Hochschule Esslingen
University of Applied Sciences

Fakultät Informationstechnik
im Studiengang Softwaretechnik und Medieninformatik

# Bachelorarbeit

## Entwurf und Implementierung einer hochperformanten, serverbasierten Kommunikationsplattform für Sensordaten im Umfeld des automatisierten Fahrens in Rust

**Michael Watzko**

Sommersemester 2018
14.02.2018 - 22.06.2018

Erstprüfer: Prof. Dr. rer. nat. Dipl.-Inform. Manfred Dausmann
Zweitprüfer: M. Sc. Kevin Erath

IT DESIGNERS GRUPPE

Firma: IT Designers GmbH
Betreuer: M. Sc. Kevin Erath

[git] • master @ 5e03a7d • 12. März 2018

# Sperrvermerk

U SHALL NOT PASS

# Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur und Hilfsmittel erstellt zu haben.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Esslingen, den 12. März 2018 _____

Michael Watzko

# Danksagungen

*„Alle Zitate aus dem Internet sind wahr!“*

Albert Einstein

*„Rust is a vampire language, it does not reflect at all!“*

https://www.youtube.com/watch?v=-Tj8Q12DaEQ

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Der Begriff „autonomes Fahren“ hat spätestens seit den Tesla Autos einen allgemeinen Bekanntheitsgrad erreicht. Damit ein Auto selbstständig fahren kann, müssen erst viele Hürden gemeistert werden. Dazu gehört zum Beispiel das Spur halten, das richtige Interpretieren von Verkehrsschildern und das Navigieren durch komplexe Kreuzungen.

Bevor ein autonomes Fahrzeug Entscheidungen treffen kann, benötigt es ein möglichst genaues Model seines Umfelds. Hierzu werden von verschiedene Sensoren wie Front-, Rück- und Seitenkameras und Abstandssensoren Informationen gesammelt und ausgewertet. Aber vielleicht kann ein Auto nicht immer selbständig genügend Informationen zu seinem Umfeld sammeln? TODO: (huhuhu Server implied huhuhu)

Externe Sensorik könnte Informationen liefern, die das Auto selbst nicht erfassen kann. Ein viel zu schneller Radfahrer hinter einer Hecke in einer unübersichtlicher Kreuzung? Eine Lücke zwischen Autos, die ausreichend groß ist, um einzufahren ohne zu bremsen? Die nächste Ampel wird bei Ankunft rot sein, ein schnelles und Umwelt belastendes Anfahren ist nicht nötig? Ideen gibt es zuhauf.

Aber was ist, wenn das System aussetzt? Die Antwort hierzu ist einfach: das Auto muss immer noch selbstständig agieren können, externe Systeme sollen nur optionale Helfer sein. Viel schlimmer ist es dagegen, wenn das unterstützende System falsche Informationen liefert. Eine Lücke zwischen Autos, wo keine ist; eine freie Fahrbahn, wo ein Radfahrer fährt; ein angeblich entgegenkommendes Auto, eine unnötig Vollbremsung, ein Auffahrunfall. Ein solches System muss sicher sein – nicht nur vor Hackern. Es muss funktional sicher sein, Redundanzen und Notfallsysteme müssen jederzeit greifen.

Aber was nützt die beste Idee, die ausgeklügelte Strategie gegenüber einer undefinierten Situation in der verwendeten Programmiersprache? Wenn nur ein einziges mal vergessen wurde, einen Rückgabewert auf den Fehlerfall zu prüfen? Was nützt es, wenn Strategien für das Freigeben von Speicher in Notfallsituationen einen Sonderfall übersehen haben? Das System handelt total unvorhersehbar.

Was wäre, wenn es eine Programmiersprache geben würde, die so etwas nicht zulässt; die fehlerhaften Strategien zur Compilezeit findet und die Compilation stoppt. Die trotz erzwungener Sicherheitsmaßnahmen, schnell und echtzeitnah reagieren kann und sich nicht

vor Geschwindigkeitsvergleichen mit etablierten, aber unsicheren Programmiersprachen, scheuen muss?

Diese Arbeit soll zeigen, dass Rust genau so eine Programmiersprache ist und sich für sicherheitsrelevante, hoch parallelisierte und echtzeitnahe Anwendungsfälle bestens eignet.

TODO: Y RUST SO FANCY

TODO: MOAR PEP

TODO: ... Programme haben fehler, sind aber nicht Computer schuld, sondern Menschen

TODO: Methoden/Vorgehensweisen können perfektßein, Programmierer leider nicht

TODO: Wäre es nicht toll, in einer Welt zu leben, in der der menschliche Faktor als Fehlerquelle in fktl. sicherheitsrelevanter Software nahezu ausgeschlossen ist?

## 1.2 Projektkontext

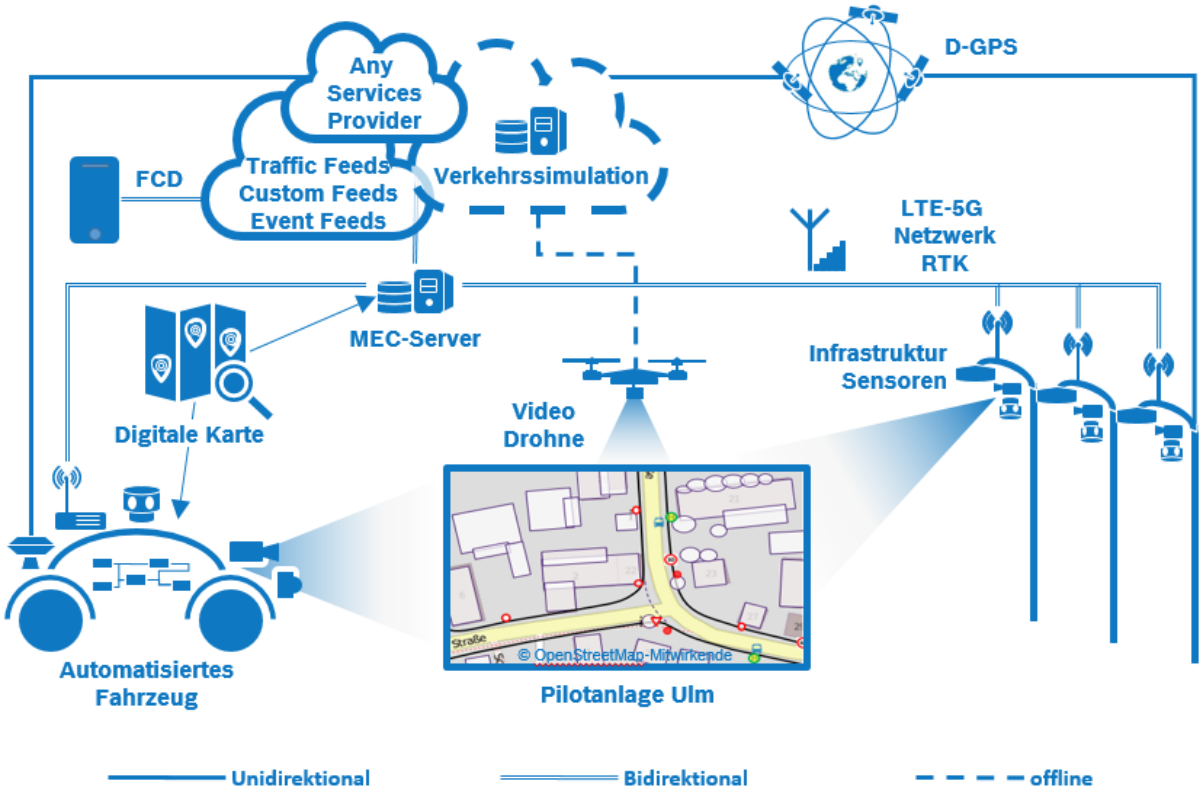


Abbildung 1.1: Übersicht über das Forschungsprojekt [MEC]

Quelle: https://www.uni-due.de/~hp0309/images/Arch_de_V1.png (modifiziert)

Diese Abschlussarbeit befasst sich mit dem Kommunikationsserver von MEC[1]-View. Das MEC-View Projekt wird durch das BMWi[2] gefördert und befasst sich mit der Thematik autonom fahrender Fahrzeuge. Es soll erforscht werden, ob und in wie weit eine durch externe Sensorik geleistete Unterstützung nötig und möglich ist, um in eine Vorfahrtstraße autonom einzufahren.

Das Forschungsprojekt ist dabei ein Zusammenschluss mehrerer Unternehmen mit unterschiedlichen Themengebieten. Die IT-Designers Gruppe beschäftigt sich mit der Implementation des Kommunikationsservers, der auf der von Nokia zur Verfügung gestellten Infrastruktur im 5G Mobilfunk als MEC Server betrieben wird. Erkannte Fahrzeuge und andere Verkehrsteilnehmer werden von den Sensoren von Osram via Mobilfunk an den Kommunikationsserver übertragen. Der Kommunikationsserver stellt diese Informationen

[1] Mobile Edge Computing
[2] Bundesministerium für Wirtschaft und Energie

dem Fusionsalgorithmus der Universität Ulm zur Verfügung und leitet das daraus gewonnene Umfeldmodell an die hochautomatisierten Fahrzeuge von Bosch und der Universität Ulm weiter. Durch hochgenaue, statische und dynamische Karten von TomTom und den Fahrstrategien von Daimler soll das Fahrzeug daraufhin autonom in die Kreuzung einfahren können.

## 1.3 Zielsetzung

Das Ziel ist es, eine alternative Implementierung des MEC-View Servers in Rust zu schaffen. Durch die Garantien (Abschnitt 2.25) von Rust wird erhofft, dass der menschliche Faktor als Fehlerquelle gemindert und somit eine fehlertolerantere und sicherere Implementation geschaffen werden kann.

Eine Ähnlichkeit in Struktur und Architektur zu der bestehender C++ Implementation ist explizit nicht vonnöten. Eventuelle Spracheigenheiten und einzigartige Features von Rust sollen im vollen Umfang genutzt werden können, ohne durch Auferzwungene und unpassende Architekturmuster benachteiligt zu werden. Es ist erwünscht eine kompetitive Implementation in Rust zu schaffen.

## 1.4 Aufbau der Arbeit

Diese Arbeit ist im wesentlichen in die folgenden Themengebiete aufgeteilt: Grundlagen, Anforderungs- und Systemanalyse, Systementwurf und Implementation und Auswertung.

Im Themengebiet Grundlagen sollen wesentliche Bestandteile dieser Arbeit erläutert und erklärt werden. Hierzu zählt zum einen die Programmiersprache Rust in ihrer Entstehungsgeschichte (siehe Abschnitt 2.1), Garantien und Sprachfeatures (siehe Abschnitt 2.25). Zum anderen die hochperformante, serverbasierte Kommunikationsplattform mit ihren Protokollen (ab Kapitel 3) und dem Systemkontext in dem diese betrieben wird.

