HOCHSCHULE LANDSHUT
HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN

FAKULTÄT INFORMATIK

Bachelorarbeit

Programmieren in Rust und Vergleich mit C/C++

Thomas Keck

Betreuer: Prof. Dr. rer. nat. Dieter Nazareth

ERKLÄRUNG ZUR BACHELORARBEIT

Keck, Thomas

Hochschule Landshut
Fakultät Informatik

Hiermit erkläre ich, dass ich die Arbeit selbständig verfasst, noch nicht anderweitig für Prüfungszwecke vorgelegt, keine anderen als die angegebenen Quellen oder Hilfsmittel benützt, sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

....

(Datum) (Unterschrift des Studierenden)

Zusammenfassung/Abstract

Beim Programmieren mit einer systemnahen Sprache besteht oft das Problem, dass viele Fehler nicht erkannt werden. Solche Fehler können beim Programmierer unbemerkt bleiben, da sie meist nur schwer zu finden sind. Eine Sprache, die dem Programmierer Vertrauen beim Entwickeln gibt, hilft dabei, kritische Fehler frühzeitig zu entdecken. Dieses Vertrauen gibt ein strenger Compiler, welcher alle Speicheradressen auf Gültigkeit überprüfen kann.

In dieser Arbeit wird die Programmierung mit der Sprache Rust gezeigt, mit den Eigenheiten, die dafür sorgen, dass Speicherfehler bereits beim Kompilieren gefunden werden können. Zudem werden Unterschiede zu C/C++ aufgezeigt.

Inhaltsverzeichnis

1 Einleitung

1.1 Was ist Rust?

Rust ist eine quelloffene Systemprogrammiersprache, die sich auf Geschwindigkeit, Speichersicherheit und Parallelität konzentriert. Entwickler nutzen Rust für ein breites Spektrum an Anwendungsgebieten: Spiel-Engines[1], Betriebssysteme[2], Dateisysteme und Browserkomponenten. [Moz]

Eine aktive Gemeinschaft von Programmierern verwaltet die Codebasis von Rust und fügt fortlaufend neue Verbesserungen hinzu. Mozilla sponsert das Open-Source-Projekt.

Rust wurde von Grund auf neu aufgebaut und enthält Elemente aus bewährten und modernen Programmiersprachen. Es verbindet die ausdrucksstarke und intuitive Syntax von High-Level-Sprachen mit der Kontrolle und Leistung einer Low-Level-Sprache. Sie verhindert Zugriffsverletzungen und gewährleistet Threadsicherheit.

Rust macht die Systemprogrammierung durch die Kombination von Leistung und Ergonomie zugänglicher. Sie bietet starke Features wie z. B. sichere Speicherverwaltung durch einen strengen Compiler und ein Typsystem für risikolose Nebenläufigkeit.

Große und kleine Unternehmen setzen Rust bereits ein, darunter:

- Mozilla: Wichtige Komponenten von Mozilla Firefox Quantum.
- Dropbox: Mehrere Komponenten wurden in Rust als Teil eines größeren Projekts geschrieben, um eine höhere Effizienz des Rechenzentrums zu erreichen.
- Yelp: Ein Framework für Echtzeit A/B-Tests, welches auf allen Yelp-Websites und -Anwendungen verwendet wird.

[1] http://arewegameyet.com
[2] z. B. Redox OS

2 Rust toolchain

Die Rust toolchain ist eine Sammlung von Werkzeugen, die dabei helfen, den Compiler aktuell zu halten und Projekte zu verwalten.

2.1 rustup

Das Rustup-Tool ist die empfohlene Installationsmethode für Rust. Das Tool ermöglicht zusätzlich die Verwaltung von verschiedenen Versionen, Komponenten und Plattformen. Um zwischen den Versionen stable, beta und nightly zu wechseln, kann auf der Kommandozeile eingegeben werden: [Rusb]

```
rustup install beta                    # release channel
rustup install nightly
rustup update                          # update all channels
rustup default nightly                 # switch to 'nightly'
```

Rust unterstützt auch das Kompilieren für andere Systeme (Cross-Compiler). So wird beispielsweise MUSL als Zielsystem verwendet:

```
# add target
rustup target add x86-64-unknown-linux-musl
# build project with target
cargo build --target=x86_64-unknown-linux-musl
```

Mit Hilfe von rustup können verschiedene Komponenten installiert werden:

- rust-docs: Lokale Kopie der Rust-Dokumentation, um sie offline lesen zu können.
- rust-src: Lokale Kopie des Quellcodes von Rust. Autokomplettierungs-Tools verwenden diese Information.
- rustfmt-preview: Zur automatischen Code-Formatierung.

```
rustup component add rustfmt-preview
```

2.2 rustc

Der Compiler von Rust, er übersetzt den Quellcode in einen binären Code, entweder als Bibliothek oder als ausführbare Datei. Die meisten Rust-Programmierer rufen rustc nicht direkt auf, sondern indirekt über Cargo. [Rusg]

2.2.1 Hauptphasen der Kompilierung

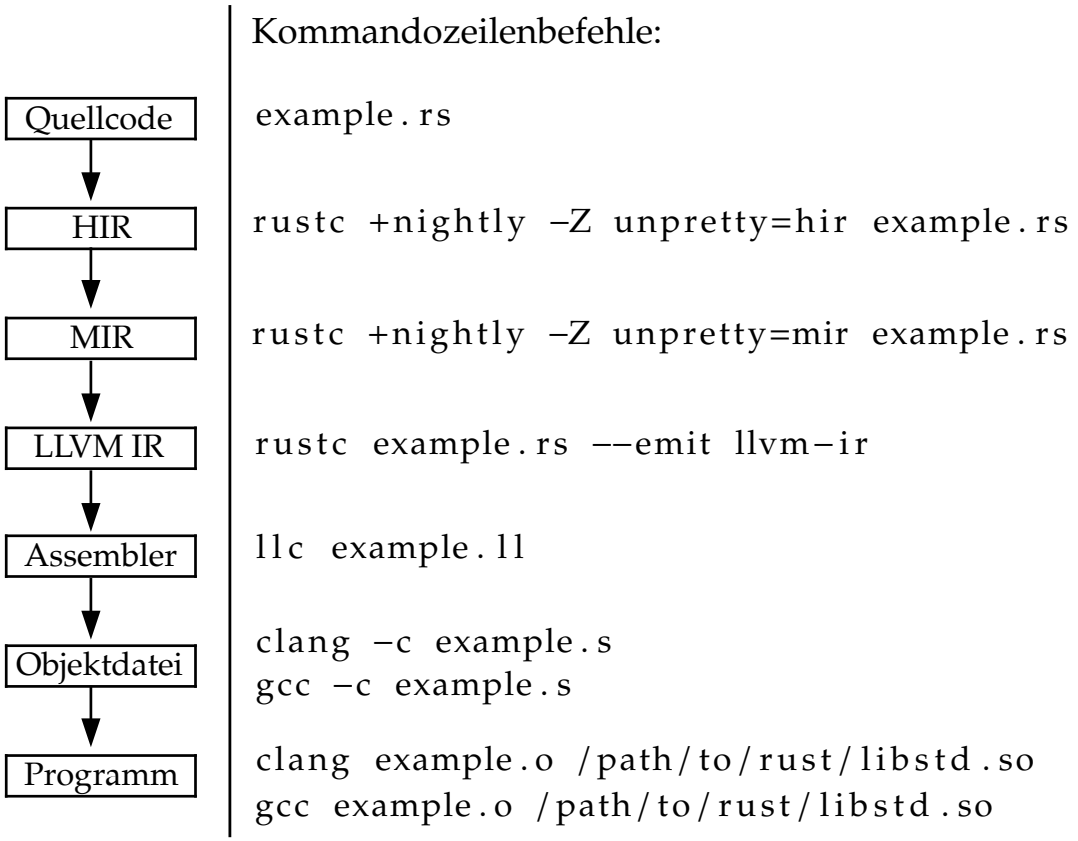

Abb. 2.1: Zwischenschritte bei der Kompilierung

Der Compiler von Rust übersetzt den Code mit mehreren Zwischenschritten, eine Darstellung hierzu ist in Abbildung 2.1 zu sehen. [Rusd]

Zuerst wird der Rust Code in eine „high level intermediate representation“ (HIR) übersetzt. Die HIR Darstellung ist ein abstrakter Syntaxbaum, welcher in einer ähnlichen Syntax aufgebaut ist wie der Rust Code. Hier findet die Typprüfung statt, welche eine Tabelle mit den Typen der Ausdrücke erstellt. Anschließend finden weitere Analysen statt, beispielsweise die Überprüfung der Zugriffe auf private Klassen oder Module (Kapselung).

Die nächste Stufe ist die „middle IR“ (MIR). Diese Darstellung ist für Menschen schlecht lesbar, aber sie eignet sich für die Überprüfung der Ownership von Variablen. Codeoptimierungen werden hier auch vorgenommen. Die Bedeutung von Ownership in Rust wird in Abschnitt 3.2 behandelt.

Aus der MIR Darstellung wird eine „low level virtual machine IR“ (LLVM IR) erzeugt. LLVM ist ein Compilerunterbau, welcher auch vom C Compiler clang verwendet wird. LLVM führt hier einige Optimierungen durch und erzeugt Objektdateien, welche dann gelinkt werden.

2.2.2 Grundlegende Verwendung

Der Kommandozeilenbefehl für das Kompilieren mit rustc ähnelt dem eines C-Programms:

```
gcc   hello.c  -o helloC          # C
rustc hello.rs -o helloRust       # Rust
```

Anders als in C muss nur der crate root[1] angegeben werden. Der Compiler kann mithilfe des Codes selbständig festellen, welche Dateien er übersetzen und linken muss. Bei einem einfachen Aufruf des Compilers ohne weitere Parameter erhält man ein ausführbares Programm wie es auch in C oder C++ der Fall ist. Üblicherweise werden bei einem Aufruf von `rustc` keine Objektdateien hinterlassen. Wird der Projektmanager Cargo verwendet, dann werden die Objektdateien gespeichert. Somit werden nur die Teile des Projekts erneut kompiliert, die umgeschrieben wurden.

2.2.3 Lint

Ein Lint ist ein Werkzeug, das zur statischen Codeanalyse verwendet wird. Der Rust-Compiler verwendet ebenfalls einen Lint, wodurch beim Kompilieren Warnungen und Fehlermeldungen ausgeben werden. Im folgendem Beispiel wird eine unbenutzte Variable deklariert, der Aufbau des Codes wird in Kapitel 3 erklärt.

```
$ cat main.rs
fn main() {
    let x = 5;
}
```

[1] Quellcode-Datei mit der main() Methode

```
$ rustc main.rs
warning: unused variable: `x`
 --> main.rs:2:9
  |
2 |     let x = 5;
  |         ^ help: consider using `_x` instead
  |
  = note: #[warn(unused_variables)] on by default
```

Das ist das `unused_variables` Lint und verweist auf eine unbenutzte Variable, die im Code keine Verwendung findet. Dies ist nicht falsch, es könnte jedoch ein Hinweis auf einen Bug sein.

2.3 Cargo

Cargo ist ein Projektmanager für Rust. Damit können Abhängigkeiten heruntergeladen und verteilbare Pakete erstellt werden, welche auf crates.io[2] hochgeladen werden können. [Rusa]

2.3.1 Projektverwaltung

Projekte können mit Hilfe von Cargo erstellt werden, dabei entsteht eine bestimmte Ordnerstruktur mit einer Cargo.toml Datei sowie dem crate root im src-Ordner. Ein Projekt kann eine Applikation (binary) oder eine Bibliothek (library) sein. Der crate root ist bei einer Applikation immer „main.rs“ und bei einer Bibliothek „lib.rs“.

```
$ cargo new hello_world --bin       # --lib for library
    Created binary (application) `hello_world` package

$ cd hello_world
$ tree .
.
├── Cargo.toml
└── src
    └── main.rs
1 directory, 2 files
```

[2]Paketeregister der Rust-Community

Die Cargo.toml enthält alle wichtigen Metainformationen, die Cargo zum Kompilieren benötigt.

```
[package]
name = "hello_world"
version = "0.1.0"
authors = ["Thomas Keck <s-tkeckk@haw-landshut.de>"]
edition = "2018"

[dependencies]
```

Die Informationen über den Autor enthält Cargo von den Umgebungsvariablen `CARGO_NAME` und `CARGO_EMAIL`. In Rust gibt es sogenannte editions, welche in der Regel in einem zeitlichen Abstand von zwei oder drei Jahren veröffentlicht werden und, ähnlich wie in C, einen Standard festlegen. Im Juli 2019 gab es zwei Editionen: 2015 und 2018. Das Pendant in C wären die C-Standards wie z. B. C90, C99 oder C11.

Zudem können hier zusätzliche Bibliotheken angegeben werden, die Cargo automatisch von crates.io herunterlädt und in das Projekt einbindet. Cargo erstellt für genauere Informationen der Abhängigkeiten eine Datei: Cargo.lock. Diese sollte nicht manuell verändert werden, da sie von Cargo selbständig gepflegt wird. Mithilfe von Cargo können auch Tests gestartet werden, genauere Information dazu sind aus Abschnitt 3.6 zu entnehmen.

Projektlayout

Die Abbildung 2.2 zeigt alle wichtigen Dateien als Baum.

- Cargo.toml und Cargo.lock werden im Wurzelverzeichnis des Projekts gespeichert.
- Quellcode-Dateien sind im `src`-Ordner vorgesehen.
- Die Standarddatei für Bibliotheken ist `src/lib.rs`.
- Die Standarddatei für ausführbare Programme ist `src/main.rs`.
- Quellcode für sekundäre ausführbare Programme sind `src/bin/*.rs`.
- Integrationstests befinden sich im Ordner `tests`, Unit-Tests werden in die jeweiligen Programmdateien geschrieben.

- Beispiele befinden sich im `examples` Ordner und
- Benchmarks im `benches` Ordner.

```
.
├── Cargo.lock
├── Cargo.toml
├── benches
│   └── large-input.rs
├── examples
│   └── simple.rs
├── src
│   ├── bin
│   │   └── another_executable.rs
│   ├── lib.rs
│   └── main.rs
└── tests
    └── some-integration-tests.rs
```

Abb. 2.2: Dateibaum eines Cargo Projekts

Wichtige Kommandozeilenbefehle für Cargo

Zum Kompilieren und Ausführen:

```
$ cargo build
$ ./target/debug/hello_world

$ cargo build --release          # optimized performance
$ ./target/release/hello_world

# build and run in one command
$ cargo run
```

Zum Testen:

```
# run all standard tests
$ cargo test

# run all tests marked as ignored
$ cargo test -- --ignored
```

2.3.2 Veröffentlichung auf crates.io

Das Paketeregister der Rust-Community, genannt crates.io, ist ein Ort für Bibliotheken, die von verschiedenen Programmierern aus der Community verwaltet werden. Eine Veröffentlichung ist permanent. Das heißt, dass keine Versionsnummern überschrieben werden können und somit der Code nicht gelöscht werden kann. Dafür gibt es keine Begrenzung für die Anzahl der Versionen, die veröffentlicht werden können.

