Zig Basics

Systemprogrammierung für das 21. Jahrhundert

David Pierre Sugar

Zig Basics

David Pierre Sugar

Munich, Germany 2024: First Edition

Self Publishers Worldwide
Seattle San Francisco New York
London Paris Rome Beijing Barcelona

Inhaltsverzeichnis

Vorwort

Zig ist eine Sprache geeignet für die Systemprogrammierung.

Das alleine macht Zig nicht besonders, jedoch verheiratet Zig die Simplizität von C mit vielen modernen Features, was vor allem Neulingen, die eine systemnahe Programmiersprache lernen wollen, zugute kommt.

Zig als Systemprogrammiersprache ist unter anderem geeignet für:

- Kryptographie
- Mikrokontrollerprogrammierung
- Dateisysteme
- Datenbanken
- Betriebssysteme
- Treiber
- Spiele
- Simulationen
- Die Entwicklung von höheren Programmiersprachen

Insbesondere Startups, aber auch große Unternehmen, haben in den letzten Jahren auf Zig als Programmiersprache und Build-System gesetzt. Darunter Uber[1], Tigerbeetle[2], und ZML[3]. Dies verwenden Zig in ganz unterschiedlichen Anwendungsbereichen, darunter Datenbanken und maschinellem Lernen.

In der Welt der Systemprogrammiersprachen reiht sich Zig neben C ein und verzichtet auf viele Konzepte die andere Programmiersprachen überkomplex machen, darunter Vererbung. Damit ist Zig erfrischend übersichtlich, was vor allem Einsteigern zu gute kommt, bietet jedoch auch viele Verbesserungen gegenüber C. Ein wichtiger Fokus liegt auf der Lesbarkeit des Codes, d.h. was man sieht wird vom Computer später auch tatsächlich so ausgeführt (mit Abstrichen natürlich). Insbesondere bedeutet das: keine versteckten Allokationen, bei denen die Sprache ohne Zutun des Entwicklers dynamisch Speicher alloziert. Alles was mit der Allokation von dynamischem Speicher zu tun hat ist in Zig explizit!

[1] https://www.uber.com/en-DE/blog/bootstrapping-ubers-infrastructure-on-arm64-with-zig/
[2] https://tigerbeetle.com/
[3] https://zml.ai/

Fun-Fact: Während der StackOverflow 2024 Developer Survey[4] gaben 6.2% der Befragten an „umfangreiche Entwicklungsarbeiten“ in Zig getätigt zu haben und 73.8% wollen die Sprache im kommenden Jahr (2025) nutzen. Damit ist Zig trotz seines Alpha-Status eine gern genutzte Programmiersprache und reiht sich von der Zahl der Anwender neben Sprachen wie Swift, Dart, Elixir und Ruby ein.

Zielgruppe

Falls Sie bereits Erfahrung mit C oder einer anderen systemnahen Programmiersprache haben und mehr über Zig erfahren wollen ist diese Buch für Sie. Wenn Sie Erfahrung mit einer höheren Programmiersprache haben und mehr über Systemprogrammierung und Zig erfahren wollen ist dieses Buch ebenfalls für Sie.

Grundsätzlich empfehle ich Ihnen parallel zum lesen dieses Buches eigene Programmierprojekte zu realisieren um praktische Erfahrung mit der Sprache zu sammeln. Beginnen Sie mit etwas einfachem, vertrauten und steigern Sie sich, sobald Sie ein Gefühl für die Sprache bekommen haben. Sie werden merken, dass die Grundlagen in Zig schnell zu erlernen sind, es gibt jedoch auch nach einiger Zeit viel zu entdecken. Sollten Sie etwas Inspiration benötigen, so kann Ihnen Project Euler[5] eventuell weiterhelfen.

Wichtig zu erwähnen ist, dass Zig derzeit noch nicht die Version 1.0 erreicht hat, d.h. die Sprache und damit auch die Standardbibliothek werden sich in Zukunft noch ändern. Damit kann es sein, dass bestimmte Beispiele mit einer zukünftigen Zig-Compiler-Version nicht mehr compilieren. Sollte das für Sie ein Dealbreaker sein, so empfehle ich Ihnen die Finger von diesem Buch zu lassen und zu warten bis Zig Version 1.0 veröffentlicht wurde.

Voraussetzungen

Die Zig-Version, die in diesem Buch verwendet wird ist 0.13.0[6]. Je nachdem wann Sie dieses Buch lesen kann es sein, dass diese Version nicht mehr aktuell ist. Bei Abweichungen von der angegebenen Version ist nicht garantiert, dass die in diesem Buch abgebildeten Beispiele compilieren.

Zwar sind die meisten Konzepte und Beispiele in diesem Buch unabhängig von einem bestimmten Betriebssystem und Architektur, jedoch geht das Buch grundsätzlich von einem x86_64 Linux System aus. Dies wird relevant wenn auf Assembler, Calling-Conventions und ähnliche Konzepte Bezug genommen wird, da diese immer sowohl von der Architektur als auch dem Betriebssystem abhängen. Sollte Ihr Computer eine dieser Anforderungen nicht erfüllen, so empfiehlt es sich ggf. ein virtuelle Maschine zu verwenden[7].

Struktur

Die ersten drei Kapitel beschäftigen sich mit den Grundlagen der Programmiersprache Zig. Das erste Kapitel bietet anhand von Beispielen einen Überblick über die Sprache. Im zweiten Kapi-

[4] https://survey.stackoverflow.co/2024/

[5] https://projecteuler.net/about

[6] https://ziglang.org/download/

[7] https://ubuntu.com/tutorials/how-to-run-ubuntu-desktop-on-a-virtual-machine-using-virtualbox#1-overview

tel werden die grundlegenden Datentypen der Programmiersprache näher beleuchtet. In Kapitel drei wird der Leser in grundlegende Konzepte der Speicherverwaltung eingeführt, die für die korrekte und sichere Entwicklung von Anwendungen unabdingbar sind.

Im zweiten Abschnitt des Buches werden wir anhand von Fallbeispielen verschiedene Einsatzszenarios von Zig näher betrachten, darunter:

- Schreiben eines Parsers
- Breakout

Zig bietet für jede Compiler-Version zusätzliche Ressourcen zum Lernen der Sprache und als Referenz[8], darunter die Language Reference und die Online-Dokumentation der Standardbibliothek. Diese können beim Entwickeln eigener Projekte aber auch beim nachvollziehen der Code-Beispiele eine große Hilfe darstellen.

Konventionen

Die folgenden Konventionen werden in diesem Buch eingehalten:

Italic: Markiert neue Begriffe, URLs, Email-Adressen, Dateinamen und -endungen.

`Konstanter Abstand`: Wird verwendet für Programmbeispiele, sowie zum benennen von Programmbausteinen, wie etwa Variablennamen oder Umgebungsvariablen.

`Konstanter Abstand Fett`: Zeigt Kommandos oder andern, vom Nutzer zu tippenden, Text.

Ziggy markiert einen Tipp bzw. einen Hinweis.

Code Beispiele

Die in diesem Buch abgebildeten Code-Beispiele finden sich auf Github unter todo zum Download.

Alle Beispiele können von Ihnen ohne Einschränkung verwendet werden. Sie brauchen die Autoren nicht explizit um Genehmigung fragen. Am Schluss geht es darum Ihnen zu helfen und nicht darum Ihnen Steine in den Weg zu legen.

Zitierungen würden uns freuen, sind jedoch keinesfalls notwendig. Ein Zitat umfasst gewöhnlich Titel, Autor, Publizist und ISBN. In diesem Fall wäre dies: ,,Zig Basics by David Pierre Sugar''.

Sollten Sie Fehler im Buch oder Code finden, die nicht auf unterschiedliche Compiler-Versionen zurückzuführen sind können Sie uns mit einem Verbesserungsvorschlag kontaktieren.

Fragen, Anmerkungen und Verbesserungen

Ich habe mein Bestes getan dieses Buch so informativ und technisch korrekt wie möglich zu gestalten. Ich bin mir jedoch auch sicher, dass dieses Buch besser sein könnte als es gerade ist.

[8] https://ziglang.org/learn/

Sollten Sie Fehler finden oder generell Feedback zu diesem Buch geben wollen, so können Sie mich unter david@thesugar.de kontaktieren. Dies gibt mir die Möglichkeit dieses Buch über die Zeit zu verbessern. Es ist mir jedoch nicht immer möglich zu antworten. Nehmen Sie es sich deswegen nicht zu Herzen wenn Sie nichts von mir hören.

Danksagung

TDB

Kapitel 1

Zig Crash Course

In diesem Kapitel schauen wir uns einige kleine Zig Programme an, damit Sie ein Gespür für die Programmiersprache bekommen. Machen Sie sich nicht zu viele Sorgen wenn Sie nicht alles sofort verstehen, in den folgenden Kapiteln werden wir uns mit den hier vorkommenden Konzept noch näher beschäftigen. Wichtig ist, dass Sie diese Kapitel nicht nur lesen sondern die Beispiel auch ausführen, um das meiste aus diesem Kapitel herauszuholen.

Zig installieren

Um Zig zu installieren besuchen Sie die Seite https://ziglang.org und folgen den Instruktionen unter „GET STARTED“[9].

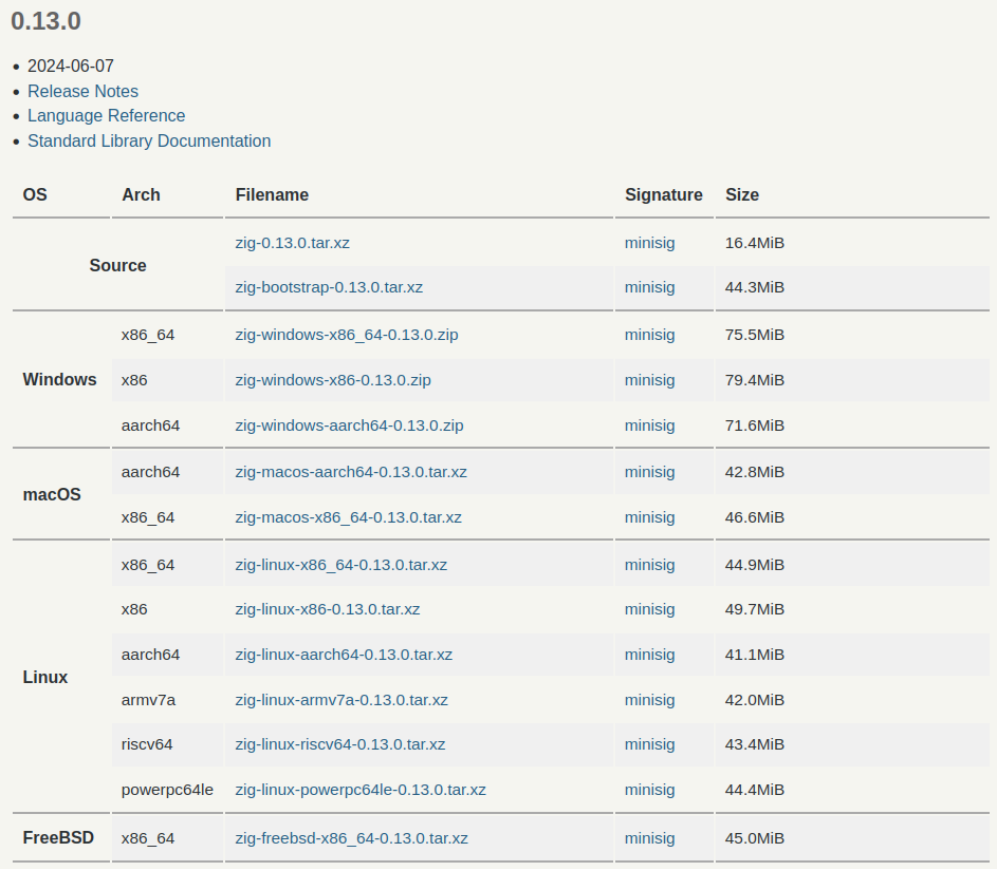
Die Installation ist unter allen Betriebssystemen relativ einfach durchzuführen. In der Download Sektion[10] finden Sie vorkompilierte Zig-Compiler für die gängigsten Betriebssysteme, darunter Linux, macOS und Windows.

Unter Linux können Sie mit dem Befehl `uname -a` Ihre Architektur bestimmen. In meinem Fall ist dies `X86_64`.

```
$ uname -a
Linux ... x86_64 x86_64 x86_64 GNU/Linux
```

Die Beispiele in diesem Buch basieren auf der Zig-Version 0.13.0, d.h. um den entsprechenden Compiler auf meinem Linux system zu installieren würde ich die Datei *zig-linux-x86_64-0.13.0.-tar.xz* aus der Download-Sektion herunterladen.

[9] https://ziglang.org/learn/getting-started/
[10] https://ziglang.org/download/



0.13.0

- 2024-06-07
- Release Notes
- Language Reference
- Standard Library Documentation

OS	Arch	Filename	Signature	Size
Source		zig-0.13.0.tar.xz	minisig	16.4MiB
		zig-bootstrap-0.13.0.tar.xz	minisig	44.3MiB
Windows	x86_64	zig-windows-x86_64-0.13.0.zip	minisig	75.5MiB
	x86	zig-windows-x86-0.13.0.zip	minisig	79.4MiB
	aarch64	zig-windows-aarch64-0.13.0.zip	minisig	71.6MiB
macOS	aarch64	zig-macos-aarch64-0.13.0.tar.xz	minisig	42.8MiB
	x86_64	zig-macos-x86_64-0.13.0.tar.xz	minisig	46.6MiB
Linux	x86_64	zig-linux-x86_64-0.13.0.tar.xz	minisig	44.9MiB
	x86	zig-linux-x86-0.13.0.tar.xz	minisig	49.7MiB
	aarch64	zig-linux-aarch64-0.13.0.tar.xz	minisig	41.1MiB
	armv7a	zig-linux-armv7a-0.13.0.tar.xz	minisig	42.0MiB
	riscv64	zig-linux-riscv64-0.13.0.tar.xz	minisig	43.4MiB
	powerpc64le	zig-linux-powerpc64le-0.13.0.tar.xz	minisig	44.4MiB
FreeBSD	x86_64	zig-freebsd-x86_64-0.13.0.tar.xz	minisig	45.0MiB

Abbildung 1: Download Seite von https://ziglang.org/download/

Mit dem *tar* Kommandozeilenwerkzeug kann das heruntergeladene Archiv danach entpackt werden.

```
$ tar -xf zig-linux-x86_64-0.13.0.tar.xz
```

Der entpackte Ordner enthält die Folgenden Dateien.

```
$ ls zig-linux-x86_64-0.13.0
doc  lib  LICENSE  README.md  zig
```

- **doc**: Die Referenzdokumentation der Sprache. Diese ist auch online, unter https://ziglang.org/documentation/0.13.0/, zu finden und enthält einen Überblick über die gesamte Sprache. Ich empfehle Ihnen ergänzend zu diesem Buch die Dokumentation zu Rate zu ziehen.
- **lib**: Enthält alle benötigten Bibliotheken, inklusive der Standardbibliothek. Die Standardbibliothek enthält viel nützliche Programmbausteine, darunter geläufige Datenstrukturen, einen JSON-Parser, Kompressionsalgorithmen, kryptographische Algorithmen und Protokolle und vieles mehr. Eine Dokumentation der gesamten Standardbibliothek findet sich online unter https://ziglang.org/documentation/0.13.0/std/.
- **zig**: Dies ist ein Kommandozeilenwerkzeug mit dem unter anderem Zig-Programme kompiliert werden können.

Um den Zig-Compiler nach dem Entpacken auf einem Linux System zu installieren, können wir diesen nach */usr/local/bin* verschieben.

```
$ sudo mv zig-linux-x86_64-0.13.0 /usr/local/bin/zig-linux-x86_64-0.13.0
```

Danach erweitern wir die `$PATH` Umgebungsvariable um den Pfad zu unserem Zig-Compiler. Dies können wir in der Datei *~/.profile* oder auch *~/.bashrc* machen[11].

```
# Sample .bashrc for SuSE Linux

# ...

export PATH="$PATH:/usr/local/bin/zig-linux-x86_64-0.13.0"
```

Nach Änderung der Konfigurationsdatei muss diese neu geladen werden. Dies kann entweder durch das öffnen eines neuen Terminalfensters erfolgen oder wir führen im derzeitigen Terminal das Kommando **`source .bashrc`** in unserem Home-Verzeichnis aus. Danach können wir zum überprüfen, ob alles korrekt installiert wurde, das Zig-Zen auf der Kommandozeile ausgeben lassen. Das Zig-Zen kann als die Kernprinzipien der Sprache und ihrer Community angesehen werden, wobei man dazu sagen muss, dass es nicht „die eine“ Community gibt.

```
$ source ~/.bashrc
$ zig zen

 * Communicate intent precisely.
 * Edge cases matter.
 * Favor reading code over writing code.
 * Only one obvious way to do things.
 * Runtime crashes are better than bugs.
 * Compile errors are better than runtime crashes.
 * Incremental improvements.
 * Avoid local maximums.
 * Reduce the amount one must remember.
 * Focus on code rather than style.
 * Resource allocation may fail;
     resource deallocation must succeed.
 * Memory is a resource.
 * Together we serve the users.
```

Mit dem Kommando **`zig help`** lässt sich ein Hilfetext auf der Kommandozeile anzeigen, der die zu Verfügung stehenden Kommandos auflistet.

Praktisch ist, dass Zig für uns ein neues Projekt, inklusive Standardkonfiguration, anlegen kann.

[11] Je nach verwendetem Terminal kann die Konfigurationsdatei auch anders heißen.

```
$ mkdir hello && cd hello
$ zig init
info: created build.zig
info: created build.zig.zon
info: created src/main.zig
info: created src/root.zig
info: see `zig build --help` for a menu of options
```

Das Kommando initialisiert den gegebenen Ordner mit Template-Dateien, durch die sich sowohl eine Executable, als auch eine Bibliothek bauen lassen. Schaut man sich die erzeugten Dateien an so sieht man, dass Zig eine Datei namens *build.zig* erzeugt hat. Bei dieser handelt es sich um die Konfigurationsdatei des Projekts. Sie beschreibt aus welchen Dateien eine Executable bzw. Bibliothek gebaut werden soll und welche Abhängigkeiten (zu anderen Bibliotheken) diese besitzen. Ein bemerkenswertes Detail ist dabei, dass *build.zig* selbst ein Zig Programm ist, welches in diesem Fall zur Compile-Zeit ausgeführt wird um die eigentlichle Anwendung zu bauen.

Die Datei *build.zig.zon* enthält weitere Informationen über das Projekt, darunter dessen Namen, die Versionsnummer, sowie mögliche Dependencies. Dependencies können dabei lokal vorliegen und über einen relativen Pfad angegeben oder von einer Online-Quelle, wie etwa Github, bezogen werden. Die Endung der Datei steht im übrigen für Zig Object Notation (ZON), eine Art Konfigurationssprache für Zig, die derzeit, genauso wie Zig selbst, noch nicht final ist.

Schauen wir in *src/main.zig*, so sehen wir das Zig für uns ein kleines Programm geschrieben hat.

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

pub fn main() !void {
    std.debug.print("All your {s} are belong to us.\n", .{"codebase"});

    const stdout_file = std.io.getStdOut().writer();
    var bw = std.io.bufferedWriter(stdout_file);
    const stdout = bw.writer();

    try stdout.print("Run `zig build test` to run the tests.\n", .{});

    try bw.flush(); // don't forget to flush!
}
```

Der Code kann auf den ersten Blick überwältigend wirken, schauen wir ihn uns deswegen Stück für Stück an.

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

Mit der `@import()` Funktion importieren wir die Standardbibliothek (`std`) und binden diese an eine Konstante mit dem selben Namen. Die Standardbibliothek ist eine Ansammlung von

nützlichen Funktionen und Datentypen, die während der Entwicklung von Anwendungen häufiger zum Einsatz kommen und deswegen vom Zig zur Verfügung gestellt werden. Die Funktion `@import()` wird nicht nur zum importieren der Standardbibliothek verwendet, sondern auch um auf Module und andere, zu einem Projekt gehörende, Quelldateien zuzugreifen.

Nach der Definition der Konstante `std` beginnt die `main` Funktion:

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

Unsere `main` Funktion beginnt, wie alle Funktionen, mit `fn` und dem Namen der Funktion. Sie gibt keinen Wert zurück, aus diesem Grund folgt auf die leere Parameterliste `()` der Rückgabetyp `void`. Das Ausrufezeichen `!` weist darauf hin, das die Funktion einen Fehler zurückgeben kann. Fehler in Zig sind eigenständige Werte, die von einer Funktion zurückgegeben werden können und sich semantisch vom eigentlichen Rückgabewert unterscheiden.

```
std.debug.print("All your {s} are belong to us.\n", .{"codebase"});
```

Als erstes gibt die `main` Funktion einen String über die Debugausgabe auf der Kommandozeile aus. Die Funktion `print` erwartet dabei einen Format-String, der mit Platzhaltern (z.B. `{s}`) versehen werden kann, sowie eine Liste an Ausdrücken (z.B. `.{"codebase"}`) deren Werte in den String eingefügt werden sollen. Der Platzhalter `{s}` gibt z.B. an, dass an der gegebenen Stelle ein String eingefügt werden soll. Neben `s` gibt es unter anderem noch `d` für Ganzzahlen und `any` für beliebige werte.