In der Anforderungs- und Systemanalyse wird der Kontext in dem das System betrieben werden soll genauer betrachtet. Umzusetzende funktionale und nicht-funktionale Anforderungen werden aufgestellt, sowie eine Übersicht von Systemen mit denen das System interagiert wird.

Das Themengebiet Systementwurf und Implementation befasst sich mit dem theoretischen und praktischen Lösen der im vorherigen Kapitel aufgestellten Anforderungen. Aufgrund der Tatsache, dass es sich hierbei um eine alternative Implementation handelt, wird zur bestehenden C++ Implementation Bezug genommen. Auf architektonische Unterschiede im Systementwurf, die sich aufgrund von Sprach- und Bibliotheksunterschiede, werden hier genauer beschrieben.

Zuletzt wird eine Auswertung der Implementation aufgezeigt. TODO: michael.write_more();

# 2 Die Programmiersprache Rust

Rust hat als Ziel, eine sichere (siehe Abschnitt 2.25) und performante Systemprogrammiersprache zu sei. Abstraktionen sollen die Sicherheit, Lesbarkeit und Nutzbarkeit verbessern aber keine unnötigen Performanceeinbußen verursachen (siehe Unterabschnitt 2.25.7).

Aus anderen Programmiersprachen bekannte Fehlerquellen – wie „dangling pointers“, „double free“ oder „memory leaks“ – werden durch strikte Regeln und mit Hilfe des Compilers verhindert (Abschnitt 2.25). Im Gegensatz zu Programmiersprachen, die dies mit Hilfe ihrer Laufzeitumgebung[1] sicherstellen, werden diese Regeln in Rust durch eine statische Lebenszeitanalyse (Abschnitt 2.21) und mit dem Eigentümerprinzip (Abschnitt 2.22) bei der Compilation überprüft und erzwungen.

Diese erlaubt Rust eine zur Laufzeit hohe Ausführgeschwindigkeit zu erreichen. Das Eigentümerprinzip (siehe Abschnitt 2.22) und die Markierung durch von Datentypen durch Merkmale (siehe Abschnitt 2.9) vereinfacht es, nebenläufige und sichere Programme zu schreiben.

Rust hat in den letzten Jahren viel an Beliebtheit gewonnen und scheint dem Anspruch, eine sichere und performante Programmiersprache zu sein, gerecht zu werden:

> *„[..]Leute, die [..] sichere Programmierung haben wollen, [..] können das bei Rust haben, ohne [..] undeterministischen Laufzeiten oder Abstraktionskosten schlucken zu müssen. “* [Lei17, Felix von Leitner in einem Blogeintrag]

> *„[..] Rust makes it safe, and provides nice tools“* [Qui, Folie 130, Federico Mena-Quintero in „Ersetzen von C Bibliotheken durch Rust“]

> *„Rust hilft beim Fehlervermeiden“* [Grü17, Federico Mena-Quintero in einem Interview]

> *„Rust is [..] a language that cares about very tight control“* [fgi17, Diskussion zwischen Programmierern auf Reddit]

[1] u.a. Java Virtual Maschine (JVM), Common Language Runtime (CLR)

## 2.1 Geschichte

In 2006 begann Graydon Hoare die Programmiersprache Rust in seiner Freizeit als Hobbyprojekt zu entwickeln [Rusa]. Als Grund nannte er seine Unzufriedenheit mit der Programmiersprache C++, in der es sehr schwierig sei, fehlerfreien, speichersicheren und nebenläufigen Programmcode zu entwickeln. Zudem beschrieb er C++ als „ziemlich fehlerträchtig“ [Sch13].

Auch Federico Mena-Quintero – Mitbegründer des GNOME-Projekts [Men] – äußerte in einem Interview mit Golem im Juli 2017 seine Bedenken an der Verwendung der „feindseligen“ Sprache C [Grü17]. In Vorträgen vermittelt er seither, wie Bibliotheken durch Implementationen in Rust ersetzt werden können [Qui].

Ab 2009 begann Mozilla die Weiterentwicklung finanziell zu fördern, als einfache Tests und die Kernprinzipien demonstriert werden konnten. Die Entwicklung der Programmiersprache, des Compilers, des Buchs, von Cargo, von crates.io und von weiteren Bestandteilen findet öffentlich einsehbar auf GitHub[2] unter `https://github.com/rust-lang` statt und wird nicht ausschließlich von Mozilla Angestellten koordiniert. Dadurch kann sich jeder an Diskussionen oder Implementation beteiligen, seine Bedenken äußern oder Verbesserungen vorschlagen.

Durch automatisierte Tests (siehe Abschnitt 2.13) in Kombination mit drei Veröffentlichungskanälen („relese“, „stable“ und „nightly“) und „feature gates“ (siehe Unterabschnitt 2.2.2) wird die Stabilität des Compilers und die der Standardbibliothek (Abschnitt 2.18) gewährleistet.

Rust ist wahlweise unter MIT oder der Apache Lizenz in Version 2 verfügbar [Rus18].

## 2.2 Anwendungsgebiet

Das Ziel von Rust ist es, das Designen und Implementieren von sicheren und nebenläufigen Programmen möglich zu machen. Gleichzeitig soll der Spagat geschaffen werden, nicht nur ein sicheres aber lediglich theoretisches Konstrukt zu sein, sondern in der Praxis eine Anwendung zu finden. Als Beweis könnte hierbei auf die Umstellung von Firefox auf Rust und Servo – ein minimaler Webbrowser komplett in Rust geschrieben – verwiesen werden [Rusa].

Interessant ist eine Diskussion von 2009, bei der „sicher aber nutzlos“ und „unsicher aber brauchbar“ Gegenübergestellt wurde. Programmiersprachen scheinen auf der Suche nach

[2] Plattform zum Hosten von git-Repositories inklusive eingebautem Issue-Tracker und Wiki. Änderungen an Quellcode können vorgeschlagen, und durch die Projektverantwortlichen übernommen werden. Bietet auch die Möglichkeit eine kontinuierlichen Integrationssoftware einzubinden, um automatisierte Tests auf momentanen Quellcode und auch für Änderungen auszuführen. Eine vorgeschlagene Änderung kann somit vor Übernahme auf Kompatibilität überprüft werden.

dem nicht existierende „Nirvana“ zu sein, das sowohl sichere als auch brauchbare Programmierung verspricht [Hoa09, ab ca Minute 58:20]. Rust möchte dieses Nirvana gefunden haben.

### 2.2.1 Kompatibilität

Da Rust den LLVM[3]-Compiler nutzt, erbt Rust auch eine große Anzahl der Zielplattformen die LLVM unterstützt. Die Zielplattformen sind in drei Stufen unterteilt, bei denen verschieden stark ausgeprägte Garantien vergeben werden. Es wird zwischen

- „Stufe 1: Funktioniert garantiert“ (u.a. X86, X86-64),
- „Stufe 2: Compiliert garantiert“ (u.a. ARM, PowerPC, PowerPC-64) und
- „Stufe 3“ (u. a. Thumb (Cortex-Microcontroller))

unterschieden [Rusb]. Diese Unterscheidung wirkt sich auch auf die Stabilisierungsphase und Implementation neuer Funktionen aus (Beispiel „128-bit Integer Support“ [wit]).

### 2.2.2 Veröffentlichungszyklus

Es stehen Versionen in drei verschieden Veröffentlichungskanälen zur Verfügung:

- **nightly**: Version die einmal am Tag mit dem aktuellen Stand des Quellcodes gebaut wird. Experimentelle und nicht fertige Features sind hier zwar enthalten, aber hinter „feature gates“ versteckt. Diese können durch ein entsprechendes Attribute (siehe Abschnitt 2.12) geöffnet werden (`#[feature(const_fn)]` ermöglicht die Definition von Konstante Funktionen, Stand 12. März 2018).
- **beta**: Alle sechs Wochen wird die aktuellste Nightly zur Beta befördert und es werden nur noch Fehler aus dieser Version getilgt. Dieser Prozess könnte auch als Reifephase bezeichnet werden.
- **stable**: Nach sechs Wochen wird die aktuellste Beta zur Stable befördert und veröffentlicht. Gleichzeitig wird auch eine neue Beta veröffentlicht.

[3] Früher „Low Level Virtual Machine“ [Wik17], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [LLVa]. Unterstützt eine große Anzahl von Zielplatformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [LLVb].

### 2.2.3 Ökosystem

Mit Rust wird nicht nur eine Programmiersprache, sondern auch ein umfassendes Ökosystem angeboten.

Cargo ist vermutlich das größte angebotene Werkzeug. Es löst Abhängigkeiten auf, indem es auf das öffentliche Verzeichnis unter `https://crates.io` zurückgreift und diese entsprechend herunterlädt und compiliert. Zum jetzigen Zeitpunkt (12. März 2018) sind über 14.000 Crates öffentliche erreichbar und nutzbar. Zudem wird durch Crago eine *Cargo.toml* verlangt, in der Metainformationen einer Crate hinterlegt sind. Dies Umfasst u.a. Name, Version, Autor, Lizenz und Abhängigkeiten.

Eine Crate kan von jedem veröffentlicht werden, insofern derjenige ein GitHub-Konto besitzt, der Name der Crate noch nicht vergeben ist und der Programmcode compiliert. Die API-Dokumentation der jeweiligen Crate wird dabei automatisiert auf `https://docs.rs` veröffentlicht.

Unter `https://www.rust-lang.org` ist die Website von Rust erreichbar und unter `https://doc.rust-lang.org` sowohl die API-Dokumentation der Standardbibliothek als auch as Hauseigene Rust Buch in Version 1 und 2. Die Entwicklung findet dagegen auf GitHub unter `https://github.com/rust-lang` statt.

Kleine Testprogramme und Experimente können auf dem „Spielplatz“ unter `https://play.rust-lang.org` compiliert und ausgeführt werden, ohne lokal etwas zu installieren.

## 2.3 Aufbau eines Projektverzeichnisses

Der Aufbau eines Rust Projektverzeichnis kann zwischen zwei verschiedenen Arten differenziert werden. Zum einen gibt es den klassische Aufbau, in dem lediglich der Programmcode liegt und der Compiler direkt aufgerufen und parametrisiert wird. Zum anderen wird der Aufbau als Crate (siehe Unterabschnitt 2.3.2) empfohlen, da dadurch Abhängigkeiten automatisch aufgelöst werden können aber auch Metainformationen bezüglich des Autors, der Version und der Abhängigkeiten hinterlegt werden müssen. Ein klassischer Aufbau ist dagegen nur selten anzutreffen.