Vor der Veröffentlichung wird ein Account benötigt. Dazu muss mit einem Github-Account auf crates.io ein API-Token generiert werden. Danach kann man sich über einen Befehl auf der Kommandozeile anmelden:

```
$ cargo login abcdefghijklmnopqrstuvwxyz012345
```

Dieser Token wird anschließend in einem lokalen Verzeichnis gespeichert und sollte nicht mit anderen geteilt werden. Eine erneute Generierung eines Tokens ist möglich.

Mithilfe von Cargo werden eigene Bibliotheken paketiert, dabei entsteht eine *.crate-Datei im Unterverzeichnis `target/package`.

```
$ cargo package
```

Dabei ist zu beachten, dass es eine Beschränkung der Uploadgröße von 10 MB für *.crate-Dateien gibt. Um die Dateigröße einzuschränken, können Dateien exkludiert bzw. inkludiert werden, dazu stehen die Schlüsselwörter `exclude` (blacklisting) und `include` (whitelisting) in der Cargo.toml-Datei zur Verfügung:

```
[package]
# ...
exclude = [
    "public/assets/*",
    "videos/*",
]
```

bzw.

```
include = [
    "**/*.rs",
    "Cargo.toml",
]
```

Zu beachten ist, dass das Schlüsselwort `include`, wenn gesetzt, `exclude` überschreibt.

Zum Hochladen muss nur noch folgender Befehl ausgeführt werden:

```
$ cargo publish
```

Dieser Befehl paketiert die Bibliothek automatisch, falls keine lokale crate-Datei gefunden wurde.

Zum Veröffentlichen einer neuen Version muss lediglich die Versionsnummer in der Cargo.toml verändert werden.

Verwalten eines crate.io basierten Pakets

Die Verwaltung eines Pakets geschieht in Rust primär auf der Kommandozeilenebene mit Cargo.

Wenn ein schwerwiegender Bug in einem bereits hochgeladenen Paket gefunden wurde, kann diese Version aus dem Index von crates.io entfernt, jedoch nicht gelöscht werden.

```
$ cargo yank --vers 1.0.1
$ cargo yank --vers 1.0.1 --undo    # undo the yank
```

Diese Pakete können immer noch heruntergeladen und in andere Projekte eingebunden werden, die bereits an „yanked“ Pakete gebunden waren. Cargo lässt dies jedoch nicht bei neu erstellten Crates[3] zu.

Ein Projekt wird meist von mehreren Entwicklern programmiert, oder die Besitzer des Projekts ändern sich im Laufe der Zeit. Folgende Befehle fügen neue Entwickler zu einem Projekt hinzu, welche dann in der Lage sind, auf crates.io zu veröffentlichen:

```
# "named" owner:
$ cargo owner --add my-buddy
$ cargo owner --remove my-buddy
```

[3] Bibliothek oder Paket in Rust

```
# "team" owner:
# syntax: github:org:team
$ cargo owner --add github:rust-lang:owners
$ cargo owner --remove github:rust-lang:owners
```

Wenn ein Team angegeben wird, ist dieses nicht befugt, neue „owner“ hinzuzufügen. Die Befehle yank und publish sind den Teammitgliedern jedoch erlaubt. Ist ein „named owner“ in einem Team, so sind alle Entwickler dieses Teams als „named owner“ eingestuft.

2.3.3 Externe Tools

Cargo versucht, die Integration von Tools von Drittanbietern zu vereinfachen, z. B. für IDEs oder anderen Build-Systemen. Hierfür gibt es mehrere Möglichkeiten:

- cargo metadata-Befehl
- message-format Argument
- benutzerdefinierte Befehle

Information über die Paketstruktur mit cargo metadata

```
$ cargo metadata
```

Dieser Befehl gibt im JSON-Format alle Metadaten über ein Projekt aus. Darunter befinden sich die Version des aktuellen Projekts sowie eine Liste der Pakete und Abhängigkeiten. Die grobe Struktur sieht so aus:

```
{
    "version": integer,
    "packages": [ {
        "id": PackageId,
        "name": string,
        "version": string,
        "source": SourceId,
        "dependencies": [ Dependency ],
        "targets": [ Target ],
        "manifest_path": string,
    } ],
```

```
    "workspace_members": [ PackageId ],
    "resolve": {
        "nodes": [ {
            "id": PackageId,
            "dependencies": [ PackageId ]
        } ]
    }
}
```

Informationen beim Kompilieren

Mit dem Argument `--message-format=json` können genauere Informationen beim Kompilieren herausgefiltert werden:

```
$ cargo build --message-format=json
```

Dadurch entsteht ein Output im JSON-Format mit Informationen über Compiler-Fehlermeldungen und -Warnungen, erzeugte Artefakte und das Ergebnis.

Benutzerdefinierte Befehle

Cargo ist so ausgelegt, dass es erweitert werden kann, ohne dass Cargo selbst modifiziert werden muss. Dazu muss ein Programm in der Form cargo-*command* in einem der `$PATH`-Verzeichnisse des Benutzers liegen. Anschließend kann es mit „cargo *command*“ aufgerufen werden. Wenn ein solches Programm von Cargo aufgerufen wird, übergibt es als ersten Parameter den Programmnamen, als zweites die Bezeichnung des Programms (*command*). Alle weiteren Parameter in der Befehlszeile werden unverändert weitergeleitet.

Beispiel:

```
// cargo-listargs
use std::env;

fn main() {
    let args: Vec<_> = env::args().collect();
    println!("{:?}", args);
}
```

Obiges Programm `cargo-listargs` gibt eine Liste der Parameter aus, die übergeben wurden. Es könnte auch in C programmiert sein. Entscheidend ist, dass der Programmname in der richtigen Form ist.

```
$ ./cargo-listargs arg1 arg2
# ["./cargo-listargs", "arg1", "arg2"]

$ cargo listargs arg1 arg2
# ["/path/to/cargo-listargs", "listargs", "arg1", "arg2"]
```

Im Internet befanden sich im Juli 2019 bereits über 40 benutzerdefinierte Befehle, die von der Rust-Community erstellt wurden. [Rush]

3 Programmierung mit Rust und Unterschiede zu C/C++

In diesem Kapitel wird die Programmierung mit Rust gezeigt. Dabei werden auch einige Unterschiede zu C und C++ erläutert.

3.1 Grundlagen

Rust ist eine kompilierte Sprache. Genau wie C und C++ erzeugt sie nach dem Kompilieren eine ausführbare Datei. Sie ist außerdem eine imperative Sprache, sie besteht also aus Anweisungen und Ausdrücken. Rust wird auch vom objektorientierten Programmierparadigma beeinflusst, bietet jedoch, im Gegensatz zu C++, keine Vererbungen von Klassen an. Sie ist eine stark typisierte und statisch typisierte Sprache.

3.1.1 Hello, world!

```
fn main() {
    println!("Hello, world!");
}
```

Genau wie in C oder C++ beginnt in Rust das Programm mit einer Startfunktion mit dem Namen `main`. Funktionen bzw. Prozeduren werden in Rust mit dem Schlüsselwort `fn` vor dem Funktionsnamen und einem Block definiert. Blöcke verhalten sich in Rust genau wie in C/C++ hinsichtlich der Sichtbarkeit von Variablen (Verschattung ist möglich).

Funktionen können auch mit einem Rückgabewert definiert werden:

```
fn example() -> bool { ...
```

Und Funktionsparameter:

```
fn example(number1: i32, number2: i32) -> bool { ...
```

Der Rückgabetyp wird mit einem Pfeil (->) angegeben. Parameter werden innerhalb der Klammern durch Kommata getrennt in der Form: `name: typ`. Der Aufruf von Funktionen ist syntaktisch wie C/C++. Funktionen werden in Unterabschnitt 3.1.6 genauer beschrieben.

Im Hello World Programm wurde `println!` ausgeführt. Das Ausrufezeichen besagt, dass es sich hierbei nicht um eine Funktion handelt, sondern um ein Makro. In C/C++ werden Makros von einem Präprozessor verarbeitet. Rust besitzt keinen Präprozessor, es wird also kein Code direkt eingefügt. Die Funktionsweise von Makros wird in Unterabschnitt 3.1.7 behandelt. Das Makro `println!` gibt Text mit einem Zeilenumbruch auf der Kommandozeile (stdout) aus.

Der erste Parameter ist immer eine Zeichenkette. Um Variablen oder Ergebnisse von Ausdrücken auszugeben, müssen im Text geschweifte Klammern, sowie weitere Parameter, welche dann an Stelle der Klammern eingefügt werden, gesetzt werden. Ohne Parameter wird nur ein Zeilenumbruch ausgegeben.

```
println!();
println!("format {} arguments", "some");
println!(
    "first = {0}, second = {1}, first again = {0}",
    1, 2
);
println!("{a} {c} {b}", a = "a", b = 'b', c = 3);
println!("{:.3}", 1.23456789); // 1.235
```

Die Klammern können auch nummeriert werden, falls Ausdrücke mehr als einmal ausgegeben werden sollen. Alternativ können Identifikatoren verwendet werden. Fließkommazahlen können beispielsweise mit einer bestimmten Anzahl an Nachkommastellen ausgegeben werden. Hier wird wie in C/C++ die letzte Stelle gerundet. Zeilen- und Blockkommentare sind syntaktisch aufgebaut wie in C/C++.

3.1.2 Variablen und Mutabilität

Variablen können in Rust mit dem Schlüsselwort `let` erstellt werden. Der Typ der Variable wird in den meisten Fällen von Rust erkannt. Weitere Informationen über Datentypen werden in Unterabschnitt 3.1.3 angesprochen, denn zunächst soll die Syntax gezeigt werden. So wird beispielsweise eine Variable `x` mit dem Wert 5 erstellt:

```
let x = 5;
```

In Rust sind Variablen standardmäßig unveränderlich. Das ist einer von vielen Faktoren, die Programmierern helfen sollen, Fehler im Code besser finden zu können. [KN18]

Ein Beispiel in Rust:

```
fn main() {
    let x = 5;
    println!("The value of x is {}", x);
    x = 6;
    println!("The value of x is {}", x);
}
```

Beim Kompilieren erscheint folgende Fehlermeldung:

```
error[E0384]: cannot assign twice to immutable variable
`x`
--> src/main.rs:4:5
  |
2 |     let x = 5;
  |         -
  |         |
  |         first assignment to `x`
  |         help: make this binding mutable: `mut x`
3 |     println!("The value of x is {}", x);
4 |     x = 6;
  |     ^^^^^ cannot assign twice to immutable variable
```

Die Fehlermeldung besagt, dass ein unveränderlicher Wert nicht verändert werden darf und schlägt vor, den Wert veränderlich, also als „mutable“, zu deklarieren. Dazu wird das Schlüsselwort `mut` verwendet:

```
let mut x = 5;
x = 10;
x = x + 5;
x += 1;
```

Die Operatoren =, +=, -=, *+, /= und %= funktionieren in Rust wie in C/C++. Die Operatoren für Postinkrement und Präinkrement wurden in Rust bewusst nicht implementiert, um den Code lesbarer zu gestalten.

In C und C++ ist jede Variablendefinition standardmäßig veränderlich. Bei gleicher Vorangehensweise würde folgendes C-Programm fehlerfrei übersetzen:

```
#include <stdio.h>

int main()
{
    int x = 5;
    printf("The value of x is %d\n", x);
    x = 6;
    printf("The value of x is %d\n", x);
    return 0;
}
```

Das Schlüsselwort `const` kann verwendet werden, um Werte in C/C++ konstant zu deklarieren.

```
const int x = 5;
```

Wird aber beispielsweise in C versucht, `x` zu verändern, indem die Speicheradresse von `x` dereferenziert und der Wert dahinter verändert wird, ist das Verhalten nicht definiert. [ISO, p. 87]

Das Verhalten in folgendem Beispielprogramm ist somit nicht definiert und kann von vielen verschiedenen Faktoren abhängen:

```
#include <stdio.h>

int main()
{
    const int x = 5;
    printf("The value of x is %d\n", x);
    *(int *)&x = 6;
    printf("The value of x is %d\n", x);
    return 0;
}
```

Ergebnis mit dem Clang Compiler in der Version 7.0.1 auf einem Linux System (keine Warnungen beim Kompilieren):

```
The value of x is 5
The value of x is 5
```

Ergebnis mit dem GCC Compiler in der Version 8.3.1 auf einem Linux System (auch hier keine Warnungen):

```
The value of x is 5
The value of x is 6
```

3.1.3 Datentypen

Jede Variable in Rust hat einen bestimmten Datentyp. Es wird unterschieden zwischen skalaren und zusammengesetzten Typen. Rust ist eine statisch typisierte Sprache, daher müssen die Typen aller Variablen zur Kompilierzeit bekannt sein. Der Compiler kann normalerweise, basierend auf Wert und Verwendungszweck einer Variable, ableiten, welcher Typ verwendet werden soll.

In Fällen, in denen viele Typen möglich sind, z. B. wenn ein String in einen numerischen Typ konvertiert werden soll, muss eine Typannotation hinzugefügt werden:

```
let unsigned_number: u32 = 42;
```

Integer Typ

Ein Integer ist ein Datentyp für ganze Zahlen. Der Typ `u32` gibt an, dass es sich um eine vorzeichenlose Ganzzahl handelt. Vorzeichenbehaftete Ganzzahltypen beginnen mit „i“ an Stelle von „u“ und die Zahl gibt an, wie viel Speicherplatz sie beansprucht. Tabelle 3.1 zeigt die Integertypen von Rust und C.