```
const stdout_file = std.io.getStdOut().writer();
var bw = std.io.bufferedWriter(stdout_file);
const stdout = bw.writer();
```

Via `std.io` können wir mit `getStdIn()`, `getStdOut()` und `getStdErr()` auf `stdin`, `stdout` und `stderr` zugreifen. Alle drei Funktionen geben jeweils eine Objekt vom Typ `File` zurück. Die Funktion `writer()` welche auf der stdout-Datei aufgerufen wird, gibt einen `Writer` zurück. Ein `Writer` ist ein Wrapper um einen beliebiges Datenobjekt (z.B. eine offene Datei, ein Array, ...) und stellt eine standartisiertes Interface zur Verfügung um Daten zu serialisieren. In unserem Fall wird der `stdout_file` Writer wiederum in einen `BufferedWriter` gewrapped, welcher nicht bei jedem einzelnen Schreibvorgang auf die Datei `stdout` zugreift, sondern erst wenn genug Daten geschrieben wurden bzw. wenn die Funktion `flush()` aufgerufen wird. Die Konstante `stdout` ist also ein `Writer` der einen `Writer` umschließt, der eine Datei umschließt, in die schlussendlich geschrieben werden soll.

```
try stdout.print("Run `zig build test` to run the tests.\n", .{});
```

Der `BufferedWriter` (`stdout`) wird verwendet um (indirekt) den String „Run zig build test to run the tests.“ nach stdout (standardmäßig die Kommandozeile) zu schreiben. Da diese Schreiboperation fehlschlagen kann wird vor den Ausdruck ein `try` gestellt. Damit wird ein potenzieller Fehler „nach oben“ propagiert, was im gegebenen Fall zu einem Programmabsturz führen würde, da `main` keine Funktion über sich besitzt. Als Alternative könnte mit einem `catch` Block der Fehler explizit abgefangen werden.

```
try bw.flush();
```

Um sicher zu gehen, dass auch alle Daten aus dem `BufferedWriter` tatsächlich geschrieben wurden, muss schlussendlich `flush()` aufgerufen werden.

Das von Zig vorbereitete „Hello, World“-Programm kann mit **`zig build run`**, von einem beliebigen Ordner innerhalb des Zig-Projekts, ausgeführt werden.

```
$ zig build run
All your codebase are belong to us.
Run `zig build test` to run the tests.
```

Im gegebenen Beispiel wurden zwei Schritte ausgeführt. Zuerst wurde der Zig-Compiler aufgerufen um das Programm in *src/main.zig* zu kompilieren und im zweiten Schritt wurde das Programm ausgeführt. Zig platziert dabei seine Kompilierten Anwendungen in *zig-out/bin* und Bibliotheken in *zig-out/lib*.

Funktionen

Zig's Grammatik ist sehr überschaubar und damit leicht zu erlernen. Diejenigen mit Erfahrung in anderen C ähnlichen Programmiersprachen wie C, C++, Java oder Rust sollten sich direkt Zuhause fühlen. Die unterhalb abgebildete Funktion berechnet den größten gemeinsamer Teiler (greatest common divisor) zweier Zahlen.

chapter01/gcd.zig

```
fn gcd(n: u64, m: u64) u64 {
    return if (n == 0)
        m
    else if (m == 0)
        n
    else if (n < m)
        gcd(m, n)
    else
        gcd(m, n % m);
}
```

Das `fn` Schlüsselwort markiert den Beginn einer Funktion. Im gegebenen Beispiel definieren wir eine Funktion mit dem Name `gcd`, welche zwei Argumente `m` und `n`, jeweils vom Typ

`u64`, erwartet. Nach der Liste an Argumenten in runden Klammern folgt der Typ des erwarteten Rückgabewertes. Da die Funktion den größten gemeinsamen Teiler zweier `u64` Ganzzahlen berechnet ist auch der Rückgabewert vom Typ `u64`. Der Körper der Funktion wird in geschweifte Klammern gefasst.

Zig unterscheidet zwischen zwei Variablen-Typen, Variablen und Konstanten. Konstanten können nach ihrer Initialisierung nicht mehr verändert werden, während Variablen neu zugewiesen werden können. Funktionsargumente zählen grundsätzlich zu Konstanten, d.h. sie können nicht verändert werden. Der Zig-Compiler erzwingt die Nutzung von Konstanten, sollte eine Variable nach ihrer Initialisierung nicht mehr verändert werden. Dies ist eine durchaus kontroverse Designentscheidung, welche aber auf das Zig-Zen zurückgeführt werden kann, das besagt: „Favor reading code over writing code". Sollten Sie also eine Variable in fremden Code sehen so können Sie sicher sein, dass diese an einer anderen Stelle manipuliert bzw. neu zugewiesen wird.

Eine Besonderheit, die Zig von anderen Sprachen unterscheidet ist, dass Integer mit beliebiger Präzision unterstützt werden. Im obigen Beispiel handelt es sich bei `u64` um eine vorzeichenlose Ganzzahl (unsigned integer) mit 64 Bits, d.h. es können alle Zahlen zwischen 0 und $2^{64} - 1$ dargestellt werden. Zig unterstützt jedoch nicht nur `u8`, `u16`, `u32` oder `u128` sondern alle unsigned Typen zwischen `u0` und `u65535`.

Alle Zig-Basistypen sind Teil des selben `union`: std.builtin.Type. Das union beinhaltet den `Int` Typ welcher ein `struct` mit zwei Feldern ist, `signedness` und `bits`, wobei `bits` vom Typ `u16` ist, d.h. es können alle Integer-Typen zwischen 0 und $2^{16} - 1$ Bits verwendet werden. Ja Sie hören richtig, der Zig-Compiler ist seit Version 0.10.0 selbst in Zig geschrieben, d.h. er ist self-hosted.

Innerhalb des Funktonskörpers werden mittels `if` verschiedene Bedingungen abgefragt. Sollte eine beider Zahlen 0 sein, so wird jeweils die andere zurückgegeben, ansonsten wird `gcd` rekursiv aufgerufen bis für eine der beiden Zahlen die Abbruchbedingung (0) erreicht ist. Wie auch bei C muss die Bedingung in runde Klammern gefasst werden. Bei Einzeilern können die geschweiften Klammern um einen Bedingungsblock weggelassen werden. In diesem Fall wird der Rückgabewert der Bedingung an den umschließenden Block gereicht.

Mittels eines `return` Statements kann von einer Funktion in die aufrufende Funktion zurückgekehrt werden. Das Statement nimmt bei bedarf zusätzlich einen Wert der an die aufrufende Funktion zurückgegeben werden soll. Im obigen Beispiel gibt `gcd` mittels `return` den Wert des ausgeführten If-Else-Asudruck zurück.

Das vollständige Programm finden Sie im zugehörigen Github-Rerpository. Mittels `zig build-exe chapter01/gcd.zig` kann das Beispiel kompiliert werden.

Unit Tests

Wie von einer modernen Programmiersprache zu erwarten bietet Zig von Haus aus Unterstützung für Tests. Tests beginnen mit dem Schlüsselwort `test`, gefolgt von einem String der den Test bezeichnet. In geschweiften Klammern folgt der Test-Block.

chapter01/gcd.zig

```
test "assert that the gcd of 21 and 4 is 1" {
    try std.testing.expectEqual(@as(u64, 1), gcd(21, 4));
}
```

Die Standardbibliothek bietet unter `std.testing` eine ganze Reihe an Testfunktionen für verschiedene Datentypen und Situationen. Im obigen Beispiel verwenden wir `ExpectEqual`, welche als erstes Argument den erwarteten Wert erhält und als zweites Argument das Resultat eines Aufrufs von `gcd`. Die Funktion überprüft beide Werte auf ihre Gleichheit und gibt im Fehlerfall einen `error` zurück. Dieser Fehler kann mittels `try` propagiert werden, wodurch der Testrunner im obigen Beispiel erkennt, dass der Test fehlgeschlagen ist.

```
$ zig test chapter01/gcd.zig
All 1 tests passed.
```

Innerhalb einer Datei sind Definitionen auf oberster Ebene (top-level definitions) unabhängig von ihrer Reihenfolge, was die Definition von Tests mit einschließt. Damit können Tests an einer beliebigen Stelle definiert werden, darunter direkt neben der zu testenden Funktion oder am Ende einer Datei. Der Zig-Test-Runner sammelt automatisch alle definierten Tests und führt dies beim Aufruf von **`zig test`** aus. Worauf Sie jedoch achten müssen ist, dass Sie ausgehend von der Wurzel-Datei (in den meisten Fällen *src/root.zig*), die konzeptionell den Eintritspunkt für den Compiler in ihr Programm oder Ihre Bibliothek darstellt, Zig mitteilen müssen in welchen Dateien zusätzlich nach Tests gesucht werden soll. Dies bewerkstelligen Sie, indem Sie die entsprechende Datei innerhalb eines Tests importieren.

```
const foo = @import("foo.zig");

test "main tests" {
    _ = foo; // Tell test runner to also look in foo for tests
}
```

Comptime

Die meisten Sprachen erlauben eine Form von Metaprogrammierung, d.h. das Schreiben von Code der wiederum Code generiert. In C können die gefürchteten Makros mit dem Präprozessor verwendet werden und Rust bietet sogar zwei verschiedene Typen von Makros, jeweils mit einer eigenen Syntax. Zig bietet mit `comptime` seine eigene Form der Metaprogrammierung. Was Zig von anderen kompilierten Sprachen unterscheidet ist, dass die Metaprogrammierung in der Sprache selber erfolgt, das heißt wer Zig programmieren kann, der hat das nötige Handwerkszeug um auch Metaprogrammierung in Zig zu betreiben.

Ein Aufgabe für die Metaprogrammierung sehr gut geeignet ist, ist die Implementierung von Container-Typen wie etwa `std.ArrayList`. Eine `ArrayList` ist ein Liste von Elementen eines

beliebigen Typen, die eine Menge an Standardfunktionen bereitstellt um die Liste zu manipulieren. Nun wäre es sehr aufwändig die `ArrayList` für jeden Typen einzeln implementieren zu müssen. Aus diesem Grund ist `ArrayList` als Funktion implementiert, welche zur Compilezeit einen beliebigen Typen übergeben bekommt auf Basis dessen einen `ArrayList`-Typ generiert.

```
var list = std.ArrayList(u8).init(allocator);
try list.append(0x00);
```

Der Funktionsaufruf `ArrayList(u8)` wird zur Compilezeit ausgewertet und gibt einen neuen Listen-Typen zurück, mit dem sich eine Liste an `u8` Objekten managen lassen. Auf diesem Typ wird `init()` aufgerufen um eine neu Instanz des Listen-Typs zu erzeugen. Mit der Funktion `append()` kann z.B., ein Element an das Ende der Liste angehängt werden. Eine stark simplifizierte Version von `ArrayList` könnte wie folgt aussehen.

chapter01/my-arraylit.zig

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

// Die Funktion erwartet als Compilezeitargument einen Typen `T`
// und gibt ein Struct zurück, dass einen Wrapper um einen Slice
// des Type `T` darstellt.
//
// Der Wrapper implementiert Funktionen zum managen des Slices
// und unterstützt unter anderem:
// - das Hinzufügen neuer Elemente
pub fn MyArrayList(comptime T: type) type {
    return struct {
        items: []T,
        allocator: std.mem.Allocator,

        // Erzeuge eine neue Instanz von MyArrayList(T).
        // Der übergebene Allocator wird von dieser Instanz gemanaged.
        pub fn init(allocator: std.mem.Allocator) @This() {
            return .{
                .items = &[_]T{},
                .allocator = allocator,
            };
        }

        pub fn deinit(self: *@This()) void {
            self.allocator.free(self.items);
        }

        // Füge da Element `e` vom Typ `T` ans ende der Liste.
        pub fn append(self: *@This(), e: T) !void {
            // `realloc()` kopiert die Daten bei Bedarf in den neuen
```

```
            // Speicherbereich aber die Allokation kann auch
            // fehlschlagen. An dieser Stelle verbleiben wir der
            // Einfachheit halber bei einem `try`.
            self.items = try self.allocator.realloc(self.items, self.items.len
+ 1);
            self.items[self.items.len - 1] = e;
        }
    };
}

pub fn main() !void {
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    const allocator = gpa.allocator();

    var list = MyArrayList(u8).init(allocator);
    defer list.deinit();

    try list.append(0xAF);
    try list.append(0xFE);

    std.log.info("{s}", .{std.fmt.fmtSliceHexLower(list.items[0..])});
}
```

Mit dem `comptime` Keyword sagen wir dem Compiler, dass das Argument `T` zur Compilezeit erwartet wird. Beim Aufruf von `MyArrayList(u8)` wertet der Compiler die Funktion aus und generiert dabei einen neuen Typen. Das praktische ist, dass wir `MyArrayList` nur einmal implementieren müssen und diese im Anschluss mit einem beliebigen Typen verwenden können.

Der `comptime` Typ `T` kann innerhalb und auch außerhalb des von der Fukntion `MyArrayList` zurückgegebenen Structs, anstelle eines expliziten Typs, verwendet werden.

Structs die mit `init()` initialisiert und mit `deinit()` deinitialisiert werden sind ein wiederkehrendes Muster in Zig. Dabei erwartet `init()` meist einen `std.mem.Allocator` der von der erzeugten Instanz verwaltet wird.

Ein weiterer Anwendungsfall bei dem Comptime zum Einsatz kommen kann ist die Implementierung von Parsern. Ein Beispiel hierfür ist der Json-Parser der Standardbibliothek (`std.json`), welcher dazu verwendet werden kann um Zig-Typen als Json zu serialisieren und umgekehrt[12].

Um ein Zig-Objekt zu de-/serialisieren werden Informationen über den Typ des Objekts und dessen Beschaffenheit benötigt. Hierzu kommen die Funktionen `@TypeOf()` und `@typeInfo()` zum Einsatz, wie das folgende Beispiel zeigt.

chapter01/reflection.zig

[12]Die JavaScript Object Notation (JSON) ist eines der gängigsten Datenformate und wird unter anderem zur Übermittlung von Daten im Web verwendet (https://en.wikipedia.org/wiki/JSON).

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

const MyStruct = struct {
    a: u32 = 12345,
    b: []const u8 = "Hello, World",
    c: bool = false,
};

fn isStruct(obj: anytype) bool {
    const T = @TypeOf(obj);
    const TInf = @typeInfo(T);

    return switch (TInf) {
        .Struct => |S| blk: {
            inline for (S.fields) |field| {
                    std.log.info("{s}: {any}", .{ field.name, @field(obj,
field.name) });
            }

            break :blk true;
        },
        else => return false,
    };
}

pub fn main() void {
    const s = MyStruct{};

    std.debug.print("{s}", .{if (isStruct(s)) "is a struct!" else "is not a
struct!"});
}
```

Anstelle eines Typen kann `anytype` für Parameter verwendet werden. In diesem Fall wird der Typ des Parameters, beim Aufruf der Funktion, abgeleitet. Zig erlaubt Reflexion (type reflection). Unter anderem erlaubt Zig die Abfrage von (Typ-)Informationen über ein Objekt. Funktionen denen ein `@` vorangestellt sind heißen Builtin-Function (eingebaute Funktion) und werden direkt vom Compiler bereitgestellt, d.h., sie können überall in Programmen, ohne Einbindung der Standardbibliothek, verwendet werden.

Die Funktion `@TypeOf()` ist insofern speziell, als dass sie eine beliebige Anzahl an Ausdrücken als Argument annimmt und als Rückgabewert den Typ des Resultats zurückliefert. Die Ausdrücke werden dementsprechend evaluiert. Im obigen Beispiel wird `@TypeOf()` genutzt um den Typen des übergebenen Objekts zu bestimmen, da `isStruct()` aufgrund von `anytype` mit einem Objekt beliebigen Typs aufgerufen werden kann.

Die eigentliche Reflexion kann mithilfe der Funktion `@typeInfo()` durchgeführt werden, die zusätzliche Informationen über einen Typ zurückliefert. Felder sowie Deklarationen von `structs`,

`unions`, `enums` und `error` Sets kommen dabei in der selben Reihenfolge vor, wie sie auch im Source Code zu sehen sind. Im obigen Beispiel testen wir mittels eines `switch` Statements ob es sich um ein `struct` handelt oder nicht und geben dementsprechend entweder `true` oder `false` zurück. Sollte es sich um ein `struct` handeln, so iterieren wir zusätzlich über dessen Felder und geben den Namen des Felds, sowie dessen Wert aus. Den Wert des jeweiligen Felds erhalten wir, indem wir mittels `@field()` darauf zugreifen. Die Funktion `@field()` erwartet als erstes Argument ein Objekt (ein Struct) und als zweites Argument einen zu Compile-Zeit bekannten String, der den Namen des Felds darstellt, auf das zugegriffen werden soll. Damit ist `@field(s, "b")` das Äquivalent zu `s.b`.

Für jeden Typen, mit dem `isStruct()` aufgerufen wird, wird eine eigene Kopie der Funktion (zur Compile Zeit) erstellt, die an den jeweiligen Typen angepasst ist. Das Iterieren über die einzelnen Felder eines `structs` muss zur Compile Zeit erfolgen, aus diesem Grund nutzt die obige Funktion `inline` um die For-Schleife zu entrollen, d.h., aus der Schleife eine lineare Abfolge von Instruktionen zu machen.

```
$ zig build-exe chapter01/reflection.zig
$ ./reflection
info: a: 12345
info: b: { 72, 101, 108, 108, 111, 44, 32, 87, 111, 114, 108, 100 }
info: c: false
```

Reflexion kann in vielen Situationen äußerst nützlich sein, darunter der Implementierung von Parsern für Formate wie JSON oder CBOR[13], da im Endeffekt nur zwei Funktionen implementiert werden müssen, eine zum Serialisieren der Daten und eine zum Deserialisieren. Mithilfe von Reflexion kann dann, vom Compiler, für jeden zu serialisierenden Datentyp eine Kopie der Funktionen erzeugt werden, die auf den jeweiligen Typen zugeschnitten ist.

Kryptographie

Ein Großteil der Anwendungen, die Sie wahrscheinlich täglich verwenden, benutzt in irgend einer Form Kryptographie. Dabei handelt es sich grob gesagt um mathematische Algorithmen, mit denen vorwiegend die Vertraulichkeit (Confidentiality), Integrität (Integrity) und Authentizität (Authenticity) von Daten gewährleistet werden kann. Typische Anwendungsbereiche die Kryptographie verwenden sind Messenger, Video Chats, Networking (TLS), Passwortmanager und Smart Cards. Zig bietet in seiner Standardbibliothek bereits jetzt eine Vielzahl and kryptographischen Algorithmen und Protokollen, wobei ein Großteil davon von Frank Denis[14], Online auch bekannt als jedisct1, beigetragen wurde. Ohne groß ein Authoritätsargument aufmachen zu wollen, ist Frank der Maintainer von libsodium[15] und libhydrogen[16], zwei viel genutzte, kryptographische Bibliotheken.

[13] https://github.com/r4gus/zbor
[14] https://github.com/jedisct1
[15] https://github.com/jedisct1/libsodium
[16] https://github.com/jedisct1/libhydrogen

Wir werden uns in einem späteren Kapitel noch genauer mit Kryptographie auseinandersetzen, machen Sie sich deshalb keine Sorgen, wenn Sie nicht alles in diesem Abschnitt auf Anhieb verstehen. Fürs erste schauen wir uns einen gängigen Anwendungsfall von Kryptographie an, die Verschlüsselung einer Datei. Angenommen wir haben eine Datei deren Inhalt geheim bleiben soll und wir wollen des weiteren überprüfen können, dass der Inhalt der Datei nicht verändert wurde. In solch einem Fall bietet sich die Verwendung eines AEAD (Authenticated Encryption with Associated Data) Ciphers an. Zig bietet unter `std.crypto.aead` verschiedene AEAD Cipher an. Die Unterschiede zwischen den Ciphern ist für dieses Beispiel Out-of-Scope. Sie müssen sich fürs erste damit begnügen mir zu glauben, dass `XChaCha20Poly1305`[17] für diese Art von Problem eine gute Wahl ist. Der Name `XChaCha20Poly1305` enthält dabei zwei Informationen, die uns Aufschluss über die Zusammensetzung des Ciphers geben:

- `XChaCha20`: Zur Verschlüsselung der Daten wird die „Nonce-eXtended" Version der `ChaCha20` Stromchiffre verwendet. `XChaCha20` erwartet einen Schlüssel und eine Nonce (Number used once: Eine Byte-Sequenz die nur einmal für eine Verschlüsselung verwendet werden darf) und leitet daraus eine Schlüsselsequenz ab, die mit dem Klartext XORed wird. Die eXtended Version verwendet dabei eine 192-Bit Nonce anstelle einer 96-Bit Nonce, was es deutlich sicherer macht diese zufällig mittels eines (kryptographisch sicheren) Zufallszahlengenerators zu erzeugen. Dieser Teil des Algorithmus ist für die Vertraulichkeit der Daten verantwortlich.
- `Poly1305`: `Poly1305` ist ein Hash, der zur Erzeugung von (one-time) Message Authentication Codes (MAC) verwendet werden kann. MACs sind sogenannte Keyed-Hashfunktionen, bei denen in einen Hash (keine Sorge, wir werden uns noch näher damit beschäftigen) ein geheimer Schlüssel integriert wird. Die Hashsumme wird dabei in unserem Beispiel über den Ciphertext, d.h. den Verschlüsselten Text, gebildet[18]. Durch den Einbezug eines Schlüssels kann nicht nur überprüft werden, dass die Integrität der Datei nicht verletzt wurde (sie wurde nicht verändert), sondern es kann auch sichergestellt werden, dass die MAC von Ihnen generiert wurde, da nur Sie als Nutzer der Anwendung den geheimen Schlüssel kennen.

chapter01/encrypt.zig

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

const argon2 = std.crypto.pwhash.argon2;
const XChaCha20Poly1305 = std.crypto.aead.chacha_poly.XChaCha20Poly1305;

var gpa = std.heap.GeneralPurposeAllocator(.{}){};
const allocator = gpa.allocator();

const Mode = enum {
    encrypt,
    decrypt,
```