### 2.3.1 Klassisch

```
src/
|-- main.rs
|-- functionality.rs
|-- module/
    |-- mod.rs
    |-- functionality.rs
    |-- submodule/
        |-- mod.rs
        |-- functionality.rs
```

Listing 2.1: Verzeichnisstruktur des Quelltext-Verzeichnisses

Das Quelldatei-Verzeichnis sollte entweder eine *main.rs* für Ausführbare Programme oder eine *lib.rs* für Bibliotheken enthalten. Während der Paketmanager

Cargo eine solche Benennung als Standardkonvention erwartet, kann bei manueller Nutzung des Compilers auch ein anderer Name für die Quelldatei vergeben werden.

Der Compiler startet in der Wurzeldatei und lädt weitere Module, die durch `mod module;` gekennzeichnet sind (ähnlich `# include "module.h"` in C/C++). Ein Modul kann dabei eine weitere Quelldatei oder ein ganzes Verzeichnis sein. Ein Verzeichnis wird aber nur als gültiges Modul interpretiert, wenn sich eine *mod.rs* Datei darin befindet. Um Datentypen und Funktionen aus einem Modul nutzen zu können, ohne dessen kompletten Pfad jedes mal auszuschreiben, müssen sie durch zum Beispiel `use module::functionality::Data;` in dem aktuellen Namensraum bekannt gemacht werden.

Wie bereits angedeutet, wird in Rust nicht eine „Klasse“, Datenstruktur oder Aufzählung pro Datei erwartet, sondern eine Quelldatei entspricht einem Modul. Diese Umfasst in vielen fällen wenige aber mehrere Datenstrukturen, zugehörige Aufzählung und Fehlertypen.

### 2.3.2 Als Crate

Eine „Crate“ (dt. Kiste/Kasten) erweitert den klassischen Aufbau um eine *Cargo.toml* Datei, in der Metainformationen zum Projekt hinterlegt werden. Durch die Benutzung des Werkzeugs „Cargo“ (dt. Fracht/Ladung) können Abhängigkeiten automatisch aufgelöst, heruntergeladen und compiliert werden.

```
1 crate/
2 |-- Cargo.toml
3 |-- src/
4     |-- ...
```

Listing 2.2: Vereinfachte Verzeichnisstruktur einer „crate“

Eine Crate kann entweder ein ausführbares Programm oder eine Bibliothek sein. Davon abhängig is die Wurzeldatei *src/main.rs* (für ein ausführbares Programm) oder *src/lib.rs* (für eine Bibliothek). Mit dem erzeuge einer Crate (`cargo new --bin meinProg` bzw. `cargo new --lib meineBib`) wird auch gleichzeitig git[4] für das Verzeichnis initialisiert.

[4] (dt. Blödmann) ist eine Software zur Versionierungs von Quelldateien, entwickelt von Linus Torvalds 2005. TODO: cite

## 2.4 Hello World

```
1 fn main() {
2     println!("Hello World");
3 }
```

Listing 2.3: „Hello World“ in Rust

Der Programmcode in Listing 2.3 gibt auf der Konsole `Hello World` aus. Das `fn` die Funktion `main` definiert und diese der Startpunkt des Programms ist, wird vermutlich wenig überraschend sein. Viel überraschender ist vermutlich eher das Ausrufezeichen in Zeile 2, da es auf den ersten Blick dort nicht hingehören sollte. In Rust haben Ausrufezeichen und Fragezeichen besondere Bedeutungen, weswegen die Verwendung in Zeile 2 trotzdem richtig ist.

Die Bedeutung des Fragezeichens dient zum schnelleren Auswerten von `Rusult<_, _>`-Werten und wird in Unterabschnitt 2.25.3 genauer erklärt. Das Ausrufezeichen kennzeichnet, dass der ansonsten augenscheinliche Funktionsaufruf tatsächlich ein Aufruf einer Makrofunktion ist.

Eine Funktion `println` gibt es nicht, auch keine aus C erwarteten Funktionen wie `printf`, `fputs` oder `sprintf`. Eine Ausgabe erfolgt durch das `println!` Makro, welches eine String durch Nutzung des `format!` Makros formatiert und erstellt. Daraufhin wird das `writeln!` Makro verwendet, um die formatierte Zeichenkette auf die Standardausgabe zu schreiben.

TODO: nopeZ? Ein kleines Beispiel, viele versteckte Mechaniken zur Laufzeitoptimierung aber trotzdem handlich und leserlich – Rust.

## 2.5 Einfache Datentypen

Die Datentypen in Rust sind im wesentlichen die üblichen Verdächtigen: `bool` für boolische Ausdrücke; `char` für ein einzelnes Unicode Zeichen; `str` für eine Zeichenkette; `u8`, `i8`, `u16`, `i16`, `u32`, `i32`, `u64`, `i64`, (bald `u128`, `i128` [Mat16]) und `usize`, `isize` für ganzzahlige Werte; `f32`, `f64` für Fließkommazahlen in einfacher und zweifacher Präzision; Arrays und Slices [Rusd].

Ganzzahlige Datentypen mit einem führenden `u` sind vorzeichenlos („unsigned“), vorzeichenbehaftete („signed“) sind dagegen mit einem `i` gekennzeichnet. Die darauf folgende Zahl gibt die Anzahl der Bits wieder, die der Datentyp groß ist. Die einzige Ausnahme bildet unter den ganzzahligen Datentypen ist `usize` bzw `isize`, da dieser immer so groß ist, wie die Architektur der Zielplattform. Bei der Indexierung eines Arrays bzw. einer Slice würde ein Datentyp, mit einer global definierten Größe, keinen Sinn ergeben. Das Maximum an adressierbaren Einträgen ist von der Architektur der Zielplattform abhängig. Fließkommazahlen sind mit einem führenden `f` gekennzeichnet.

Durch dieses Schema bei der Bezeichnung der Datentypen wird eine Verwirrung wie zum Beispiel in C unterbunden, wo die primitiven Datentypen ( `short` , `int` , `long` , ..) keine definierte Größe haben, sondern dies abhängig vom eingesetzten Compiler und der Zielplattform ist [DD13, S. 187]. Erst ab C99 wurden zusätzliche, aber optionale, ganzzahlige Datentypen mit bestimmten Größe definiert [GD14, S. 141].

Konstanten können eindeutig einem Datentyp zugewiesen werden, indem dieser angehängt wird: `4711u16` ist vom Datentyp `u16` . Des weiteren dürfen Ziffern durch beliebiges setzen von `_` getrennt werden, um die Lesbarkeit zu erhöhen: `1_000_000_f32` . Eine Schreibweise in Binär ( `0b0000_1000_u8` ), in Hexadezimal ( `0xFF_08_u16` ) oder Oktal ( `0o64_u8` ) ist auch möglich. Konstante Zeichen und Zeichenketten können auch automatisch in Bytes gewandelt werden ( `b'b'` entspricht `0x62_u8` und `b"abc"` entsricht `&[0x61_u8, 0x62_u8, 0x63_u8]` ).

Arrays haben immer eine zur Compilezeit bekannte Größe und müssen auch immer mit einem Wert initialisiert werden (siehe Unterabschnitt 2.25.1). Dynamische Arrays auf dem Stack gibt es (noch? [Rusl]) nicht, stattdessen wird auf die Vektor Implementation der Standardbibliothek verwiesen (siehe Abschnitt 2.18). Die Notation für Arrays ist `[<Füllwert>; <Größe>]` , wobei die Größe ein konstanter Wert sein muss. `[0_u8; 128]` steht demnach für ein 128 Byte langes Array, das mit 0-en vom Datentyp `u8` gefüllt ist.

„Slices“ (dt. Scheiben/Stücke) bezeichnet Rust Referenzen oder Referenzen auf Teilbereiche von Arrays und Slices. Die Größe einer Slice wird dabei mit der Referenz auf die Startposition gespeichert und oft auch „fat pointer“ genannt. Ein zusätzlicher Parameter für die Größe eines Buffers, wie in C üblich, ist somit unnötig. Die Notation ähnelt dabei die eines Arrays, aber ohne Größenspezifikation: `&[<Datentyp>]` . Eine Slice ist immer eine Referenz. Um eine Slice auf ein Array oder eine andere Slice zu erhalten, muss der Start- und Endindex des Teilbereiches angegeben werden. Falls kein Start- oder Endindex angegeben wurde, wird das jeweilige Limit übernommen.

Folgendes Beispiel soll die Notation von Arrays und Slices Beispielhaft verdeutlichen:

```
1 fn main() {
2     let b : [u8; 10] = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
3     for b in &b[2..5] {
4         print!("{}, ", b);
5     }
6 }
```

Listing 2.4: Beispiel eines Arrays und einer Slice

Das in Listing 2.4 gezeigte Programm, gibt auf der Konsole `2, 3, 4,` aus.

## 2.6 Zusammengesetzen Datentypen

Die Programmiersprache Rust kennt neben den einfachen Datentypen (Abschnitt 2.5) weitere Möglichkeiten Daten zu organisieren:

- ein Tupel, das mehrere Werte namenlos zusammenfasst: `(f32, u8)` : `a.0 = 1.0_f32`,
- eine Datenstruktur, die wie in C Datentypen namenbehaftet zusammenfasst: `struct Punkt { x: f32, y: f32 }` : `p.x = 1.0_f32`,
- und Aufzählungen: `enum Bildschirm { Tv, Monitor, Leinwand }`.

Im Vergleich zu C kann ein Eintrag in einem `enum` gleichzeitig Daten wie eine Datenstruktur oder ein Tupel halten, oder lediglich einen Ganzzahlwert repräsentieren. Mit dem `type` Schlüsselwort können Aliase erstellt oder im Falle von FFI (siehe Abschnitt 2.26) aufgelöst werden: `type Vektor = (f32, f32);` Felder einer Struktur können zudem mit `pub` oder `pub(crate)` gekennzeichnet werden (siehe Abschnitt 2.10).

Seit Version 1.19 ist auch der Datentyp `union` in Rust verfügbar [Rus17]. Eine `union` kann aber nur in `unsafe`-Blöcken verwendet werden, da der Compiler eine Ordnungsgemäße Nutzung nicht überprüfen kann. Für diese Abschlussarbeit hat der Datentyp aber keine Relevanz und wird daher nicht weiter erwähnt.