	Rust		C/C++	
Länge	signed	unsigned	signed	unsigned
8-bit	i8	u8	__int8_t	__uint8_t
16-bit	i16	u16	__int16_t	__uint16_t
32-bit	i32	u32	__int32_t	__uint32_t
64-bit	i64	u64	__int64_t	__uint64_t
128-bit	i128	u128	__int128_t	__uint128_t
arch	isize	usize		

Tab. 3.1: Integertypen in Rust und C/C++

In C sollen mit `short` und `long` verschieden lange ganzzahlige Werte zur Verfügung stehen, soweit dies praktikabel ist; `int` wird normalerweise die natürliche Größe für eine bestimmte Maschine sein. `short` belegt oft 16 Bits, `long` 32 Bits und `int` entweder 16 oder 32 Bits. Es steht jedem Übersetzer frei, sinnvolle Größen für seine Maschine zu wählen, nur mit den Einschränkungen, dass `short` und `int` wenigstens 16 Bits haben, `long` mindestens 32 Bits, und dass `short` nicht länger als `int` und `int` nicht länger als `long` sein darf. [KR90]

So können die Größen der üblich verwendeten Ganzzahltypen in C ausgegeben werden:

```
printf("%zu\n", sizeof(char));      // 1:  8-bit
printf("%zu\n", sizeof(short));     // 2: 16-bit
printf("%zu\n", sizeof(int));       // 4: 32-bit
printf("%zu\n", sizeof(long));      // 8: 64-bit
```

In Rust hängen die Typen `isize` und `usize` von der Art des Zielsystems ab, für das kompiliert wird: 64 Bits für 64-Bit-Architekturen und 32 Bits für 32-Bit-Architekturen. Rust ist eine statisch typisierte Sprache, daher darf dies nicht zur Laufzeit entschieden werden.

In Rust wird standardmäßig der Integertyp i32 verwendet. Dieser Typ ist im Allgemeinen der schnellste Typ, auch auf 64-Bit-Systemen. Die Typen `isize` und `usize` können zum Indexieren von Felder verwendet werden.

Weitere Typen in Rust

- Fließkomma-Typen: `f32` und `f64` (Standard ist `f64`)
- Wahrheitswert: `bool` true / false
- Zeichentyp `char`: Unicode, das heißt chinesische, japanische und koreanische Zeichen, Emoji, Leerzeichen mit Nullbreite sind möglich

Tupel

Ein Tupel ist ein allgemeiner Weg, um einige andere Werte mit verschiedenen Typen zu einem Verbindungstyp zu gruppieren. Tupel haben eine feste Länge und können somit nicht größer oder kleiner werden, nachdem sie deklariert wurden.

```
fn main() {
    let tup: (i32, f64, u8) = (500, 6.4, 1);
    let (x, y, z) = tup;
    println!("The value of y is: {}", y);
    println!("The value of z is: {}", tup.2);
}
```

Tupel können auf einzelne Variablen zerlegt werden. Dazu werden die Variablennamen nach dem Schlüsselwort `let` in Klammern geschrieben. Auf einzelne Elemente aus dem Tupel kann mit einem Index nach einem Punkt zugegriffen werden.

3.1.4 Kontrollstrukturen

Kontrollstrukturen definieren die Reihenfolge, in der Berechnungen durchgeführt werden. Die Entscheidung, ob Code ausgeführt werden soll oder nicht, abhängig davon, ob eine Bedingung wahr ist, und die Entscheidung, wiederholt Code auszuführen, während eine Bedingung wahr ist, sind grundlegende Bausteine in den meisten Programmiersprachen. In Rust gibt es Ausdrücke, die einen Wert repräsentieren und Anweisungen, welche mit Strichpunkten voneinander getrennt werden.

Die häufigsten Konstrukte, mit denen der Ausführungsfluss von Rust-Code gesteuert werden können, sind `if`-Ausdrücke und Schleifen.

`if`-Anweisungen

Mit `if-else`-Anweisungen werden Entscheidungen formuliert. Der `else`-Teil ist optional.

Beispiel:

```
if number < 5 {
    println!("condition was true");
} else {
    println!("condition was false");
}
```

Diese Anweisungen sind syntaktisch wie C oder C++, mit dem Unterschied, dass keine Klammern bei dem Ausdruck mit dem Wahrheitswert benötigt werden.

In Rust muss dieser Ausdruck `true` oder `false` sein. Folgendes Programm wird also nicht übersetzt, weil es sich hier um den Zahlenwert `i32` handelt:

```
let number = 3;
if number {                                    // type must be bool
    println!("number was three");
}
```

C und C++ prüfen bei einer `if`-Anweisung mit einem Integer Wert nur, ob dieser den Wert 0 hat. Da in Rust ein boolescher Typ benötigt wird, muss das obige Programm umgeschrieben werden:

```
let number = 3;
if number != 0 {
    println!("number was something other than 0");
}
```

Auch in Rust können mehrere Bedingungen in einer `else if`-Anweisung behandelt werden:

```
let number = 6;

if number % 4 == 0 {
    println!("number is divisible by 4");
} else if number % 3 == 0 {
    println!("number is divisible by 3");
} else if number % 2 == 0 {
    println!("number is divisible by 2");
} else {
    println!("number is not divisible by 4, 3, or 2");
}
```

In Rust kann die `if`-Anweisung ein Ausdruck sein, um z. B. Werte von Variablen zu definieren. Ähnlich wie der `?:` Operator in C/C++.

So wird der ?: Operator wird in C verwendet:

```
bool condition = true;
int number = condition ? 5 : 6;
```

Der Wert von `number` ist also 5, weil `condition` wahr ist. Dieser Operator aus C/C++ kann auch in Rust mit einer `if`-Anweisung programmiert werden:

```
let condition = true;
let number = if condition {
    5
} else {
    6
};
```

Hier fällt auf, dass kein Strichpunkt innerhalb der geschweiften Klammern steht. Wenn in Rust am Ende eines Blocks eine Anweisung ohne Strichpunkt steht, wird der Block von Rust als Ausdruck gewertet. In Verbindung mit einer `if`-Anweisung kann so der `?:` Operator aus C/C++ programmiert werden.

Da Rust eine statisch typisierte Sprache ist, muss der Typ beim Kompilieren bekannt sein. Das bedeutet, dass bei dieser `if`-Anweisung der Typ einheitlich sein muss. Im letzten Beispiel ist `number` vom Typ `i32`.

`match`

In Rust gibt es keine `switch`-Anweisung wie in C/C++. Die Alternative in Rust sind `match`-Anweisungen bzw. -Ausdrücke. Die Syntax sieht so aus:

```
let number = 30;

match number {
    1 => println!("number is 1"),
    2 | 3 | 5 | 7 | 11 => println!("small prime num"),
    20..=30 => println!("number is between 20 and 30"),
    _ => println!("number was something different"),
}
```

Nach dem Schlüsselwort `match` muss ein Ausdruck geschrieben werden. Dieser kann dann unterschieden werden. Es unterscheidet zwischen einem einzelnen Wert, mehreren Werten und einem Wertebereich. Der Unterstrich ist vergleichbar mit `default` aus C/C++. Nach dem Pfeil (=>) folgt der Code, der bedingt ausgeführt werden soll. Kommata trennen die Fälle voneinander.

Der Wert des Ausdrucks kann weitergegeben werden, indem ein Bezeichner verwendet wird. Dieser kann mit `if` und einem booleschen Ausdruck zusätzlich überprüft werden. Es kann nach dem Pfeil auch ein Block mit mehreren Anweisungen folgen:

```
match number {
    n if n < 100 => {
        println!("number is smaller than 100");
        println!("and the number is {}", n);
    }
    n => println!("the number is {}", n),
}
```

`match` kann auch als Ausdruck interpretiert werden:

```
let result = match number {
    1 => "one",
    2..=10 => "between two and ten",
    _ => "i don't know",
};
```

Das Schlüsselwort `break` kann in Rust nur in Schleifen verwendet werden, daher ist ein „Fallthrough“ hier nicht nicht möglich.

Schleifen

Rust hat drei Arten von Schleifen: loop, while und for.

`loop`-Schleifen führen Code so oft aus, bis sie explizit mit dem Schlüsselwort `break` gestoppt werden. `loop`-Schleifen können in Rust auch als Ausdruck interpretiert werden, wenn nach dem `break` ein Ausdruck angefügt wird:

```
fn main() {
    let mut counter = 0;
    let result = loop {
        counter += 1;
        if counter == 10 {
            break counter * 2;      // loop returning 20
        }
    };
    println!("The result is {}", result);
}
```

`while`-Schleifen gibt es auch in C und C++. In Rust ist die Syntax ähnlich:

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    let mut index = 0;
    while index < 5 {
        println!("the value is: {}", a[index]);
        index = index + 1;
    }
}
```

`while`-Schleifen können nicht als Ausdruck interpretiert werden, da nicht sichergestellt werden kann, dass ein `break` aufgerufen wird. Rusts Kontrollflussanalyse behandelt `while true` anders als `loop`, da der Compiler weiß, dass `loop` immer ausgeführt wird. Bei der Zuweisung von deklarierten Variablen macht dies einen Unterschied:

```
let var;
loop {
    var = 1;
    break;
}
println!("{}", var);  // works fine

let var;
while true {
    var = 1;
    break;
}
println!("{}", var);  // possibly uninitialized variable
```

Um ein Feld oder andere iterierbare Typen (z. B. Vektoren) zu durchlaufen, kann mit einer `for`-Schleife gearbeitet werden. Diese Schleifen funktionieren in Rust anders als in C oder C++. Sie erinnern an `for-each`-Schleifen aus Java. Das bedeutet, dass der Code innerhalb der Schleife für jedes Element aus einer Sammlung einmal ausgeführt wird.

```
fn main() {
    let a = [10, 20, 30, 40, 50];
    for element in a.iter() {
        println!("the value is: {}", element);
    }
}
```

Der Aufruf von `iter()` erzeugt einen Iterator, welcher in `for`-Schleifen verwendet werden kann. Mit `for`-Schleifen kann in Rust die Stabilität des Programms erhöht werden, da Zugriffe außerhalb eines Felds verhindert werden können.

3.1.5 Felder

Eine andere Möglichkeit, Sammlungen mehrerer Werte zu bilden, sind Felder. Im Gegensatz zu einem Tupel, muss jedes Element eines Felds denselben Typ haben. Felder in Rust unterscheiden sich von Felder in einigen anderen Sprachen, da Felder in Rust eine feste Länge haben, die zur Kompilierzeit festgelegt wird.

In Rust werden die Werte, die direkt in ein Feld gespeichert werden sollen, als eine durch Kommata getrennte Liste in eckigen Klammern geschrieben:

```
let a = [1, 2, 3, 4, 5];
```

Ein Feld ist ein einzelner Speicherbereich, der auf dem Stack reserviert ist. Sie sind nützlich, um sicherzustellen, dass immer eine feste Anzahl von Elementen vorhanden ist. Felder können in Rust nicht wie in C oder C++ auf dem Heap erstellt werden, da die Größe zur Kompilierzeit bekannt sein muss. Die Elemente innerhalb eines Felds können jedoch Zeiger auf den Heap beinhalten. Alternativ kann der Vektortyp verwendet werden, welcher die Werte immer auf dem Heap alloziert und dessen Größe dynamisch zur Laufzeit verändert werden kann. Wenn ein Feld einer Funktion übergeben werden soll, kann der Slice Typ verwendet werden, Unterabschnitt 3.2.3 zeigt die Verwendung von Slices. Die Begriffe Stack und Heap haben in Rust dieselbe Bedeutung wie in C und C++.

Die Schreibweise des Feldtyps erfolgt mit eckigen Klammern mit dem Typen der Elemente, gefolgt von einem Semikolon und der Anzahl der Elemente für das Feld:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

Eine ähnliche Schreibweise wird zum Initialisieren eines Felds verwendet, das für jedes Element denselben Wert enthält:

```
let a = [3; 5];
```

Das Feld a enthält 5 Elemente mit dem Wert 3. Folgender Code erzeugt das gleiche Feld:

```
let a = [3, 3, 3, 3, 3];
```

Es kann mithilfe eines Index auf die Elemente zugegriffen werden:

```
let first = a[0];
let second = a[1];
```

Wird auf ein Element zugegriffen, das außerhalb des Bereichs ist, beendet sich das Programm mit folgender Meldung:

```
thread 'main' panicked at 'index out of bounds: the len
is 5 but the index is 10', src/main.rs:5:13
note: Run with 'RUST_BACKTRACE=1' environment variable
to display a backtrace.
```

Mit statischen Indizes können diese Fehler beim Kompilieren erkannt werden:

```
let array = [1, 2, 3, 4, 5];

array[0];
array[10];                  // compiler error: idx is 10
array[(2 * 9  + 2) / 4];    // compiler error: idx is 5

let index = 200;
array[index];               // runtime error
```

Wenn in C oder C++ auf ein Element außerhalb des Bereichs eines Felds zugegriffen wird, fällt dies beim Testen wesentlich seltener auf, da normalerweise das Programm weiter ausgeführt wird.

Das ist ein Beispiel der Sicherheitsprinzipien von Rust. Viele Low-Level-Programmiersprachen verzichten auf diesen Check, sodass ein ungültiger Speicherbereich indexiert werden kann. Rust verhindert dies durch ein sofortiges Beenden des Programms.

3.1.6 Funktionen

Die grobe Struktur von Funktionen wurde bereits mit dem Hello World Programm in Unterabschnitt 3.1.1 gezeigt. Ein Programm, das eine Beispielfunktion enthält, könnte so aussehen:

```
fn main() {
    println!("Hello, world!");
    another_function(42);
}
fn another_function(n: i32) {
    println!("Another function with number {}.", n);
}
```

Eine Funktion mit Parameter enthält den Namen der Variable, sowie den mit einem Doppelpunkt getrennten Typen. Bei mehreren Parametern werden diese mit Komma getrennt.

Bei Funktionen mit Rückgabewerten, kann am Ende des Funktionskörpers ein Ausdruck als Rückgabewert ohne Strichpunkt geschrieben werden, weil der Block von Rust daduch als Ausdruck interpretiert werden kann. Wenn eine Funktion früh beendet werden soll, kann wie in C/C++ das Schlüsselwort `return` mit einem Ausdruck als Rückgabewert benutzt werden. Der Rückgabetyp der Funktion muss mit einem Pfeil (->) angegeben werden.

```
fn main() {
    let result = add(12, 34);
    println!("The result is {}.", result);
}
fn add(n1: i32, n2: i32) -> i32 {
    n1 + n2
}
```

Anders als in C/C++ müssen in Rust Funktionen nicht vor der aufrufenden Funktion definiert oder deklariert werden.

3.1.7 Makros

Bisher wurde in den Codebeispielen regelmäßig das Makro `println!` verwendet. Mit Makros kann Code geschrieben werden, der anderen Code generiert. Dieses Konzept wird Metaprogrammierung genannt. Im Gegensatz zu Funktionen können Makros eine variable Anzahl an Parametern übergeben werden. Makros werden wie in C/C++ extrahiert, bevor der Compiler den Code interpretiert. Der Nachteil von Makros gegenüber Funktionen ist, dass sie komplexer sind. Sie sind schwerer zu lesen, zu verstehen und zu pflegen. Makros müssen, anders als Funktionen, vor der aufrufenden Funktion bekannt gemacht werden.

Makros erinnern an `match`-Anweisungen. Das heißt, sie unterscheiden zwischen Muster bzw. Patterns. Ein Makro kann beispielsweise so definiert werden:

```
macro_rules! my_macro {
    (1) => { "one" };
    (2) => { "two" };
    (3) => { "three" };
}
```

Für die Definition eines Makros muss das Schlüsselwort `macro_rules!` zusammen mit dem Namen angegeben werden. Danach folgen in einem Block die Muster, die erkannt werden sollen. Ein Pfeil (=>) zeigt auf einen Block mit den Anweisungen, die durchgeführt werden, bzw. mit dem Ausdruck, der zurückgegeben werden soll.