[17] https://datatracker.ietf.org/doc/html/rfc7539
[18] Dies wird als Encrypt-than-Mac bezeichnet.

```
};

pub fn main() !void {
    var password: ?[]const u8 = null;
    var mode: ?Mode = null;

    // Als erstes parsen wir die übergebenen Kommandozeilenargumente.
    // Diese bestimmen zum einen mit welchem Passwort die Daten
    // verschlüsselt werden sollen und zum anderen den Modus, d.h.
    // ob ver- bzw. entschlüsselt werden soll.
    var ai = try std.process.argsWithAllocator(allocator);
    defer ai.deinit();

    while (ai.next()) |arg| {
        // `std.mem.eql` kann dazu verwendet werden zwei Strings mit einander
zu vergleichen...
        if (arg.len > 11 and std.mem.eql(u8, "--password=", arg[0..11])) {
            password = arg[11..];
        } else if (arg.len >= 9 and std.mem.eql(u8, "--encrypt", arg[0..9]))
{
            mode = .encrypt;
        } else if (arg.len >= 9 and std.mem.eql(u8, "--decrypt", arg[0..9]))
{
            mode = .decrypt;
        }
    }

     // Sollten nicht alle benötigten Argumente übergeben worden sein, so
beenden wir den Prozess.
    if (password == null or mode == null) {
           std.log.err("usage: ./encrypt --password=<password> [--encrypt|--
decrypt]", .{});
        return;
    }

    // Als nächstes lesen wir die übergebenen Daten von `stdin` ein.
    const stdin = std.io.getStdIn();
    const data = try stdin.readToEndAlloc(allocator, 64_000);
    defer {
        // Wir überschreiben die Daten bevor wir den Speicher wieder freigeben.
        @memset(data, 0);
        allocator.free(data);
    }

    if (mode == .encrypt) {
        // Bei der Verschlüsselung müssen wir eine Reihe an (öffentlichen)
        // Parametern festlegen, die bei der Entschlüsselung wiederverwendet
```

```
        // werden müssen.

        // Als erstes müssen wir ein Schlüssel von unserem Passwort ableiten.
        // Hierfür verwenden wir die Argon2id Key-Derivation-Function (KDF).
        var salt: [32]u8 = undefined;
        std.crypto.random.bytes(&salt);

        var key: [XChaCha20Poly1305.key_length]u8 = undefined;
        try argon2.kdf(allocator, &key, password.?, &salt, .{
          // Die Parameter bestimmen wie aufwendig die Brechnung des Schlüssels
`key` ist.
            // Damit wird verhindert, diesen durch "Brute-Forcing" brechen zu
können.
            .t = 3,
            .m = 4096,
            .p = 1,
        }, .argon2id);

        // Nun können wir die Daten ver-/ bzw. entschlüsseln.

        // Der TAG wird von der encrypt() Funktion erzeugt und später
        // von decrypt() überprüft.
        var tag: [XChaCha20Poly1305.tag_length]u8 = undefined;

        // Für jede Verschlüsselung muss eine neue, einzigartige Nonce
        // verwendet werden. Da wir die eXtended Version von ChaCha20
        // verwenden, kann diese durch einen kryptographisch sicheren
        // Zufallszahlengenerator festgelegt werden.
        var nonce: [XChaCha20Poly1305.nonce_length]u8 = undefined;
        std.crypto.random.bytes(&nonce);

        XChaCha20Poly1305.encrypt(data, &tag, data, "", nonce, key);

          // Der Salt, Nonce und Tag müssen mit den verschlüsselten Daten
serialisiert werden,
        // da wir diese später zur Entschlüsselung benötigen.
        const stdout = std.io.getStdOut();
        try std.fmt.format(stdout.writer(), "{s}:{s}:{s}:{s}", .{
            // Wir serialisieren die Binärdaten in Hexadezimal.
            std.fmt.fmtSliceHexLower(salt[0..]),
            std.fmt.fmtSliceHexLower(nonce[0..]),
            std.fmt.fmtSliceHexLower(tag[0..]),
            std.fmt.fmtSliceHexLower(data),
        });
    } else {
        // Da wir die Daten in Hexadezimal serialisiert haben, müssen wir diese
        // wieder voneinander trennen und in Binärdaten umwandeln.
```

```
        var si = std.mem.split(u8, data, ":");

        const salt = si.next();
        if (salt == null or salt.?.len != 32 * 2) {
            std.log.err("invalid data (missing salt)", .{});
            return;
        }
        var salt_: [32]u8 = undefined;
        _ = try std.fmt.hexToBytes(&salt_, salt.?);

        const nonce = si.next();
        if (nonce == null or nonce.?.len != XChaCha20Poly1305.nonce_length *
2) {
            std.log.err("invalid data (missing nonce)", .{});
            return;
        }
        var nonce_: [XChaCha20Poly1305.nonce_length]u8 = undefined;
        _ = try std.fmt.hexToBytes(&nonce_, nonce.?);

        const tag = si.next();
        if (tag == null or tag.?.len != XChaCha20Poly1305.tag_length * 2) {
            std.log.err("invalid data (missing tag)", .{});
            return;
        }
        var tag_: [XChaCha20Poly1305.tag_length]u8 = undefined;
        _ = try std.fmt.hexToBytes(&tag_, tag.?);

        const ct = si.next();
        if (ct == null) {
            std.log.err("invalid data (missing cipher text)", .{});
            return;
        }

        const pt = try allocator.alloc(u8, ct.?.len / 2);
        defer {
            @memset(pt, 0);
            allocator.free(pt);
        }

        _ = try std.fmt.hexToBytes(pt, ct.?);

        // Danach können wir die deserialisierten Daten verwenden um
        // den Ciphertext zu entschlüsseln.
        var key: [XChaCha20Poly1305.key_length]u8 = undefined;
        try argon2.kdf(allocator, &key, password.?, &salt_, .{
            .t = 3,
            .m = 4096,
```

```
            .p = 1,
        }, .argon2id);

        try XChaCha20Poly1305.decrypt(pt, pt, tag_, "", nonce_, key);

        const stdout = std.io.getStdOut();
        try std.fmt.format(stdout.writer(), "{s}", .{pt});
    }
}
```

In diesem Beispiel laufen eine Vielzahl von Konzepten zusammen, die sie im Laufen diese Buches noch häufiger antreffen werden. Unsere Anwendung erwartet Daten, z.B. den Inhalt einer Datei, über `stdin`, sowie zwei Kommandozeilenargumente: `--password` und `--encrypt` bzw. `--decrypt`. Basierend auf diesen Argumenten werden die übergebenen Daten entweder verschlüsselt oder entschlüsselt und nach `stdout` geschrieben.

Wir beginnen mit einigen Top-Level-Deklarationen, damit wir den Pfad zu Datenstrukturen, wie etwa `XChaCha20Poly1305`, nicht immer ausschreiben müssen. Weiterhin definieren wir ein Enum `Mode` welches zwei operationelle Zustände ausdrücken kann, Verschlüsselung (`encrypt`) und Entschlüsselung (`decrypt`).

Innerhalb von `main` parsen wir zuerst die übergebenen Argumente, indem wir durch die Funktion `argsWithAllocator()` einen Iterator über die Kommandozeilenargumente beziehen und mithilfe dessen über die einzelnen Argumente iterieren. Iteratoren sind ein häufig wiederzufindendes Konzept und lassen sich hervorragend mit `while` Schleifen kombinieren. Solange `ai.next()` ein Element zurückliefert, wird diese an `arg` gebunden und die Schleife wird fortgeführt. Liefert `next()` den Wert `null` zurück, so wird automatisch aus der Schleife ausgebrochen.

Danach stellen wir sicher, dass sowohl ein Passwort als auch ein Modus vom Nutzer spezifiziert wurden. Sollte eines der beiden Argumente fehlen, so wird ein entsprechender Fehler gelogged und der Prozess vorzeitig beendet.

Als nächstes wird eine über `stdin` übergebene Datei eingelesen und an die Konstante `data` gebunden. Da der für die Datei benötigte Speicher dynamisch alloziert wird muss dieser wider freigegeben werden. Hierfür wird eine `defer`-Block verwendet, der vor Beendigung der Anwendung ausgeführt wird. Innerhalb dieses Blocks wird zusätzlich der Speicherinhalt mittels `@memset` überschrieben.

Der Umgang mit Sicherheitsrelevanten Daten ist durchaus herausfordernd. Grundsätzlich muss darauf geachtet werden, dass sensible Daten nicht zu lange im Speicher verweilen. Voraussetzung hierfür ist, dass Sie überhaupt wissen wo überall sensible Daten abgespeichert werden. Zum einen können Sie Daten, nachdem diese nicht mehr benötigt werden, überschreiben. Sie sollten jedoch

auch weniger offensichtlich Angriffsvektoren, wie das Swappen von Hauptspeicher, im Hinterkopf behalten.

Sowohl für die Ver- als auch Entschlüsselung muss zuerst ein geheimer Schlüssel vom übergebenen Passwort, mittels einer Key-Derivation-Funktion, abgeleitet werden. Für diese Beispiel wird *Argon2id*[19], der Gewinner der 2015 Password Hashing Competition, verwendet. Die Berechnung eines Schlüssels durch Argon2 hängt von den Folgenden (öffentlichen) Parametern ab:

- Salt: eine zufällige Sequenz die in die Schlüsselberechnung einfließt.
- Time: Die Anzahl an Iterationen für die Berechnung.
- Memory: Die Speicher-Kosten für die Berechnung.
- Parallelismus: Die Anzahl an parallelen Berechnungen.

Time, Memory und Parallelismus bestimmen wie aufwändig die Ableitung eines Schlüssels ist. Grundsätzlich gilt: je aufwendiger desto besser, jedoch schlägt sich dies auch in einer längeren Wartezeit nieder (spielen Sie deshalb gerne mit den Parametern). Alle Parameter werden bei der Verschlüsselung festgelegt und müssen mit dem Ciphertext zusammen gespeichert werden, da bei der Entschlüsselung die selben Parameter wieder in die KDF einfließen müssen um den Selben Schlüssel vom Passwort abzuleiten.

Zur Verschlüsselung wird eine zufällige Nonce generiert welche zusammen mit den zu verschlüsselnden Daten, einem Zeiger auf ein Array für den Tag, zusätzliche Daten (in diesem Fall der leere String `""`) und dem abgeleiteten Schlüssel an `encrypt` übergeben werden. Die Funktion verschlüsselt daraufhin die Daten. Danach wird der Salt, die Nonce, der Tag, sowie die verschlüsselten Daten, getrennt durch ein `:`, in die Standardausgabe `stdout` geschrieben.

Für die Entschlüsselung wird dieser String anhand der `:`, mittels `split`, aufgeteilt. Sollten die eingelesenen Daten nicht im erwarteten Format vorliegen, das heißt Salt, Nonce, Tag oder Ciphertext fehlen, so wird ein Fehler ausgegeben und die Anwendung beendet. Andernfalls, werden die eingelesenen Parameter verwendet um den Ciphertext, mittels `decrypt`, wieder zu entschlüsseln.

Das kleine Verschlüsselungsprogramm kann wie folgt verwendet werden:

```
$ cat hello.txt
Hello, World!
$ cat hello.txt | ./encrypt --password=supersecret --encrypt > secret.txt
$ cat secret.txt
828dfa14efa4b1f8242a8258a411301bd79bc4b7528294500305a4e9baaecbba:
85e4593786697e4e49212131a8e6e6bb68d25f43613dd870:ec666e95ebe1fa4c
53a1183379ae0dbd:80a7fe0475834364229c15dfb96d
$ cat secret.txt | ./encrypt --password=supersecret --decrypt
Hello, World!
```

Graphische Applikationen

[19] https://en.wikipedia.org/wiki/Argon2

Ein weiterer Anwendungsfall für Zig ist die Entwicklung graphischer Applikationen. Hierfür existiert eine Vielzahl an Bibliotheken, darunter GTK und QT. Was beide Bibliotheken gemeinsam haben ist, dass sie in C beziehungsweise C++ geschrieben sind. Normalerweise würde das die Entwicklung von Bindings voraussetzen, um die Bibliotheken in anderen Sprachen nutzen zu können. Zig integriert jedoch direkt C, wodurch C-Bibliotheken direkt verwendet werden können[20].

In diesem Abschnitt zeige ich Ihnen, wie sie eine simple GUI-Applikation mit GTK4 und Zig schreiben können. Hierfür müssen Sie zuerst einen neuen Projektordner anlegen.

```
$ mkdir gui
$ cd gui
$ zig init
info: created build.zig
info: created build.zig.zon
info: created src/main.zig
info: created src/root.zig
info: see `zig build --help` for a menu of options
```

Danach fügen Sie `gtk4` als Bibliothek zu Ihrer Anwendung hinzu. Hierfür öffnen Sie `build.zig` mit einem Texteditor und erweitern die Datei um die folgenden Zeilen:

chapter01/gui/build.zig

```
//...
const exe = b.addExecutable(.{
    //...
});
// Fügen Sie die folgenden beiden Zeilen hinzu
exe.linkLibC();
exe.linkSystemLibrary("gtk4");
//...
```

Stellen Sie sicher, dass Sie die Developer-Bibliothek von GTK4 auf Ihrem System installiert haben. Unter Debian/Ubuntu können Sie diese über den APT-Paket-Manager installieren.

```
sudo apt install libgtk-4-dev
```

Führen Sie danach **`zig build`** aus um zu überprüfen, dass Zig die benötigte Bibliothek auf Ihrem System findet. Erzeugen Sie als nächstes die Datei *src/gtk.zig* und fügen Sie den Folgenden Code hinzu:

chapter01/gui/src/gtk.zig

[20] Mit wenigen Einschränkungen. Zig scheitert zurzeit noch an der Übersetzung einer Makros.

```
pub usingnamespace @cImport({
    @cInclude("gtk/gtk.h");
});

const c = @cImport({
    @cInclude("gtk/gtk.h");
});

/// g_signal_connect re-implementieren
pub fn z_signal_connect(
    instance: c.gpointer,
    detailed_signal: [*c]const c.gchar,
    c_handler: c.GCallback,
    data: c.gpointer,
) c.gulong {
    var zero: u32 = 0;
    const flags: *c.GConnectFlags = @as(*c.GConnectFlags, @ptrCast(&zero));
    return c.g_signal_connect_data(
        instance,
        detailed_signal,
        c_handler,
        data,
        null,
        flags.*,
    );
}
```

Zig ist zwar ziemlich gut darin mit C zu integrieren, jedoch werden Sie von Zeit zu Zeit noch auf Probleme stoßen. In den meisten Fällen lässt sich dies jedoch relativ einfach lösen. Innerhalb von `src/gtk.zig` inkludieren wir zuerst die GTK4 Header-Datei `gtk.h`. Wie Ihnen vielleicht aufgefallen ist, haben wir an keiner Stelle innerhalb von `build.zig` auf diese Datei verwiesen. Zig reicht es in den aller meisten Fällen aus, wenn Sie die Bibliothek benennen die Sie einbinden möchten und fügt die benötigten Pfade automatisch hinzu.

Das Schlüsselwort `usingnamespace` sorgt dafür, dass wir auf alle in `gtk.h` deklarierten Objekte, über `gtk.zig`, direkt zugreifen können.

Eine in `gtk.h` deklarierte Funktion, die wir später noch benötigen, ist `g_signal_connect`. Diese lässt sich leider nicht ohne weiteres direkt verwenden (einer der seltenen Fälle bei denen Zig derzeit noch versagt). Aus diesem Grund implementieren wir die Funktion selber und nennen unsere Implementierung `z_signal_connect`.

Nun haben wir alles vorbereitet und können uns um die eigentliche Anwendung kümmern. Ersetzen Sie den Code in `src/main.zig` mit dem folgenden Programm:

chapter01/gui/src/main.zig

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

fn onActivate(app: *gtk.GtkApplication) void {
    const window: *gtk.GtkWidget = gtk.gtk_application_window_new(app);

    gtk.gtk_window_set_title(
        @as(*gtk.GtkWindow, @ptrCast(window)),
        "Zig Basics",
    );
    gtk.gtk_window_set_default_size(
        @as(*gtk.GtkWindow, @ptrCast(window)),
        920,
        640,
    );

    gtk.gtk_window_present(@as(*gtk.GtkWindow, @ptrCast(window)));
}

pub fn main() !void {
    const application = gtk.gtk_application_new(
        "de.zig.basics",
        gtk.G_APPLICATION_FLAGS_NONE,
    );
    _ = gtk.z_signal_connect(
        application,
        "activate",
        @as(gtk.GCallback, @ptrCast(&onActivate)),
        null,
    );
    _ = gtk.g_application_run(
        @as(*gtk.GApplication, @ptrCast(application)),
        0,
        null,
    );
}
```

Ganz oben importieren wir die Standardbibliothek, als auch die Datei `gtk.zig` unter dem Namen `gtk`. Danach folgt die Funktion `onActivate`, welche verwendet wird um ein GTK-Fenster zu erzeugen. Schauen wir uns aber zuerst die `main` Funktion an.

Innerhalb von `main` wird als erstes, mithilfe von `gtk_application_new`, ein Anwendungsobjekt erzeugt, welches an die Konstante `application` gebunden wird. Als nächstes wird an Callback registriert, der durch das Signal `activate` aufgerufen wird. Als Callback nutzen wir die Funktion `onActivate`. Nachdem die Anwendung mittels `g_application_run` die GTK-Anwendung gestartet hat, wird das `activate` Signal ausgelöst, wodurch `onActivate` aufgerufen wird.

Die Funktion `onActivate` erzeugt als erstes ein neues Fenster für die Anwendung und weist dem Fenster, mithilfe von `get_window_set_title`, den Titel *Zig Basics* zu. Danach wird eine Fenstergröße von *920 x 640* Pixeln festgelegt, bevor das Fenster mit `gtk_window_present` angezeigt wird.

Innerhalb des Root-Verzeichnisses des Projekts können Sie mit **`zig build run`** die Anwendung starten. Nach dem Starten des Programms sollten Sie ein leeres Fenster sehen.

Abbildung 2: Leeres GKT4-Fenster

Nur ein leeres Fenster ist etwas langweilig, deshalb fügen wir als nächstes noch einen Button hinzu, der den Text *„Hello, World!“* auf der Kommandozeile ausgibt. Ich weis, ein Button ist nicht viel spannender als ein leeres Fenster, er sollte jedoch als Beispiel genügen.

Zuerst muss ein Callback definiert werden, der aufgerufen wird sobald der Button vom Nutzer gedrückt wird.

chapter01/gui/src/main.zig

```
fn onButtonClicked(_: *gtk.GtkWidget, _: gtk.gpointer) void {
    std.log.info("Hello, World!", .{});
}
```

Callbacks in GTK erwarten zwei Argumente, einen Zeiger auf das Widget (z.B. der Button) welches den Callback ausgelöst hat und optional einen Zeiger auf Daten, die an die Funktion übergeben werden sollen. Da wir weder das Widget noch Daten benötigen, werden die Parameternamen durch `_` ersetzt. Damit stellen wir den Compiler zufrieden der erwartet, dass alle deklarierten Variablen verwendet werden, Parameter eingeschlossen.

Allgemein setzt sich eine GTK Anwendung aus Widgets (Bausteinen) zusammen. Alles was in einem Fenster angezeigt wird, wird intern als Baumstruktur, bestehend aus Objekten vom Typ `GtkWidget`, abgebildet, wobei das Fenster selber die Wurzel des Baums ist. `GtkWidget` ist dabei

ein generischer Typ, das heist er umfasst Verhalten das von allen Bausteinen geteilt wird, egal ob es sich dabei um einen Button, Text oder eine andere graphische Komponente handelt.

Fügen Sie den folgenden Code zwischen dem Aufruf von `gtk_window_set_default_size` und `z_signal_connect` ein.

chapter01/gui/src/main.zig

```
const button = gtk.gtk_button_new_with_label("Click Me!");
gtk.gtk_window_set_child(
    @as(*gtk.GtkWindow, @ptrCast(window)),
    @as(*gtk.GtkWidget, @ptrCast(button)),
);
_ = gtk.z_signal_connect(
    button,
    "clicked",
    @as(gtk.GCallback, @ptrCast(&onButtonClicked)),
    null,
);
```

Da alle Bausteine als `GtkWidget` verwendet werden können ist es teilweise nötig einzelne Zeiger auf den richtigen, von einer Funktion erwarteten, Parametertypen zu casten. Der Ausdruck `@as(*gtk.GtkWindow, @ptrCast(window))` bedeutet zum Beispiel: betrachte den Zeiger `window` als einen Zeiger zu einem `GtkWindow`.

Mit der Funktion `gtk_window_set_child` kann ein Widget als Kind des gegebenen Fensters gesetzt werden. Danach registrieren wir noch einen Callback für den Button, der bei einem Click (Signal *„clicked“*) ausgelöst wird. Als Callback verwenden wir die Funktion `onButtonClicked`, die wir zuvor definiert hatten.

Nachdem Sie das Programm mit **`zig build run`** gestartet haben sollten Sie innerhalb des Fensters einen Button sehen, der das Fenster ausfüllt.

Abbildung 3: GKT4-Fenster mit Button

Beim clicken des Buttons sollte *„Hello, World!“* auf der Kommandozeile ausgegeben werden. Herzlichen Glückwunsch! Sie haben ihre erste graphische Benutzerobefläche in Zig programmiert.

Zig als C Build-System

Zig integriert nicht nur hervorragend mit C sondern kann auch als Build-System für C und C++ Projekte verwendet werden. Damit stellt Zig unter anderem eine Alternative zu Make oder CMake dar.

Genau wie für Zig Projekte können Sie auch zu Ihren C und C++ Projekten einen `build.zig` Datei hinzufügen, welche den Build-Prozess beschreibt. Außerdem macht es Sinn eine `build.zig.zon` Datei hinzuzufügen, die zusätzliche Metadaten zu Ihrem Projekt liefert.

In diesem Beispiel möchte ich ihnen Zeigen, wie Sie eine kleine Bibliothek in C schreiben und diese im Anschluss in einem weiteren Projekt verwenden können.

Bibliothek schreiben

Legen Sie zuerst einen Ordner *math* an und initialisieren Sie diesen mit `cd math && zig init`. Entfernen Sie danach alle Dateien in *src* und fügen Sie die Datei *math.c* hinzu, welche den folgenden Code enthält:

chapter01/math/src/math.c

```c
#include "math.h"

int add(int a, int b) {
    return a + b;
}
```

Definieren Sie danach eine zugehörige Header-Datei:

chapter01/math/src/math.h

```
#ifdef __cplusplus
extern "C" {
#endif

    int add(int, int);

#ifdef __cplusplus
}
#endif
```

Unsere Bibliothek enthält genau eine Funktion `add()`, deren Prototyp in der Header-Datei deklariert wird. Damit die Bibliothek auch mit C++ verwendet werden kann, prüfen wir ob `__cplusplus` definiert ist und umschließen die Deklaration falls nötig mit `extern "C" { }`. Durch `extern "C"` sagen wir dem C++-Compiler, dass er keine zusätzlichen Parameterinformationen dem Namen hinzufügen soll, der zum linken verwendet wird.

Öffnen Sie danach *build.zig* und entfernen Sie alles bis auf den folgenden Code:

chapter01/math/build.zig

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

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

    const lib = b.addSharedLibrary(.{
        // Wir verwenden mymath damit es keine Überschneidungen mit
        // der math.h aus der C-Standardbibliothek gibt!
        .name = "mymath",
        .target = target,
        .optimize = optimize,
    });
    lib.addCSourceFiles(.{
        .files = &.{"src/math.c"},
        .flags = &.{"-std=gnu11"},
    });

    lib.addIncludePath(b.path("src"));
    lib.installHeader(b.path("src/math.h"), "mymath.h");

    lib.linkLibC();
    b.installArtifact(lib);
}
```

Mit `addSharedLibrary()` erzeugen Sie eine neue dynamische Bibliothek. Sollten Sie eine statische Bibliothek benötigen, können Sie diesen Aufruf einfach durch `addStaticLibrary()` austauschen. Die Funktion erwartet ein Argument vom Typ `SharedLibraryOptions` mit dem die Eigenschaften der Bibliothek beeinflusst werden können. Dazu zählt unter anderem der Name der Bibliothek, sowie die Zielarchitektur (`target`) und der Optimierungsgrad (`optimize`).