## 2.7 Funktionen, Ausdrücke und Statements

Funktionen werden durch `fn` gekennzeichnet, gefolgt mit dem Funktionsnamen, der Parameterliste und zuletzt der Datentyp für den Rückgabewert. Selbst wenn kein expliziter Rückgabetyp angegeben wird, wird formal `()` zurück gegeben; `()` entspricht etwa `void` aus bekannten Programmiersprachen. Die Parameterliste unterscheidet sich von bekannten Programmiersprachen wie C und Java, indem zuerst der Variablenname und darauf folgend der Datentyp notiert wird.

```
1 fn add(a: f32, b: f32) -> f32 {
2     a + b
3 }
```

Listing 2.5: Beispiel einer Funktion

Obwohl in Zeile 2 von Listing 2.5 kein `return` zu sehen ist, wird trotzdem das Ergebnis der Addition zurückgegeben. Dies liegt daran, da in Rust vieles ein Ausdruck ist und somit einen Rückgabewerte liefert [Ruse]. Auch ein if-else ist ein Ausdruck und kann einen Rückgabewert haben. Ein bedingter Operator (?:) is somit unnötig, da stattdessen ein

if-else verwendet werden kann: `let a = if b { c } else { d };`. Auch eine Zeile mit einen Semikolon hat formal einen Rückgabewert: `()`.

## 2.8 Implementierung einer Datenstruktur

Zu einer Datenstruktur oder Aufzählung kann ein individuelles Verhalten implementiert werden. In dieser Kombination ähneln diese Konstrukte sehr einer Klasse aus bekannten objektorientierten Programmiersprachen, wie zum Beispiel Java, C# oder C++ (siehe auch Abschnitt 2.24).

Einen Konstruktor gibt es jedoch nicht; lediglich die Konvention, eine statische Funktion `new` stattdessen zu verwenden [Rusf]:

```
1  struct Punkt {
2      x: f32,
3      y: f32,
4  }
5
6  impl Punkt {
7      pub fn new(x: f32, y: f32) -> Punkt {
8          Punkt { x, y }
9      }
10 }
```

Listing 2.6: Punkt Datenstruktur mit einem „Konstruktor“

In seltenen Fällen wird auch `Default` implementiert (siehe Abschnitt 2.9), wodurch eine statische Funktion `default()` als Konstruktor ohne Parameter bereitgestellt wird.

Da eine Funktionsüberladung nicht möglich ist, soll bei weiteren Konstruktoren ein sprechender Name verwendet werden. Der `Vec<_>` der Standardbibliothek (siehe Abschnitt 2.18) bietet zum Beispiel zusätzlich `Vec::with_capacity(capacity: usize)` an, um einen Vektor mit einer bestimmten Größe zu initialisieren.

Für Funktionen können auch die Zugriffsmodifikatoren festgelegt werden (siehe Abschnitt 2.10).

## 2.9 Generalisierung durch Traits

Ähnlich wie Java oder C# bietet Rust durch einen eigenen Typ die Möglichkeit, ein gewünschtes Erscheinungsbild zu generalisieren, ohne gleichzeitig eine Implementation vorzugeben. Im Rust wird dieser Typ „Trait“ (dt. Merkmal) genannt.

Für Merkmale werden Funktionen in einem entsprechenden `trait <Name> { }`-Block ohne Rumpf deklariert. Optional kann auch ein Standardrumpf implementiert werden, der bei einer Spezialisierung überschrieben werden darf.

Merkmale unterscheiden sich in ihrer Handhabung gegenüber anderen Datentypen, da sie im allgemeinen keine bekannte Größe zur Compilezeit haben. Während dies in Programmiersprachen wie Java und C# automatisch durch TODO: vtables umgesetzt wird (TODO: virtuelle methoden, vergleich c++, speicher, indirekt), muss in Rust dies der Entwickler entscheiden. TODO: dabei dabei dabei dabei dabei dabei Dabei gibt es mehrere Vorgehensweisen:

- Leihen mittels Referenz: `fn foo(bar: &Bar)` oder `fn foo(bar: &mut Bar)` – ein Unterschied zu anderen Datentypen ist nicht zu erkennen.
- Datentyp, der das geforderte Merkmal implementiert, auf den Heap verschieben und die TODO: Eigentümerschaft über den Heapbereich übertragen (TODO: resultiert in einfacher Pointer-übergabe): `fn foo(bar: Box<Bar>)` (auch „Trait-Object“ genannt)
- Als TODO: spezialisierte Funktion: `fn foo<T: Bar>(bar: T)` (auch für Felder in Aufzählungen oder Datenstrukturen möglich)

Eine Deklaration `fn foo(bar: Bar)` für das Merkmal `Bar` ist nicht möglich, da zur Compilezeit eine eindeutige Größe nicht bekannt ist. Der zu reservierende Speicher für die Variable kann nicht bestimmt werden.

Eine spezialisierte Funktion verhält sich ähnlich wie eine TODO: Templateklasse in C++. Der Compiler erzeugt für jede Spezialisierung eine Kopie der Funktion und setzt den entsprechenden Typ ein. Dies ermöglicht auch Optimierungen der Funktion für den eingesetzten Typ TODO: cite?, vergrößert aber das Compilat.

Im folgenden werden oft anzutreffende und wichtige Merkmale aus der Standardbibliothek kurz erläutert:

- `Send`: Markiert einen Datentyp als zwischen Threads übertragbar. Automatisch für alle Datentypen bei denen auch alle beinhalteten Datentypen von Typ Send sind TODO: rly? cite?. Manuelle Implementation ist TODO: unsafe. `!Sized` verhindert dagegen, dass ein Wert zu anderen Threads übertragen werden darf. Somit können ansonsten textuelle Beschränkungen, wie für der OpenGL-Kontext, durch den Compiler überprüft und erzwungen werden. CTODO: Beispiel OpenGL?
- `Sync`: Markiert einen Datentype als Thread sicher, d.h. mehrere Threads dürfen gleichzeitig lesend darauf zugreifen, während `!Sync` dies verhindert. TODO: automatisch? manuell? Verlangt, dass alle beinhalteten Datentypen auch `Sync` sind.
- `Sized`: Verlangt eine zu Compilezeit bekannte Größe. `!Sized` erlaubt dagegen eine unbekannte Größe zur Compilezeit.

- `Copy`: Markiert einen Datentyp, der durch einfaches Speicherkopieren TODO: memcpy vervielfacht werden kann (TODO: bsp skalare datentypen). Verlangt, dass alle beinhalteten Datentypen auch `Copy` sind. Alle einfachen Datentypen sind bereits `Copy`.
- `Clone`: Markiert einen Datentyp der vervielfacht werden kann, dabei jedoch Laufzeitkosten verursacht. Stelle eine Funktion zum clonen bereit, die explizit aufgerufen werden muss (zbsp Zähler inkrementieren). Verlangt, dass alle beinhalteten Datentypen auch `Clone` sind. Alle einfachen Datentypen sind bereits `Clone`.
- `Debug` und `Display`: Verlangt die Implementation von Funktionen um als Text dargestellt zu werden, entweder mit mehr Zusatzinformationen (`Debug`) oder TODO: schön (`Display`). Verlangt, TODO: ..
- `Deafult`: Verlangt die Implementation einer statische Methode `default()`, die wie ein leerer Standardkonstruktor von Java oder C# wirkt. Verlangt, dass alle beinhalteten Datentypen auch `Deafult` sind.
- `PartialEq`: Verlangt die Implementation einer Funktion um mit Instanzen des gleichen Typs verglichen werden zu können. Im Vergleich zu `Eq` erlaubt `PartialEq`, dass Typen keine volle Äquivalenzrelation haben. Dies ist zum Beispiel für den Vergleich von Fließkommazahlen wichtig, da laut IEE754 `Nan` ungleich zu allem ist, auch zu sich selbst (`Nan != Nan`) [Wik18b][BO17, S. 272-275][Rusi].
- `Eq`: Erlaubt dem Compiler einen Vergleich auf Bit-Ebene durchzuführen, ungeachtet des Datentyps [Rusj].
- `PartialOrd`: Verlangt die Implementation einer Funktion um mit Instanzen des gleichen Typs sortieren zu können. Erlaubt aber auch, dass Werte zueinander nicht sortierbar sind. Dies ist zum Beispiel für Fließkommazahlen wichtig, da laut IEE754 `Nan` nicht sortiert werden kann (weder `Nan <= 0` noch `Nan > 0` ergibt `true`) [Wik18b][BO17, S. 275-277][Rusk].
- `Ord`: Erzwingt im Gegensatz zu `PartialOrd`, dass Werte immer sortierbar sind.
- `Drop`: Verlangt die Implementation einer Funktion, die kurz vor der Speicherfreigabe eines Objekts aufgerufen wird (ähnlich Destruktor aus C++).

Mit dem Attribute `#[derive(..)]` ist eine automatisierte Implementation genannter Merkmale oft möglich, insofern die jeweiligen Bedingungen erfüllt sind. So kann im allgemeinen `#[derive(Clone)]` genutzt werden um eine Datenstruktur oder eine Aufzählung automatisch klonbar zu machen oder `#[derive(Debug)]` um automatisch alle Felder in Text wandeln zu können. Ein ergonomisches aber auch Fehler reduzierendes Feature.

## 2.10 Zugriffsmodifikatoren

Zugriffsmodifikatoren erlauben es in Rust, Module, TODO: Re-exporte / use, Datenstrukturen, Aufzählungen, Merkmale und Funktionen gegenüber Nutzern einer Crate und anderen Modulen sichtbar zu machen. Der Standardmodifikator limitiert die Sichtbarkeit auf das Modul, in dem die Deklaration stattgefunden hat und wird durch keine Nennung eines Zugriffsmodifikators erreicht. Um die Sichtbarkeit auf die gesamte Crate zu erhöhen, wird ein `pub(crate)` vorangestellt. Mit `pub` ist die Deklaration für alle sichtbar.

## 2.11 Musterabgleich

Der `match` Ausdruck ist ein sehr mächtiges Werkzeug in Rust und entspricht einem stark erweiterten `switch` aus Programmiersprachen wie C, Java oder C#. Mit ihm ist es möglich einen Wert eine Aufzählung aufzulösen und auf eventuell beinhaltete Werte zuzugreifen oder zu konsumieren.

```
1 fn main() {
2     let value : Option<&str> = Some("text");
3     match value {
4         Some(value) => println!("Wert ist: {}", value),
5         None => println!("Kein Wert"),
6     };
7 }
```

Listing 2.7: Kompletter `match` Ausdruck

In Zeile 3 von Listing 2.7 wird `value` aufgelöst. In dem Beispiel ist `value` aus Zeile 2 und 3 `Some("text")`, weswegen Zeile 4 ausgeführt, `value` konsumiert und `Wert ist: text` auf der Konsole ausgegeben wird.