```
println!("{} {} {}",
    my_macro!(1),
    my_macro!(2),
    my_macro!(3)
);
// prints "one two three"
```

Die Parameter beim Aufruf eines Makros müssen immer mit einem Muster übereinstimmen.

```
my_macro!(4); // compiler error: unexpected token '4'
```

Die Muster von Makros unterscheiden sich von Mustern aus `match`, da sie auch Rust Code erkennen können. Komplexere Muster können verwendet werden:

```
macro_rules! my_print_macro {
    ( $( $x:expr ),* ) => {
        $(
            println!("{}", $x);
        )*
    };
}
```

Auch hier befindet sich das Muster innerhalb von Runden Klammern, gefolgt von einem Pfeil. Die Syntax `$()` fängt ein Codemuster auf; `$x:expr` besagt, dass ein Ausdruck gelesen werden soll. Um eine beliebige Anzahl an Ausdrücken zu filtern, kann `,*` nach einem `$()` verwendet werden. Wiederholungsoperatoren geben

Makros in Rust mehr Flexibilität als in C/C++. Es gibt insgesamt drei verschiedene Arten:

- `*` 0 oder mehr Wiederholungen,
- `+` 1 oder mehr Wiederholungen,
- `?` 0 oder 1 Wiederholung.

Der Ausdruck `$()*` führt für jedes gefundene Muster den Code einmal aus. Die Ausdrücke (expr) können mit einem Dollarzeichen verwendet werden. Die Funktionsweise ist vergleichbar mit einer `for`-Schleife.

Die Entwickler von Rust haben angekündigt, dass dieses System erneuert wird. Die Erstellung von Makros mit `macro_rules!` wird langfristig also nicht mehr unterstützt werden.

3.2 Ownership

Ownership ist ein einzigartiges Merkmal von Rust. Es ermöglicht Speichersicherheit zur Kompilierzeit, ohne die Notwendigkeit eines Garbage Collectors. Daher ist es für Programmierer wichtig, das Ownership-Prinzip als Programmierer zu verstehen. In diesem Kapitel werden Ownership und damit zusammenhängende Eigenschaften beschrieben.

3.2.1 Funktionsweise von Ownership

Alle Computerprogramme müssen Arbeitsspeichermanagement betreiben. Manche Sprachen benutzen einen Garbage Collector, welcher während der Programmlaufzeit nach nicht mehr verwendeten Speicher sucht und diesen anschließend frei gibt. In andern Sprachen muss der Programmierer explizit Speicher zuweisen und freigeben. Rust geht anders vor: Speicher wird durch ein System von Besitz und einen Satz an Regeln gestützt, welche der Compiler zur Kompilierzeit überprüft, sodass das Programm zur Laufzeit nicht gebremst wird.

Regeln

- Jeder Wert in Rust hat eine Variable, die als Eigentümer bezeichnet wird.
- Es gibt für jeden Wert nur einen einzelnen aktiven Eigentümer.

- Wenn der Besitzer den Gültigkeitsbereich der Lebensdauer verlässt, wird der Wert gelöscht.

Die Lebensdauer und Sichtbarkeit von Variablen verhalten sich in Rust wie in C/C++. Verschattungen von Variablen sind daher möglich, ohne dass die Lebensdauer von verschatteten Variablen beeinflusst wird.

Beispiel: String

Um die Eigentumsregeln zu veranschaulichen, ist ein komplexerer Datentyp notwendig, als die, die in Unterabschnitt 3.1.3 behandelt wurden, denn diese werden auf dem Stack gespeichert. In diesem Beispiel sollen Daten, die auf dem Heap gespeichert werden, betrachtet werden, um zu veranschaulichen, wie Rust erkennt, wann diese Daten zu bereinigen sind. Hierfür wird im Folgenden der Typ `String` verwendet. Diese Aspekte gelten auch für andere komplexe Datentypen, die von der Standardbibliothek bereitgestellt werden.

Wenn ein Rust-Programm eine Eingabe von einem Benutzer speichern möchte, muss ein Datentyp verwendet werden, welcher eine variable Länge haben kann. Einen solchen String kann man in Rust zum Beispiel mit der `from`-Funktion erstellen:

```
let mut s = String::from("hello");
s.push_str(", world!");              // appends a literal
println!("{}", s);                   // 'hello, world!'
```

Der Aufruf `String::from()` erzeugt ein Objekt mit einer Zeichenkette, die auf den Heap zugewiesen wird. In Abschnitt 3.4 wird objektorientierte Programmierung erläutert. Zunächst ist wichtig zu verstehen, dass der Heap verwendet wird, um einen veränderlichen String zu erzeugen, dessen Größe zur Kompilierzeit unbekannt sein kann. Das heißt:

1. Speicherplatz muss vom Betriebssystem zur Laufzeit bereitgestellt werden und

2. der Speicherplatz muss wieder freigegeben werden, wenn der String nicht mehr benötigt wird.

Der erste Teil wird manuell vom Programmierer beim Aufruf von `String::from` initiiert. Der zweite Teil geschieht automatisch intern durch die Methode `drop`, sobald der Besitzer den Gültigkeitsbereich verlässt. Zum Beispiel am Ende eines Blocks.

```
{
    let s = String::from("hello");
    // s is valid from this point forward

    // do stuff with s
}                       // this scope is now over, and s is
                        // no longer valid
```

In C++ wird dieses Muster der Freigabe von Ressourcen am Ende der Lebensdauer eines Elements manchmal als *Ressourcenbelegung ist Initialisierung (Resource Acquisition Is Initialization, kurz RAII)* bezeichnet. [Str15, p. 71]

Dieses Muster hat einen tiefgründigen Einfluss auf die Schreibweise von Rust-Code. Es mag einfach erscheinen, aber das Verhalten von Code kann in komplizierten Situationen unerwartet sein, wenn mehrere Variablen die Daten verwenden sollen, die auf dem Heap zugewiesen wurden.

Variablen und Daten: Move

Mehrere Variablen können in Rust auf unterschiedliche Weise mit denselben Daten interagieren. Ein Beispiel mit Integer:

```
let x = 5;
let y = x;
```

Hier wird der Wert 5 an die Konstante `x` gebunden und anschließend eine Kopie von `x` an die Konstante `y` gebunden. Der Wert wird hier kopiert, da Integer ein primitiver Datentyp ist. Eigentum primitiver Datentypen kann nicht übergeben werden (Move nicht möglich). Sie werden vollständig kopiert. In der Regel kann kein Eigentum einfacher Skalarwerte übertragen werden. Einige der Typen sind:

- Alle Integer-Typen, z. B. `u32`
- Boolean-Typ, `bool` mit den Werten `true` und `false`
- Alle Fließkommazahlen wie z. B. `f64`
- Zeichentyp `char`
- Tupel, wenn sie nur vorher genannte Typen beinhalten, z. B. `(i32, i32)`, jedoch nicht `(i32, String)`

Beispiel mit String:

```
let s1 = String::from("hello");
let s2 = s1;
```

Dies sieht dem vorherigen Code sehr ähnlich, aber die Funktionsweise ist nicht dieselbe, denn Stringobjekte benutzen den Heap.

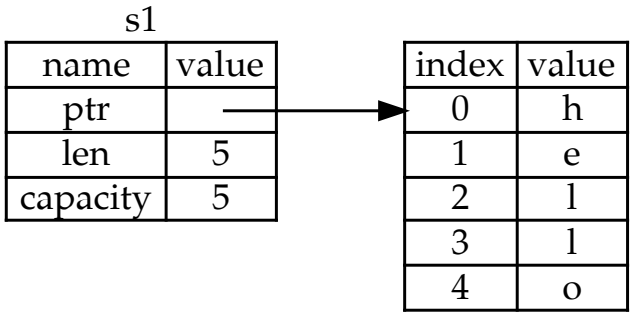

Abb. 3.1: Repräsentation des Speichers eines String

Abbildung 3.1 zeigt die Bestandteile eines String. Er besteht aus drei Teilen, zu sehen in der linken Tabelle: ein Pointer auf den Speicher, der den String enthält, die Länge und die Kapazität. Wenn die Kapazität nicht mehr ausreicht, vergrößert sie sich automatisch. Dabei wird der Speicherplatz nach jeder erneuten Allokation verdoppelt. Diese Datengruppe wird auf dem Stack gespeichert. In der rechten Tabelle ist sich der Speicher auf dem Heap dargestellt.

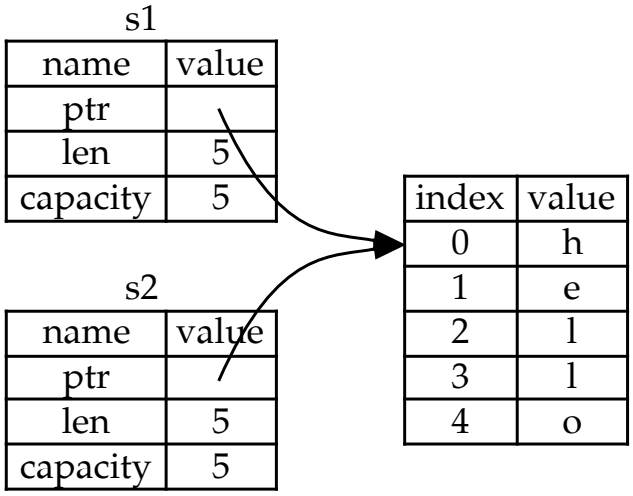

Abb. 3.2: Repräsentation des Speichers eines String: Kopie

Im letzten Codebeispiel werden bei der Zuweisung von `s1` zu `s2` nur die Stringdaten, also ein Pointer, die Länge und die Kapazität kopiert, welche sich auf dem Stack befinden. Die Daten des Heaps werden nicht kopiert. Abbildung 3.2 zeigt eine mögliche Darstellung.

Wenn Rust eine vollständige Kopie gemacht hätte, würden die Daten wie auf Abbildung 3.3 aussehen und die Operation `s2 = s1` wäre hinsichtlich der Laufzeitleistung sehr teuer.

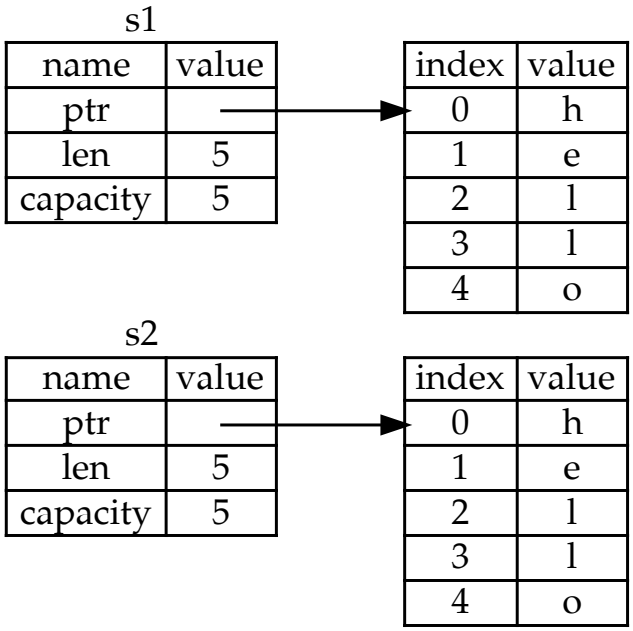

Abb. 3.3: Repräsentation des Speichers eines String: Vollständige Kopie

Sobald eine Variable außerhalb des Gültigkeitsbereichs der Lebensdauer gelangt, wird der Speicher aus dem Heap gelöscht. Aber da der Speicherbereich in Abbildung 3.2 von zwei Variablen referenziert wird, würde zwei mal der Speicherbereich freigegeben werden. Das ist bekannt als „double free“-Error und kann zu Speicherbeschädigungen und Sicherheitsanfälligkeiten führen.

Um die Speichersicherheit zu gewährleisten, hält Rust `s1` für ungültig und muss somit nichts löschen, wenn `s1` die Lebensdauer verlässt.

Das hat zur Folge, dass `s1` nicht mehr genutzt werden kann, nachdem es `s2` zugewiesen wurde:

```
let s1 = String::From("hello");
let s2 = s1;
println!("{}, world!", s1);          // error
```

Daraus entsteht folgende Fehlermeldung:

```
error[E0382]: borrow of moved value: ‘s1‘
 --> src/main.rs:5:28
3 |      let s1 = String::from("hello");
  |          -- move occurs because ‘s1‘ has type
‘std::string::String‘, which does not implement the
‘Copy‘ trait
4 |      let s2 = s1;
  |               -- value moved here
5 |      println!("{}, world!", s1);
  |                             ^^ value borrowed here
after move
```

Diese Art von Kopieren, welche die erste Variable ungültig macht, wird in Rust „move“ genannt (`s1` was *moved* into `s2`). Was also tatsächlich passiert, wird in Abbildung 3.4 dargestellt.

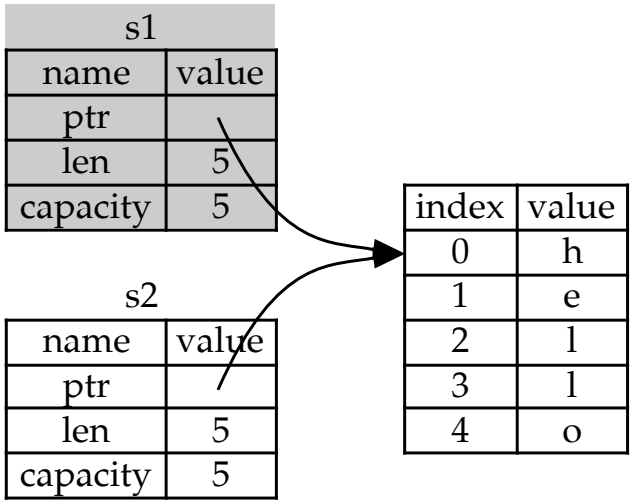

Abb. 3.4: Repräsentation des Speichers eines String: Moved

Folglich gibt es in Rust keinen „double free“-Error, da der Speicher nur einmal freigegeben wird, nämlich wenn `s2` den Gültigkeitsbereich verlässt.

Variablen und Daten: Clone

Wenn eine vollständige Kopie erstellt werden soll, kann die Methode `clone` benutzt werden:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Der Code kompiliert ohne Fehlermeldungen und erzeugt explizit das in Abbildung 3.3 gezeigte Verhalten, bei dem auch die Heap-Daten kopiert werden.

Ownership und Funktionen

Die Semantik bezüglich Ownership für die Übergabe eines Wertes an eine Funktion ähnelt derjenigen, die einer Variablen einen Wert zuweist. Beim Übergeben von Variablen an eine Funktion wird genauso wie bei einer Zuweisung die Ownership übergeben (Move), wenn es sich um keinen einfachen Skalarwert handelt. Wie in C/C++ werden die Daten bei der Übergabe an Funktionen kopiert (in Rust: Abbildung 3.4), wenn kein Pointer verwendet wird, also „call by value“. Pointer werden in Rust Referenzen genannt. Die Funktionsweise von Referenzen wird in Unterabschnitt 3.2.2 erklärt.

Folgendes Programm zeigt, wie die Ownership von Variablen übergeben werden:

