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 jede Variable und Speicheradresse prüft.

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 System-Programmiersprache, 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 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. Es verhindert Segmentierungsfehler und gewährleistet Threadsicherheit. Dadurch können Entwickler Code schreiben, der ehrgeizig, schnell und korrekt ist.

Rust macht die Systemprogrammierung durch die Kombination von Leistung und Ergonomie zugänglicher. Es bietet starke Features wie Zero-Cost-Abstraktionen, sichere Speicherverwaltung durch einen strengen Compiler und Typsystem sowie 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 Zielsysteme, dabei kann rustup helfen. So kann man beispielsweise MUSL verwenden:

```
# 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. [Ruse]

2.2.1 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. Es müssen somit keine Objektdateien erstellt werden.

2.2.2 Lints

Ein Lint ist ein Werkzeug, das zur Verbesserung des Quellcodes verwendet wird. Der Rust-Compiler enthält eine Reihe von Lints, dadurch werden beim Kompilieren Warnungen und Fehlermeldungen ausgeben. Beispiel:

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

$ 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
```

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

Das ist das `unused_variables` Lint und besagt, dass eine Variable deklariert wurde, 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
```

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"
```

[2] Paketeregister der Rust-Community

```
[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.

Projekt-Layout

```
.
├── 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. 2.1: Dateibaum eines Rust Projekts

- 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.

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 bei 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.

```
[package]
# ...
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

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

Wenn ein Teamname angegeben wird, sind diese 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.

[3] Bibliothek oder Paket in Rust

2.3.3 Externe Tools

Cargo versucht, die Integration von Tools von Drittanbietern zu vereinfachen, z. B. für IDEs oder anderen Build-Systemen. Dazu verfügt Cargo über 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 von Cargo aufgerufen werden mit „cargo *command*“. 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 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. [Rusf]

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

Zu den Grundlagen einer jeden Programmiersprache gehört der Umgang mit Variablen und Datentypen. Kontrollstrukturen definieren dabei die Reihenfolge, in der Berechnungen durchgeführt werden.

3.1.1 Variablen und Mutabilität

In Rust sind Variablen standardmäßig unveränderlich. Das ist einer von vielen Faktoren, die Programmierer helfen sollen, Fehler im Code besser auffinden 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.

In C oder C++ ist jede Variablendefinition standardmäßig veränderlich, das heißt 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, ein konstant deklarierter Wert zu verändern, indem eine Variable mit einem nicht konstanten Typ verwendet 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
```

3.1.2 Datentypen

Jede Variable in Rust hat einen bestimmten Datentyp. In Rust wird unterschieden zwischen skalaren und zusammengesetzten Typen. Rust ist eine statisch typisierte Sprache, somit müssen die Typen aller Variablen zur Kompilierzeit bekannt sein. Der Compiler kann normalerweise ableiten, welcher Typ verwenden soll basierend auf dem Wert und wie er verwendet wird. 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 guess: u32 = "42".parse().expect("Not a number!");
```

Ohne Angabe des Typs würde der Rust-Compiler eine Fehlermeldung ausgeben.

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. Folgende Tabelle zeigt die Integertypen von Rust und C:

In C sollen mit `short` und `long` verschieden lange ganzzahlige Werte zur Verfügung stehen, soweit dies praktikabel ist; `int` wird normalerweise die natürliche

	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++

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]

Beispiel:

```
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 Computers ab, auf dem das Programm ausgeführt wird: 64 Bits bei 64-Bit-Architekturen und 32 Bits bei 32-Bit-Architekturen.

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 Arrays verwendet werden.