Wenn nur ein konkreter Fall von Bedeutung ist, kann dies in der verkürzten Schreibweise `if let` notiert werden:

```
1 fn main() {
2     let mut value : Option<u32> = Some(4);
3     if let Some(ref mut value) = value {
4         *value += 1;
5     }
6     println!("{:?}", value); // "Some(5)"
7 }
```

Listing 2.8: Vereinfachte `if let` Ausdruck

TODO: Dekonstruktion von Werten mittels Pattern Matching

Ein weiterer Unterschied von Listing 2.8 gegenüber Listing 2.7 ist in Zeile 3 das Schlüsselwort `ref`, wodurch der Konsum des Wertes verhindert wird. Das Schlüsselwort `mut` erlaubt zudem eine Änderung des Wertes, weswegen `value` in Zeile 4 vom Typ `&mut u32` ist. Die Dereferenzierung mit Addition wird somit ermöglicht.

Als Wildcard für sowohl nicht benötigte Werte, als auch alle weiteren Fälle kann `_` verwendet werden: `if let Some(_) = value { println!("It's something!"); }`

Weitere Möglichkeiten, Muster zu erkennen sind ab Seite 221 in [BO17] in detaillierter Ausführung zu finden. Dazu gehören unter anderem die „guard expression“, „bindings“ und „ranges“. Aufgrund des Umfangs wird hier auf eine weitere Vertiefung verzichtet.

## 2.12 Attribute

TODO: Unterscheidung Methode, Datentyp oder main.rs/lib.rs

## 2.13 Automatisierte Tests

TODO: arg1

## 2.14 Namens- und Formatierkonvention / Styleguide

```
1 enum MY_ENUM {
2     AN_ENTRY,
3     ANOTHER_ENTRY,
4 }
```

Listing 2.9: Beispiel für nicht Styleguide konformer Aufzählung

[warning]: **type ‘MY_ENUM‘ should have a camel case name such as ‘MyEnum‘**
[warning]: **variant ‘AN_ENTRY‘ should have a camel case name such as ‘AnEntry‘**
[warning]: **variant ‘ANOTHER_ENTRY‘ should have a camel case name such as ‘AnotherEntry‘** warning: unused variable: ‘a‘ [Rusc]

## 2.15 Formatierung

`format!` all se things

TODO: formatierung

TODO: let, optionaler datentyp, macros, generics, () statt void

TODO: official format/naming convetion, use, function, macro

TODO: Variables, Structs, Enums, Traits

TODO: type safety langauge

TODO: Rust -> MIR -> assembler

TODO: MIR/assemblerbeispiele?
[BO17]

TODO: pattern matching

## 2.16 Niemals nichts und niemals unbehandelte Ausnahmen

TODO: core, datatypes, arrays slices, no null „billion dollar mistake“

Rust kennt `null` (-Pointer) nicht, bietet aber in `core` (??) `Option<_>` als Ersatz an. Dieser Datentyp erzwingt eine Prüfung vor dem Zugriff auf den optionalen Wert.

TODO: IMMER INITIALISIERT, sonst kein zugriff erlaubt

Für die Fehlerbehandlung wird nicht auf ein Exception-Handling zurückgegriffen, sondern ein eigener Datentyp angeboten, der entweder den Rückgabewert enthält, oder aber einen Fehler: `Result<_, _>` (siehe Unterabschnitt 2.25.3).

Durch den TODO: Fragenzeichenoperator kann trotzdem ein ähnliches Verhalten wie beim auftreten einer Ausnahme in Java oder C++ erzielt werden. TODO: example?

TODO: ref if let Ok(_)

## 2.17 Besorgter Compiler

TODO: many warnings

## 2.18 Standardbibliothek

Die Rust Community ist darum bemüht, die Standardbibliothek sehr leichtgewichtig zu halten. Nicht eindeutig als fundamental eingestufte Funktionalität wird lieber als Crate auf `https://crates.io` angeboten, anstatt in die Standardbibliothek übernommen zu werden TODO: prove via cite. Mit dieser Entscheidung soll auch eine Entwicklung unabhängig von den Releasezyklen von Rust ermöglicht werden TODO: cite.

Die Standardbibliothek wird selbst als eine Crate (siehe Unterabschnitt 2.3.2) zur Verfügung gestellt, auf die standardmäßige eine Abhängigkeit besteht. Für die Verwendung von Rust im Embedded-Bereich, kann diese Abhängigkeit, die für Microcontroller sehr umfangreich ist, durch `#![no_std]` unterbunden werden. Daraufhin sind nur noch die in der `core` Crate zur Verfügung gestellten, fundamentalen Sprachkonstrukte verwendbar.

Da diese Abschlussarbeit keine Anwendung im Embedded-Bereich findet, wird der volle Funktionsumfang der Standardbibliothek genutzt. Wichtige aber auch Bekannte Datentypen sind:

- **std::vec::Vec**: Ein Vektor (wie eine Liste), bei dem die Werte in einem dynamisch groß allokierten Speicherbereich auf dem Heap liegen. Ist **der** Ersatz für dynamische Arrays. TODO: graph stack(pointer) + heap(vec+elements) + stack(slice)

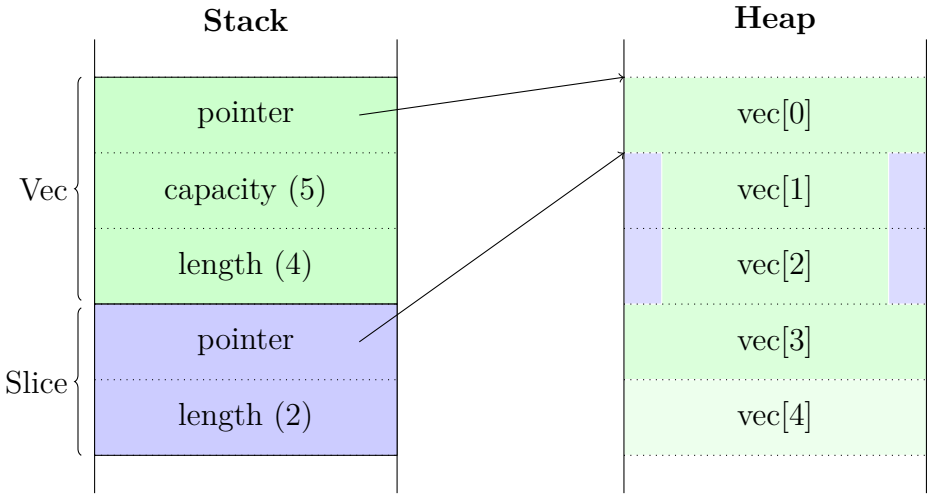


Abbildung 2.1: Speicherlayout Vec und Slice [BO17, S. 63]

- **std::boxed::Box**: Ein Speicherbereich auf dem Heap für einen beliebigen Datentyp.
- **std::string::String**: Eine UTF-8 encodierte, vergrößer- und verkleinerbare Zeichenkette auf dem Heap.
- TODO: more?!TODO: containers, collections, rc, arc, mutex, rwlock, platform abstractions: threads, tcp, udp TODO: println!, writeln!, format!

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: static type system with local type inference

TODO: per default: stack

TODO: formatting rules

## 2.19 Alles hat einen Rückgabewert

TODO: () ??, Statement vs

## 2.20 use mod pub

## 2.21 Speichermanagement

Rust benutzt ein „statisches, automatisches Speicher Management – keinen Garbage Collector“ [Gil17]. Das bedeutet, die Lebenszeit einer Variable wird statisch während der Compilezeit anhand des Geltungsbereichs ermittelt (siehe Abschnitt 2.21). Durch diese statische Analyses findet der Compiler heraus, wann der Speicher einer Variable wieder freigegeben werden muss. Dies ist nämlich genau dann, wenn der Geltungsbereich des Eigentümers zu Ende ist. Weder ein GC[5], der dies zur Laufzeit nachverfolgt, noch ein manuelles eingreifen durch den Entwickler (zum Beispiel durch `free(*void)`, wie in C/C++ üblich) nötig. Der menschliche Faktor als Fehlerquelle wird wieder unterbunden, ohne Laufzeitkosten zu erzeugen.

```
1 fn main() { // neuer Scope
2     let mut a = Box::new(5); // 5 kommt auf den Heap
3     { // neuer Scope
4         let b = Box::new(10); // 10 kommt auch auf den Heap
5         *a += *b; // a ist nun 15
6     } // Lebenszeit von b zuende, Speicher wird freigeben
7     println!("a: {}", a); // Ausgabe: "a: 15"
8 } // Lebenszeit von a zuende, Speicher wird freigegeben
```

Listing 2.10: Geltungsbereich von Variablen

[5] Garbage Collector

Als Alternative kann eine Variable oder Datenstruktur auch vorzeitig durch Aufruf von `std::mem::drop(_)` freigegeben werden. Die optionalen Implementation von `std::op::Drop` TODO: trait? ref? kommt der Implementation des Destruktors aus C++ gleich.

TODO: ßtatic automatic memory management no garbage collection

TODO: while compiling, does not compile on error / unprovable code, trait Drop

TODO: autodrop, auto file close

## 2.22 Eigentümer- und Verleihprinzip

Bereits 2003 beschreibt Bruce Powel Douglass im Buch „Real-Time Design Patterns“, dass „passive“ Objekte ihre Arbeit nur in dem TODO: Thread-Kontext ihres „aktiven“ Eigentümers tätigen sollen [Dou03, S. 204]. In dem beschriebenen „Concurrency Pattern“ wird eine klare Zuordnung getätigt, welche Objekte welchem anderen Objekt als Eigentümern zugeordnet sind, um eine sicherere Nebenläufigkeit zu schaffen TODO: shit.

Diese Philosophie setzt Rust direkt in der Sprache um, so darf eine Variable immer nur einen Eigentümer haben. Zusätzlich zu einem immer eindeutig identifizierbaren Eigentümer für eine Variable, kann diese auch ausgeliehen werden; entweder exklusiv mit sowohl Lese- als auch Schreiberlaubnis, oder mehrfache mit nur Leseerlaubnis.

Eigentümerschaft kann auch übertragen werden, der vorherige Eigentümer kann danach nicht mehr auf den Wert zugreifen.

Die Garantie nur einen Eigentümer, eine exklusive Schreiberlaubnis oder mehrere Leseerlaubnisse auf eine Variable zu haben, wird durch die statische Lebenszeitanalyse garantiert (siehe Abschnitt 2.21). Da dies zur Compilezeit geschieht, ist eine Überprüfung zur Laufzeit nicht nötig, weshalb diese Philosophie keinen negativen Einfluss auf die TODO: Ausführgeschwindigkeit hat.