```
fn main() {
    let s = String::from("hello");
    takes_ownership(s);

    let x = 5;
    makes_copy_because_scalar(x);

    println!("{}", s); // compiler error
    println!("{}", x); // this works!
}

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}
```

```
fn makes_copy_because_scalar(some_integer: i32) {
    println!("{}", some_integer);
}
```

Wenn `s` nach der Methode `takes_ownership` genutzt wird, würde Rust einen Fehler bei der Kompilierung ausgeben. Diese statischen Checks sollen Fehler im Programm verhindern.

Funktionen können Ownership durch einen Rückgabewert auch wieder zurückgeben:

```
fn main() {
    let s1 = gives_ownership();
    let s2 = takes_and_gives_back(s1);
    println!("{}, world!", s1); // error
    println!("{}, world!", s2); // this works
}

fn gives_ownership() -> String {
    let some_string = String::from("hello");
    some_string
}

fn takes_and_gives_back(a_string: String) -> String {
    a_string
}
```

3.2.2 Referenzen und Borrowing

Referenzen können verwendet werden, um einen Zeiger zu übergeben, sodass nicht der Ganze Datentyp kopiert werden muss. Also „call by reference“.

Eine Funktion, welche die Länge eines Strings berechnet und die Ownership des verwendeten Strings wieder zurückgibt, könnte so aussehen:

```
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("The length of {} is {}.", s2, len);
}
```

```
fn calculate_length(s: String) -> (String, usize) {
    let length = s.len();
    (s, length)
}
```

Dies ist jedoch viel Aufwand für ein Konzept, das gebräuchlich sein sollte. Daher können in Rust Referenzen auf Variablen übergeben werden.

So wird die `calculate_length` Funktion definiert und benutzt, die eine Referenz auf ein Objekt als Parameter enthält, anstatt die Ownership zu übernehmen:

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of {} is {}.", s1, len);
}
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Der gesamte Tupelcode in der Variablendeklaration und der Funktionsrückgabewert ist weg. Außerdem wird `&s1` in `calculate_length` und in seiner Definition `&String` anstelle von `String` benutzt.

Die &-Zeichen erstellen Referenzen, die auf einen bestimmten Wert verweisen, ohne dessen Ownership zu beanspruchen. Abbildung 3.5 zeigt ein Diagramm.

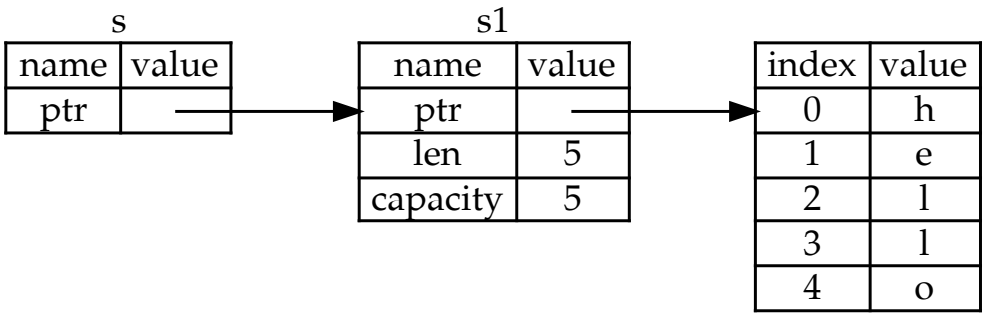

Abb. 3.5: Diagramm von Zeiger `&String s` auf `String s1`

Mit der `&s1` Syntax kann eine Referenz erstellt werden, die sich auf den Wert von `s1` bezieht, ihm aber nicht gehört. Der Wert auf den er verweist wird nicht gelöscht, wenn die Referenz den Gültigkeitsbereich verlässt.

Ebenso verwendet die Signatur der Funktion & um anzuzeigen, dass der Typ des Parameters eine Referenz ist.

Das Gegenteil der Referenzierung mit & ist die Dereferenzierung, die mit dem Dereferenzierungsoperator * erzielt wird.

Die Lebensdauer von Referenzen, ist die, eines normalen Funktionsparameters, jedoch wird der Wert nicht aus dem Speicher gelöscht, da keine Ownership übergeben wurde. Referenzierung im Funktionsparameter wird in Rust „borrowing“ genannt. Wie in der realen Welt kann Eigentum an andere ausgeliehen werden. Wenn die andere Person es nicht mehr braucht, kann diese es wieder an den Eigentümer zurückgeben.

Genauso wie Variablen standardmäßig unveränderlich sind, gilt dies auch für Referenzen. Es kann nichts verändert werden, auf das nur mit & referenziert wird. Folgender Code würde daher nicht funktionieren:

```
fn main() {
    let s = String::from("hello");
    change(&s);
}
fn change(some_string: &String) {
    some_string.push_str(", world"); // compiler error
}
```

Veränderbare Referenzen

Kleine Veränderungen beheben den Fehler aus dem letzten Codebeispiel:

```
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}
fn change(some_string: &mut String) {
    some_string.push_str(", world"); // works now
}
```

Die Variable s muss durch Kennzeichnung mit `mut` veränderbar gemacht werden. Außerdem wird nun eine veränderbare Referenz übergeben (`&mut`). Die Funktion erwartet nun eine entsprechende Referenz als Parameter in der Form: `some_string: &mut String`.

Es gibt jedoch eine große Einschränkung: Es darf nur eine veränderbare Referenz auf ein bestimmtes Datenelement in einem bestimmeten Bereich aktiv sein. Folgender Code würde somit nicht kompilieren:

```
let mut s = String::from("hello");
let r1 = &mut s;
let r2 = &mut s;                    // error
println!("{}, {}", r1, r2);
```

Diese Einschränkung erlaubt kontrollierte Verwendung von Variablen. Mithilfe dieser Regel verhindert Rust sogenannte „data races“. Diese ähneln „race conditions“ und treten auf, wenn drei Verhaltensweisen auftreten:

- Zwei oder mehr Pointer greifen zeitgleich auf den selben Speicherbereich zu.
- Mindestens einer der Pointer schreibt.
- Kein Mechanismus wird verwendet, um den Zugriff der Daten zu synchronisieren.

Diese „data races“ können undefiniertes Verhalten auslösen und sind schwer festzustellen und zu beheben. In Rust-Programmen gibt es dieses Problem nicht, da gefährdeter Code nicht fehlerfrei kompilieren kann.

Es dürfen auch keine Kombinationen aus veränderbaren und unveränderbaren Referenzen erstellt werden. Folgender Code führt zu einem Compilerfehler:

```
let mut s = String::from("hello");
let r1 = &s;      // ok
let r2 = &s;      // ok
let r3 = &mut s; // error
```

Bei unveränderlichen Referenzen sollten man nicht damit rechnen müssen, dass sich der Wert dahinter verändern kann. Mehrere unveränderbare Referenzen sind in Ordnung, da mehrere lesende Zugriffe sich untereinander nicht beeinflussen.

„Dangling References“

In Programmiersprachen mit Pointern kann es vorkommen, dass diese auf einen Bereich im Speicher zeigen, der bereits freigegeben wurde. Das sind sogenannte „dangling pointer“. In Rust garantiert der Compiler, dass Referenzen immer auf einen gültigen Bereich zeigen. Folgender Code wird also beim Kompilieren eine Fehlermeldung ausgeben:

```
fn main() {
    let reference_to_nothing = dangle();
}
fn dangle() -> &String {
    let s = String::from("hello");
    &s
}
```

Der String `s` wird nach der Funktion `dangle` aus dem Speicher gelöscht. Die Referenz, die zurückgegeben wird, zeigt auf einen ungültigen Bereich im Speicher. Der Compiler erkennt, dass der Rückgabewert eine „dangling reference“ ist und gibt eine Fehlermeldung aus. Damit der String im Speicher bleibt, muss die Ownership von `s` zurückgegeben werden:

```
fn no_dangle() -> String {
    let s = String::from("hello");
    s
}
```

Regeln: Referenzen

- Es kann entweder nur eine veränderbare oder eine beliebige Anzahl an unveränderlichen Referenzen gleichzeitig benutzt werden.
- Referenzen müssen immer auf einen gültigen Bereich zeigen.

3.2.3 Slice Typ

Ein weiterer Datentyp, welcher keine Ownership annimmt sind Slices. Sie werden benutzt um eine zusammenhängende Folge von Elementen einer Sammlung anstatt auf die gesamte Sammlung zu verweisen.

String Slice

In Rust kann mit sogenannten „string slices“ ein bestimmter Bereich eines Strings referenziert werden.

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```

Die Syntax der Klammern beschreibt `[starting_index..ending_index]`. Intern wird die Startposition und die Länge gespeichert. Die Länge errechnet sich aus der Differenz von `starting_index` und `ending_index`. `world` ist ein Pointer auf das siebte Byte von `s` mit einer Länge von 5. Abbildung 3.6 zeigt ein Diagramm des Speichers.

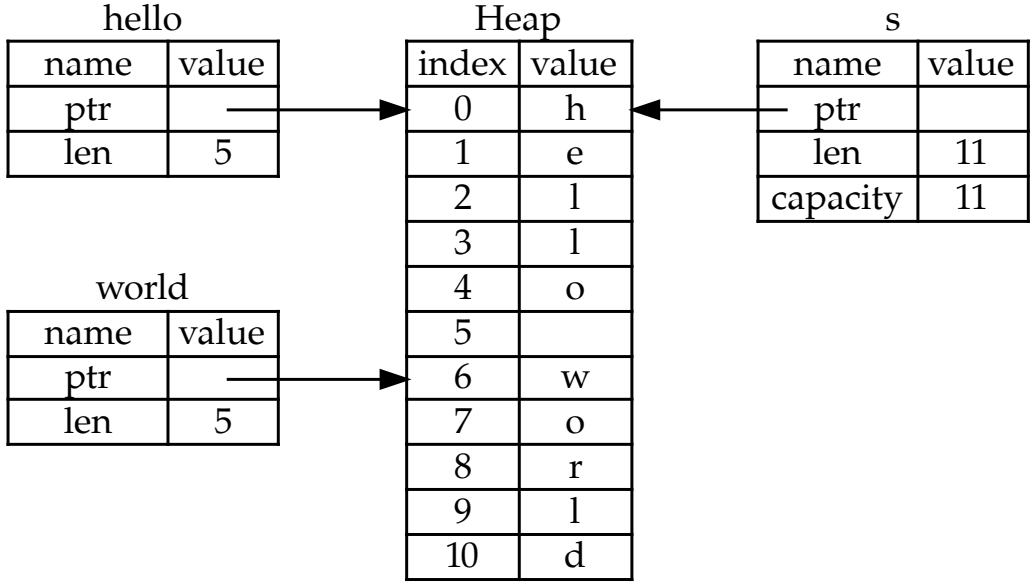

Abb. 3.6: Diagramm von String Slices

Dadurch entsteht eine Abhängigkeit, welche es Rust erlaubt, bereits beim Kompilieren festzustellen, ob String Slices auf einen gültigen Bereich im Speicher zeigen. So wird der Code weniger anfällig für Laufzeitfehler.

```
let s = "Hello, world!";
```

Der Typ von `s` ist `&str`: Es ist ein String Slice, der auf einen bestimmten Bereich einer Zeichenkette verweist. Deshalb sind String Literale unveränderlich. `&str` ist eine unveränderliche Referenz.

Andere Slices

String Slices sind speziell für Strings. Es gibt auch einen allgemeineren Slice Typ für Felder, bei dem wie bei String Slices auch hier Teile von Felder referenziert werden können:

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

Der Typ von `slice` ist `&[i32]`. Auch hier wird ein Pointer auf das erste Element sowie die Länge des Slice gespeichert.

Wie in Unterabschnitt 3.1.5 erwähnt, können Slices dazu verwendet werden, Felder an Funktionen über einen „call by reference“ zu übergeben:

```
fn main() {
    let array = [1, 2, 3, 4, 5];
    call_by_reference(&array);
}

fn call_by_reference(slice: &[i32]) {
    for element in slice.iter() {
        println!("{}", element);
    }
}
```

3.2.4 Lifetime Operator

Folgende Funktion gibt eine Referenz eines Inputs zurück:

```
fn ref_test(a: &i32, b: &i32) -> &i32 {
    a
}
```

Bei Aufruf dieser Funktion entsteht ein Problem:

```
fn main() {
    let a = 549;

    let e;
    {
        let tmp = a;
        e = ref_test(&a, &tmp); // e is a reference of a
    }
    // tmp goes out of scope

    // cant use e because tmp does not live long enough
    println!("{}", e);
}
```

Der Compiler kann nicht feststellen, ob die Lebensdauer des Rückgabewerts von der Lebensdauer der übergebenen Referenzen abhängig ist. Darum müssen die Lebenszeiten mit dem Lifetime Operator im Funktionskopf angegeben werden.