C-Quelldateien können der Bibliothek `lib` , mit der Methode `addCSourceFiles()` , hinzugefügt werden. Zusätzlich zu den Quelldateien können auch Flags angegeben werden, die beim kompilieren mit berücksichtigt werden sollen. `addCSourceFiles()` kann dabei beliebig oft aufgerufen werden.

```
lib.addCSourceFiles(.{
    .files = &.{"src/math.c"},
    .flags = &.{"-std=gnu11"},
});
```

Zig sucht automatisch nach benötigten Headern im System (zum Beispiel in */usr/include*), jedoch kann es auch nötig sein weitere Pfade anzugeben, in denen für das Projekt benötigte Header liegen. Dies erfolgt durch die `addIncludePath()` Methode, welche auf dem mit `addStaticLibrary()` erzeugten Objekt aufgerufen wird. Der Pfad kann relativ zum Root-Verzeichnis des Projekts angegeben werden.

```
lib.addIncludePath(b.path("src"));
```

Wenn die eigene Bibliothek Header exportieren soll, die für die Verwendung der Bibliothek benötigt werden, so kann dies durch `installHeader()` erfolgen. Die Methode erwartet als erstes Argument einen Pfad zu der zu exportierenden Header-Datei und als zweites den Namen für die Header-Datei, unter welchem diese exportiert werden soll. Im gegebenen Fall exportieren wir die Header Datei unter dem Namen *mymath.h*, damit diese sich nicht mit der *math.h* aus der Standardbibliothek überschneidet.

```
lib.installHeader(b.path("src/math.h"), "mymath.h");
```

Bei der Verwendung von C wird in den meisten Fällen LibC benötigt, die Standardbibliothek für die Programmiersprache C. Diese kann mit `linkLibC()` gelinkt werden.

```
lib.linkLibC();
```

Mit `installArtifact()` kann Zig angewiesen werden die Bibliothek zu bauen.

```
b.installArtifact(lib);
```

Nach dem Ausführen von **`zig build`** sollten Sie die folgenden Dateien in *zig-out* vorfinden:

```
$ ls -R zig-out/
zig-out/:
include  lib

zig-out/include:
math.h

zig-out/lib:
libmath.a
```

Um die Bibliothek systemweit verwenden zu können, müssen die Dateien an die entsprechenden Stellen im Dateisystem kopiert werden. Unter Linux ist dies zum Beispiel */usr/local/include* für *math.h*. Für dieses Beispiel wird dies jedoch nicht benötigt.

Bibliothek einbinden

Um die im vorherigen Abschnitt implementierte Bibliothek zu verwenden, erzeugen wir mit `mkdir user && cd user && zig init` ein neues Projekt, im selben Verzeichnis, in dem auch *math* liegt.

Für diesen Teil des Beispiels verwenden wir C++, damit Sie mir auch glauben wenn ich Ihnen sage, dass Sie Zig sowohl für C als auch C++ Projekte verwenden können. Löschen Sie alle Dateien unter *src* und fügen Sie die Datei *src/main.cpp* hinzu.

chapter01/user/src/main.cpp

```
#include <iostream>
#include "mymath.h"

int main()
{
    std::cout << "4 + 3" << add(4, 3) << "\n";
}
```

Fügen Sie als nächstes unter *build.zig.zon* die *math* Bibliothek als Abhängigkeit hinzu:

chapter01/user/build.zig.zon

```
//...
.dependencies = .{
    .mymath = .{
        .path = "../math",
    },
},
//...
```

Entfernen Sie dann alles bis auf den folgenden Code aus *build.zig*:

chapter01/user.build.zig

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

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

    const math_dep = b.dependency("mymath", .{
        .target = target,
        .optimize = optimize,
    });

    const exe = b.addExecutable(.{
        .name = "user",
        .target = target,
        .optimize = optimize,
    });
    exe.addCSourceFiles(.{
        .files = &.{"src/main.cpp"},
        .flags = &.{},
    });
    exe.linkLibrary(math_dep.artifact("mymath"));
    exe.linkLibCpp();
    b.installArtifact(exe);

    // Erlaubt es mit `zig build run` die Anwendung auszuführen.
    const run_cmd = b.addRunArtifact(exe);
    run_cmd.step.dependOn(b.getInstallStep());
    if (b.args) |args| {
        run_cmd.addArgs(args);
    }
    const run_step = b.step("run", "Run the app");
    run_step.dependOn(&run_cmd.step);
}
```

Zwei Besonderheiten dieses Build-Skripts sind zum einen der Aufruf von `b.dependency()`, sowie `exe.linkLibrary()`.

Mit `b.dependency()` können sie, über den Namen (in diesem Fall `mymath`), auf die in *build.zig.zon* definierten Abhängigkeiten zugreifen. Als zusätzliches Argument erwartet die Methode sowohl die Zielarchitektur als auch den Optimierungsgrad.

```
const math_dep = b.dependency("mymath", .{
    .target = target,
    .optimize = optimize,
});
```

Die `mymath` Abhängigkeit enthält sowohl die dynamische Bibliothek, als auch die zugehörige Header-Datei *mymath.h* (beziehungsweise die Bauanleitung für diese). Mit `exe.linkLibrary()` können wir die Bibliothek mit unserer Executable linken. Hierzu greifen wir mit `math_dep.artifact("mymath")` auf die Bibliothek zu und übergeben diese an `linkLibrary()` [21].

```
exe.linkLibrary(math_dep.artifact("mymath"));
```

Mit **`zig build run`** können Sie die C++-Anwendung bauen und ausführen.

```
$ zig build run
4 + 3 = 7
```

[21] Eigentlich greifen wir mit `artifact()` auf den Compile-Step für `mymath` zu.

Kapitel 2

Grundlagen

Zig ist eine kompilierte Sprache, d.h. sie wird, bevor der Programmcode ausgeführt werden kann, in eine Sprache übersetzt die vom Prozessor verstanden wird. Die Übersetzungsarbeit übernimmt dabei ein Compiler.

Zig verfügt über viel Datentypen, darunter vorzeichenbehaftete und -unbehaftete Ganzzahlen (Integer), Fließkommazahlen (Float), Booleans und Stirngs. Weiterhin besitzt Zig eine Vielzahl an Collection-Typen, darunter Arrays, Tuples.

Zig unterscheidet bei Variablen zwischen Variablen und Konstanten, welche Werte speichern, die über einen Namen referenziert werden. Der Name einer Variable beziehungsweise Konstante wird auch als Identifier bezeichnet. Konstanten sind nach ihrer Initialisierung nicht mehr veränderbar, während Variablen neu zugewiesen werden können. Durch die Unterscheidung zwischen Variablen und Konstanten kann die Absicht hinter einer Variablen-Deklaration eindeutiger ausgedrückt werden.

Zusätzlich zu simplen Typen stellt Zig zusätzliche Collection-Typen, darunter Hash-Maps und Array-Listen, über die Standardbibliothek bereit.

Weiterhin unterstützt Zig optionale Typen, welche die Abwesenheit eines Wertes ausdrücken. Das heißt ein optionaler Typ kann entweder einen Wert besitzen oder keinen. Optionals ersetzen unter anderem NULL-Pointer, wodurch vielen, aus C bekannten Speicherfehlern, vorgebeugt werden kann.

In Zig sind Fehler ebenfalls Werte, das heißt anstatt eine Exception zu werfen können Funktionen einen Fehler-Wert an die Aufrufende Funktion zurückgeben, welche potenzielle Fehler behandeln muss bevor auf den eigentlichen Rückgabewert zugegriffen werden kann.

Konstanten und Variablen

Konstanten und Variablen bestehen aus einem Namen in Snake-Case (`buffer` oder `private_key`) und einem Typen (zum Beispiel `u8` oder `[]const u8`). Sie werden verwendet um Werte vom entsprechenden Typ zu binden (zum Beispiel `13` oder `"Hello, World!"`). Konstanten können nach ihrer Initialisierung nicht mehr neu zugewiesen werden.

Variablen-Deklarationen

Konstanten und Variablen müssen vor ihrer Verwendung deklariert und initialisiert werden. Konstanten werden mit dem `const` Schlüsselwort deklariert, während für Variablen `var` verwendet wird.

```
const es256 = "ES256";
var retries = 3;
retries -= 1;
```

In diesem Beispiel wird ein Konstante mit dem Namen `es256` deklariert und ihr wird der Wert `"ES256"` zugewiesen. Danach wird eine Variable mit dem Namen `retries` deklariert und der Wert `3` zugewiesen. Die Anzahl an Versuchen muss als Variable deklariert werden, da die Anzahl dekrementiert wird.

Sollte eine Variable sich nach ihrer Initialisierung nicht mehr verändern muss diese immer als Konstante deklariert werden! Dies wird vom Compiler sichergestellt.

Konstanten und Variablen müssen bei ihrer Deklarationen auch initialisiert werden. Alternativ kann ihnen auch der Wert `undefined` zugewiesen werden, was so viel bedeutet wie „der Wert der Variable ist zu diesem Zeitpunkt undefiniert“.

```
var later: u8 = undefined;
later = 3;
```

Wichtig zu betonen ist bei der Verwendung von `undefined`, dass Zig den Typ der Variable nicht ableiten kann. Deshalb muss der Typ, bei der Deklaration der Variable, explizit mit angegeben werden.

Typ-Annotationen

Durch Typ-Annotation kann der Typ einer Konstante oder Variable angegeben werden. Hierzu wird hinter dem Variablen-Namen ein „`:`“ angehängt, gefolgt vom Namen des Typen, der verwendet werden soll.

```
const hello: []const u8 = "Hello, World!";
```

Im obigen Beispiel wird eine Konstante mit dem Namen `hello` vom Typ `[]const u8` (String) deklariert.

In vielen Fällen kann Zig den Typ einer Variable ableiten, zum Beispiel durch den verwendeten Initialisierungswert. Es gibt jedoch auch Situationen, bei denen der Typ einer Variable klar ausgedrückt werden muss. Ein solcher Fall betrifft die Verwendung von `undefined`, bei dem der Compiler keinen Typen für die Variable ableiten kann. Es gibt jedoch auch Situationen, bei denen der Compiler den falschen Typen für eine Variable bestimmt.

```
var i = 0;
while (i < 10) : (i += 1) {}
```

Im obigen Beispiel wird der Variable `i` der Typ `comptime_int` zugewiesen, da Integer-Literale ebenfalls vom Typ `comptime_int` sind. Variablen vom Typ `comptime_int` müssen jedoch zur Kompilierzeit bekannt sein, was im gegebenen Fall nicht zutrifft, da `i` zur Laufzeit, innerhalb der Schleife, inkrementiert wird.

Dies führt zu dem folgenden Fehler beim Kompilieren:

```
$ zig build-exe chapter02/integer.zig
error: variable of type 'comptime_int' must be const or comptime
    var i = 0;
        ^
note: to modify this variable at runtime, it must be given an explicit fixed-
size number type
```

Um da Problem zu lösen muss der Variable `i` ein Integer-Typ mit einer bekannten Größe zugewiesen werden. Für Zähler-Variablen ist dies oft `usize`.

```
var i: usize = 0;
while (i < 10) : (i += 1) {}
```

Variablen benennen

Die Namen von Konstanten und Variablen müssen mit einem Buchstaben oder Underscore (`_`) beginnen, gefolgt einer beliebigen Anzahl an Buchstaben oder Ziffern. Dabei ist darauf zu achten, dass der Name nicht mit dem Identifier eines Schlüsselworts überlappt. Zum Beispiel ist es nicht erlaubt eine Konstante `const` zu nennen.

```
const pi = 3.14;
const private_key = "\x01\x02\x03\x04";
```

Es ist Konvention, die Namen von Variablen und Konstanten in Snake-Case zu schreiben, das heißt Wörter werden in Kleinbuchstaben geschrieben, getrennt durch einen Unterstrich (`_`).

Die Namen von Variablen dürfen niemals die Namen von Variablen aus einem umschließenden Scope überschatten, das heißt sie dürfen nicht den selben Namen besitzen.

Sollte ein Name nicht die genannten Bedingungen erfüllen, so kann die `@""` Syntax verwendet werden.

```
const @"π" = 3.14;
```

Die `@""` Syntax kann auch verwendet werden um Schlüsselwörter als Variablen-Namen verwenden zu können. Dies sollte jedoch vermieden werden um Verwirrung vorzubeugen.

Lokale Variablen

Lokale Variablen erscheinen innerhalb von Funktionen, Comptime-Blöcken und `@cImport`-Blöcken.

Einer lokalen Variable kann das `comptime` Schlüsselwort vorangestellt werden. Dadurch ist der Wert der Variable zur Kompilierzeit bekannt und das Laden und Speichern der Variable passiert während der semantischen Analyse des Programms, anstatt zur Laufzeit.

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

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

    x += 1;
    y += 1;

    try std.testing.expect(x == 2);
    try std.testing.expect(y == 2);

    // Da `y` zur Kompilierzeit bekannt ist und die Bedingung ebenfalls
    // zur Kompilierzeit überprüft werden kann, wird der gegebene
    // Block vom Compiler weg-optimiert, wodurch der Compile-Error
    // nicht ausgelöst wird.
    if (y != 2) {
        @compileError("wrong y value");
    }
}
```

Die Life-Time einer lokalen Variable, das heißt der Zeitraum in dem die Variable existiert, beginnt und endet mit dem Block, indem sie deklariert wurde.

Container-Level Variablen

Container-Level Variablen werden außerhalb einer Funktion, Comptime-Blocks oder `@cImport` -Blocks deklariert und sind vergleichbar mit globalen Variablen in anderen Sprachen.

Jedes syntaktische Konstrukt in Zig, welches als Namensraum dient und Variablen- oder Funktionsdeklaraionen umschließt, wird als Container bezeichnet. Weiterhin können Container selbst Typdeklarationen sein, welche instantiiert

werden können. Dazu zählen `struct`s, `enum`s, `union`s und sogar Quellcode-Dateien mit der Dateiendung *.zig*.

Der Initialisierungswert einer container-level Variable ist implizit `comptime`. Ist die deklarierte Variable eine Konstante, so ist ihr Wert zur Kompilierzeit bekannt, andernfalls ihr Wert zur Laufzeit bekannt.

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

var x: i32 = sub(y, 10);
const y: i32 = sub(34, 9);

fn sub(a: i32, b: i32) i32 {
    return a - b;
}

test "Container-Level Variablen" {
    try std.testing.expect(x == 15);
    try std.testing.expect(y == 25);
}
```

Die Life-Time einer container-level Variable ist statisch, das heißt die Variable existiert während der gesamten Laufzeit des Programms.

Statisch-lokale Variablen

Es ist möglich lokale Variablen mit einer statischen Life-Time zu deklarieren, indem ein Container innerhalb einer Funktion verwendet wird.

chapter02/static_local_variable.zig

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

fn next() i32 {
    const S = struct {
        var x: i32 = 0;
    };

    defer S.x += 1;
    return S.x;
}

test "Statische, lokale Variable" {
    try std.testing.expect(next() == 0);
    try std.testing.expect(next() == 1);
    try std.testing.expect(next() == 2);
}
```

Kommentare

Kommentare können genutzt werden um die Funktionsweise von Programmabschnitten zu Dokumentieren. Dabei unterscheidet Zig zwischen drei Arten von Kommentaren.

Normale Kommentare beginnen mit `//` und können an einer beliebigen Stelle im Code platziert werden. Alles was auf `//` innerhalb einer Zeile folgt ist Teil des Kommentars.

```
// Das ist ein Kommentar
```

Doc-Kommentare können für die Dokumentation einzelner Programmteile genutzt werden und beginnen mit `///`. Mehrere, hintereinander folgende Doc-Kommentare bilden einen zusammenhängenden Block und erlauben es Kommentare über mehrere Zeilen hinweg zu verfassen. Doc-Kommentare sind kontextabhängig und dokumentieren was auch immer dem Kommentar folgt.

chapter02/docs.zig

```
//! Ein Modul bestehend aus einem Struct `Color` und
//! einer Funktion `add(u32, u32) u32`.

const std = @import("std");

/// Eine Farbe bestehend aus Red, Green und Blue.
pub const Color = struct {
    r: u8,
    g: u8,
    b: u8,
};

/// Addition zweier Zahlen.
///
/// # Argumente
/// * `a`- Die erste Zahl
/// * `b`- Die zweite Zahl
///
/// # Rückgabewert
/// Das Resultat von `a + b`.
pub fn add(a: u32, b: u32) u32 {
    return a + b;
}

test "Main Test" {
    _ = Color;
    try std.testing.expect(add(3, 4) == 7);
}
```

Top-Level-Kommentare beginnen mit `//!` und dokumentieren den umschließenden Container. Sie werden in der Regel genutzt um Module zu dokumentieren.

Mit `zig test -femit-docs <your-code>.zig` können die Doc- und Top-Level-Kommentare in eine HTML-Seite umgewandelt werden. Zig wird hierfür einen neuen Ordner mit dem Namen *docs* anlegen. Mit `python3 -m http.server` kann ein HTTP-Server gestartet werden um die Dokumentation anzuzeigen.

Zurzeit scheint es jedoch noch Probleme mit dem Erzeugen zu geben.

Ganzzahlen (Integer)

Integer sind Ganzzahlen, das heißt sie Besitzt keine Bruchkomponente und können entweder vorzeichenbehaftet (*signed*) oder vorzeichenunbehaftet (*unsigned*) sein.

Zig unterstützt Ganzzahlen mit einer beliebigen Bitbreite. Der Bezeichner eines jeden Integer-Typen beginnt mit einem Buchstaben `i` (signed) oder `u` (unsigned) gefolgt von einer oder mehreren Ziffern, welche die Bitbreite in Dezimal darstellen. Als Beispiel, `i7` ist eine vorzeichenbehaftete Ganzzahl der sieben Bit zur Kodierung der Zahl zur Verfügung stehen. Die Aussage, dass die Bitbreite beliebig ist entspricht dabei nicht ganz der Wahrheit. Die maximal erlaubte Bitbreite beträgt $2^{16} - 1 = 65535$. Beispiele für Integer sind:

Typ	Wertebereich
`i7`	-2^6 bis $2^6 - 1$
`i32`	-2^{31} bis $2^{31} - 1$
`u8`	0 bis $2^8 - 1$
`u64`	0 bis $2^{64} - 1$

Darstellung von Integern im Speicher

Vorzeichenbehaftete Ganzzahlen werden im Zweierkomplement dargestellt[22]. In Assembler wird nicht zwischen vorzeichenbehafteten und vorzeichenunbehafteten Zahlen unterschieden. Alle mathematischen Operationen werden von der CPU auf Registern, mit einer festen Bitbreite (meist 64 Bit auf modernen Computern), ausgeführt. Dabei entspricht jede, vom Computer ausgeführte, arithmetische Operationen effektiv einem „Rechnen mit Rest“, auch bekannt als modulare Arithmetik[23]. Die Bitbreite m der Register (z.B. 64) repräsentiert dabei den Modulo 2^m. Damit entspricht ein 64 Bit Register dem Restklassenring $\mathbb{Z}_{2^{64}} = \{0, 1, 2, ..., 2^{64} - 1\}$ und jegliche Addition zweier Register resultiert in einem Wert der ebenfalls in $\mathbb{Z}_{2^{64}}$ liegt, d.h. auf *x86_64*

[22] https://en.wikipedia.org/wiki/Two's_complement
[23] https://de.wikipedia.org/wiki/Modulare_Arithmetik

wäre die Instruktion `add rax, rbx` äquivalent zu $\text{rax} = \text{rax} + \text{rbx} \bmod 2^{64}$. Diese Verhalten überträgt sich analog auf Ganzzahlen in Zig.