TODO: Split example, explain more

```
1  fn main() {
2      let mut a = Box::new(1.0_f32); // Eigentümer der neuen
3                                     // Heap-Variable ist a
4
5      {
6          let b = &a; // a wird an b mit Lesezugriff verliehen
7          let c = &a; // a wird an c mit Lesezugriff verliehen
8
9          println!("a: {}", a); // "a: 1"
10         println!("b: {}", b); // "b: 1"
```

```
11          println!("c: {}", c); // "c: 1"
12
13          // let d = &mut a; // Nicht erlaubt: Es existieren
14                             // verliehene Lesezugriffe
15
16          // *a = 7_f32; // Nicht erlaubt: Es existieren
17                         // verliehene Lesezugriffe
18
19      } // Ende von b und c, a nicht mehr verliehen
20
21      {
22          let e = &mut a; // Leihe a mit Schreiberlaubnis
23          **e = 9_f32;    // Setze Inhalt von a
24
25          // println!("a: {}", a); // Nicht erlaubt: exklusiver
26                                   // Zugriff an e verliehen
27
28          println!("e: {}", e); // "e: 9"
29
30      } // Ende von e, a nicht mehr verliehen
31
32      println!("a: {}", a); // "a: 9"
33      let f = a; // Neuer Eigentümer der Heap-Variable ist f
34      // *a = 12.5_f32; // Nicht erlaubt: Nicht mehr Eigentümer
35      // *f = 12.5_f32; // Nicht erlaubt: f nicht änderlich
36      println!("f: {}", f); // "f: 9"
37  }
```

Listing 2.11: Eigentümer und Referenzen von Variablen

TODO: missing move? orly

## 2.23 Rust als funktionale Programmiersprache

TODO: functional programming -> no global state, no exceptions, find literature TODO: prove via code

## 2.24 Rust als Objekt-Orientierte Programmiersprache

TODO: trait TODO: prove via design patterns, a few? from faq:: Is Rust object oriented? It is multi-paradigm. Many things you can do in OO languages you can do in Rust, but

not everything, and not always using the same abstraction you're accustomed to.

## 2.25 Versprechen von Rust

> *„It's not bad programmers, it's that C is a hostile language“* [Qui, S. 54]
>
> *„I'm thinking that C is actively hostile to writing and maintaining reliable code“* [Qui, S. 129]

Im Gegensatz zu anderen Programmiersprachen wirbt Rust mit Versprechen und Garantien, die dafür sorgen sollen, Fehler zu vermeiden. In einer perfekten Welt wären viele dieser Maßnahmen nicht nötig, aber leider sind Programmierer auch nur Menschen. Menschen machen Fehler. Deswegen hat Rust einige Interessante Mechaniken eingeführt, bekannte Fehlerquellen zu unterbinden und erzwingt die Einhaltung indem andere Vorgehensweisen meist ausgeschlossen werden.

Dieses Kapitel beschäftigt sich mit den wichtigsten und bekanntesten dieser Mechaniken.

### 2.25.1 Kein undefiniertes Verhalten

Bei der Entwicklung von Rust wird ein sehr großer Fokus darauf gelegt, keine undefinierten Zustände zu erlauben. Daher ist es normalerweise nicht möglich, ein undefiniertes Verhalten oder einen undefinierten Zustand zu erzeugen. Die Ausnahme macht dabei die Nutzung von `unsafe`, für zum Beispiel FFI (siehe Abschnitt 2.26). Für diese Fälle gibt es eine überschaubare Liste von Szenarien, aus denen ein undefinierter Zustand bzw. undefiniertes Verhalten resultieren kann [Rusg].

Als einfaches Beispiel eines undefinierten Zustandes in C ist eine Variable, die deklariert wurde, aber der noch kein Wert zugewiesen wurde. In manchen Szenarien hat die Variable dann den Wert der in diesem Moment an der entsprechenden Stelle im Speicher stand, in anderen Szenarien wird der Speicher vom Betriebssystem, Allokator oder von vom Compiler eingefügten Befehlen mit 0en gefüllt – eine sichere Aussage ist nicht möglich.

Rust dagegen lässt keinen zugriff auf Variablen zu, die nicht zuvor initialisiert wurden [BO17, S. 126]. Der Compiler stoppt mit einem Fehler: „**error[E0381]: use of possibly uninitialized variable: 'a'**“.

### 2.25.2 Keine vergessene Null-Pointer Prüfung

> *„I call it my billion-dollar mistake. It was the invention of the null reference in 1965“* [Hoa09, Tony Hoare, QCon Software Konferenz in London, 2009]
> TODO: cant find moment in video / presentation of this qutoe!?

Wie in Abschnitt 2.16 beschrieben, kennt Rust keinen `NULL`-Pointer. Daher ist es auch nicht möglich, durch Nachlässigkeit auf den falschen Speicher zuzugreifen, da eine Referenz immer gültig ist. Für fälle, in denen es eventuell keinen Wert gibt, bietet Rust stattdessen den `Option<_>` Datentyp an. `Option<_>` ist eine Aufzählung, die entweder `None` ist, oder `Some(_)` mit einem Wert. Auf den Wert kann nicht zugegriffen werden, ohne zu prüfen, ob wirklich ein `Some(_)` vorliegt. Dies kann durch `match` oder verkürzt durch ein `if let Some(wert) = optional { /* tu etwas mit wert */ }` geschehen (siehe Abschnitt 2.11).

### 2.25.3 Keine vergessene Fehlerprüfung

```
1 #include <stdio.h>
2
3 void main(void) {
4    FILE *file = fopen("private.key", "w");
5    fputs("42", file);
6 }
```

Listing 2.12: Negativbeispiel: Fehlende Fehlerprüfung in C

In Listing 2.12 sind mindestens zwei Fehler versteckt, die aber keinen Compileabbruch auslösen, sondern sich zur Laufzeit zeigen können. Der erste Fehler ist eine fehlende Überprüfung des Rückgabewertes von `fopen` in Zeile 4, da dieser `null` ist, falls das Öffnen der Datei fehlgeschlagen ist. Der Versuch in die Datei zu schreiben in Zeile 5 kann daraufhin in einen Speicherzugriffsfehler resultieren und das Programm abstürzen lassen. TODO: vergleich Java/C++(++) exceptions (vergessen von fehlerbehandlung in c++(++) trotzdem möglich)

In Rust wird weder eine Ausnahme geworfen, noch ein Rückgabewert zurück gegeben, der ohne Prüfung verwendet werden kann:

```
1  use std::fs::File;
2  use std::io::Write;
3
4  fn main() {
5      match File::open("private.key") {
6          Err(e) => println!("Fehler aufgetreten: {}", e),
7          Ok(mut file) => {
8              let _ = write!(file, "42");
9          }
10     }
11 }
```

Listing 2.13: Positivbeispiel: Keine fehlende Fehlerprüfung in Rust

Der Rückgabewert von `File::open("private.key")` in Zeile 5 von Listing 2.13 ist vom Typ `Result<File, Error>`. Auf den eigentlichen Rückgabewert `File` kann nicht ohne eine Fehlerprüfung zugegriffen werden, da dies `Result` verhindert. Eine Fehlerprüfung kann wie in Zeile 5 mit einem `match` passieren, oder auch mit anderen Funktionen wie `.unwrap()`, `.unwrap_or()` ... https://doc.rust-lang.org/std/result/enum.Result.html die dann aber eine `panic!` TODO: ref auslösen, falls ein Fehler vorliegt – somit wird ein undefiniertes Verhalten unterbunden TODO: ref.

TODO: Der zweite Fehler...? Durch die Lebenszeitanalyse TODO: ref in Rust ist der Geltungsbereich der `File` Variable bekannt, deshalb wird in dem Beispiel in Rust in Listing 2.13 die Datei auch wieder ordnungsgemäß geschlossen, während dies im C Beispiel in Listing 2.12 nicht der Fall ist.

TODO: explain result

TODO: explain shorthand '?'

### 2.25.4 No dangling pointer

TODO: src https://www.youtube.com/watch?v=d1uraoHM8Gg

### 2.25.5 Sichere Nebenläufigkeit

TODO: „Safety is invisible“ [BO17, S. 41]

TODO: Send, Sync, No dataraces weil Ownership Abschnitt 2.22, Channel, Mutex, RwLock

TODO: Datarace benötigt immer einen schreibenden + min einen lesenden gleichzeitig

### 2.25.6 Slice out of index

### 2.25.7 Zero Cost Abstraction

Trotz der vielen verwendeten Abstraktionen möchte Rust dadurch möglichst keine weitere Laufzeitkosten erzeugen.

Der `Option<_>` Datentyp kann zum Beispiel tatsächlich als Pointer dargestellt werden, der bei `NULL` `None` ist und ansonsten `Some(_)` [BO17, S. 100]. Somit wird eine Überprüfung erzwungen, ohne dabei Laufzeitkosten erzeugt zu haben.

TODO: nothing special? Ein weiteres Beispiel ist der atomare Referenzzähler von `Arc<_>`. Der Zähler ist im Heap direkt vor dem beinhalteten Wert und nicht in einem extra Speicherbereich TODO: cite. Ein weiteren indirekten Speicherzugriff mit Laufzeitkosten wird somit verhindert.

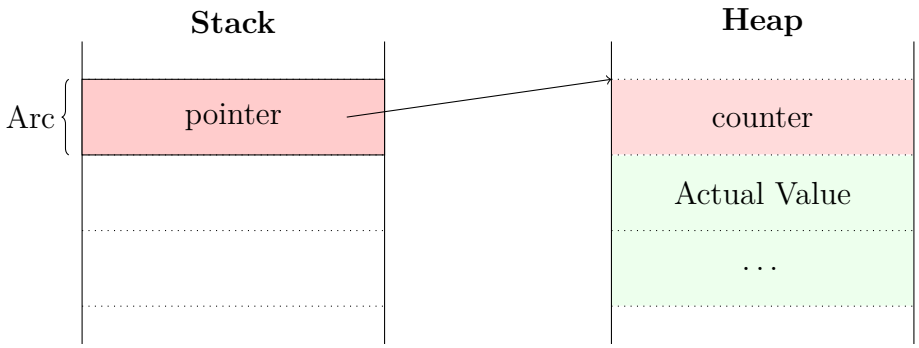


Abbildung 2.2: Speicherlayout Arc TODO: cite

## 2.26 Einbinden von Bibliotheken

#### Externe Datentypen

Rust bietet durch das Foreign Function Interface[6] die Möglichkeit, andere (System-)Bibliotheken einzubinden. Entsprechende Strukturen und Funktionen werden durch einen `extern`-Block oder im Falle von Strukturen stattdessen optional mit einem `#[repr(C)]` gekennzeichnet.

In einem Beispiel, soll die Nutzung von Foreign Function Interface demonstriert werden.