```
// doesn't work yet
fn ref_test<'a>(a: &'a i32, b: &'a i32) -> &'a i32 {
    a
}
```

Lifetimes werden in Rust mit einem Apostroph geschrieben. Obiger Code definiert eine Lebensdauer `'a` in den spitzen Klammern. Den Parametern und dem Rückgabewert wird diese Lebensdauer zugeordnet. Der Compiler interpretiert daraus, dass beide Parameter die gleiche Lebensdauer haben könnten. Damit der Code aus der `main` Funktion fehlerfrei kompiliert werden kann, muss der Parameter `b` mit einer anderen Lebensdauer markiert werden:

```
// now it works
fn ref_test<'a, 'b>(a: &'a i32, b: &'b i32) -> &'a i32 {
    a
}
```

Jetzt weiß der Compiler, dass die Lebendauer des Rückgabewerts abhängig vom ersten Parameter `a` ist. Es kann also sichergestellt werden, dass der Wert hinter der Referenz `e` nach dem Block weiterlebt.

3.3 Modulsystem

Dieses System von Rust ermöglicht die Erstellung von Modulen, welche mit Pfaden angesprochen werden können. Hiermit können auch externe Pakete verwendet werden:

Module lassen den Code in Gruppen einteilen:

```
mod sound {
    mod instrument {
        mod string {
            fn guitar() {}
        }
    }
    mod voice {}
}
```

Das Beispiel definiert das Modul `sound` mit zwei inneren Modulen `instrument` und `voice` und ein Modul `string` in `instrument` mit der Funktion `guitar`. Der gesamte Modulbaum ist unter dem impliziten Modul namens `crate` verwurzelt.

Diese Art von Baum erinnert an das Dateisystem eines Betriebssystems. Genau wie Verzeichnisse in einem Dateisystem, kann im Code durch Module Code organisiert werden. Eine weitere Ähnlichkeit zu Dateisystemen besteht darin, dass zum Verweisen auf ein Element dessen Pfad verwendet wird.

Es gibt zwei Arten von Pfaden:

- Ein absoluter Pfad beginnend beim crate root mithilfe des crate-Namens oder dem Schlüsselwort `crate`.
- Ein relativer Pfad beginnend beim aktuellen Modul, er verwendet `self`, `super` oder einen Bezeichner im aktuellen Modul.

Das Trennzeichen besteht aus zwei Doppelpunkten (`::`). Folgende `main`-Funktion könnte unter dem letzten Codebeispiel folgen:

```
fn main() {
    // absolute path
    crate::sound::instrument::string::guitar();

    // relative path
    sound::instrument::string::guitar();
}
```

Dieser Code würde jedoch nicht kompilieren, da alle Elemente (Funktionen, Methoden, Strukturen, Enums und Konstanten) standardmäßig privat sind. Um ein Element öffentlich zu machen, kann das Schlüsselwort `pub` verwendet werden.

Elemente ohne das Schlüsselwort `pub` sind privat, wenn der Modulbaum des aktuellen Moduls „nach unten“ betrachtet wird. Elemente ohne `pub` sind jedoch öffentlich, wenn sich diese auf derselben Ebene befinden, oder der Baum „nach oben“ betrachtet wird. In einem Dateisystem verhält es sich ähnlich: Ohne Berechtigung auf einen Ordner kann dieser nicht betrachtet werden. Auf einen zugänglichen Ordner können dessen Vorgängerverzeichnisse auch eingesehen werden.

Um die Funktion `guitar` in der `main`-Methode aufrufen zu können, müssen einige Module sowie die Funktion öffentlich gemacht werden:

```
mod sound {
    pub mod instrument {
        pub mod string {
            pub fn guitar() {}
        }
    }
    mod voice {}
}
```

Relative Pfade können mit `super` auch auf übergeordnete Module zugreifen, ähnlich wie bei einem Dateisystem mit „`..`“:

```
mod sound {
    pub mod instrument {
        pub fn clarinet() {
            super::breathe_in();
        }
    }
    fn breathe_in() {
        // Function body code
    }
}
```

Häufig genutzte Pfade können mit dem Schlüsselwort `use` gekürzt werden. Das ist vergleichbar mit einer Verknüpfung in einem Dateisystem. Es können absolute und relative Pfade benutzt werden.

```
use crate::sound::instrument;

fn main() {
    instrument::clarinet();
}
```

Externe Pakete können mit `use` eingebunden werden. Dafür muss zunächst das Paket via Cargo.toml hinzugefügt werden:

```
[dependencies]
rand = "0.6.5"
```

Im Code würde das so aussehen:

```
use rand::Rng;

fn main() {
    let secret = rand::thread_rng().gen_range(1, 101);
}
```

Der Quellcode kann mit dem Schlüsselwort `mod` auf mehrere Dateien aufgeteilt werden. Dabei ist der Name der Datei der Name des Moduls. Ordnerstrukturen können auch erstellt werden. Dabei ist der Ordner selbst auch ein Modul, dessen Definition in der Datei `mod.rs` definiert werden kann.

Datei **src/main.rs**

```
mod sound;

fn main() {
    sound::instrument::clarinet();
}
```

Datei **src/sound.rs**

```
pub mod instrument {
    pub fn clarinet() {
        // Function body
    }
}
```

3.4 Objektorientierung

In der Programmiergemeinschaft herrscht kein Konsens darüber, welche Merkmale eine Sprache besitzen muss, um objektorientiert zu sein. Rust wird von vielen Programmierparadigmen beeinflusst, darunter auch von objektorientierter Programmierung (OOP). Objektorientierte Sprachen teilen üblicherweise drei Gemeinsamkeiten: Objekte, Kapselung und Vererbung.

Das Buch *Design Patterns: Elements of Reusable Object-Oriented Software* von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides definiert OOP auf diese Weise:

> Object-oriented programs are made up of objects. An object packages both data and the procedures that operate on that data. The procedures are typically called methods or operations.

Nach dieser Definition ist Rust objektorientiert. Strukturen und Enums enthalten Daten, `impl` Blöcke stellen Methoden für diese bereit.

Eine Struktur in Rust ist vergleichbar mit `struct` in C. Rust kann diese Strukturen jedoch mit Methoden erweitern, sodass diese wie Klassen verwendet werden können. So wird eine Struktur mit drei Deklarationen `id`, `age` und `wage` definiert:

```
struct Employee {
    id: i32,
    age: i32,
    wage: f64,
}

impl Employee {
    fn new(id: i32, age: i32, wage: f64) -> Self {
        Employee { id, age, wage }
    }
    fn birthday(&mut self) {
        self.age += 1;
    }
}

fn main() {
    let mut joe = Employee::new(1, 32, 48000.0);
    joe.birthday();
}
```

Die Methode `new` ist der Konstruktor, dessen Rückgabetyp `Self` ist. `Self` ist der Typ von `impl`. Er könnte auch mit `-> Employee` definiert sein. Alternativ kann eine Instanz auch ohne Konstruktor erstellt werden:

```
let mut joe = Employee {
    id: 1,
    age: 32,
    wage: 48000.0,
};
```

Wenn in Methoden auf das eigene Objekt zugegriffen werden soll, muss dieses als ersten Funktionsparameter übergeben werden. Diese Übergabe geschieht automatisch: beim Aufruf `joe.birthday();` muss innerhalb der Klammern nichts geschrieben werden.

3.4.1 Stack oder Heap?

In C++ werden Objekte auf dem Stack gespeichert, wenn kein `new` Operator verwendet wird. In Rust werden Objekte zunächst auch auf dem Stack gespeichert.

C++ Code:

```
Employee e(1, 32, 48000.0);
```

Rust Code:

```
let e = Employee::new(1, 32, 48000.0);
```

Soll in C++ ein Objekt auf dem Heap alloziert werden, wird der `new` Operator verwendet. In Rust muss ein sogenannter Smart Pointer erstellt werden. Die Funktionsweise von Smart Pointer wird in Unterabschnitt 3.4.2 erläutert. Zunächst ist wichtig zu verstehen, dass in Rust jeder Datentyp auf dem Heap gespeichert werden kann.

C++ Code:

```
Employee *e = new Employee(1, 32, 48000.0);
delete e;
```

Rust Code:

```
let e = Box::new(Employee::new(1, 32, 48000.0));
```

3.4.2 Smart Pointer

Smart Pointer in Rust verhalten sich ähnlich wie Referenzen, mit dem Unterschied, dass sie mit einem Konstruktoraufruf instanziiert werden können. Sie sind also Strukturen bzw Klassen, die einen Pointer auf einen beliebigen Typ beinhalten. Genau wie Referenzen können sie mit dem * Operator dereferenziert werden. Es gibt verschiedene Smart Pointer, häufig genutzt werden folgende drei Arten:

- `Box<T>` wird verwendet, um Werte dem Stack zuzuweisen.
- `Rc<T>` erlaubt das Teilen von Ownership, realisert wird das durch Zählen von Referenzen.

- `Arc<T>` erlaubt das Teilen von Ownership für mehrere Threads, wenn ein Synchronisierungsmechanismus benutzt wird.

Verwendung von `Box<T>`

Der `Box<T>` Typ kann auch dazu verwendet werden, Ownership von großen Datentypen zu übertragen. So kann das Kopieren von vielen Daten ausgeschlossen werden.

```
fn main() {
    let large = Box::new(LargeStruct::new());
    take_boxed_ownership(large);
    // large is now invalid
}

fn take_boxed_ownership(x: Box<LargeStruct>) {
    // do stuff
}
```

3.4.3 Kapselung

Ein weiterer Aspekt, welcher häufig mit OOP in Verbindung steht, ist die Idee der Kapselung. Das bedeutet, dass die Implementierung eines Objekts nicht für den Code verfügbar ist, der dieses Objekt verwendet.

Mit Modulen kann in Rust gekapselt werden. Aber auch Strukturen und Enums, sowie dessen Werte, sind standardmäßig privat und können mit dem Schlüsselwort `pub` öffentlich gemacht werden.

```
pub mod people {
    pub struct Employee {
        pub id: i32,   // public
        age: i32,      // private
        wage: f64,     // private
    }
}
```

3.4.4 Vererbung

Vererbung ist ein Mechanismus, mit dem ein Objekt von der Definition eines anderen Objekts erben kann, wodurch die Daten und das Verhalten des übergeordneten Objekts übernommen werden, ohne dass sie erneut definiert werden müssen.

Wenn eine Sprache Vererbung können muss, um eine objektorientierte Sprache zu sein, dann ist Rust keine. Es gibt keine Möglichkeit eine Struktur zu definieren, die Felder und Methodenimplementierungen der übergeordneten Struktur erbt.

Ein Beispiel der Vererbung in C++:

```
class Employee {
  public:
    string first_name, family_name;
    char middle_initial;
    Date hiring_date;
    short department;
    // ...
};

class Manager : public Employee {
    list<Employee *> group;
    short level;
    // ...
};
```

Hier besitzt der Manager die gleichen Felder wie ein normaler Mitarbeiter, zusätzlich zu den eigenen Feldern.

Warum verzichtet Rust auf Vererbung?

In letzter Zeit ist die Vererbung als Entwurfslösung in vielen Programmiersprachen in Ungnade gefallen, da häufig die Gefahr besteht, dass mehr Code als erforderlich gemeinsam genutzt wird. Unterklassen sollten nicht immer alle Merkmale ihrer übergeordneten Klasse gemeinsam haben, dies geschieht jedoch mit Vererbung. Das kann das Design eines Programms weniger flexibel machen. Außerdem wird die Möglichkeit eingeführt, Methoden für Unterklassen aufzurufen, die keinen Sinn ergeben, oder Fehler verursachen, da die Methoden nicht für die Unterklasse gelten. Darüber hinaus können in einigen Sprachen nur Unterklassen von einer Klasse erben, wodurch die Flexibilität der Sprache weiter einschränkt wird.

3.4.5 Traits

Ein `trait` ist eine Sammlung von Methoden, die für einen unbekannten Typ definiert werden: `Self`. Sie können auf Methoden zugreifen, die im selben Trait definiert sind. Traits können für jeden Datentyp implementiert werden. [Rusf]

```
pub trait Animal {
    fn noise(&self) -> &str;
}
```

Dieses Trait definiert ein Tier, welches Geräusche von sich geben kann.

```
pub struct Sheep {
    pub name: String,
}
impl Animal for Sheep {
    fn noise(&self) -> &str {
        "baaaaaah"
    }
}
```

Die Struktur `Sheep` implementiert den Trait, sodass ein Schaf als Tier gesehen wird. Es kann nun eine Funktionen geschrieben werden, die ein beliebiges Tier erwartet:

```
pub fn make_animal_speak(animal: &dyn Animal) {
    println!("Animal says {}.", animal.noise());
}
```

Es können auch Vektoren mit Trait-Objekten erstellt werden:

```
let mut animals: Vec<Box<dyn Animal>> = Vec::new();
animals.push(Box::new(Sheep {}));
```

Das Schlüsselwort `dyn` ist optional, dient aber zur besseren Unterscheidung zwischen einem Trait (dyn) und einer Struktur und sollte daher bei Traitobjekten immer verwendet werden. Da `Animal` nur ein Trait ist und keine Struktur, wird das Schlüsselwort benutzt.

Der Typ `Box` ist ein Pointer mit einer festen Speichergröße, mit dem Werte auf dem Heap gespeichert werden können. Er wird benötigt, da zur Kompilierzeit die Größe des Datentyps noch nicht bekannt ist.

Trait-Objekte benutzen dynamische Bindung (dynamic dispatch)

Wenn Trait-Objekte für Funktionen verwendet werden, muss Rust die dynamische Bindung benutzen. Der Compiler kennt nicht alle Typen, die möglicherweise für den Code verwendet werden. Er weiß also nicht, welche Methode für welchen Typ der Aufruf implementiert ist. Daher verwendet Rust zur Laufzeit Zeiger im Trait-Objekt, um zu ermitteln, welche Methode aufgerufen werden soll. Wenn diese Suche ausgeführt wird, fallen Laufzeitkosten an, die beim statischen Binden (static dispatch) nicht auftreten würden. Die dynamische Bindung verhindert auch, dass der Compiler den Code einer Methode direkt an die entsprechende Stelle kopiert, wodurch Optimierungen verhindert werden können. Jedoch erhält man durch Traits zusätzliche Flexibilität im Code. Es ist also ein Kompromiss.

3.5 Generische Programmierung

Ähnlich wie in C++ (Templates) gibt es auch in Rust die Möglichkeit, Typen als Argumente zu übergeben, ohne Informationen zu verlieren. Das bedeutet: Es können Funktionssignaturen oder Strukturen erstellt werden für Elemente, die viele verschiedene konkrete Datentypen verwenden können.

Beispiel in Rust:

```
#[derive(Debug)]
struct Point<T> {
    x: T,
    y: T,
}

impl<T> Point<T> {
    fn new(x: T, y: T) -> Point<T> {
        Point { x: x, y: y }
    }
}

fn main() {
    let p1 = Point::new(1, 2);
    let p2 = Point::new(1.0, 2.0);
}
```

Die Klasse `Point` beinhaltet zwei Werte, deren Typen erst bei der Erstellung des Objekts bekannt werden. Beide Variablen `x` und `y` haben den gleichen Typ `T`. Die Variable `p1` ist ein `Point<i32>` und `p2` ein `Point<f64>`.

Es können auch Typen verwendet werden, die bestimmte Traits implementieren müssen:

```
fn animal_noise<T: Animal>(animal: T) {
    println!("This animal makes {}.", animal.noise());
}
```

C++ und Rust implementieren Generics so, dass Code mit generischen Typen nicht langsamer läuft als mit konkreten Typen. Rust erreicht dies dadurch, dass beim Kompilieren generischer Code in spezifischen Code umgewandelt wird und die konkreten Typen eingetragen werden. Dieser Prozess wird Monomorphisierung genannt.

3.6 Unit-Tests

Bei der Entwicklung von Rust wurde auf ein möglichst hohes Maß an Fehlerfreiheit in Programmen geachtet. Fehlerfreiheit ist jedoch nur schwer nachzuweisen. Das Typensystem von Rust kann nicht jede Art von Fehler verhindern, daher bietet Rust Unterstützung für das Schreiben automatisierter Softwaretests.

In C oder C++ gibt es Frameworks, welche von Dritten angeboten werden. Ein Beispiel ist Google Test[1].

Für Unit-Tests in Rust wird nach Konvention in jeder Datei ein Modul mit dem Namen `tests` erstellt mit der Annotation `#[cfg(test)]`. Hier werden Testfunktionen definiert mit der Annotation `#[test]`. Der Test `it_works` wird mit dem Befehl `cargo test` gestartet.