Das Zweierkomplement einer Zahl $a \in \mathbb{Z}_m$ ist das additive Inverse a' dieser Zahl, d.h. $a + a' \equiv 0$. Dieses kann mit $a' = m - a$ berechnet werden. Für `i8` wäre das additive Inverse zu $a = 4$ die Zahl $a' = 2^8 - 4 = 256 - 4 = 252$. Addiert man beide Zahlen modulo 256, so erhält man wiederum das neutrale Element 0, $a + a' \bmod 256 = 4 + 252 \bmod 256 = 256 \bmod 256 = 0$. Das Zweierkomplement hat seinen Namen jedoch nicht von der Subtraktion, sondern von der speziellen Weise wie das additive Inverse einer Zahl bestimmt wird. Dieser Vorgang kann wie folgt beschrieben werden:

1. Gegeben eine Zahl in Binärdarstellung, invertiere jedes Bit, d.h. jede 1 wird zu einer 0 und umgekehrt.
2. Addiere 1 auf das Resultat und ignoriere mögliche Überläufe.

Für das obige Beispiel mit der Zahl 4 vom Typ `i8` sieht dies wie folgt aus:

$$\begin{aligned} 00000100_2 &= 4_{16} && \text{invertiere alle Bits der Zahl 4} \\ 11111011_2 &= 251_{16} && \text{addiere 1 auf die Zahl 251} \\ 11111100_2 &= 252_{16} \end{aligned}$$

Integer-Literale

Zur Compile-Zeit bekannte Literale vom Typ `comptime_int` haben kein Limit was ihre Größe (in Bezug auf die Bitbreite) und konvertieren zu anderen Integertypen, solange das Literal im Wertebereich des Typen liegt.

```
// Variable `i` vom Typ `comptime_int`
var i = 0;
```

Optional können die Prefixe `0x`, `0o` und `0b` an ein Literal angehängt werden um Literale in Hexadezimal, Octal oder Binär anzugeben, z.B. `0xcafebabe`.

Um größere Zahlen besser lesbar zu machen, kann ein Literal mit Hilfe von Unterstrichen aufgeteilt werden, z.B. `0xcafe_babe`.

Laufzeit-Variablen

Um die Variable zur Laufzeit modifizieren zu können, muss ihr eine expliziter Type mit fester Bitbreite zugewiesen werden. Dies kann auf zwei weisen erfolgen.

1. Deklaration der Variable `i` mit explizitem Typ, z.B. `var i: usize = 0`.
2. Verwendung der Funktion `@as()`, von welcher der Compiler den Type der Variable `i` ableiten kann, z.B. `var i = @as(usize, 0)`.

Ein häufiger Fehler, der aber schnell behoben ist, ist die Verwendung einer Variable vom Typ `comptime_int` in einer Schleife.

```
var i = 0;
while (i < 100) : (i += 1) {}
```

Was zu einem entsprechenden Fehler zur Kompilierzeit führt.

```
$ zig build-exe chapter02/integer.zig
error: variable of type 'comptime_int' must be const or comptime
    var i = 0;
        ^
note: to modify this variable at runtime, it must be given an explicit fixed-
size number type
```

Der Zig-Compiler ist dabei hilfreich, indem er neben dem Fehler auch einen Lösungsansatz bietet. Nachdem der Variable `i` ein expliziter Typ zugewiesen wird (`var i: usize`) compiliert das Programm ohne weitere Fehler.

Integer-Operatoren

Zig unterstützt verschiedene Operator für das Rechnen mit Integern, darunter `+` (Addition), `-` (Subtraktion), `*` (Multiplikation) und `/` (Division).

Die Verwendung dieser Operatoren führt bei einem Überlauf jedoch zu undefiniertem Verhalten (engl. undefined behavior). Aus diesem Grund stellt Zig spezielle Versionen dieser Operatoren zur Verfügung, darunter:

- Operatoren für Sättigungsarithmetik: Alle Operationen laufen in einem festen Intervall zwischen einem Minimum und einem Maximum ab welches nicht unter- bzw. überschritten werden kann.
  - Addition (`+|`): `@as(u8, 255) +| 1 == @as(u8, 255)`
  - Subtraktion (`-|`): `@as(u32, 0) -| 1 == 0`
  - Multiplikation (`*|`): `@as(u8, 200) *| 2 == 255`
- Wrapping-Arithmetik: Dies ist äquivalent zu modularer Arithmetik.
  - Addition (`+%`): `@as(u32, 0xffffffff) +% 1 == 0`
  - Subtraktion (`-%`): `@as(u8, 0) -% 1 == 255`
  - Multiplikation (`*%`): `@as(u8, 200) *% 2 == 144`

Integer-Bounds

Auf den Minimal- und Maximalwert eines Integers kann mit `std.math.minInt` und `std.math.maxInt` zugegriffen werden.

```
try testing.expect(minInt(i128) == -170141183460469231731687303715884105728);
try testing.expect(maxInt(i128) == 170141183460469231731687303715884105727);
```

Beide Funktionen erwarten als Argument den Integer-Typ, für den das Minimum oder Maximum bestimmt werden soll. Da beide Funktionen `comptime` sind wird der Rückgabewert zur Kompilierzeit bestimmt.

Fließkommazahlen (Float)

Fließkommazahlen haben eine Bruchkomponente, wie etwa `3.14` oder `-0.5`.

Im Gegensatz zu Integern erlaubt Zig keine beliebige Bitbreite für Fließkommazahlen. Zur Verfügung stehen:

Typ	Repräsentation
`f16`	IEEE-754-2008 binary16
`f32`	IEEE-754-2008 binary32
`f64`	IEEE-754-2008 binary64
`f80`	IEEE-754-2008 80-bit extended precision
`f128`	IEEE-754-2008 binary128

Der Typ `f32` entspricht dem Typ `float` (single precision) in C, während `f64` dem Typ `double` (double precision) entspricht. Je nach Prozessortyp stehen dedizierte Maschineninstruktionen für zumindest einen Teil der Typen zur Verfügung, was eine effizientere Verwendung ermöglicht. Auf *x86_64* Prozessor stehen z.B. Instruktionen für single und double Precision zur Verfügung.

Float-Literale

Literale sind immer vom Typ `comptime_float`, welcher äquivalent zum größtmöglichen Fließkommatypen (`f128`) ist, und können zu jedem beliebigen Fließkommatypen konvertiert werden. Enthält ein Literal keinen Bruchteil, so ist eine Konvertierung zu einem Integertyp ebenfalls möglich.

Alle Float-Literale haben einen Dezimalpunkt (`.`). Sie können entweder als Dezimalzahl angegeben werden (ohne Präfix) oder als Hexadezimalzahl (mit dem Präfix `0x`). Optional kann ein Exponent mit angegeben werden. Für Dezimalzahlen wird hierfür ein `E` oder `e` verwendet und für Hexadezimalzahlen ein `P` oder `p`.

Für Dezimalzahlen mit einem Exponenten `e` wird die angegebene Fließkommazahl mit 10^e multipliziert:

- `123.0E+77` = $123.0 * 10^{77}$

Für Hexadezimalzahlen mit einem Exponenten `p` wird die Fließkommazahl mit 2^p multipliziert:

- `0x103.70p-5` = $103.70_{16} * 2^{-5}$

```
const fp = 123.0E+77;
const hfp = 0x103.70p-5;
```

Darstellung von Floats im Speicher

Die interne Darstellung einer Fließkommazahl besteht für das Format *IEEE-754* aus einem Vorzeichenbit, gefolgt von einem Exponenten und einem Bruch. Wie viele Bits jeweils für Exponent und Bruch zur Verfügung stehen ist abhängig von der Bitbreite der Fließkommazahl. Für *IEEE-754 binary32* sieht dies wie folgt aus:

31	30	...	23	22	...	0
s	exponent (e)			fraction (f)		

Diese Darstellung entspricht der Gleichung $(-1)^s * 1.f * 2^{e-127}$. Der Bruch f entspricht einer normalisierten, binär kodierten Fließkommazahl, d.h. die Zahl wird um eine entsprechende Anzahl an Stelle verschoben, sodass genau eine führende Eins vor dem Komma steht. Als Beispiel entspricht die Fließkommazahl 3.25 in binär der Zahl 11.01 oder anders ausgedrückt $11.01 * 2^0$. Um die Zahl zu normalisieren wird diese nun um eine Stelle nach rechts verschoben $1.101 * 2^1$. Die Zahl nach der führenden Eins (101) entspricht f und der Exponent e ist die Summe des Exponenten der normalisierten Darstellung und einem Bias (im Fall von `f32` ist dieser 127), d.h. $e = 1 + 127 = 128_{16} = 10000000_2$. Damit wird 3.25 wie folgt kodiert:

31	30	...	23	22	...	0
0	10000000_2			$10100000000000000000000_2$		

Aufgrund der Darstellung von Fließkommazahlen kann sich die Ausführung bestimmter Operationen, wie ein Tests auf Gleichheit (`==`), als trickreich herausstellen. Ein Beispiel ist die wiederholte Addition der Fließkommazahl 0.1. Die Summe $\sum_{k=1}^{10} 0.1$ ist erwartungsgemäß 1.0, je nach Präzision der Fließkommazahl gilt jedoch $\sum_{k=1}^{10} 0.1 \neq 1.0$.

Konvertierung von numerischen Typen

Von Zeit zu Zeit kann es nötig sein einen numerischen Typen in einen anderen zu konvertieren. Hierfür stehen die eingebauten Funktionen `@intCast()` und `@floatCast()` zur Verfügung.

Integer Konvertierung

Die Funktion `@intCast(anytype)` konvertiert einen Integer zu einem anderen Integer, wobei der numerische Wert beibehalten wird. Der Typ des Rückgabewertes wird dabei vom Compiler abgeleitet. Hierzu muss `@as()` in Kombination mit `@intCast()` verwendet werden. Alternativ kann einer Variable auch explizit ein Typ zugewiesen werden, an den der konvertierte Wert gebunden werden soll.

chapter02/conversion.zig

```zig
test "Konvertierungs-Test: pass" {
    var a: u16 = 0x00ff; // runtime-known
    _ = &a;
    const b: u8 = @intCast(a);
    _ = b;
    const c = @as(u8, @intCast(a));
    _ = c;
}
```

Grundsätzlich kann zwischen einer Narrowing- und Widening-Konvertierung unterschieden werden. Bei ersterer ist der Ziel-Typ kleiner als der ursprüngliche Typ. Hierdurch kann es passieren das der numerische Wert „out-of-range“ ist, das heißt der Ziel-Typ hat nicht genug Bits um den Wert im Speicher darzustellen. Ein solcher Fall führt zu undefiniertem Verhalten, wobei Zig, je nach Optimierung, das „Abschneiden“ von Bits erkennt und den Prozess vorzeitig beendet.

chapter02/conversion.zig

```zig
test "Konvertierungs-Test: fail" {
    var a: u16 = 0x00ff; // runtime-known
    _ = &a;
    const b: u7 = @intCast(a);
    _ = b;
}
```

```
$ zig test conversion.zig
thread 363318 panic: integer cast truncated bits
zig-book/code/chapter02/conversion.zig:13:19:          0x103cdad          in
test.Konvertierungs-Test: fail (test)
    const b: u7 = @intCast(a);
                  ^
zig-linux-x86_64-0.13.0/lib/compiler/test_runner.zig:157:25:   0x1048099   in
mainTerminal (test)
        if (test_fn.func()) |_| {
                            ^
zig-linux-x86_64-0.13.0/lib/compiler/test_runner.zig:37:28: 0x103e11b in main
(test)
        return mainTerminal();
                              ^
zig-linux-x86_64-0.13.0/lib/std/start.zig:514:22:        0x103d259        in
posixCallMainAndExit (test)
            root.main();
                     ^
zig-linux-x86_64-0.13.0/lib/std/start.zig:266:5: 0x103cdc1 in _start (test)
    asm volatile (switch (native_arch) {
```

```
    ^
???:?:?: 0x0 in ??? (???)
```

Sollte es erlaubt sein einen Wert, bei einer Narrowing-Konvertierung, abzuschneiden, so kann entweder `@truncate` oder alternativ der Und-Operator (`&`) in Kombination mit einer Bit-Maske verwendet werden.

Widening-Konvertierungen wiederum sind unkritisch und damit immer erfolgreich.

Float Konvertierung

Die Funktion `@floatCast(anytype)` konvertiert einen Float zu einem anderen Float, wobei der numerische Wert an Präzision verlieren kann. Die Konvertierung von Floats ist dabei sicher. Der Typ des Rückgabewerts wird wie bei der Konvertierung von Integern abgeleitet.

chapter02/conversion.zig

```
test "Float Konvertierung" {
    var a: f32 = 1234567.0; // runtime-known
    _ = &a;
    const b: f16 = @floatCast(a);
    _ = b;
}
```

Typen Alias

Alle primitiven Typen in Zig haben `type` als ihren Meta-Typ und können selbst an Konstanten gebunden werden. Damit erlaubt Zig die Definition eines Alias für einen bestehenden Typen.

Ein Alias ist nützlich, um einen Typen bei einem Namen zu referenzieren. Beispielsweise werden Universally Unique Identifier (UUID)[24] als 128-Bit-Zahl kodiert.

https://github.com/r4gus/uuid-zig/blob/master/src/core.zig

```
/// Universally Unique IDentifier
///
/// A UUID is 128 bits long, and can guarantee uniqueness across space and
time (RFC4122).
pub const Uuid = u128;
```

Nachdem ein Alias definiert wurde kann dieser überall verwendet werden, wo auch der ursprüngliche Bezeichner des Typen verwendet werden kann.

https://github.com/r4gus/uuid-zig/blob/master/src/v4.zig

[24] https://en.wikipedia.org/wiki/Universally_unique_identifier

```
/// Create a version 4 UUID using a user provided RNG
pub fn new2(r: std.rand.Random) Uuid {
    // Set all bits to pseudo-randomly chosen values.
    var uuid: Uuid = r.int(Uuid);
    // Set the two most significant bits of the
    // clock_seq_hi_and_reserved to zero and one.
    // Set the four most significant bits of the
    // time_hi_and_version field to the 4-bit version number.
    uuid &= 0xffffffffffffff3fff0fffffffffffff;
    uuid |= 0x00000000000000800040000000000000;
    return uuid;
}
```

Booleans

Zig besitzt einen primitiven Boolean Typ `bool`. Ein Boolean ist ein Daten-Typ der zwei mögliche Werte annehmen kann `true` oder `false`. Er ist nach Georg Boole[25] benannt, der die Boolsche-Algebra definierte.

Booleans werden primär in Conditional-Satements und -Expressions (Kontrollstrukturen) verwendet[26], zum Beispiel in Kombination mit If-Then-Else Blöcken, um zu bestimmen, welcher Block ausgeführt werden soll.

```
const name = "Sesam, öffne dich!";

if (std.mem.eql(u8, name, "Sesam, öffne dich!") {
    std.log.info("Ruhm und Reichtum!", .{});
} else {
    std.log.info("...", .{});
}
```

Die Funktion `std.mem.eql` überprüft ob die zwei gegebenen Strings, bezogen auf ihren Inhalt, gleich sind. Falls ja wird der Wert `true` zurückgegeben, andernfalls der Wert `false`.

Im Gegensatz zu C verhindert Zig, dass numerische Werte als Booleans verwendet werden[27].

defer

Mit dem `defer` Schlüsselwort können Ausdrücke und Blöcke markiert werden, die beim Verlassen eines Blocks ausgeführt werden sollen. Solche `defer`-Ausdrücke und -Blöcke werden in der umgekehrten Reihenfolge ausgeführt, in welcher sie definiert wurden.

chapter02/defer.zig

[25] https://en.wikipedia.org/wiki/George_Boole
[26] https://en.wikipedia.org/wiki/Conditional_(computer_programming)
[27] In C ist der Wert `0` äquivalent zu `False` und alle verbleibenden Werte äquivalent zu `True`.

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

fn myDefer() void {
    defer {
        print("Wird als zweites ausgeführt\n", .{});
    }

    defer print("Wird als erstes ausgeführt\n", .{});

    if (false) {
        defer print("Wird nie ausgeführt\n", .{});
    }
}

test "defer test #1" {
    myDefer();
}
```

`defer`s werden dabei nur ausgeführt, wenn Sie beim Ausführen eines Blocks auch erreicht wurden. Im obigen Beispiel kommt zuerst ein `defer`-Block vor, gefolgt von einem `defer`-Ausdruck. Da `defer`s in umgekehrter Reihenfolge ausgeführt werden, wird beim Verlassen der Funktion zuerst „*Wird als erstes ausgeführt*“ auf der Kommandozeile ausgegeben, gefolgt von „*Wird als zweites ausgeführt*“. „*Wird nie ausgeführt*“ wird nicht ausgegeben, da die If-Bedingung immer `false` ist und somit der If-Block nie ausgeführt wird.

`defer`s eignen sich besonders gut zum aufräumen von Ressourcen. Ein Beispiel hierfür ist die Deallokation von dynamisch alloziertem Speicher. Es ist gängige Praxis, dass auf die dynamische Allokation von Speicher ein `defer` folgt, welches den Speicher wieder frei gibt, sollte dieser nach dem Verlassen des umschließenden Blocks nicht mehr benötigt werden.

```
var mem = try std.heap.c_allocator.alloc(T: u8, n: 16);
defer std.heap.c_allocator.free(mem);
// do something...
```

Optionals

In Situationen bei denen eine Wert fehlen kann, können Optionals verwendet werden. Ein Optional repräsentiert zwei mögliche Zustände: Entweder es ist ein Wert vorhanden oder es ist kein Wert vorhanden. Dies ist nicht zu verwechseln mit undefinierten Werten, die bei der Verwendung von `undefined` vorkommen!

Als ein Beispiel stellt die Zig Standardbibliothek die Funktion `std.math.cast` zur Verfügung. Diese erlaubt das Konvertieren eines Integer in einen anderen Integer-Typen. Falls der gegebene Wert nicht in den neuen Integer-Typen passt, so wird von der Funktion `null` zurückgegeben.

```
const std = @import("std");
const cast = std.math.cast;

test "Integer Konvertierung" {
    try std.testing.expect(cast(u8, @as(u32, 300)) == null);
    try std.testing.expect(cast(u8, @as(u32, 255)).? == @as(u8, 255));
}
```

Ein optionaler Typ besteht aus einem beliebigen Typen, dem ein `?` vorangestellt wird, zum Beispiel `?u32` oder `?[]const u8`.

null

Um einer optionalen Variable einen wertlosen Zustand zuzuweisen, wird der Wert `null` verwendet.

```
var optional: ?u32 = 7;
optional = null;
```

Sollte eine optionale Variable einen Wert besitzen, so wird dieser als „ungleich zu `null`“ betrachtet. Dies kann mit dem Gleicheits- (`==`) beziehungsweise Ungelichheits-Operator (`!=`) abgefragt werden.

chatper02/optionals.zig

```
test "Optional" {
    const num: ?u8 = std.math.cast(u8, @as(u32, 255));

    if (num != null) {
        try std.testing.expect(num.? == 255);
    } else {
        try std.testing.expect(1 == 0); // fail
    }
}
```

Mit dem `?`-Operator kann auf den Wert eines Optionals zugegriffen werden. Es sollte jedoch sichergestellt werden, dass ein Wert existiert!

Das obige Beispiel kann auch wie folgt geschrieben werden:

chatper02/optionals.zig

```
test "Optional #2" {
    const num: ?u8 = std.math.cast(u8, @as(u32, 255));

    if (num) |n| {
```

```
        try std.testing.expect(n == 255);
    } else {
        try std.testing.expect(1 == 0); // fail
    }
}
```

Optionale Variablen können als Bedingung, innerhalb einse If-Statements, verwendet werden. Sollte `num` einen Wert besitzen, so wird dieser an `n` gebunden und der If-Block wird betreten. Andernfalls wird der Else-Block ausgeführt.

Variablen die nicht als „optional“ deklariert wurden enthalten garantiert immer einen Wert! Dies vereinfacht es, Fälle bei denen ein Wert fehlen kann, wie etwa der Zeiger bei einer verketteten Liste, zu handhaben. Optionals erzwingen es, explizit auf den Wert einer optionalen Variable zuzugreifen. Entweder durch Verwendung des `?`-Operators oder eines If-Statements. Damit wird verhindert, dass ein optionaler Wert aus Versehen als nicht optionaler Wert gehandhabt wird.

Je nach Situation kann wie folgt mit fehlenden Werten umgegangen werden:

- Überspringe den Code der auf den eigentlichen Wert angewandt werden würde.
- Bereitstellen eines Fallback-Werts.
- Propagiere den `null` Wert an die darüber liegende Funktion oder beende den Prozess vorzeitig.

chatper02/optionals.zig

```
test "Handling" {
    var num: ?u8 = std.math.cast(u8, 250);

    // Überspringe Block falls `num == null`
    if (num) |*n| {
        n.* += 1;
    }
    try std.testing.expect(num.? == 251);

    // Stelle einen Fallback-Wert bereit
    const num2: u8 = if (std.math.cast(u8, 256)) |n| n else 255;
    try std.testing.expect(num2 == 255);
}
```

Neben If-Statements können Optionals auch in While-Schleifen verwendet werden. Sollte das verwendete Optional `null` sein so wird aus der Schleife ausgebrochen, andernfalls wird eine Iteration der schleife Durchlaufen.

chatper02/optionals.zig

```
test "while" {
    const S = struct {
        pub fn next() ?u2 {
            const T = struct {
                var v: ?u2 = 0;
            };

            defer {
                if (T.v) |*v| {
                    if (v.* == 3) T.v = null else v.* += 1;
                }
            }

            return T.v;
        }
    };

    const stdout = std.io.getStdOut();

    while (S.next()) |value| {
        try stdout.writer().print("{d}\n", .{value});
    }
}
```

In diesem Beispiel wird eine Funktion `next()` definiert, die eine statische, lokale Variable `v` besitzt. Diese wird mit `0` initialisiert. Bei jedem Aufruf von `next()` wird der aktuelle Wert von `v` zurückgegeben. Der `defer` Block wird vor der Rückkehr aus der Funktion ausgeführt und inkrementiert `v`, jedoch nur falls `v` nicht gleich drei ist. Sollte `v` gleich drei sein, so wird `v` der `null`-Wert zugewiesen.

Verwendet man den Rückgabewert von `next()` als Bedingung einer While-Schleife so wird der Rückgabewert an `value` gebunden, solange dieser nicht gleich `null` ist, das heißt `value` hat den Typ `u2`.

Führt man den Test aus, so sieht man, dass die Zahlen 0 bis 3 auf der Kommandozeile ausgegeben werden, bevor aus der Schleife ausgebrochen wird.