[6] Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [Wik18a]

```c
typedef struct PositionOffset {
    long position_north;
    long position_east;
    long *std_dev_position_north; // OPTIONAL
    long *std_dev_position_east;  // OPTIONAL

    // ...
} PositionOffset_t;
```

Listing 2.14: Ausschnitt von „PositionOffset“ TODO: ref mecview lib in C, autgen ASN

Die Struktur in Listing 2.14 muss zur Nutzung in Rust zuerst bekannt gemacht werden. Dabei gibt es mehrere Möglichkeiten:

1. Falls der Aufbau der Struktur nicht von Bedeutung ist, kann es ausreichen, den Datentyp lediglich bekannt zu machen: `#[repr(C)] struct PositionOffset;`
2. Der Aufbau ist wie bei Punkt 1 unbedeutend, es soll aber ausdrücklich auf einen externen Datentyp hingewiesen werden: `extern { type PositionOffset; }` [Rusm] (TODO: nightly)
3. Der Inhalt der Struktur ist von Bedeutung, da darauf zugegriffen werden soll oder in Rust eine Instanz erzeugbar sein soll. In diesem Fall muss die Struktur komplett wiedergegeben werden:

```rust
use std::os::raw::c_long;

#[repr(C)]
pub struct PositionOffset {
    pub position_north: c_long,
    pub position_east: c_long,
    pub std_dev_position_north: *mut c_long,
    pub std_dev_position_east: *mut c_long,
    // ...
}
```

Listing 2.15: Ausschnitt von „PositionOffset“ TODO: ref mecview lib in Rust

In Listing 2.15 ist die Struktur „PositionOffset“ definiert, die durch das Attribut `#repr(C)` wie eine C-Struktur im Speicher organisiert wird. Somit ist sie kompatibel zu der C-Struktur aus Listing 2.14.

Wenn auf eine C-Struktur zugegriffen wird, sollten auch, wie in Listing 2.15 zu sehen, spezielle Datentypen (`c_long`, `c_void`, `c_char`, ...) verwendet werden, um die Kompatibilität mit verschiedenen Systemen und C-Compilern zu wahren. TODO: u32 immer 32bit, aber int nicht immer gleich (Beispiel!?) –> Probleme

Ein C-Pointer `*long` wird in Rust „Raw-Pointer“ genannt und entweder `*mut c_long` oder `*const c_long` geschrieben. Der Unterschied ist wie zwischen `&mut c_long` und `&c_long` und dient dem TODO: Rusttypsystem!? ref!? zur Unterscheidung TODO: Erzwinungt im Besitz von entsprechender Mutability zu sein, während es für die C-Seite keinen Unterschied macht [Rush]:

| Referenz in Rust | Raw-Pointer in Rust | C-Pointer |
|---|---|---|
| `&mut c_long` | `*mut c_long` | `long*` |
| `&c_long` | `*const c_long` | `long*` |

Abbildung 2.3: Vergleich Rust Raw-Pointer und Referenz zu C-Pointer

### Externer Funktionsaufruf

Während eine Struktur, die eine externe Struktur wiedergibt, sich optional in einem `extern {}` Block befinden kann, ist es zwingend, eine externe Funktionen darin bekannt zu machen:

```
1  use std::os::raw::c_void;
2
3  #[link(name = "messages", kind = "static")]
4  extern {
5      type asn_TYPE_descriptor_s;
6      type asn_enc_rval_t;
7
8      fn uper_encode_to_buffer(
9          type_descriptor: *const asn_TYPE_descriptor_s,
10         struct_ptr: *const c_void,
11         buffer: *mut c_void,
12         buffer_size: usize,
13     ) -> asn_enc_rval_t;
14 }
```

Listing 2.16: Externe Funktionsdefinition der ASN.1 Funktion zum Enkodieren

Wie in Listing 2.16 zu sehen ist, können auch `extern {}` Blöcke mit Attributen versehen werden. Zwingend ist bei der Verwendung eines `#[link(..)]` Attributes der Name der

Bibliothek, auf die sich der im `extern {}` Block stehende Code bezieht. Optional kann auch wie in Listing 2.16 die Art der TODO: Linkung (dylib, static) angegeben werden.

Die Art der Definition einer externen Funktion unterscheidet sich nicht von einer normalen Funktionsdefinition. Es sollten aber, wie in Abschnitt 2.26 beschrieben, zu C bzw. der externen Sprache kompatiblen Datentypen verwendet werden.

## 2.27 Kernfeatures

TODO: nothing on heap unless specified (Box, Vec, other container)

TODO: closures are fast, orly, p.310

https://www.youtube.com/watch?v=d1uraoHM8Gg
TODO: no need for a runtime, all static analytics
TODO: memory safety
TODO: data-race freedom
TODO: active community
TODO: concurrency: no undefined behavior
TODO: ffi binding Foreign Function Interface
TODO: zero cost abstraction
TODO: package manager: cargo

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: static type system with local type inference
TODO: explicit notion of mutability
TODO: zero-cost abstraction *(do not introduce new cost through implementation of abstraction)
TODO: errors are values not exceptions TODO: no null
TODO: ßtatic automatic memory management no garbage collection
TODO: often compared to GO and D ( 44min)

## 2.28 Schwächen

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: compile-times
TODO: Rust is a vampire language, it does not reflect at all!
TODO: depending on the field -> majority of libraries?

## 2.29 Performance Fallstricke

TODO: [Llo]

## 2.30 Beispiele von Verwendung von Rust

TODO: firefox

https://www.youtube.com/watch?v=-Tj8Q12DaEQ
TODO: GTK binding heavily to rust

TODO: unstable TODO: ffi

# 3 Hochperformante, serverbasierte Kommunikationsplattform

## 3.1 Kein Echtzeitsystem

### 3.1.1 Was ist ein Echtzeitsystem

TODO: hard real-time [Dou03, S. 75]

TODO: soft real-time [Dou03, S. 76]

### 3.1.2 Echtzeitnah

### 3.1.3 Funktionale Sicherheit

### 3.1.4 Was ist dann ein hochperformantes System

### 3.1.5 Low-Latency + Entwurfsmuster + Patterns? + Algorithmen?

TODO: Hochperformant -> parallel?

TODO: Design Pattern, Gamma et al, four important aspects

TODO: Real Time Design Patterns Buch: Ab Seite 141, verschiedene Systempatterns, microkernel [Dou03, S. 151]? channel architektur pattern [Dou03, S. 167]?

TODO: Message Queuing Pattern [Dou03, S. 207]

TODO: Clean Architecture / Clean Code

## 3.2 Serverbasierte Kommunikationsplattform

## 3.3 Protokolle

### 3.3.1 ASN.1

Die Notationsform ASN.1[1] ermöglicht abstrakte Datentypen und Werte zu beschreiben [Jr93]. Die Beschreibungen können anschließend zu Quellcode einer theoretisch[2] beliebigen Programmiersprache compiliert werden. Beschriebene Datentypen werden dadurch als native Konstrukte dargestellt und können mittels einer der standardisierten (oder auch eigenen [ITUb]) Encodierungen serialisiert werden.

Um den Austausch zwischen verschiedenen Anwendungen und Systemen zu ermöglichen, sind von der TODO: ITU bereits einige Encodierungen standardisiert [ITU15a, S. 8]. Für diese Arbeit ist aber einzig der PER Standard relevant, da der Server diese Encodierung verwenden muss, um mit den Sensoren und den Autos zu kommunizieren (siehe TODO: ref requirements / analyse).

Die anderen bekannteren Verfahren werden deshalb nur kurz erwähnt:

- **BER** (Basic Encoding Rules): Flexible binäre Encodierung [Wik18c], spezifiziert in X.690 [ITU15b]
- **CER** (Canonical Encoding Rules): Reduziert BER mit der Restriktion die Enden von Datenfelder speziell zu Markieren anstatt deren Größe zu übermitteln, eignet sich gut für große Nachrichten [Wik18c], spezifiziert in X.690 [ITU15b]
- **DER** (Distinguished Encoding Rules): Reduziert BER durch die Restriktion Größeninformationen zu Datenfeldern in den Metadaten zu übermitteln, eignet sich gut für kleine Nachrichten [Wik18c], spezifiziert in X.690 [ITU15b]
- **XER** (XML Encoding Rules): Beschreibt den Wechsel der Darstellung zwischen ASN.1 und XML, spezifiziert in X.693 [ITU15c]

TODO: isdn

[ITUa]

> *„ ASN.1 has a long record of accomplishment, having been in use since 1984. It has evolved over time to meet industry needs, such as PER support for the bandwidth-constrained wireless industry and XML support for easy use of common Web browsers. “* [ITUa]

[1] Abstract Syntax Notation One

[2] Es gibt keine Einschränkungen seitens des Standards, aber entsprechende Compiler zu finden erweist sich als schwierig TODO: ref impl Schwierigkeiten mit ASN+Rust

### PER

Die Packed Encoding Rules werden in in X.691 [ITU15a] beschrieben. Sie beschreiben eine Encodierung, die genutzt werden kann, um beschriebene Datentypen möglichst kompakt – also in wenigen Bytes – zu serialisieren.

TODO: sources: Für den Einsatz im Mobilfunknetz ist diese Encodierung sehr beliebt, da bei der Übermittlung einer Nachricht kein anderen Kommunikationsteilnehmer auf dieser Frequenz eine weiter Nachricht übermitteln kann. Eine kürzere Nachricht blockiert eine Frequenz kürzer, weshalb kürzere Nachrichten einen höheren Durchsatz erlaubt. Im Mobilfunkbereich ist dies von besonderer Bedeutung, da das Medium von vielen Teilnehmern gleichzeitig geteilt wird. TODO: michael.refactor_this_shit()

### 3.3.2 Kurze Erwähnung Protobuf?

# 4 Aufgabenstellung

# 5 Anforderungen

TODO: Safety / Funktionale Sicherheit Da bei Fehlern möglicherweise andere Verkehrsteilnehmer zu Schaden kommen können, müssen diverse Sicherheitsrichtlinien beachtet werden. Die Industrienorm ISO 26262 beschreibt dabei verschiedene Vorgehensweisen, unter anderem eine FBA[1], Risikoabschätzung durch Einstufung nach ASILs[2] und beschreibt Gegenmaßnahmen.

## 5.1 Funktionale Anforderungen

## 5.2 Nichtfunktionale Anforderungen

## 5.3 Kein Protobuf weil

[1] Fehlerbaumanalyse
[2] Automotive Safety Integrity Levels

TODO: was man machen soll

# 6 Systemanalyse

## 6.1 Systemkontextdiagramm

## 6.2 Schnitstellenanalyse

## 6.3 C++ Referenzsystem

## 6.4 Use-Cases

TODO: was wirklich umgesetzt sein wird

# 7 Systementwurf

## 7.1 Architektur

## 7.2 Änderungen bedingt durch Rust

# 8 Implementierung

### 8.0.1 Unerwartete Schwierigkeiten

TODO: Schwierigkeiten: FFI binding, manuell -> meh, also generieren

# 9 Auswertung

# 10 Zusammenfassung und Fazit

# Literatur


[BO17] Jim Blandy und Jason Orendorff. Programming Rust. Fast, Safe Systems Development. O'Reilly Media, Dez. 2017. ISBN: 1491927283.