```
#[cfg(test)]
mod tests {
    #[test]
    fn it_works() {
        assert_eq!(2 + 2, 4);       // assert equal
        assert_ne!(2 + 2, 22);      // assert not equal
    }
}
```

[1] https://github.com/google/googletest

Folgende Makros sind hilfreich bei der Erstellung von Tests:

- `assert!(a)`: Prüft einen booleschen Ausdruck; Panik bei `false`.
- `assert_eq!(a,b)`: Vergleicht zwei Werte; Panik bei unterschiedlichen Werten.
- `assert_ne!(a,b)`: Wie `assert_eq!`; Panik bei gleichen Werten.
- `panic!()`: Generiert Panik und der Test schlägt fehl.

Diese Makros können zusätzlich einen Text für mehr Informationen ausgeben:

```
assert_eq!(a, b, "Testing equality of {} and {}", a, b);
```

3.7 Error Handling

3.7.1 Fehler mit `panic!`

Wenn im Code ein Fehler auftritt, kann das Makro `panic!` benutzt werden. In Abschnitt 3.6 wurde es bereits benutzt, um Tests fehlschlagen zu lassen. Beim Aufruf dieses Makros gibt das Programm eine Fehlermeldung aus, räumt Stack und Heap auf und beendet sich. Paniken werden immer zur Laufzeit ausgelöst.

Wenn also eine Panik auftritt, wird das Programm standardmäßig beendet. Das bedeutet, dass das Programm selbst noch den Speicher der Objekte wieder freigibt. Das Zurückgehen des Stacks und das Aufräumen, ist jedoch eine Menge Arbeit. Die Alternative ist, sofort abzubrechen, wodurch das Programm ohne Bereinigung beendet wird und der vom Programm verwendete Speicher vom Betriebssystem bereinigt werden muss. Wenn in einem Projekt die resultierende Binärdatei so klein wie möglich gestaltet werden soll, kann bei einer Panik vom „Abwickeln“ auf „Abbrechen“ gewechselt werden, indem in der Cargo.toml `panic = 'abort'` in die entsprechenden Abschnitte geschrieben wird:

```
[profile.release]
panic = 'abort'
```

Eine Panik kann beispielsweise auch ausgelöst werden, wenn im Code auf ein Element eines Vektors zugegriffen wird, welches nicht innerhalb des Bereichs liegt:

```
fn main() {
    let v = vec![1, 2, 3];
    v[99];
}
```

In diesem Codebeispiel wird mithilfe eines Makros ein Vektor mit drei Zahlen erstellt. Es wird dann auf das 99. Element zugegriffen. Dadurch entsteht folgende Fehlermeldung zur Laufzeit:

```
thread 'main' panicked at 'index out of bounds: the len
is 3 but the index is 99'
```

In C und C++ würden bei Felder keine Fehlermeldungen erscheinen und es würde ein Zahlenwert ausgegeben werden, der davon abhängig ist, was in diesem Moment im Speicher steht. In C++ können alternativ Listen erstellt werden (z. B. `vector<int>`), die zur Laufzeit überprüfen können, ob die Zugriffe innerhalb des Bereichs der Liste stattfinden.

3.7.2 Fehler mit `Result`

Viele Fehler sind nicht schwerwiegend genug, um ein vollständiges Anhalten des Programms zu erfordern. Wenn beispielsweise eine nicht existierende Datei geöffnet werden soll, könnte diese erstellt werden, anstatt das Programm zu beenden.

Das Enum `Result` ist in der Standardbibliothek definiert:

```
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

Bei `T` und `E` handelt es sich um generische Typparameter. Dabei stellt `T` den Typ des Werts dar, der in einem Erfolgsfall innerhalb der `Ok` Variante zurückgegeben wird und `E` die Art des Fehlers, der in einem Fehlerfall innerhalb der `Err` Variante zurückgegeben wird.

So wird eine Funktion aufgerufen, die ein `Result` zurückgibt, weil sie fehlschlagen könnte.

```
use std::fs::File;
fn main() {
    let f = File::open("hello.txt");
}
```

In der Standardbibliothek steht beschrieben, welchen Typ die Funktion zurückgibt. `f` ist vom Typ `Result<File, Error>`. Mithilfe des `match`-Statements kann

getestet werden, ob die Funktion korrekt ausgeführt wurde, oder, bei Misserfolg, welche Art von Fehler entstanden ist.

```
use std::fs::File;
use std::io::ErrorKind;

let f = match f {
    Ok(file) => file,
    Err(error) => match error.kind() {
        ErrorKind::NotFound => {
            panic!() /* do stuff */
        },
        _ => panic!("could not open file"),
    },
};
```

Der Aufruf `error.kind()` gibt ein Enum `ErrorKind` zurück, womit auf verschiedene Fehlerszenarien reagiert werden kann.

Fehler weitergeben

Beim Schreiben einer Funktion, welche möglicherweise einen Fehler aufruft, kann dieser an den aufrufenden Code zurückgegeben werden, anstatt ihn in dieser Funktion zu behandeln. Somit wird dem aufrufendem Code mehr Kontrolle gegeben. Anstelle des Aufrufs von `panic!`, wird ein `return`-Statement verwendet. Alternativ kann der ?-Operator verwendet werden.

3.7.3 ?-Operator

Wird ein ? hinter ein `Result` geschrieben und der Wert ist ein `Ok`, wird der Wert innerhalb des `Ok` von diesem Ausdruck zurückgegeben und die Funktion wird fortgesetzt. Wenn es sich bei dem Wert um einen Fehler handelt, wird der Fehler von der gesamten Funktion zurückgegeben, als ob das Schlüsselwort `return` verwendet worden wäre. Dieser Operator kann nur innerhalb von Funktionen verwendet werden, welche ein `Result` als Rückgabetyp haben. Dadurch wird die Implementierung vereinfacht und die Lesbarkeit des Codes verbessert.

Codebeispiel:

```
use std::io;
use std::Read;
use std::fs::File;

fn read_username() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?;
    let mut s = String::new();
    f.read_to_string(&mut s)?;
    Ok(s)
}
```

Die Methode `File::open` gibt ein `Result<File, io::Error>` zurück. In Verbindung mit dem ?-Operator wird bei erfolgreichem öffnen der Datei die Variable `f` mit dem Typ `File` gesetzt. Ansonsten wird die Funktion beendet und gibt den `io::Error` in einem `Err` verpackt zurück. Also `Err(io::Error)`.

3.7.4 Error Handling in C und C++

In C gibt es keine direkte Unterstützung für die Fehlerbehandlung. Es gibt aber Möglichkeiten, wie sie dennoch durchgeführt werden kann. Viele C-Funktionsaufrufe geben im Fehlerfall `-1` oder `NULL` zurück, sodass mit einer `if`-Anweisung entsprechend reagiert werden kann. Die globale Variable `errno` kann zusätzlich zur Unterscheidung verwendet werden.

C++ bietet einige hilfreiche Features. Die `throw`-Anweisung kann Ausnahmen an einen Handler übergeben, ähnlich wie es beispielsweise in Java üblich ist. Zusätzlich können statische Assertionen benutzt werden, um zur Kompilierzeit bestimmte Eigenschaften sicherzustellen.

3.8 Dokumentieren mit rustdoc

Ein wichtiger Bestandteil bei der Programmierung in Rust ist das Erstellen von Dokumentationen. Das Tool `rustdoc` ist vergleichbar mit `javadoc` für Java. Es können im Code spezielle Kommentare geschrieben und daraus eine Dokumentation mit HTML, CSS und JavaScript erstellt werden.

3.8.1 Grundlegende Verwendung

Folgendes Beispiel zeigt die grundsätzliche Verwendung der rustdoc Kommentare. Sie werden mit drei, statt zwei Schrägstrichen gekennzeichnet. Alternativ kann die Annotation `#[doc]` verwendet werden.

```
/// foo is a function
pub fn foo() {}
```

Alternative Schreibweise:

```
#[doc = "foo is a function"]
pub fn foo() {}
```

Es können in den Kommentarzeilen Elemente aus Markdown verwendet werden, um z. B. Überschriften oder Listenaufzählungen zu erstellen.

Die Dokumentation kann auf der Kommandozeile mit einem Cargo-Befehl generiert werden:

```
$ cargo doc
```

Nach dem Kompilieren ist die `index.html` anschließend zu finden unter dem Pfad `./target/doc/[crate-name]/index.html`.

3.8.2 Dokumentationstests

Rust erlaubt die Ausführung von Tests innerhalb der Codebeispiele in der Dokumentation. Codebeispiele werden mit drei Akzentzeichen (‘‘‘) am Anfang und am Ende des Codes gekennzeichnet.

```
/// # Examples
/// ‘‘‘
/// let x = 5;
/// assert_eq!(x, 5);
/// ‘‘‘
```

Bei einem Aufruf von `cargo test` wird dieser Code ausgeführt. Der Code muss also kompilierbar sein. Alle `assert`-Funktionen können hier verwendet werden.

Es gibt ein paar nützliche Annotationen, mit denen Dokumentationstests zusätzlich angepasst werden können.

- `ignore`: Der Code wird von Rust ignoriert.

```
/// ```ignore
/// fn foo() {
/// ```
```

- `should_panic`: Der Test sollte eine Panik auslösen.

```
/// ```should_panic
/// assert!(false);
/// ```
```

- `no_run`: Der Code wird kompiliert aber nicht ausgeführt.

```
/// ```no_run
/// loop {
///     println!("Hello, world");
/// }
/// ```
```

- `compile_fail`: Der Code soll beim Kompilieren einen Fehler ausgeben.

```
/// ```compile_fail
/// let x = 5;
/// x += 2; // shouldn't compile!
/// ```
```

3.9 WebAssembly

WebAssembly ist eine Technologie, mit der performante Webanwendungen programmiert werden können. Es handelt sich dabei um einen Binärcode, der aus verschiedenen Programmiersprachen generiert werden kann. Darunter C, C++ und Rust. Rust bietet Programmierern eine Low-Level-Kontrolle und zuverlässige Leistung. Es ist frei von Pausen durch garbage collection aus JavaScript und Programmierer haben Kontrolle über Pointer, Monomorphisierungen und Speicherlayout.

Zum Erstellen von WebAssembly Projekten in Rust werden die Cargo-Pakete `wasm-pack`, `cargo-generate` und der Paketmanager für JavaScript NPM benötigt. [Ruse]

3.9.1 Beispielprojekt in Rust anlegen

Rust bietet eine Vorlage auf Github an, die mit Cargo heruntergeladen werden kann.

```
$ cargo generate --git https://github.com/rustwasm/wasm-pack-template
```

Dadurch entsteht ein Rustprojekt mit einer Cargo.toml sowie zwei Quellcodedateien `lib.rs` und `utils.rs` im `src`-Ordner.

Die Cargo.toml ist vorkonfiguriert für die Erstellung von `.wasm` Bibliotheken. Die Quelldateien definieren ein Beispielprogramm, welches die Funktion `alert` aus JavaScript in Rust importiert und eine Funktion `greet` von Rust exportiert.

Um das Projekt zu bauen, müssen die Rust-Quelldateien in `.wasm` Binärdateien kompiliert und eine JavaScript API generiert werden, um das generierte WebAssembly zu benutzen. Dies geschieht mit einem Befehl:

```
$ wasm-pack build
```

Die generierten Dateien werden im Ordner `pkg` gespeichert. Dort befindet sich die `.wasm`-Datei und eine `.js`-Datei, welche die WebAssembly-Daten importiert und die in Rust programmierte `greet` Funktion als JavaScript-Funktion verpackt.

Webseite

Um den Code auf einer Webseite zu benutzen, kann mit Node.js die Projektvorlage `create-wasm-app` verwendet werden:

```
$ npm init wasm-app www
```

Nun müssen die Abhängigkeiten installiert werden. Folgender Befehl muss im Unterordner `./www` ausgeführt werden:

```
$ npm install
```

In der Datei `./www/package.json` muss bei den Abhängigkeiten das Projekt hinzugefügt werden:

```
"devDependencies": {
    "name-of-project": "file:../pkg",
    // ...
}
```

Im JavaScript-Code kann nun `name-of-project` importiert werden. Jetzt muss die neue Abhängigkeit installiert werden und der Webserver kann gestartet werden. Im Verzeichnis `./www`:

```
$ npm install
$ npm run start
```

Im Browser kann unter der URL http://localhost:8080/ die Webseite aufgerufen werden.

4 Performance

4.1 zero-cost in Rust

Das Typsystem in Rust ermöglicht viele sogenannte zero-cost-Abstraktionen. Dadurch kann der Code in Rust leserlich und überschaubar gestaltet werden, ohne dabei auf bestimmte Aspekte zu achten, die in anderen Programmiersprachen für einen Performanceverlust sorgen würden.

Folgender Code zeigt eine zero-cost-Abstraktion in Rust [McL18]:

```
struct Point2D {
    x: f64,
    y: f64,
}

impl Point2D {
    fn move_right(self, distance: f64) -> Point2D {
        Point2D {
            x: self.x + distance,
            y: self.y,
        }
    }
}

fn do_stuff(input: f64) -> f64 {
    let p1 = Point2D{ x: 3.0, y: 7.0 };
    let p2 = p1.move_right(input);
    p2.x
}
```

Die Funktion `do_stuff` bekommt eine Zahl, addiert sie mit `3` und gibt diese wieder zurück. Im Code wurde dazu eine Struktur `Point2D` mit der Funktion

`move_right` verwendet. Der Compiler erkennt, dass der Code der Funktion auf eine einfache Addition verkürzt werden kann:

```
fn do_stuff(input: f64) -> f64 {
    3.0 + input
}
```

Ein anderes Beispiel ist die Verwendung von Traits als Input für Funktionen:

```
trait Logger {
    fn error(&self, message: &str);
    fn info(&self, message: &str);
}

struct NullLogger {}

impl Logger for NullLogger {
    fn error(&self, message: &str) {}
    fn info(&self, message: &str) {}
}

struct PrintLogger {}

impl Logger for PrintLogger {
    fn error(&self, message: &str) {
        println!("ERROR: {}", message);
    }
    fn info(&self, message: &str) {
        println!("INFO: {}", message);
    }
}

fn log_example(logger: &dyn Logger) {
    logger.info("example info");
    logger.error("example error");
}
```

Hier weiß Rust nicht, welche Art von Logger die Funktion `log_example` benutzt. Die Optimierung der Funktion wird verhindert, da der Compiler erst zur Laufzeit

wissen kann, wie die Funktionen `logger.info` und `logger.error` implementiert sind. Siehe „dynamic dispatch“ in Abschnitt 3.4.5.

