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

für Franziska

Inhaltsverzeichnis

Zig Crash Course 1
- Zig installieren 1
- Funktionen 4
- Unit Tests 6
- Comptime 6
- Kommandozeilenargumente 8
- Parallelität 9
- Cross Compilation 9

Einfache Typen 11

Speicherverwaltung 13
- Grundlagen 13
- Lifetimes 16
  - Static Memory 16
  - Automatic Memory 17
  - Dynamic Memory 20
- Häufige Fehler 23
  - Speicherzugriffsfehler (Access Errors) 23
    - Buffer Overflow/ Over-Read 23
    - Invalid Page Fault 24
- Zusammenfassung 24

Vorwort

Zig ist eine Sprache geeignet für die Systemprogrammierung.

TODO

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.

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.

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

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[4] 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.

Struktur

Die ersten zwei Kapitel TBD…

Zig bietet für jede Compiler-Version zusätzliche Ressourcen zum Lernen der Sprache und als Referenz[5], 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.

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.

Kontakt

Bei Fragen zu diesem Buch kontaktieren Sie bitte david (at) thesugar.de.

Danksagung

TDB

[4] https://projecteuler.net/about
[5] https://ziglang.org/learn/

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“[6].

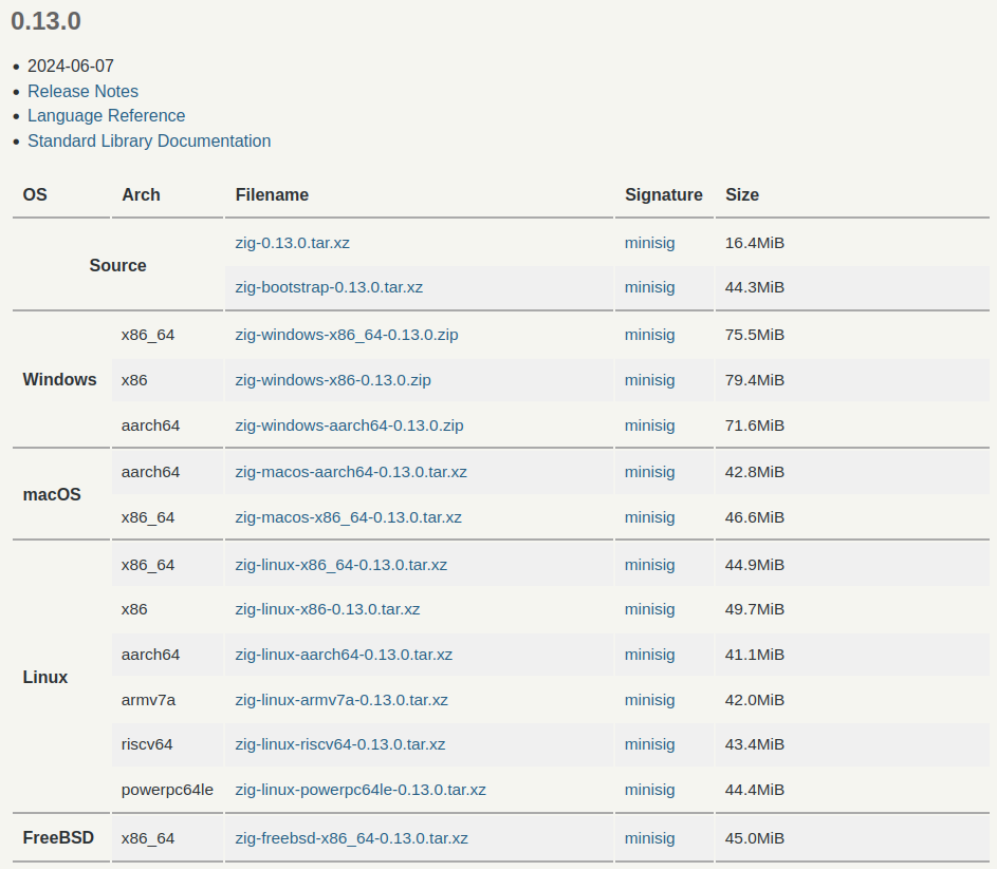
Die Installation ist unter allen Betriebssystemen relativ einfach durchzuführen. In der Download Sektion[7] finden Sie vorkompilierte Zig-Compiler für die gängigsten Betriebssysteme, darunter Linux, macOS und Windows, sowie Architekturen.

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.

[6] https://ziglang.org/learn/getting-started/
[7] 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 der Compiler, den wir im Laufe dieses Buchs exzessiv verwenden werden.

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[8].

```
# 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.