[DD13] P.J. Deitel und H. Deitel. C for Programmers with an Introduction to C11. Deitel Developer Series. Pearson Education, 2013. ISBN: 9780133462074.

[Dou03] B.P. Douglass. Real-time Design Patterns: Robust Scalable Architecture for Real-time Sys Addison-Wesley object technology series Bd. 1. Addison-Wesley, 2003. ISBN: 9780201699562.

[fgi17] fgilcher. Subreddit Rust. fgilcher kommentiert. Englisch. 3. Nov. 2017. URL: https://www.reddit.com/r/rust/comments/7amv58/just_started_learning_rust_and_was_wondering_does/dpb9qew/ (besucht am 14. 02. 2018).

[Gil17] Florian Gilcher. GOTO 2017. Why is Rust Successful? Englisch. 6. Dez. 2017. URL: https://www.youtube.com/watch?v=-Tj8Q12DaEQ (besucht am 21. 02. 2018).

[GD14] J. Goll und M. Dausmann. C als erste Programmiersprache: Mit den Konzepten von C11. SpringerLink : Bücher. Springer Fachmedien Wiesbaden, 2014. ISBN: 9783834822710.

[Grü17] Sebastian Grüner. „C ist eine feindselige Sprache". Der Mitbegründer des Gnome-Projekts Deutsch. 22. Juni 2017. URL: https://www.golem.de/news/rust-c-ist-eine-feindselige-sprache-1707-129196.html (besucht am 14. 02. 2018).

[Hoa09] Tony Hoare. Null References: The Billion Dollar Mistake. Englisch. 25. Aug. 2009. URL: https://www.infoq.com/presentations/Null-References-The-Billion-Dollar-Mistake-Tony-Hoare (besucht am 06. 03. 2018).

[ITUa] International Telecommunication Union (ITU). Introduction to ASN.1. ASN.1 Project. Englisch. URL: https://www.itu.int/en/ITU-T/asn1/Pages/introduction.aspx (besucht am 23. 02. 2018).

[ITUb] International Telecommunication Union (ITU). The Encoding control notation. ASN.1 Pro Englisch. URL: https://www.itu.int/en/ITU-T/asn1/Pages/ecn.aspx (besucht am 23. 02. 2018).

[ITU15a] International Telecommunication Union (ITU). „Information technology – ASN.1 encoding rules. Specification of Packed Encoding Rules (PER)". Englisch. In: (Aug. 2015).

[ITU15b] International Telecommunication Union (ITU). „Information technology – ASN.1 encoding rules. Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)“. Englisch. In: (Aug. 2015).

[ITU15c] International Telecommunication Union (ITU). „Information technology – ASN.1 encoding rules. XML Encoding Rules (XER)“. Englisch. In: (Aug. 2015).

[Jr93] Burton S. Kaliski Jr. A Layman's Guide to Subset ASN.1, BER, and DER. An RSA Labor Englisch. 1. Nov. 1993. URL: http://luca.ntop.org/Teaching/Appunti/asn1.html (besucht am 23. 02. 2018).

[Lei17] Felix von Leitner. Fefes Blog. D soll Teil von gcc werden. Deutsch. 22. Juni 2017. URL: https://blog.fefe.de/?ts=a7b51cac (besucht am 14. 02. 2018).

[Llo] Llogiq. Llogiq on stuff. Rust Performance Pitfalls. Englisch. URL: https://llogiq.github.io/2017/06/01/perf-pitfalls.html (besucht am 14. 02. 2018).

[LLVa] LLVM.org. The LLVM Compiler Infrastructure Project. LLVM Overview. Englisch. URL: https://llvm.org/ (besucht am 19. 02. 2018).

[LLVb] LLVM.org. The LLVM Compiler Infrastructure Project. LLVM Features. Englisch. URL: https://llvm.org/Features.html (besucht am 19. 02. 2018).

[Mat16] Niko Matsakis. GitHub. Tracking issue for 128-bit integer support (RFC 1504). Englisch. 2016. URL: https://github.com/rust-lang/rust/issues/35118 (besucht am 05. 03. 2018).

[MEC] MEC-View. MEC-View. Deutsch. URL: http://mec-view.de/ (besucht am 19. 02. 2018).

[Men] Federico Mena-Quintero. Federico Mena-Quintero. Englisch. URL: https://people.gnome.org/~federico/ (besucht am 06. 03. 2018).

[Qui] Federico Mena Quintero. Replacing C library code with Rust. What I learned with librsvg. Englisch. URL: https://people.gnome.org/~federico/blog/docs/fmq-porting-c-to-rust.pdf (besucht am 14. 02. 2018).

[Rusa] Rust. The Rust Programming Language. Englisch. URL: https://www.rust-lang.org/en-US/faq.html (besucht am 16. 02. 2018).

[Rusb] Rust. The Rust Programming Language. Rust Platform Support. Englisch. URL: https://forge.rust-lang.org/platform-support.html (besucht am 19. 02. 2018).

[Rusc] Rust-Lang. Style Guidelines. Englisch. URL: https://doc.rust-lang.org/1.0.0/style/README.html (besucht am 23. 02. 2018).

[Rus17] Rust-Lang. Announcing Rust 1.19. Englisch. 20. Juli 2017. URL: https://blog.rust-lang.org/2017/07/20/Rust-1.19.html (besucht am 08. 03. 2018).

[Rus18] Rust-Lang. GitHub. rust/Copyright. Englisch. 2018. URL: https://github.com/rust-lang/rust/blob/master/COPYRIGHT (besucht am 12. 03. 2018).

[Rusd] Rust-Lang/Book. The Rust Programming Language. Primitive Types. Englisch. URL: https://doc.rust-lang.org/book/first-edition/primitive-types.html (besucht am 21. 02. 2018).

[Ruse] Rust-Lang/Book. The Rust Programming Language. Statements and expressions. Englisch. URL: https://doc.rust-lang.org/reference/statements-and-expressions.html (besucht am 05. 03. 2018).

[Rusf] Rust-Lang/Book. The Rust Programming Language. Constructors. Englisch. URL: https://doc.rust-lang.org/beta/nomicon/constructors.html (besucht am 05. 03. 2018).

[Rusg] Rust-Lang/Book. The Rust Programming Language. Behavior considered undefined. Englisch. URL: https://doc.rust-lang.org/reference/behavior-considered-undefined.html (besucht am 07. 03. 2018).

[Rush] Rust-Lang/Book. The Rust Programming Language. Unsafe Rust. Englisch. URL: https://doc.rust-lang.org/book/second-edition/ch19-01-unsafe-rust.html#dereferencing-a-raw-pointer (besucht am 20. 02. 2018).

[Rusi] Rust-Lang/Doc. Rust. core::cmp::PartialEq. Englisch. URL: https://doc.rust-lang.org/core/cmp/trait.PartialEq.html (besucht am 08. 03. 2018).

[Rusj] Rust-Lang/Doc. Rust. core::cmp::PartialEq. Englisch. URL: https://doc.rust-lang.org/core/cmp/trait.Eq.html (besucht am 08. 03. 2018).

[Rusk] Rust-Lang/Doc. Rust. core::cmp::PartialOrd. Englisch. URL: https://doc.rust-lang.org/core/cmp/trait.PartialOrd.html (besucht am 08. 03. 2018).

[Rusl] Rust-Lang/RFCs. GitHub. support alloca. Englisch. URL: https://github.com/rust-lang/rfcs/issues/618 (besucht am 08. 03. 2018).

[Rusm] Rust-Lang/RFCs. GitHub. Tracking issue for RFC 1861: Extern types. Englisch. URL: https://github.com/rust-lang/rust/issues/43467 (besucht am 20. 02. 2018).

[Sch13] Julia Schmidt. Graydon Hoare im Interview zur Programmiersprache Rust. Deutsch. 12. Juli 2013. URL: https://www.heise.de/-1916345 (besucht am 16. 02. 2018).

[Wik17] Wikipedia. LLVM — Wikipedia, Die freie Enzyklopädie. 2017.

[Wik18a] Wikipedia. Foreign function interface — Wikipedia, The Free Encyclopedia. 2018.

[Wik18b] Wikipedia. NaN — Wikipedia, The Free Encyclopedia. 2018.

[Wik18c] Wikipedia. X.690 — Wikipedia, The Free Encyclopedia. 2018.

[wit] withoutboats. GitHub. Tracking issue for 128-bit integer support (RFC 1504). Englisch. URL: https://github.com/rust-lang/rust/issues/35118#issuecomment-362689905 (besucht am 07. 03. 2018).

# Glossar

*Foreign Function Interface* Beschreibt den Mechanismus wie ein Programm das in einer Programmiersprache geschrieben ist, Funktionen aufrufen kann, die einer einer anderen Programmiersprache geschrieben wurden. [Wik18a] . 27, 30

*git* (dt. Blödmann) ist eine Software zur Versionierungs von Quelldateien, entwickelt von Linus Torvalds 2005. TODO: cite . V, 7, 10

*GitHub* Plattform zum Hosten von git-Repositories inklusive eingebautem Issue-Tracker und Wiki. Änderungen an Quellcode können vorgeschlagen, und durch die Projektverantwortlichen übernommen werden. Bietet auch die Möglichkeit eine kontinuierlichen Integrationssoftware einzubinden, um automatisierte Tests auf momentanen Quellcode und auch für Änderungen auszuführen. Eine vorgeschlagene Änderung kann somit vor Übernahme auf Kompatibilität überprüft werden. . 7, 9

*LLVM* Früher „Low Level Virtual Machine“ [Wik17], heute Eigenname; ist eine „Ansammlung von modularen und wiederverwendbaren Kompiler- und Werkzeugtechnologien“ [LLVa]. Unterstützt eine große Anzahl von Zielplatformen, u.a. X86, X86-64, PowerPC, PowerPC-64, ARM, Thumb, ... [LLVb]. . 8

# Abkürzungsverzeichnis

*ASIL* Automotive Safety Integrity Level. 37

*ASN.1* Abstract Syntax Notation One. 33

*BMWi* Bundesministerium für Wirtschaft und Energie. 3

*FBA* Fehlerbaumanalyse. 37

*GC* Garbage Collector. 21

*MEC* Mobile Edge Computing. 3, 4

# Abbildungsverzeichnis

# Listings