---|
| `unsigned char *` | `*u8`, `[*]u8`, `[]u8`, `[*:0]u8`, `[:0]u8` |
| `std::string` | `[:0]u8` |
| `strcmp()` | `std.mem.orderZ()` |
| `T [count]`<br>`std::array<T,count>` | `[count]T` |
| `reinterpret_cast<`<br>`    unsigned char *>(&a)` | `std.mem.asBytes(&a)` |
| `reinterpret_cast<`<br>`    unsigned char *>(ptr)`<br>`as_bytes(span)` | `std.mem.sliceAsBytes(slice)` |
| `reinterpret_cast<T *>`<br>`    (unsigned_char_ptr)` | `std.mem.bytesAsValue(T, slice)`<br>`std.mem.bytesAsSlice(T, slice)` |
| `std::bitcast<>()` | `@as(), @bitCast()` |
| `(int_type) int_value` | `@as(), @intCast(), @truncate()` |
| `(flt_type) flt_value` | `@as(), @fltCast()` |
| `(flt_type) int_value` | `@as(), @floatFromInt()` |
| `(int_type) flt_value` | `@as(), @intFromFloat()` |
| `(int_type) bool_value` | `@as(), @intFromBool()` |
| `fprintf(stderr, "%d\n", n);` | `std.debug.print("{}\n", .{n});` |
| `printf("%d\n", n);` | `const out = std.io`<br>`    .getStdOut().writer();`<br>`try out.print("{}\n", .{n});` |
| `FILE *f = fopen(name, "rb");` | `const file = try std.fs.cwd().`<br>`    .openFile(name, .{});` |
| `fclose(f);` | `file.close();` |
| `fread(......, f);` | `try file.reader(). ....;` |

```
BOOST_VERIFY(c);                    std.debug.assert(c);

typedef B A;
using A = B;                        const A = B;

struct S { ..... };                 const S = struct { ..... };

union U { ..... };                  const U = union { ..... };

enum E { ..... };                   const E = enum { ..... };

                                    pub fn negate(
template<class T>                       T: type,
T negate(T x) {                         x: T,
    return -x;                      ) T {
}                                       return -x;
                                    }

                                    pub fn negate(
                                        x: anytype,
                                    ) @TypeOf(x) {
                                        return -x;
                                    }

                                    pub fn add(
template<class T, T N>                  T: type,
T add(T x) {                            comptime n: T,
    return x + N;                       x: T,
}                                   ) T {
                                        return x + n;
                                    }

template<class T>                   fn S(T: type) type {
struct S { ..... };                     return struct { ..... };
                                    }

template<class T>                   const A = B;
using A = B<T>;

template<class T>                   fn A(T: type) type {
using A = .....;                        return .....;
                                    }

if constexpr (c) .....              if (comptime c) .....

std::vector                         std.ArrayList
```

| | |
|---|---|
| `boost::static_vector` | `std.BoundedArray` |
| `std::atomic` | `std.atomic.Value`<br>`@atomic....()`<br>`@cmpxchg....()` |
| `new T` | `try alloc.create(T)` |
| `delete p;` | `alloc.destroy(p)` |
| `new T[n]` | `try alloc.alloc(T, n)`<br>`try alloc`<br>`    .allocSentinel(T, n, snt)` |
| `delete [] p;` | `alloc.free(p)` |

# History

**1.0.0alpha (June 9, 2025)**

first public revision

**1.1.0alpha (June 10, 2025)**

- Added: pointers to fields
- Added: special features of string literals
- Added: reference to `std.mem.orderZ`
- Added: new entries to the construct correspondence table
- Fixed: added missing pointer to the `vftbl` field types in the discussion of C++-style polymorphism and type-erased interfaces.

# Detailed table of contents

# Index