```
$ zig test optionals.zig
0
1
2
3
All 4 tests passed.
```

Zeiger (Pointer)

Zig unterscheidet zwischen zwei Arten von Zeigern, *single-item* und *many-item* Pointer.

Ein single-item Pointer `*T` zeigt auf exakt einen Wert im Speicher und kann mit der Syntax `ptr.*` dereferenziert werden. Mit Hilfe des Address-of-Operators `&` kann ein single-item Pointer bezogen werden.

```
// Definiere eine Variable vom Typ u8
var v: u8 = 128;
// Beziehe einen Zeiger auf `v`
const v_ptr: *u8 = &v;
// Dereferenziere den Zeiger `v_ptr` und addiere 1 zu `v`
v.* += 1;
```

Ein multi-item Pointer `[*]T` zeigt auf eine lineare Sequenz an Werten im Speicher mit unbekannter Länge. Der Zeiger eines Slice (`.ptr`) ist ein multi-item Pointer. Allgemein teilen Slices und multi-item Pointer die selbe Index- und Slice-Syntax.

- `ptr[i]`
- `ptr[start..end]`
- `ptr[start..]`

Genau wie C erlaubt auch Zig Zeigerarithmetik auf multi-item Pointer.

chapter02/pointer.zig

```
var array = [_]i32{ 1, 2, 3, 4 };

var array_ptr = array[0..].ptr;

std.log.info("{d}", .{array_ptr[0]});
array_ptr += 1;
std.log.info("{d}", .{array_ptr[0]});
```

Nach dem Compilieren mit **`zig build-exe chapter02/pointer.zig`** können wir die Beispiel Anwendung ausführen und sehen, dass die ersten beiden Zahlen von `array` ausgegeben werden, obwohl wir den selben Index für `array_ptr` verwenden. Grund dafür ist, dass wir den Zeiger selbst, zwischen dem ersten und zweiten Aufruf von `std.log.info()`, inkrementiert haben.

```
$ ./pointer
info: 1
info: 2
```

Ein weit verbreitetes Konzept in C sind `NULL`-terminierte Strings, d.h. ein `0`-Byte wird hinter den letzten Character eines Strings geschrieben und markiert so dessen Ende. Zig bietet etwas sehr ähnliches, nämlich sentinel-terminated Pointer, auf die im nächsten Abschnitt noch näher eingegangen wird.

Arrays und Slices

Zig besitzt eine Vielzahl an Datentypen um eine (lineare) Sequenz an Werten im Speicher darzustellen, darunter:

- Der Typ `[N]T` repräsentiert ein Array vom Typ `T` bestehend aus `N` Werten. Die Größe eines Arrays ist zur Compilezeit bekannt und Arrays werden grundsätzlich auf dem Stack alloziert. Damit kann ein Array weder erweitert noch verkleinert werden.
- Der Typ `[]T` bzw. `[]const T` repräsentiert ein Slice vom Typ `T`, bestehend aus einem Zeiger und einer Länge. Die Länge eines Slices ist zur Laufzeit bekannt. Slices referenzieren eine Sequenz von Werten. Dies kann z.B. ein Array sein oder auch eine auf dem Heap gespeicherte Sequenz. Die von einem konstanten Slice `[]const T` referenzierten Werte können gelesen, jedoch nicht verändert werden, während die Werte eines Slices `[]T` sowohl gelesen als auch verändert werden können.

Sowohl Arrays als auch Slices erlauben den Zugriff auf deren Länge durch den Ausdruck `.len`.

chapter02/slices.zig

```
var a = [_]u8{ 1, 2, 3, 4 };
std.log.info("length of a is {d}", .{a.len});
const s = &a;
std.log.info("length of a is still {d}", .{s.len});
```

Mit dem Address-Of Operator `&` kann ein Slice für ein Array erzeugt werden. Alternativ kann auch der Ausdruck `a[0..]` verwendet werden, der einen Bereich innerhalb des Arrays beschreibt. Grundsätzlich liegt das erste Element einer Sequenz immer an Index 0 und es kann mit `a[0]` auf dieses zugegriffen werden. Das letzte Element liegt immer an der Stelle `a.len - 1` und es kann mit `a[a.len - 1]` darauf zugegriffen werden. Der Index muss dabei immer ein Integer vom Typ `usize` oder ein Literal sein, das zu diesem Typ konvertiert werden kann. Die Verwendung anderer Typen als Index führt zu einem Fehler zur Compilezeit.

Auf den Zeiger eines Slices kann mit `.ptr` zugegriffen werden, z.B. `s.ptr`.

Zig überprüft bei dem Zugriff auf eine Array oder Slice zur Laufzeit, dass der Index innerhalb des Speicherbereichs der Sequenz liegt. Ließt eine Anwendung über die Grenzen der Sequenz, so führt dies zu einem Fehler zur Laufzeit der den Prozess beendet. Dies verhindert typische Speicherfehler wie Buffer-Overflows and Buffer-Overreads die in Sprachen wie C weit verbreitet sind und in der Vergangenheit zu Hauf von Angreifern ausgenutzt wurden um Anwendungen zu exploiten.

chapter02/slices.zig

```
var i: usize = 0;
while (true) : (i += 1) {
```

```
    a[i] += 1;
}
```

```
$ zig build-exe slices.zig -Doptimize=ReleaseFast
$ ./slices
info: length of a is 4
info: length of a is still 4
thread 1232 panic: index out of bounds: index 4, len 4
slices.zig:14:10: 0x103544c in main (slices)
        a[i] += 1;
         ^
start.zig:514:22: 0x1034c99 in posixCallMainAndExit (slices)
            root.main();
                     ^
start.zig:266:5: 0x1034801 in _start (slices)
    asm volatile (switch (native_arch) {
    ^
???:?:?: 0x0 in ??? (???)
Aborted (core dumped)
```

Arrays

Es gibt eine Vielzahl von Möglichkeiten um Arrays in Zig zu definieren. Die einfachste Möglichkeit ist, eine Sequenz von Werten in geschweiften Klammern anzugeben.

```
const prime: [5]u8 = .{2, 3, 5, 7, 11};
const names = [3][]const u8{"David", "Franziska", "Sarah"};
```

Für den Fall, dass initial keine Werte bekannt sind kann ein Array mit `undefined` initialisiert werden. In diesem Fall ist der Inhalt des Speichers undefiniert.

```
const some: [1000]u8 = undefined;
```

Arrays können aber auch mit einem bestimmten Wert initialisiert werden. Im unteren Beispiel wird das gesamte Array mit `0` Werten initialisiert.

```
const some: [1000]u8 = .{0} ** 1000;
```

Die Länge eines Arrays muss immer zur Compilezeit bekannt sein. Dementsprechend können keine Variablen zur Angabe der Länge verwendet werden, außer die Variable ist vom Typ `comptime_int`. Sollte ein Array benötigt werden, dessen Länge nur zur Laufzeit bekannt ist, so

muss der Speicher entweder manuell alloziert oder auf einen Kontainertypen wie `ArrayList` aus der Standardbibliothek zurückgegriffen werden[28].

Viel Funktionen die über Sequenzen arbeiten erwarten ein Slice und kein Array. Zig konvertiert dabei nicht automatisch Arrays zu Slices, d.h. bei einem Aufruf muss explizit der Address-Of Operator `&` auf das Array angewandt werden oder alternativ ein Slice mit dem `[]` Operator festgelegt werden.

chapter02/coersion.zig

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

pub fn main() void {
    const a: [5]u8 = .{ 1, 2, 3, 4, 5 };

    foo(&a);
    foo(a[1..]);
}

fn foo(s: []const u8) void {
    for (s) |e| {
        std.log.info("{d}", .{e});
    }
}
```

```
$ ./coersion
info: 1
info: 2
info: 3
info: 4
info: 5
info: 2
info: 3
info: 4
info: 5
```

Slices

Slices `[]T` werden ohne Angabe einer Länge geschrieben und repräsentieren eine lineare Sequenz an Werten. Konzeptionell ist ein Slice eine Zeiger vom Typ `std.builtin.Type.Pointer`. Schaut man sich die Definition von `Slice` in *zig/src/mutable_value.zig*[29] an, so sieht man, dass ein Slice durch einen Zeiger (`ptr`), der auf den Beginn des referenzierten Speicherbereichs zeigt, sowie eine Länge (`len`) beschrieben wird.

github.com/ziglang/zig/src/mutable_value.zig

[28] Mehr dazu in folgenden Kapiteln.

[29] https://github.com/ziglang/zig/blob/624fa8523a2c4158ddc9fce231181a9e8583a633/src/mutable_value.zig

```
pub const Slice = struct {
    ty: InternPool.Index, // wir ignorieren dieses Feld :)
    /// Must have the appropriate many-ptr type.
    ptr: *MutableValue,
    /// Must be of type `usize`.
    len: *MutableValue,
};
```

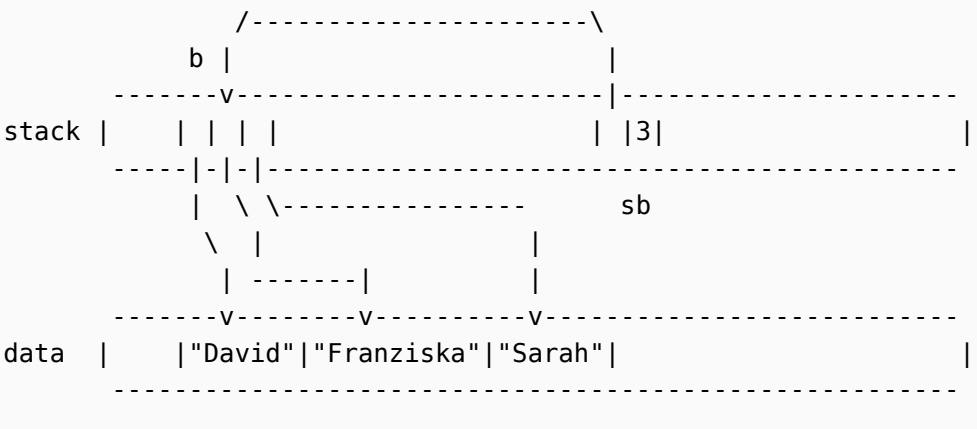
Je nach Typ einer Variable bzw. eines Parameters konvertiert Zig die Referenz zu einem Struct automatisch in ein Slice.

```
const b: [3][]const u8 = .{ "David", "Franziska", "Sarah" };

// Zig konvertiert die Referenz automatisch zu einem Slice.
const sb: []const []const u8 = &b;
_ = sb;

// `rb` ist ein Pointer zu einem Array.
const rb: *const [3][]const u8 = &b;
_ = rb;
```

Da `b` eine Konstante ist, muss auch das Slice `sb` (`[]const T`), sowie der Pointer `rb` auf das Array (`*const [N]T`) konstant sein. Wäre `b` eine Variable, so wäre auch das `const`, in Bezug auf das Slice bzw. den Pointer, optional, je nachdem ob das Array durch die jeweilige Referenz verändert werden soll oder nicht.



Mithilfe des `[]` Operators können Slices für einen bestehenden Speicherbereich angegeben werden. Innerhalb der eckigen Klammern muss dafür ein Bereich spezifiziert werden, der durch das Slice eingegrenzt werden soll:

- `[0..]` : Der gesamte Bereich, vom ersten bis zum letzten Element.
- `[N..M]` : Ein Bereich beginnend ab Index `N` (eingeschlossen) und endend bei Index `M` (ausgeschlossen).

```
const name = "David";
// Die ersten drei Buchstaben
try std.testing.expectEqualStrings("Dav", name[0..3]);
// Die letzten zwei Buchstaben
try std.testing.expectEqualStrings("id", name[3..]);
// Die mittleren drei Buchstaben
try std.testing.expectEqualStrings("avi", name[1..4]);
```

Um Buffer-Overreads vorzubeugen überprüft Zig, dass die angegeben Indices valide sind. Sind die Indices zur Compilezeit bekannt, so führt ein invalider Index zu einem Compile-Fehler, andernfalls zu einer Panic zur Laufzeit.

chapter02/slice_error.zig

```
const a = "this won't work";
// ...
const n: usize = 20;
std.log.info("{s}", .{a[1..n]});
```

Versucht man den obigen Code mit `zig build-exe chapter02/slice_error.zig` zu Compilieren so erhält man den folgenden Fehler:

```
error: end index 20 out of bounds for array of length 15 +1 (sentinel)
    std.log.info("{s}", .{a[1..n]});
```

Sentinel-Terminierte Slices

Slices definieren einen Speicherbereich, durch einen Zeiger und eine Länge. Dadurch wird der Speicherbereich explizit eingegrenzt, was in vielen Fällen die aus C bekannten NULL-Terminatoren überflüssig macht. Jedoch unterstützt Zig auch NULL-terminierte Strings.

Allgemein werden Slices, deren Ende durch einen bestimmten Wert begrenzt wird (zum Beispiel ein NULL-Byte), als sentinel-terminated Slices bezeichnet. Der Sentinel (Wächter) ist eine vordefinierter Wert der das Slice abschließt.

Sentinel-terminierte Slices werden mit der `[:x]T` Syntax definiert und besitzen wie auch alle anderen Slices eine Länge, auf die über das `.len` Feld zugegriffen werden kann. Die Länge ist dabei die Länge des Slices ohne den Sentinel-Wert!

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

test "slice test" {
    const name: [:0]const u8 = "Pierre";

    try std.testing.expect(name.len == 5);
```

```
    try std.testing.expect(name[6] == 0);
}
```

Sentinel-terminierte Slices können auch mit der `data[start..end :x]` Syntax erzeugt werden, wobei `data` eine Zeiger auf mehrere Werte, ein Array oder ein Slice seien muss. Der Wert `x` ist der Sentinel.

```
const arr = [_]u8{'h', 'e', 'l', 'l', 'o', 0, 'h', 'e', 'l', 'l', 'o', 0};
const s = arr[0..5 :0];
```

Wichtig dabei ist, dass das mit `data[start..end :x]` erzeugte Slice auch tatsächlich vom angegebenen Sentinel terminiert wird! Sollte dies nicht der Fall sein, führt dies je zu undefiniertem Verhalten. Je nach gewählter Optimierung führt dies im besten Fall zur einer Panic zur Laufzeit, die den Prozess vorzeitig beendet.

Im Allgemeinen werden in Zig Slices, gegenüber sentinel-terminated Pointern, präferiert. Der Grund hierfür ist, dass Slices über Bounds-Checking verfügen und so gängige Speicherfehler abgefangen werden können. Es gibt jedoch auch Situationen, in denen many-item Pointer bzw. sentinel-terminated Pointer explizit benötigt werden, z.B. beim Arbeiten mit C Code. Auf die Interoperabilität zwischen Zig und C wird in einem späteren Kapitel noch näher eingegangen.

Errors

Während der Ausführung von Zig-Code kann ein Programm auf Fehler zur Laufzeit stoßen. Dabei kann es sich zum Beispiel um eine fehlende Datei handeln, die nicht geöffnet werde kann.

Im Gegensatz zu Optionals, welche die Abwesenheit eines Wertes kommunizieren können, geben Fehler mehr Aufschluss über den Grund, warum der Aufruf einer Funktion fehlgeschlagen ist. Außerdem unterstützt Zig das Propagieren von Fehlern.

Zig betrachtet Errors als Werte, die in einem Error-Set zusammengefasst werden. Ein Error-Set ist vergleichbar zu einem Enum, wobei jedem Error-Bezeichner ein eindeutiger ganzzahliger Wert größer 0 zugewiesen wird[30]. Wird ein Error-Bezeichner (zum Beispiel `error.OutOfMemory`) mehrfach definiert, so wird diesem immer der selbe numerische Wert zugewiesen.

Error-Sets können mit dem `error` Schlüsselwort definiert werden. Ein Error-Typ wird deklariert, indem dem Basistypen der Name des zugehörigen Error-Sets, gefolgt von einem `!`, vorangestellt wird. Angenommen eine Funktion gibt potenziell einen Fehler aus dem Error-Set `MyErrors` oder `void` (kein Rückgabewert) zurück, dann kann der Rückgabewert der Funktion wie folgt geschrieben werden: `MyErrors!void`. Um einen Error zurück zu geben kann der entsprechende Error-Wert, genau wie andere Rückgabewerte, mit `return` an die aufrufende Funktion gereicht werden.

[30] Standardmäßig ist der einem Error zugrunde liegende Integer-Typ ein `u16`.

chapter02/errors.zig

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

const MyErrors = error{
    IsNotEight,
};

/// Check if the given number is eight.
/// Returns an error if `n` is not equal 8!
fn checkNumber(n: u8) MyErrors!void {
    if (n != 8) return MyErrors.IsNotEight;
}

test "Error test #1" {
    try std.testing.expectError(MyErrors.IsNotEight, checkNumber(7));
}
```

Da den gleichen Error-Bezeichnern der gleiche numerische Wert zugewiesen wird, kann im obigen Beispiel anstelle von `MyErrors.IsNotEight` auch `error.IsNotEight` zurückgegeben werden. Zig erlaubt mit der Syntax `error.<NameDesErrors>` die Definition von Errors innerhalb eines impliziten Error-Sets. Dies ist die Kurzform für `(error{<NameDesErrors>}).<NameDesErrors>`.

```
fn checkNumber(n: u8) MyErrors!void {
    if (n != 8) return error.IsNotEight;
}
```

Error-Set Coercion

Angenommen es existieren zwei Error-Sets, wobei das eine Error-Set eine Teilmenge des Anderen darstellt. In einem solchen Fall erlaubt Zig die Coercion, das heißt das Umwandeln, von der Tielmenge in die Obermenge.

chapter02/errors.zig

```
const FileOpenError = error{
    AccessDenied,
    OutOfMemory,
    FileNotFound,
};

const AllocationError = error{
    OutOfMemory,
};
```

```
fn coerce(err: AllocationError) FileOpenError {
    return err;
}

test "Error-Set Coercion" {
    try std.testing.expect(FileOpenError.OutOfMemory ==
coerce(AllocationError.OutOfMemory));
}
```

Was jedoch nicht funktioniert ist die Umwandlung einer Obermenge in eine Teilmenge!

Globales Error-Set

Zig erlaubt es das explizite Error-Set links vom `!` wegzulassen, zum Beispiel `!void`. In diesem Fall ist das Error-Set implizit `anyerror`, das globalen Error-Set, dem alle Errors der gesamten Compilation-Unit angehören. Jedes Error-Set kann in `anyerror` umgewandelt werden. Außerdem kann eine Element aus dem globalen Error-Set explizit in ein nicht globales Error-Set gecastet werden.

```
fn checkNumber(n: u8) !void {
    if (n != 8) return error.IsNotEight;
}
```

Im obigen Beispiel wird der Rückgabewert automatisch in einen Wert vom Typ `anyerror!void` umgewandelt.

In vielen Fällen ist es praktisch das Error-Set einer Funktion von Zig ableiten zu lassen. Je nach Anwendungsfall kann dies jedoch auch Nachteile mit sich bringen. Vor allem bei der Entwicklung von Modulen, die mit anderen Programmieren geteilt werden, sollten Sie sich angewöhnen explizite Error-Sets zu verwenden.

catch

Mit `catch` können Errors, die von einer Funktion zurückgegeben werden, abgefangen und entsprechend behandelt werden.

chapter02/errors.zig

```
pub fn main() void {
    const n = 7;
    checkNumber(n) catch |e| {
        std.log.err("The number {d} is not equal 8: {any}", .{ n, e });
    };
}
```

Das `catch` folgt direkt hinter dem Aufruf der Funktion. Optional kann der Fehler-Wert an eine Variable (im obigen Fall `e`) gebunden werden. Der `catch`-Block (eingegrenzt durch geschweifte Klammern `{}`) wird nur ausgeführt, falls die Funktion einen Error als Rückgabewert liefert.

`catch` eignet sich ebenfalls um im Fehlerfall einen Default-Wert bereitzustellen.

chapter02/errors.zig

```
test "Default-Wert" {
    const n = std.fmt.parseInt(u64, "0xdeaX", 16) catch 16;
    try std.testing.expect(n == 16);
}
```

In diesem Beispiel ist `n` entweder gleich dem entpackten Rückgabewert von `parseInt()` oder, falls `parseInt()` einen Error zurück gibt, 16. Wie zu sehen ist muss nicht zwangsläufig ein Block auf `catch` folgen, genauso zulässig ist ein Ausdruck. Der entpackte Rückgabewert der Funktion und der Ausdruck rechts vom `catch` müssen den selben Typ besitzen (in diesem Beispiel `u64`).