```
$ 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

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

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, inklusive Test, 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!
}

test "simple test" {
    var list = std.ArrayList(i32).init(std.testing.allocator);
    defer list.deinit(); // try commenting this out
    try list.append(42);
    try std.testing.expectEqual(@as(i32, 42), list.pop());
}
```

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, kann aber auf das Zig-Zen zurückgeführt werden 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 die andere zurückgegeben, ansonsten wird `gcd` rekursiv aufgerufen bis eine beider Zahlen 0 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 Inhalt des Bedingungsblocks 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 Bedingungsblocks 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, 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, d.h., wer Zig programmieren kann, der hat alles 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 ein eigener `ArrayList`-Typ generiert wird.

```
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 die eigentliche Funktionalität der unserer ArrayList nur einmal implementieren müssen.

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[9].

Kommandozeilenargumente

[9]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).

Parallelität
Cross Compilation

Kapitel 2

Einfache Typen

Foo bra

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");
```

```zig
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[10]. 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 allo-

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

zierten 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 einem 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 Variablen und Konstanten, die im globalen Scope einer Anwendung deklariert werden, in der .data oder .bss Section eines Programms. Speicher für diese Sektionen wird beim Start eines Prozesses gemapped und er bleibt bis zur Terminierung des Prozesses valide. Variablen die dies betrifft haben eine statische Life-Time, 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 global deklariert und ist damit statisch. Selbes gilt für die lokale Variable `x`. Im gegensatz zu C werden statische, lokalen Variablen nicht mit dem static Keyword deklariert sondern innerhalb eines lokalen Structs. 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 betrachtet. Dementsprechend gibt es eigentlich keine global deklarierten Variablen, sondern nur Variablen die in Structs 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 Funktions-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[11].

[11]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.

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.

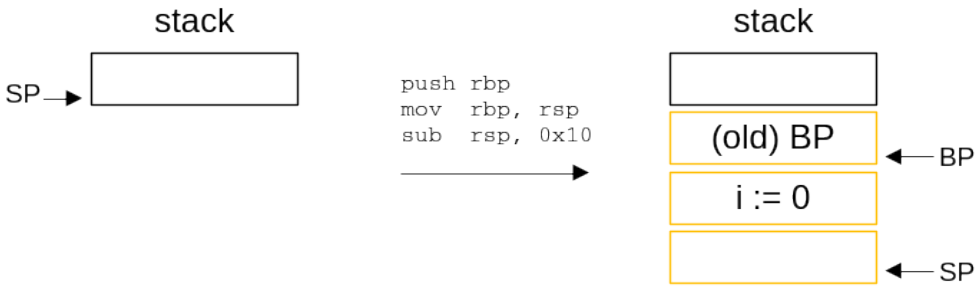


Abbildung 2: Stack-Frame von `main()`

```
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
 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.

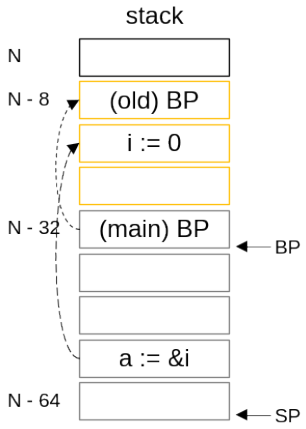


Abbildung 3: Stack-Frame von `foo()`

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, dealloziert 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.

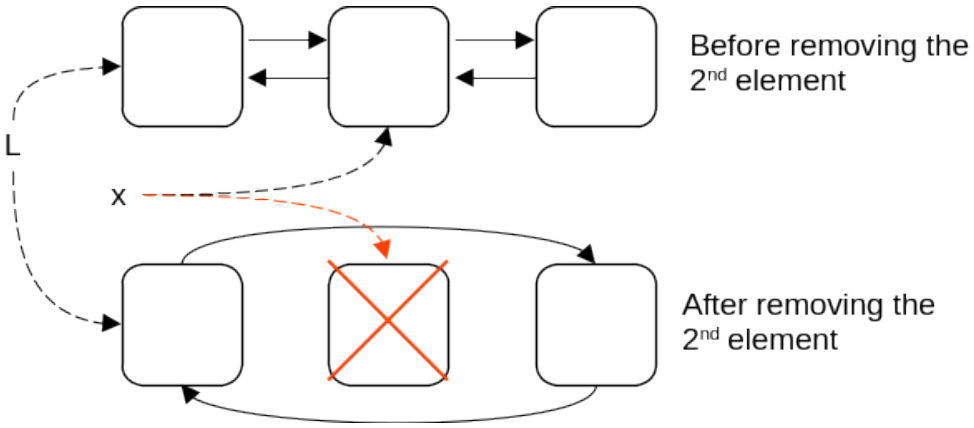


Abbildung 4: Beispiel einer verketteten Liste

```
// 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 deallociert 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 deallociert. 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 Objekts 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 Slices arbeiten und nicht mit rohen Zeigern 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

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.