Mithilfe von generischen Funktionen können solche Optimierungen dennoch vorgenommen werden, da der Compiler für jede Art von Logger eine eigene Funktion erstellt, also „static dispatch“ verwendet. Die generische Funktion könnte so aussehen:

```
fn log_example<L: Logger>(logger: &L) {
    logger.info("example info");
    logger.error("example error");
}
```

Der Vorteil der Benutzung von generischen Funktionen besteht darin, dass Rust mehr Optimierungen vornehmen kann. Der Nachteil ist jedoch, dass die ausführbaren Binärdateien mehr Speicherplatz benötigen und das Kompilieren länger dauert.

4.2 Benchmark-Tests

Um festzustellen, wie performant Rust-Programme im Verhältnis zu anderen Sprachen laufen, können verschiedene Arten von Programmen ausgeführt und gemessen werden. In diesem Kapitel werden Daten aus dem Computer Language Benchmarks Game[1] zitiert. Die Programme können von Usern hochgeladen werden und sind im Internet frei einsehbar. Sie wurden so programmiert, dass sie übliche Mittel der jeweiligen Sprache verwenden. Beispielsweise wird in C++ der `new` Operator verwendet.

In den Balkendiagrammen ist Rust 100% und die Prozentzahlen für C und C++ werden von den Ergebnissen aus Rust berechnet.

Rust	rustc 1.35.0 (3c235d560 2019-05-20) LLVM version 8.0
C gcc	gcc (Ubuntu 8.3.0-6ubuntu1) 8.3.0
C++ g++	g++ (Ubuntu 8.3.0-6ubuntu1) 8.3.0

Tab. 4.1: Compiler der Benchmark-Tests

[1] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

4.2.1 Die Benchmarkprogramme

binary trees In diesem Test geht es darum, einen vollständigen Binärbaum zu erstellen, den Baum „herunterzuklettern“ und dabei die Knotenpunkte zu zählen und den Baum anschließend vom Speicher zu löschen.

fannkuch-redux Eine Permutation von {1,...,n}, z. B. {4,2,1,5,3}. Die erste Zahl ist 4, es sollen also die ersten 4 Elemente getauscht werden. Das wird so oft wiederholt bis die 1 an der ersten Stelle ist: {5,1,2,4,3} {3,4,2,1,5} {2,4,3,1,5} {4,2,3,1,5} {1,3,2,4,5}.

fasta Generierung von DNA-Sequenzen durch Kopieren aus gegebenen Sequenzen und Zufallszahlen. Dabei werden lineare und binäre Suchfunktionen verwendet.

k-nucleotide Eine Hash-Tabelle wird zum Zählen von Vorkommen bestimmter Zeichenketten aus einer DNA-Sequenz benutzt.

mandelbrot Ein Mandelbrot wird auf eine $N * N$ Bitmap geplottet.

n-body Die Modellierung von Umlaufbahnen von Gasriesen mit einem symplektischen Integrator, also die Bewegung von Partikeln in einem Raum, dessen Masse von einem einzelnen Objekt dominiert wird.

pidigits Berechnung von Pi mithilfe von Langzahlarithmetik.

regex-redux Reguläre Ausdrücke anwenden um Übereinstimmungen zu finden.

reverse-complement Rückwärtskomplement einer DNA-Sequenz berechnen.

spectral-norm Berechnung der Spektralnorm einer Matrix.

4.2.2 Ausführungszeit

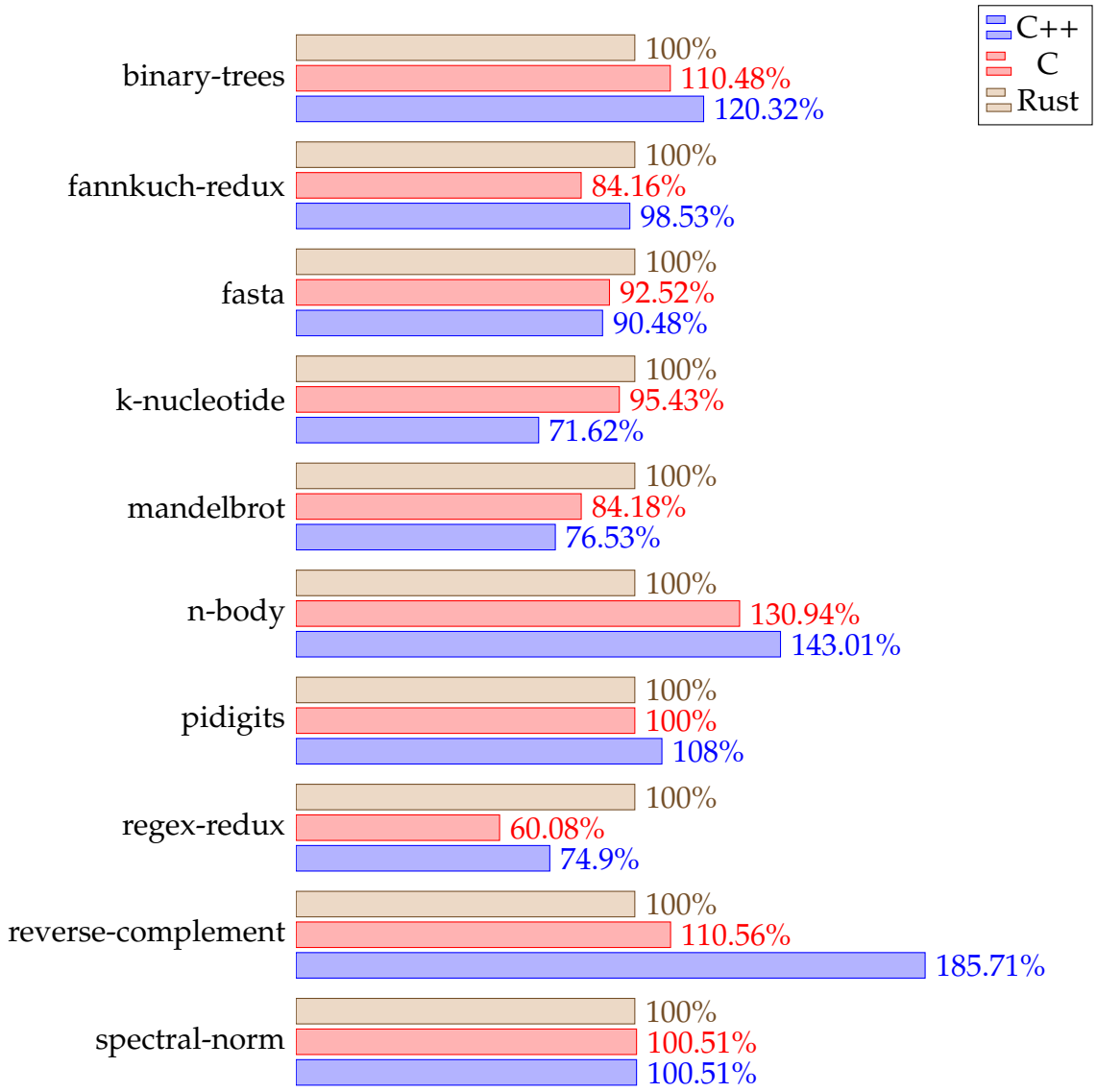

Abb. 4.1: Ausführungszeit: Rust im Verhältnis zu C und C++

In diesen Tests läuft C schneller als C++, jedoch verfügt C++ über viele zusätzliche Funktionen, die die Programmierung erleichtern können. Zusätzliche Funktionen besitzt auch die Sprache Rust, die durchschnittlich ähnlich wie C performt.

Besonders bei Berechnungen mit Fließkommazahlen im Test „n-body“ ist Rust sehr performant, verliert aber bei anderen Aufgaben gegenüber C/C++. Dank der schnellen Ausführungszeit, eignet sich Rust für eingebettete Systeme.

Bei eingebetteten Systemen ist das Verhalten bei einer Panik nicht definiert. Das Verhalten bei einer Panik kann in Rust jedoch neu definiert werden. [Rusc]

4.2.3 Speicherverbrauch

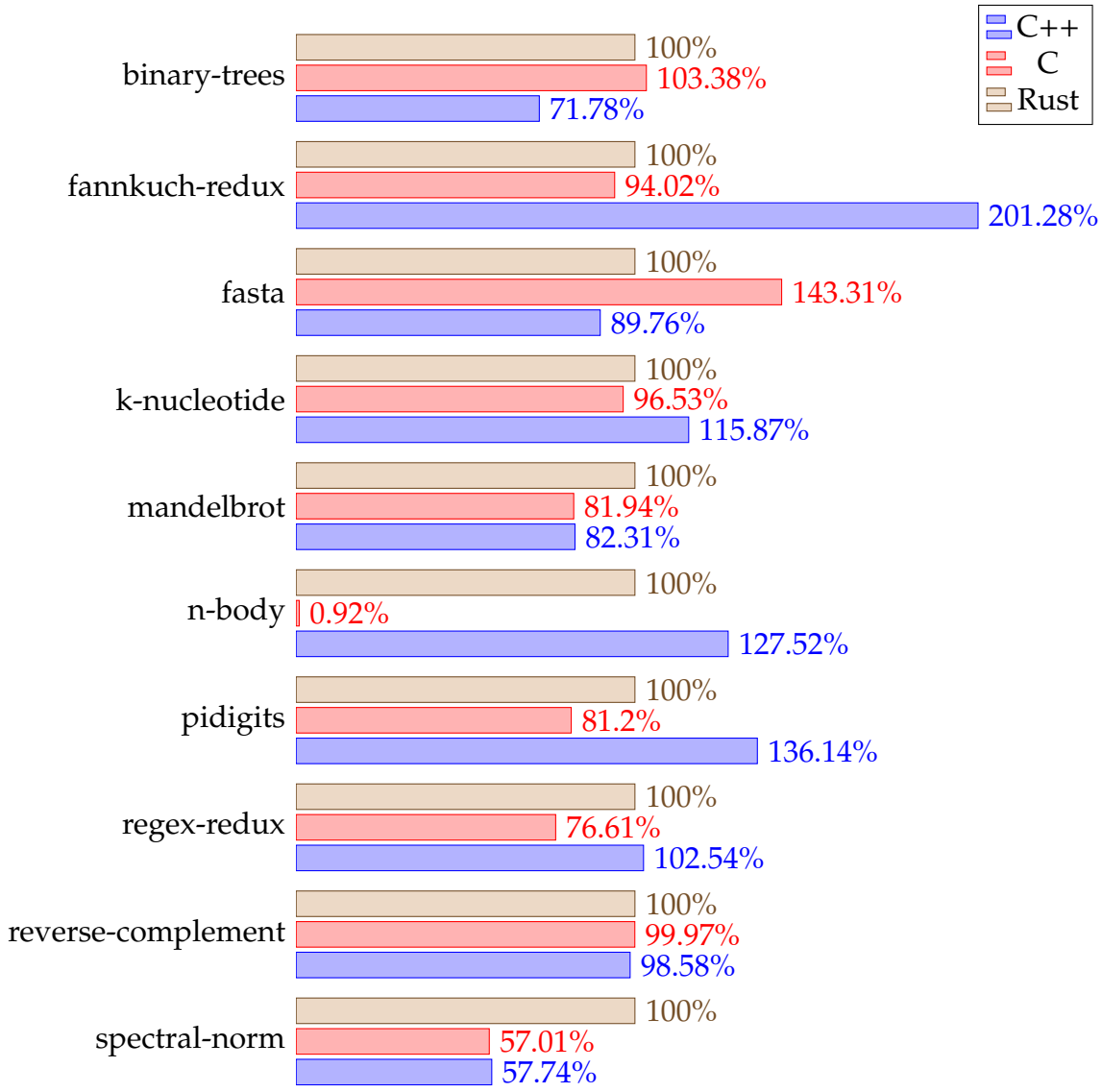

Abb. 4.2: Speicherverbrauch: Rust im Verhältnis zu C und C++

In C kann dank der Low-Level-Orientierung viel Speicher eingespart werden. Jedoch gehen hier die Ergebnisse weit auseinander, wie z. B. bei den Tests „fannkuch-redux“ und „n-body“.

Rust benötigt in den meisten Fällen nicht auffallend viel Speicher und ist dadurch als Systemprogrammiersprache geeignet.

Literaturverzeichnis

[ISO] ISO. *N2176 — International Standard ISO/IEC 9899:2017 Programming languages — C*. International Organization for Standardization, Geneva, Switzerland.

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[KR90] Brian W. Kernighan and Dennis M. Ritchie. *Programmieren in C*. Carl Hanser Verlag GmbH & Co. KG, 2 edition, 1990.

[McL18] Tim McLean. An introduction to structs, traits, and zero-cost abstractions - rust kw meetup. https://youtu.be/Sn3JklPAVLk, May 2018.

[Moz] Mozilla Foundation. Rust language. https://research.mozilla.org/rust/.

[Rusa] Rust Project Developers. The cargo book. https://doc.rust-lang.org/cargo/.

[Rusb] Rust Project Developers. The edition guide. https://doc.rust-lang.org/stable/edition-guide/.

[Rusc] Rust Project Developers. The embedded rust book. https://rust-embedded.github.io/book/.

[Rusd] Rust Project Developers. Guide to rustc development. https://rust-lang.github.io/rustc-guide/.

[Ruse] Rust Project Developers. Rust and webassembly. https://rustwasm.github.io/docs/book/.

[Rusf] Rust Project Developers. Rust by example. https://doc.rust-lang.org/stable/rust-by-example/.

[Rusg] Rust Project Developers. The rustc book. https://doc.rust-lang.org/rustc/index.html.

[Rush] Rust Project Developers. Third party cargo subcommands. https://github.com/rust-lang/cargo/wiki/Third-party-cargo-subcommands.

[Str15] Bjarne Stroustrup. *Die C++-Programmiersprache*. Carl Hanser Verlag GmbH & Co. KG, 2015.

Abbildungsverzeichnis

Tabellenverzeichnis