Alternativ kann auch ein Block mit einem frei wählbaren Bezeichner (zum Beispiel `blk`) verwendet werden. Der Bezeichner muss dabei die selben Anforderungen wie ein Variablen-Name erfüllen.

```
const n = std.fmt.parseInt(u64, "0xdeaX", 16) catch blk: {
    break :blk 16;
}
try std.testing.expect(n == 16);
```

Mittels `break` kann der Default-Wert `16` in den umschließenden Block gereicht werden, wo er an die Konstante `n` gebunden wird. Das Literal wird dabei automatisch vom Typ `comptime_int` in einen `u64` umgewandelt.

try

In vielen Fällen reicht es aus, beim Auftreten eines Errors, selbst einen Error an die aufrufende Funktion zurückzugeben. Dies wird als Fehler-Propagierung bezeichnet und kann in Zig durch die Verwendung von `try` umgesetzt werden. Hierzu wird vor den Aufruf einer Funktion, die einen Fehler-Typen als Rückgabetyp besitzt, das Schlüsselwort `try` gesetzt.

```
fn foo(str: []const u8) !void {
    const n = try std.fmt.parseInt(u64, str, 16);
    _ = n;
}
```

Das Schlüsselwort `try` evaluiert den zugehörigen Ausdruck und kehrt im Fehlerfall mit dem selben Error aus der Funktion zurück. Andernfalls wird der Rückgabewert der aufgerufenen Funktion entpackt.

Dies ist die Kurzform für den folgenden Code:

```
fn foo(str: []const u8) !void {
    const n = std.fmt.parseInt(u64, str, 16) catch |e| return e;
    _ = n;
}
```

errdefer

Es gibt Situationen, bei denen Code nur im Fehlerfall ausgeführt werden soll, zum Beispiel um Speicher zu de-allozieren, der nicht mehr benötigt wird. Für solche Fälle kann `errdefer` verwendet werden, das die gleichen Eigenschaften wie `defer` aufweist, mit dem großen Unterschied das `errdefer` nur ausgeführt wird, sollte die Funktion einen Fehler zurückgeben.

```
fn alwaysFail(a: std.mem.Allocator) ![]const u8 {
    const mem = try a.alloc(u8, 13);
    errdefer a.free(mem);

    @memcpy(mem, "Hello, World!");

    if (std.mem.eql(u8, "Hello", mem[0..5])) {
        // Weil der Fehler `HelloError` zurückgegeben wird,
        // wird auch der von `mem` referenzierte Speicher
        // freigegeben.
        return error.HelloError;
    }

    return mem;
}
```

Das praktische ist, dass durch `errdefer` die Allokation und Deallokation sehr nahe beieinander liegen können. Dies macht es einfacher sicherzustellen, dass im Fehlerfall kein Speicherleck (engl. Memory-Leak) entsteht.

Sowohl `defer` als auch `errdefer` beziehen sich auf den umschließenden Block. Dadurch wird `errdefer` nicht ausgeführt, sollte der Error außerhalb des Blocks zurückgegeben werden!

Error-Sets zusammenführen

Errors-Sets können mit dem `||`-Operator zusammengeführt werden.

```
const A = error{
    Foo,
};
```

```
const B = error{
    Bar,
};

const AB = A || B;
```

Kapitel 3

Speicherverwaltung

Im Vergleich zu anderen Sprachen, wie etwa Java oder Python, muss der Speicher in Zig manuell verwaltet werden. Dies bringt einige Vorteile mit sich, birgt aber auch Risiken, die bei Nichtbeachtung zu Schwachstellen in den eigenen Anwendungen führen können. Was Zig von anderen Sprachen mit manueller Speicherverwaltung hervorhebt ist die explizite Verwendung und Verwaltung von Allokatoren, in der Programmiersprache repräsentiert durch den `Allocator` Typ. Dies kann von anderen Programmiersprachen kommenden Entwicklern anfangs ungewohnt vorkommen, bietet jedoch ein hohes Maß an Flexibilität, da Speicher zur Laufzeit dynamisch von verschiedenen Speicherquellen alloziert werden kann.

Grundlagen

In den meisten Fällen kann ein Programm von zwei verschiedenen Quellen Speicher allozieren, dem Stack und dem Heap. Wird eine Funktion aufgerufen, so alloziert diese Speicher auf dem Stack der von den lokalen Variablen und Parametern zur Speicherung der zugehörigen Werte verwendet wird. Dieser, von einer Funktion allozierte, Speicherbereich wird als Stack-Frame bezeichnet. Die Allokation eines Stack-Frames wird durchgeführt, indem der Wert eines spezielles CPU-Register, der sog. Stack-Pointer welcher auf das Ende des Stacks zeigt, verringert wird. Die Anzahl an Bytes um die der Stack-Pointer verringert werden muss um alle lokalen Variablen halten zu können wird vom Compiler zur Compilezeit berechnet und in entsprechende Assemblerinstruktionen übersetzt.

Durch die Einschränkung, dass die Größe eines Stack-Frames zur Compilezeit bekannt sein muss, lassen sich bestimmte Aufgaben schwer lösen. Angenommen Sie wollen eine Zeichenkette unbekannter Länge von Ihrem Programm einlesen lassen, um diese später zu verarbeiteten. Eine Möglichkeit um die Zeichenkette zu speichern wäre innerhalb der `main` Funktion eine Variable vom Typ Array mit fester Länge zu deklarieren, jedoch ist dieser Ansatz sehr unflexibel da Sie in dem gegebenen Szenario die Länge der zu erwartenden Zeichenkette nicht kennen. Bei besonders kurzen Zeichenketten verschwenden Sie ggf. Speicher während sich besonders lange Zeichenketten nicht einlesen lassen, da nicht genügend Speicher auf dem Stack alloziert wurde. Um Probleme solcher Art besser lösen zu können, kann Speicher dynamisch zur Laufzeit eines Programms alloziert werden. Der Heap kann als linearer Speicherbereich betrachtet werden, der von einem Allokator verwaltet wird. Wird Speicher zur Laufzeit benötigt, so kann der Allokator durch einen Funktionsaufruf angewiesen werden eine bestimmte Menge an Bytes zu allozieren.

Der Allokator sucht ein Stück Speicher mit der passenden Länge heraus, markiert dieses als alloziert und gibt einen Zeiger auf den Beginn des Speicherbereichs zurück. Wird der Speicher nicht mehr benötigt, so kann der Allokator durch einen weiteren Funktionsaufruf aufgefordert werden den Speicher wieder frei zu geben. In C und C++ verwenden Sie i.d.R. `malloc` und `free` um Speicher zu allozieren bzw. freizugeben, in den wenigsten Fällen müssen Sie sich jedoch Gedanken um den zu verwendenden Allokator machen. Im Gegensatz dazu verwenden Sie in Zig immer explizit einen Allokator.

In vielen Fällen, vor allem als Neuling, ist die Unterscheidung zwischen den vielen verschiedenen Arten von Allokatoren, welche die Zig Standartbibliothek bereitstellt, weniger interessant. Wird ein Standard-Allokator, im Sinne von `malloc` und `free`, benötigt, so kann auf den GeneralPurposeAllocator zurückgegriffen werden.

```
const Gpa = std.heap.GeneralPurposeAllocator(.{});
var gpa = Gpa{};
const allocator = gpa.allocator();
```

Die Funktion `GeneralPurposeAllocator` erwartet ein Konfigurationsstruct als Argument zur Compilezeit und gibt einen neuen `GeneralPurposeAllocator`-Typ zurück der der Konstante `Gpa` zugewiesen wird. In den meisten Fällen kann durch Verwendung von `.{}` als Argument die Standardkonfiguration übernommen werden. Danach kann der `Gpa` Allokator-Typ verwendet werden um ein neues Allokator-Object zu erzeugen und an die Variable `gpa` zu binden. Durch Aufruf der `allocator()` Funktion auf dem Objekt kann schlussendlich auf den eigentlichen Allokator zugegriffen werden. Dies mag auf den ersten Blick kompliziert wirken, vor allem im Vergleich zu anderen Sprachen wo Funktionen wie `malloc()` scheinbar immer zur Verfügung stehen, in den meisten Fällen reicht es aber aus, den Allokator einmal am Anfang der Anwendung zu instanziieren. Danach kann dieser zur Allokation von Speicher verwendet werden. Der Allocator-Typ erlaubt es verschiedene Allokatoren durch das selbe, standardisierte Interface zu verwenden. Das bedeutet, dass Entwickler von Bibliotheken bzw. Modulen das gesamte dynamische Speichermanagement durch einen Typen (Allocator) handhaben können, während die Verwender von besagten Bibliotheken die freie Wahl bezüglich des dahinter liegenden Allokators besitzen.

Beim Allozieren von Speicher wird in Zig grundsätzlich zwischen der Allokation von exakt einem Objekt und der Allokation mehrerer Objekte unterschieden. Soll speicher für genau ein Objekt alloziert werden, so muss `create()` zum allozieren und `destroy()` zum Freigeben des Speichers verwendet werden. Andernfalls können die Funktionen `alloc()` und `free()` verwendet werden. Die Funktion `create()` erwartet einen Typen (`type`) als Argument und alloziert daraufhin Speicher für exakt eine Instanz dieses Typen. Eine Allokation kann jedoch fehlschlagen, z.B. weil kein ausreichender Speicher auf dem Heap vorhanden ist. Aus diesem Grund gibt `create()` nicht direkt einen Zeiger auf den allozierten Speicher zurück, sondern einen Fehler-Typ. Damit werden Entwickler gezwungen sich bewusst zu machen, dass eine Allokation fehlschlagen kann. Dies spiegelt sich auch im Zig-Zen wieder, in welchem es u.a. heißt: „Resource

allocation may fail; resource deallocation must succeed'' (auf Deutsch: Die Allokation von Resourcen kann fehlschlagen; die deallokation von Resourcen muss gelingen).

```
// chapter03/hello_world.zig
const std = @import("std");

const Gpa = std.heap.GeneralPurposeAllocator(.{});
var gpa = Gpa{};
const allocator = gpa.allocator();

pub fn main() !void {
    const T = u8;
    const L = "Hello, World".len;

    // Hier allozieren wir Speicher für L Elemente vom Typ `u8` .
    const hello_world = allocator.alloc(T, L) catch {
        // Im Fall, dass der Speicher nicht alloziert werden kann,
        // geben wir eine Fehlermeldung aus und beenden den
        // Prozess ordnungsgemäß.
        std.log.err("We ran out of memory!", .{});
        return;
    };
    // Defer wird vor dem Verlassen der Funktion ausgeführt.
    // Es ist 'good practice' Speicher mittels `defer` zu
    // deallozieren.
    defer allocator.free(hello_world);

    // Wir kopieren "Hello, World" in den allozierten Speicher.
    @memcpy(hello_world, "Hello, World");

    // Nun geben wir den String auf der Kommandozeile aus.
    const stdout_file = std.io.getStdOut().writer();
    var bw = std.io.bufferedWriter(stdout_file);
    const stdout = bw.writer();

    try stdout.print("{s}\n", .{hello_world});
    try bw.flush();
}
```

Das obige Programm gibt „Hello, World'' auf der Kommandozeile aus, jedoch allozieren wir vor der Ausgabe, zur Veranschaulichung, Speicher für den auszugebenden String auf dem Heap. Die Funktion `alloc()` erwartet als Argument den Typ, für den Speicher alloziert werden soll (`u8`), sowie die Anzahl an Elementen. Aus dem Typ `T` und der Anzahl `L` berechnet sich die Anzahl an Bytes die benötigt werden um `L` mal den Typ `T` im Speicher zu halten (`@sizeOf(T) * L`). Wie bereits erwähnt kann die Speicherallokation fehlschlagen, aus diesem Grund müssen wir denn Fehlerfall berücksichtigen bevor wir auf den Rückgabewert von `alloc()` zugreifen können[31].

[31] Anstelle eines catch Blocks hätten wir an dieser Stelle auch try verwenden können.

Da `alloc()` Speicher für mehr als ein Objekt alloziert, gibt die Funktion anstelle eines Zeigers auf den Typ `T` einen Slice vom Typ `T` zurück. Ein Slice ist ein Wrapper um einen Zeiger, der zusätzlich die Länge des referenzierten Speicherbereichs kodiert. Nach außer verhält sich ein Slice wie ein Zeiger in C, d.h. mit dem Square-Bracket-Operator `[]` kann auf einzelne Element zugegriffen werden, jedoch wird vor jedem Zugriff überprüft, ob der angegebene Index innerhalb des allozierten Bereichs liegt um Out-of-Bounds-Reads zu vorzubeugen. Slices ersetzen in vielen Fällen null-terminierte Strings, was dabei hilft Speicherfehlern vorzubeugen.

Ein wichtiger Punkt der zu jeder Allokation gehört ist die Deallokation des allozierten Speichers. In Zig kann diese direkt nach dem Aufruf von `alloc()` bzw. `create()` platziert werden, indem dem Aufruf von `free()` der defer Operator vorangestellt wird. Defer sorgt dafür, dass vor dem Verlassen eines Blocks, im obigen Beispiel ist dies der Funktionsblock von main, alle defer Blöcke ausgeführt werden und zwar in umgekehrter Reihenfolge in der sie deklariert werden. Dies ist vor allem zum Aufräumen von Ressourcen sehr hilfreich.

Sehen Sie beim Lesen von Zig-Code keinen `defer` Block zur Bereinigung von Speicher direkt nach einer Allokation sollten Sie erst einemal stutzig werden. Es gibt aber auch Situationen, z.B. bei der Verwendung eines ArenaAllocators, in denen nicht jede einzelne Allokation manuell bereinigt werden muss. In solchen Fällen ist es aber durchaus nützlich für Leser Ihres Quellcodes, wenn Sie durch ein Kommentar ersichtlich machen, dass das Fehlen einer Deallokation beabsichtigt ist.

Lifetimes

Bei der Verwendung von Programmiersprachen mit manuellem Speichermanagement ist die Berücksichtigung der Lifetime (Lebenszeit) von Objekten essenziell um Speicherfehler zu vermeiden. Die Lifetime eines Objekts beschreibt ein abstraktes Zeitintervall zur Laufzeit, in welchem ein bestimmtes Objekt oder eine Sequenz von Objekten im Speicher existieren und auf diese zugegriffen werden darf. Die Art wie bzw. wo Speicher für ein Objekt alloziert wird hat dabei großen Einfluss auf dessen Lebenszeit. Im Allgemeinen beginnt die Lifetime eines Objekts mit dessen Erzeugung und endet wenn der Speicher des Objekt wieder freigegeben wird. Bezogen auf die Art der Allokation kann grob zwischen den folgenden Fällen unterschieden werden:

- Statische Allokation
- Automatische Allokation
- Dynamische Allokation

Static Memory

In Zig, wie auch in C, befinden sich statische Variablen und Konstanten, die im globalen Scope, bzw. im Fall von Zig in einem Container[32], einer Anwendung deklariert werden, in der .data oder .bss Section eines Programms. Speicher für diese Sektionen wird beim Start eines Prozesses

[32]Ein Container in Zig ist jedes Konstrukt, das als Namensraum (engl. namespace) dient. Dazu zählen u.a. Structs aber auch Sourcedateien.

gemapped und er bleibt bis zur Terminierung des Prozesses valide. Variablen die dies betrifft haben eine statische Lifetime, d.h. sie sind vom Start eines Prozesses bis zu dessen Beendigung valide. Selbes gilt für statische, lokale Variablen.

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

const hello = "Hello, World";

pub fn main() void {
    const local_context = struct {
        var x: u8 = 128;
    };

    std.log.info("{s}, {d}", .{ hello, local_context.x });
}
```

Die Konstante `hello` wird im umschließenden Container, dargestellt durch die Quelldatei, deklariert und ist damit zum einen statisch, zum anderen ist sie aufgrund des `const` Modifiers zur Compilezeit bekannt. Selbes gilt für die lokale, statische Variable `x`. Im gegensatz zu C werden statische, lokalen Variablen nicht mit dem static Keyword deklariert sondern innerhalb eines lokalen Structs welches ebenfalls einen Container darstellt. Lokale, statische Variablen können nützlich sein um z.B. einen gemeinsamen „Shared-State“ zwischen Aufrufen der selben Funktion zu verwirklichen.

In Zig wird jede Translationunit, d.h. jede Datei in der Quellcode liegt, als Struct und damit als Container betrachtet. Dementsprechend gibt es eigentlich keine global deklarierten Variablen wie man sie aus C kennt, sondern nur statische Variablen die in Containern deklariert werden.

Automatic Memory

Objekte die durch Deklaration bzw. Definition innerhalb eines (Funktions-)Blocks erzeugt werden erben ihre Lifetime von dem umschließenden Block. Für Variablen und Parameter von Funktionen bedeutet dies, dass sich ihre Lifetime an der Lifetime eines Stack-Frames orientiert. Bei jedem Funktionsaufruf wird für den Aufruf ein Stück zusammenhängender Speicher auf dem Stack alloziert (der Stack-Frame) welcher groß genug ist um alle lokalen Variablen und Parameter zu halten. Der Frame wird dabei durch zwei Spezialregister der CPU, dem Stack-Pointer (SP) und dem Base-Pointer (BP), eingegrenzt. Der Stack-Pointer zeigt dabei auf das Ende vom Stack.

Es gibt verschiedene Arten von Stacks, jedoch ist die wohl häufigst auftretende Form der Full-Descending-Stack. Das bedeutet, dass der Stack nach unten „wächst“ (Descending), d.h. von höheren zu niedrigeren Speicheradressen, und der Stack-Pointer auf das erste valide Element des Stacks zeigt (Full).

```
// chapter03/stack_01.zig
const std = @import("std");

pub fn main() !void {
    var i: usize = 0; // Begin der Lifetime von 'i' --/
                      //                              /
    try foo(&i); // foo referenziert 'i'              /
} // Ende der Lifetime von 'i' ----------------------/

pub fn foo(a: *u64) !void { // Begin der Lifetime von 'a' --/
    a.* += 1;               //                             /
} // Ende der Lifetime von 'a' ---------------------------/
```

Das obige Programm übergibt eine Referenz auf die Variable `i` als Argument an die Funktion `foo()`, welche `i` inkrementiert. Die Lifetime der Variable startet mit ihrer Definition und endet mit dem Funktionsblock von main. Innerhalb der Lifetime darf `i` von anderen Programmteilen, in diesem Fall der Funktion `foo()`, referenziert und ggf. modifiziert werden.

Die Lifetime der Referenz `a` zu `i` beginnt mit dem Funktionsblock von `foo()` und endet mit dem Ende des Funktionsblocks. Wichtig ist, dass die Lifetime einer Referenz immer innerhalb der Lifetime des referenzierten Objekts liegen muss. Überschreitet die Lifetime einer Referenz die Lifetime des referenzierten Objekts so spricht man von einem dangling Pointer (auf Deutsch hängender Zeiger). Die Verwendung solcher dangling Pointer können zu schwerwiegenden Programmfehlern führen, da der referenzierte Speicher als undefiniert gilt.

Um das Verhalten der Anwendung besser nachvollziehen zu können, besteht die Möglichkeit mithilfe des Programms objdump die kompilierte Anwendung zu disassemblieren: **`objdump -d -M intel stack_01`**. Der Eintrittspunkt einer jeden Anwendung ist dabei die `main` Funktion.

```
00000000010349b0 <stack_01.main>:
 10349b0: push   rbp        ; Begin Prolog ---|
 10349b1: mov    rbp,rsp    ;                  |
 10349b4: sub    rsp,0x10   ; End Prolog -----|
 10349b8: mov    QWORD PTR [rbp-0x8],0x0 ; i = 0
 10349bf:       00
 10349c0: lea    rdi,[rbp-0x8]           ; &i
 10349c4: call   10348e0 <stack_01.foo>
 10349cb: add    rsp,0x10   ; Begin Epilog --|
 10349cf: pop    rbp        ; End Epilog ----|
 10349d0: ret
```

Jede Funktion besitzt ein Symbol (im Fall von main ist dies `stack_01.main`) welches repräsentativ für die Adresse der ersten Instruktion steht. Beim Funktionsaufruf wird diese Adresse in das Instruktions-Zeiger-Register (Instruction Pointer - IP) geschrieben, welcher immer auf die nächste auszuführende Instruktion zeigt. Jede Funktion beginnt mit dem sogenannten Funkti-

ons-Prolog, welcher einen neuen Stack-Frame für den Funktionsaufruf erzeugt, und endet mit dem Funktions-Epilog, welcher den Stack-Frame wieder entfernt, d.h. den Stack in den Zustand vor dem Funktionsaufruf zurückversetzt.

Im Prolog wird zuerst der Zustand des Base Pointers (BP), welcher auf den oberen Teil des derzeitigen Stack-Frames zeigt, auf den Stack gepushed, um diesen im Epilog wieder herstellen zu können. Danach wird der BP mit dem Wert des SP überschrieben, d.h. BP und SP zeigen beide auf den alten BP auf dem Stack. Danach werden 16 Bytes (0x10) für die Variablen und Parameter von main auf dem Stack alloziert, indem der SP um die entsprechende Anzahl an Bytes verringert wird. Innerhalb des allozierten Speicherbereichs wird die Variable `i` mit dem Wert 0 initialisiert. Die Adresse der Variable `i` (BP - 8) wird an die Funktion `foo()` mittels des RDI Registers übergeben[33].

In einer kompilierten Anwendung existieren Variablen nur implizit, d.h. es gibt keine Symbole oder ähnliches mit denen z.B. die Variable `i` klar identifiziert werden kann. In Assembler ist eine Variable lediglich ein Bereich im (Haupt-)-Speicher in welchem der Wert der Variable gespeichert ist.

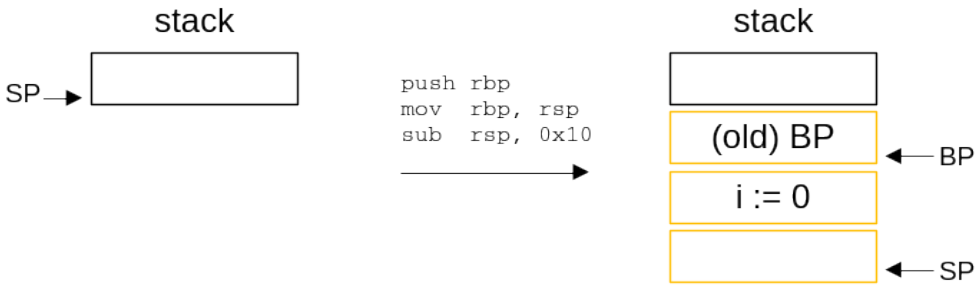


Abbildung 4: Stack-Frame von `main()`

```
00000000010348e0 <stack_01.foo>:
 10348e0: push   rbp
 10348e1: mov    rbp,rsp
 10348e4: sub    rsp,0x20
 10348e8: mov    QWORD PTR [rbp-0x18],rdi
 10348ec: mov    QWORD PTR [rbp-0x8],rdi
 10348f0: mov    rax,QWORD PTR [rdi]
 10348f3: add    rax,0x1
 10348f7: mov    QWORD PTR [rbp-0x10],rax
 10348fb: setb   al
 10348fe: jb     1034902 <stack_01.foo+0x22>
 1034900: jmp    1034924 <stack_01.foo+0x44>
 1034902: movabs rdi,0x101ed99
 1034909:       00 00 00
```

[33]Wer mehr über Assembler-Programmierung lernen möchte, dem empfehle ich das Buch „x86-64 Assembly Language Programming with Ubuntu'' von Ed Jorgensen. Diese ist öffentlich zugänglich und bietet einen sehr guten und verständlichen Einstieg.

```
103490c: mov    esi,0x10
1034911: xor    eax,eax
1034913: mov    edx,eax
1034915: movabs rcx,0x101dfb0
103491c:       00 00 00
103491f: call   1034940 <builtin.default_panic>
1034924: mov    rax,QWORD PTR [rbp-0x18]
1034928: mov    rcx,QWORD PTR [rbp-0x10]
103492c: mov    QWORD PTR [rax],rcx
103492f: xor    eax,eax
1034931: add    rsp,0x20
1034935: pop    rbp
1034936: ret
```

Nach dem Aufruf von `foo()` wird zuerst ein neuer Stack-Frame für den Funktionsaufruf erzeugt. Innerhalb dieses Stack-Frames wird der Parameter `a` mit der Adressen von `i` initialisiert. Was auffällt ist, dass die in RDI gespeicherte Adresse gleich mehrmals auf den Stack geschrieben wird und zusätzlich direkt dereferenziert wird um den Wert von `i` in das Regsiter RAX zu laden. Schaut man sich jedoch den gesamten Funktionskörper an so sieht man, dass von der Speicherstelle BP - 24 die Adresse von `i` zum Zurückschreiben des inkrementierten Werts geladen wird. Damit ist BP - 24 in diesem Fall der Parameter `a`. Dementsprechend sieht der Stack nach aufruf von `foo()` wie folgt aus.

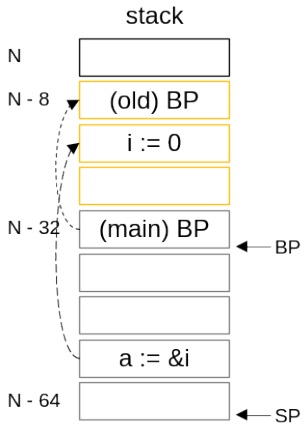


Abbildung 5: Stack-Frame von `foo()`

Diese im Funktionsprolog vollzogenen Schritte werden vor dem Verlassen der Funktion, im Epilog, umgekehrt, d.h. beim ausführen der ret Instruktion befindet sich der Stack, bezogen auf sein Layout, im selben Zustand wie vor dem Fuktionsaufruf. Was sich natürlich geändert hat ist der Wert der Variable `i`.

Dynamic Memory

Wir haben uns die Allokation von dynamischem Speicher anhand des `GeneralPurposeAllocator` am Anfang dieses Kapitels schon etwas angeschaut. Die Lifetime von dynamsich allozierten Objekten ist etwas tückischer als die von statisch oder automatisch

allozierten. Der Grund ist, dass bei komplexeren Programmen sowohl die Allokation als auch die Deallokation eines Objekts an verschiedenen Stellen im Code passieren kann, z.B. abhängig von einer Bedingung.

Ein Beispiel hierfür ist eine verkettete Liste, bei der alle Element dynamisch auf dem Heap alloziert werden. Bezogen auf die Allokation würde es in diesem Szenario mindestens eine Stelle geben und zwar der Bereich des Codes, in dem ein neues Listenelement erzeugt wird. Bei der Deallokation eines Elements muss zumindest unterschieden werden, ob ein Element aus der Liste entfernt wird oder ob die gesamte Liste, zum Ende des Programms, deallociert werden soll, wobei letzteres als ein Sonderfall angesehen werden kann. Ein weiterer Aspekt auf den geachtet werden muss ist, dass nach dem Löschen eines Elements der Liste, alle Referenzen auf dieses Element invalide sind, d.h. es darf nicht mehr auf den Speicher zugegriffen werden.

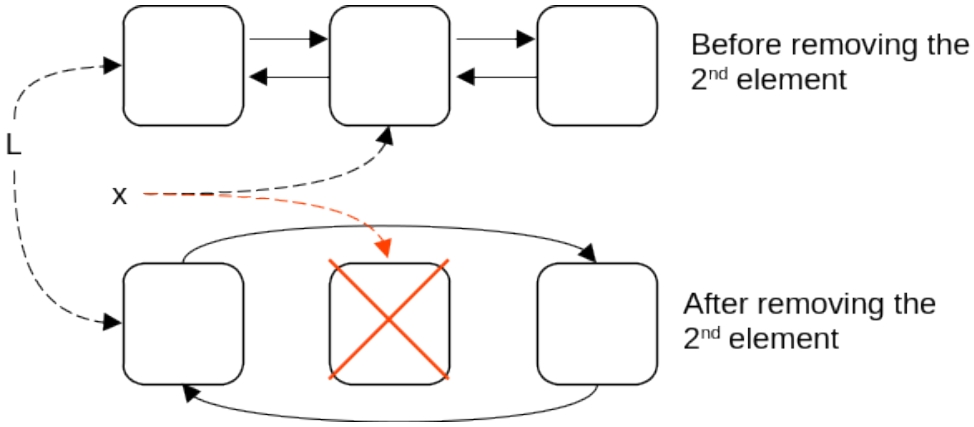


Abbildung 6: Beispiel einer verketteten Liste

```zig
// chapter03/linked-list.zig
const std = @import("std");

// Element einer verketteten Liste mit zwei (optionalen)
// Zeigern auf das nächste und vorherige Element.
const Elem = struct {
    prev: ?*Elem = null,
    next: ?*Elem = null,
    i: u32,

    pub fn new(i: u32, allocator: std.mem.Allocator) !*@This() {
        var self = try allocator.create(@This());
        self.i = i;
        return self;
    }
};

pub fn main() !void {
    // Hier fassen wir die Erzeugung eines neune Allokator-Typen
    // und dessen Instanziierung in einen Ausdruck zusammen...
```

```
    var gpa = std.heap.GeneralPurposeAllocator(.{}){};
    // ... und binden dann den Allokator an eine Konstante.
    const allocator = gpa.allocator();

    // Als nächstes erzeugen wir (manuell) eine verkettete
    // Liste mit drei Elementen.
    var lhs = try Elem.new(1, allocator);
    defer allocator.destroy(lhs);
    var middle = try Elem.new(2, allocator);
    var rhs = try Elem.new(3, allocator);
    defer allocator.destroy(rhs);

    lhs.next = middle;
    middle.prev = lhs;

    middle.next = rhs;
    rhs.prev = middle;

    // Die Konstante L referenziert das erste Element aus
    // der Liste (`lhs`).
    const L = lhs;
    // Ausgehend vom ersten Element geben wir alle Werte der
    // Liste nacheinander aus.
    std.log.info("Wert von lhs: {d}", .{L.i});
    std.log.info("Wert von middle: {d}", .{L.next.?.i});
    std.log.info("Wert von rhs: {d}", .{L.next.?.next.?.i});

    // Die Konstante `x` referenziert das mittlere Element...
    const x = middle;
    std.log.info(
    "Wert von Elem referenziert von x vor deallokation: {d}",
    .{x.i});

    // ... welches als nächstes aus der Liste (manuell) entfernt wird.
    lhs.next = middle.next;
    rhs.prev = middle.prev;
    allocator.destroy(middle);
    // ... ab diesem Zeitpunkt ist `x` ein dangling Pointer und
    // darf nicht mehr dereferenziert werden...

    // ... wir machen es trotzdem aber der Wert des referenzierten
    // Objekts ist ab diesem Zeitpunkt undefiniert.
    std.log.info(
    "Wert von Elem referenziert von x NACH deallokation: {d}",
    .{x.i});
}
```

In folgendem Beispiel erzeugen wir eine verkettete Liste mit drei Elementen. Als nächstes definieren wir eine Konstante `L`, die das erste Element der Liste `lhs` referenziert und geben nach und nach, durch Dereferenzierung, die Werte aller drei Elemente aus. Da weder `lhs`, `middle` noch `rhs` bis zum Zeitpunkt der Ausgabe dealloziert wurden, ist die Dereferenzierung erlaubt. Wie Sie vielleicht gesehen haben ist die dritte Ausgabe, die letzte Stelle an der `L` dereferenziert wird. Damit überschreitet die Lifetime von L zwar theoretisch die Lifetime von `lhs`, `middle` und `rhs`, in der Praxis spielt dies für die Korrektheit der Anwendung jedoch keine Rolle. Anders sieht es mit der Konstanten `x` aus. Zwischen der ersten und zweiten Dereferenzierung von `x` wird `middle` aus der Liste entfernt und dealloziert. Damit ist `x` ab der Deallokation von middle ein dangling Pointer was zum Problem wird, da `x` später noch einmal derefernziert wird. Auch nach der Deallokation zeigt `x` weiterhin auf eine existierende Speicherstelle, jedoch wurde diese durch die Deallokation freigegeben, d.h. die Daten an dieser Stelle sind undefiniert. Das hält Zig jedoch nicht davon ab den referenzierten Speicher als Instanz von Elem zu interpretieren, was sich auch in der Kommandozeilenausgabe widerspiegelt.

```
$ ./linked-list
info: Wert von lhs: 1
info: Wert von middle: 2
info: Wert von rhs: 3
info: Wert von Elem ... von x vor deallokation: 2
info: Wert von Elem ... von x NACH deallokation: 2863311530
```

Diese Art von Speicherfehler wird als Use-After-Free bezeichnet und kann unter den richtigen Bedingungen von Angreifern genutzt werden, den Kontrollfluss des Programms zu übernehmen, sollte es für den Angreifer möglich sein die Speicherstelle zu kontrollieren.

Etwas das Sie sich grundsätzlich Angewöhnen sollten ist, Referenzen die Sie nicht mehr benötigen zu invalidieren. Eine Möglichkeit dies zu tun ist anstelle eines Pointers einen optionalen Pointer zu verwenden.

```
var x: ?*Elem = middle;
// ...
lhs.next = middle.next;
rhs.prev = middle.prev;
allocator.destroy(middle);
x = null; // wir invalidieren x direkt nach der Deallokation
```

Häufige Fehler

Im Gegensatz zu speichersicheren (engl. memory safe) Sprachen wie etwa Rust, bietet Zig einige Fallstricke, die das Leben als Entwickler schwer machen können, aber nicht müssen! In diesem Abschnitt werden wir uns einige davon näher anschauen und ich zeigen Ihnen, wie Zig Ihnen dabei hilft sicheren Code zu schreiben.

Speicherzugriffsfehler (Access Errors)

Speicherzugriffsfehler sind eine typische Fehlerquelle und haben in der Vergangenheit schon zu so einigen Exploits geführt. Allgemein handelt es sich dabei um einen Oberbegriff für Programmierfehler, durch die unzulässig auf eine Speicherstelle zugegriffen wird. Zu den Speicherzugriffsfehlern gehören der Buffer-Overflow, Buffer-Over-Read, Invalid-Page-Fault und Use-After-Free. Den Use-After-Free haben wir uns im Kontext von Lifetimes schon angeschaut, an dieser Stelle möchte ich Ihnen die verbleibenden Fehler etwas näher bringen.

Buffer Overflow/ Over-Read

Der Buffer-Overflow und Buffer-Over-Read sind nah miteinander verwandt, kommen jedoch jeweils mit ihren eigenen Problemen. Beim Buffer-Overflow wird Speicher außerhalb eines validen Bereichs beschrieben. Dies ist meist das Resultat der unzureichenden Überprüfung der Grenzen eines Objekts, z.B. eines Arrays. Ein klassisches Beispiel ist ein Array, dass an einer Stelle indiziert wird die außerhalb der Grenzen des Arrays liegt.

```
var x: [10]u8 = .{0} ** 10;
x[10] = 1; // Index 10 ist out-of-bounds -> buffer overrread!
```

Diese Art von Fehlern können genutzt werden um Daten von naheliegenden Objekten oder sogar Adressen zu überschreiben. In der Vergangenheit wurde diese Art von Fehler von Angreifern genutzt um Schadcode in Anwendungen einzuschleusen, die Rücksprungadresse zu überschreiben und so die Kontrolle über den Prozess zu übernehmen. Moderne Compiler injizieren deswegen sogenannte Stack-Canaries, einen randomisierten Wert der von einem Angreifer nicht erraten werden kann und der vor der Rückkehr in die aufrufende Funktion überprüft wird, in Stack-Frames die potenziell von einem Buffer-Overflow betroffen sein könnten. Ist ein Stack-Frame von einem Buffer-Overflow betroffen und wurde die Rücksprungadresse überschrieben, so bedeutet dies, dass auch der Canary überschrieben wurde. In diesem Fall wird der Prozess zur Sicherheit beendet. Wie das Zig-Zen so schön sagt: „Laufzeit-Crashes sind besser als Bugs'' (engl. „Runtime crashes are better than bugs“).

Im obigen Fall wird der Buffer-Overflow schon zur Compile-Zeit erkannt, da Arrays eine zur Compile-Zeit bekannte Länge besitzen (Zig-Zen: „Compile errors are better than runtime crashes“).

```
error: index 10 outside array of length 10
    x[10] = 1;
```

Allozieren wir den Speicher jedoch dynamisch so kann der Compiler uns nicht mehr vor unserem Fehler bewahren.

```
var x = try allocator.alloc(u8, 10);
x[10] = 1;
```

Da wir in Zig jedoch in den meisten Fällen mit Arrays oder Slices arbeiten und nicht mit rohen Zeigern und Zig für beide Datenstrukturen die Grenzen bei einem Speicherzugriff überprüft,

wird der Buffer-Overflow zumindest zur Laufzeit erkannt und der Prozess beendet (Zig-Zen: „Runtime crashes are better than bugs“).

```
thread 7940 panic: index out of bounds: index 10, len 10
buffer-overflow.zig:12:6: 0x1037424 in main (buffer-overflow)
    x[10] = 1;
```

Invalid Page Fault

Moderne Betriebssystem schirmen den Hauptspeicher ab und stellen jedem Prozess stattdessen virtuellen Speicher zur Verfügung, der aus Sicht des Prozesses nahezu unbegrenzt ist. Bei bedarf werden Teile des Hauptspeichers, sogenannte Pages, in den virtuellen Addressraum eines Prozesses gemapped, wodurch dieser den Speicher lesend, so wie schreibend, nutzen kann.

Ein Invalid-Page-Fault ist eine Fehler bei dem ein Prozess versucht auf einen Speicherbereich zuzugreifen, der nicht durch eine Page gedeckt wird. Ein klassisches Beispiel hierfür ist die `NULL`-Pointer-Exception, bei dem innerhalb eines Programms versucht wird einen `NULL`-Zeiger zu dereferenzieren. Grund hierfür ist, dass die meisten Betriebssysteme keine physische Page auf den Speicherbereich mappen, der die Adresse `0x0000000000000000` mit einschließt. Wird eine Page-Fault durch das Betriebssystem erkannt, so erzeugt der Kernel einen Segmentation-Fault (Schutzverletzung) für den betroffenen Prozess, welcher diesen abnormal beendet. Jeder der schon einmal in C programmiert hat wird den entsprechenden Fehler `"sigsegv segmentation fault"` schon einmal auf der Kommandozeile gesehen haben.

Use After Free

Bei einem Use-After-Free wird Speicher, nach dem Ende dessen Lifetime, verwendet. Grundsätzlich hält Zig sie nicht davon ab, durch einen dangling Pointer, auf Speicher zuzugreifen, nachdem dieser wieder freigegeben wurde. Eine Möglichkeit das Risiko für Use-After-Free Bugs in Ihrem Code zu reduzieren, ist durch die Verwendung von optionalen Zeigern `?*T` bzw. `?*const T`.

```
```

Zusammenfassung

Bei dem Arbeiten mit Referenzen bzw. Slices sind zwei Fragen von essenzieller Bedeutung: Umschließt die Lifetime des referenzierten Objekts die der Referenz und wenn nein, habe ich dafür gesorgt, dass nach dem Ende der Lifetime des Objekts nicht mehr versucht wird auf dieses zuzugreifen. Fall Sie diese Fragen nicht beantworten können besteht eine hohe Wahrscheinlichkeit, dass sich Speicherfehler in Ihren Code einschleichen.

Kapitel 4

Control Flow

For-Schleifen

Es kann äußerst nützlich sein über den Inhalt eines Arrays oder Slices zu iterieren. Eine Möglichkeit dies zu tun ist mit Hilfe einer for-Schleife.

chapter02/loop.zig

```
const names = [_][]const u8{ "David", "Franziska", "Sarah" };

for (names) |name| {
    std.log.info("{s}", .{name});
}
```

Eine for-Schleife beginnt mit dem Schlüsselwort `for`, gefolgt von einer Sequenz, über die iteriert werden soll, in runden Klammern. Danach wird ein Bezeichner zwischen zwei `| |` angegeben. Dem Bezeichner wird für jede Iteration der aktuelle Wert zugewiesen, d.h. für das obige Beispiel wird im ersten Schleifendurchlauf `name` der Wert `"David"` zugewiesen, im zweiten Durchlauf `"Franziska"` und so weiter. Nachdem über alle Elemente iteriert wurde, wird automatisch aus der Schleife ausgebrochen.

Eine Besonderheit von Zig ist, dass innerhalb einer for-Schleife simultan über mehrere Sequenzen iteriert werden kann.

```
for (names, 0..) |name, i| {
    std.log.info("{s} ({d})", .{ name, i });
}
```

Die Sequenzen werden, getrennt durch ein Komma, innerhalb der runden Klammer angegeben. Selbes gilt für die Bezeichner, an die die einzelnen Werte der Sequenzen gebunden werden. Im obigen Beispiel wird als zweite Sequenz `0..` angegeben, d.h. eine Sequenz von Ganzzahlen beginnend bei 0. Zig sorgt dabei automatisch dafür, dass `names` und `0..` über die selbe Länge verfügen, indem das Ende von `0..` automatisch bestimmt wird, d.h. für das gegebene Beispiel ist `0..` äquivalent zu `0..3`.

Sollten Sie über mehrere Arrays bzw. Slices gleichzeitig iterieren, so müssen sie sicherstellen, dass alle die selbe Länge besitzen!

```
const dishes = [_][]const u8{ "Apfelstrudel", "Pasta", "Quiche" };

for (names, dishes) |name, dish| {
    std.log.info("{s} likes {s}", .{ name, dish });
}
```

Mit dem Schlüsselwort `break` kann aus einer umschließenden Schleife ausgebrochen werden, d.h. das Programm wird unter der Schleife fortgeführt.

```
for (1..5) |i| {
    std.log.info("{d}", .{i});
    if (i == 2) break;
}
```

Mit dem Schlüsselwort `continue` können sie den restlichen Körper der Schleife überspringen und mit der nächsten Iteration beginnen. Sollten `continue` in der letzten Iteration der Schleife ausgeführt werden, so wird aus dieser ausgebrochen.

```
for (1..5) |i| {
    if (i == 2) continue;
    std.log.info("{d}", .{i});
}
```

Schleifen können auch geschachtelt werden. Wenn Sie innerhalb einer der inneren Schleifen, aus einer der Äußeren ausbrechen wollen, müssen Sie sogenannte Label verwenden, mit der sie einer bestimmten Schleife einen Namen geben können. Labels kommen vor dem `for` Schlüsselwort und enden immer mit einem `:`. Sie können sowohl mit `break` als auch `continue` verwendet werden.

```
outer: for (names) |name| {
    for (dishes) |dish| {
        std.log.info("({s}, {s})", .{ name, dish });
        // Da wir an dieser stelle aus der äußeren Schleife ausbrechen
        // ist nur eine Ausgabe auf der Kommandozeile zu sehen.
        break :outer;
    }
}
```

Zig erlaubt auch die Verwendung von for-Loops in Ausdrücken.

```
const pname = outer: for (names) |name| {
    if (name.len > 0 and (name[0] == 'p' or name[0] == 'P'))
        break :outer name;
} else blk: {
    break :blk "no name starts with p!";
};
std.log.info("found: {s}", .{pname});
```

In diesem Beispiel suchen wir nach einem Namen der mit dem Buchstaben P bzw. p beginnt. Sollte aus der Schleife mit `break` ausgebrochen werden, so wird der `else` Block nicht ausgeführt. Da `names` keinen solchen Namen beinhaltet wird der `else` Block aufgerufen und der String `"no name starts with p!"` der Konstanten `pname` zugewiesen.

Neben Schleifen können auch `if` / `else` Blöcken Label zugewiesen werden. Dies erlaubt es, mittels `break`, Werte aus dem Block heraus zu reichen, wie oben zu sehen ist.

Sie können das Beispiel mit **`zig build-exe chapter02/loop.zig && ./loop`** compilieren und ausführen.