Weitere Typen in Rust

- Fließkomma-Typen: `f32` und `f64` (Standard ist `f64`)
- Boolean: `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);
}
```

Arrays

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

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

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

Arrays sind nützlich, um Daten auf dem Stack statt auf dem Heap zuweisen zu können oder um sicherzustellen, dass immer eine feste Anzahl von Elementen vorhanden sind. Ein Array ist jedoch nicht so flexibel wie der Vektortyp. Ein Vektor ist ein ähnlicher Auflistungstyp, der von der Standardbibliothek bereitgestellt wird und dessen Größe dynamisch verändert werden kann.

Die Schreibweise des Array-Typs erfolgt mit eckigen Klammern mit dem Typen der Elemente gefolgt von einem Semikolon und die Anzahl der Elemente für das Array:

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

Eine ähnliche Schreibweise wird zum Initialisieren eines Arrays verwendet, das für jedes Element den selben Wert enthält. Dabei wird der Anfangswert, dann ein Semikolon und die Länge des Arrays angegeben:

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

Das Array a enthält 5 Elemente, die zunächst alle auf den Wert 3 gesetzt wurden. Folgender Ausdruck erzeugt das gleiche Array:

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

Ein Array ist ein einzelner Speicherbereich, der auf dem Stack reserviert ist. Es kann mithilfe eines Indexes auf die Elemente zugegriffen werden:

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

Wenn in C oder C++ auf ein Element außerhalb des Bereichs eines Arrays zugegriffen wird, fällt dies beim Testen wesentlich weniger 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 sofortiges Beenden des Programms.

3.1.3 Funktionen

In Rust werden Funktionen und Variablennamen als Konvention in snake case geschrieben. 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 Typen mit einem Doppelpunkt getrennt. Bei mehreren Parametern werden diese mit Komma getrennt.

Bei Funktionen mit Rückgabewerten kann am Ende des Funktionskörpers der Rückgabewert als Expression ohne Semikolon geschrieben werden. Wenn eine Funktion früh beendet werden soll, kann ein `return` mit Rückgabewert benutzt werden. Der Rückgabetyp der Funktion muss mit einem Pfeil (->) geschrieben werden.

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

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. 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 der Expression benötigt wird.

In Rust muss der Typ der Expression ein Boolean sein. Folgendes Programm wird also nicht übersetzt:

```
fn main() {
    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 Boolean Typ benötigt wird, muss das obige Programm umgeschrieben werden:

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

Mehrere Bedingungen können auch in Rust 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 eine Expression verpacken um z. B. Werte von Variablen zu definieren, ähnlich wie der `?:`-Operator in C/C++:

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

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`.

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. Schleifen können in Rust auch als Expression verwendet werden wenn an dem `break` eine Expression 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;
    }
}
```

Das Programm gibt alle Werte aus dem Array aus. Alternativ kann hier mit einer `for`-Schleife gearbeitet werden. For-Schleifen funktionieren in Rust anders als in C oder C++, sie gleichen ehr einer `for-each`-Schleife 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);
    }
}
```

Mit `for`-Schleifen kann in Rust die Stabilität des Programms erhöht werden, da Zugriffe außerhalb eines Arrays verhindert werden können.

3.2 Ownership

Ownership ist ein einzigartiges Merkmal von Rust. Es ermöglicht Speichersicherheit zur Kompilierzeit ohne die Notwendigkeit eines Garbage Collectors. Daher ist es wichtig, das Ownership-Prinzip als Programmierer zu verstehen. In diesem Kapitel werden Ownership und damit zusammenhängende Eigenschaften wie Borrowing, Slices und wie Rust Daten im Arbeitsspeicher ablegt, 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, welches 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 aktiven Eigentümer.

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

Beispiel: String

Um die Eigentumsregeln zu veranschaulichen, ist ein komplexerer Datentyp notwendig als die, die in Unterabschnitt 3.1.2 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, welche 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!'
```

Um einen veränderlichen String zu erzeugen, muss Speicherplatz auf dem Heap zugewiesen werden, dessen größe zur Kompilierzeit unbekannt ist. 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

    // doo 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 eine feste Größe im Speicher haben und auf dem Stack gespeichert werden.

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 Strings werden in Rust im Heap gespeichert.

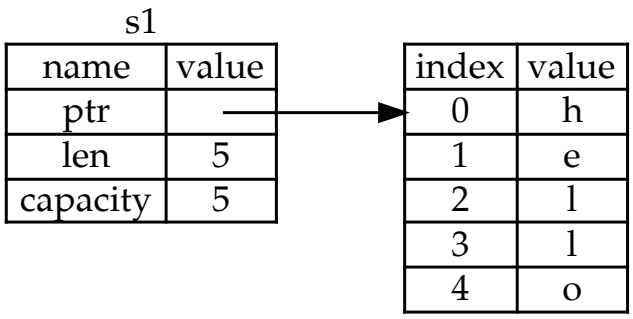

Abb. 3.1: Repräsentation des Speichers eines String

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. Diese Datengruppe wird auf dem Stack gespeichert. In der rechten Tabelle ist sich der Speicher auf dem Heap dargestellt.

Bei der Zuweisung von `s1` zu `s2` werden nur die String-Daten kopiert, das heißt der Pointer, die Länge und die Kapazität, welche sich auf dem Stack befinden. Die Daten des Heaps werden nicht kopiert.

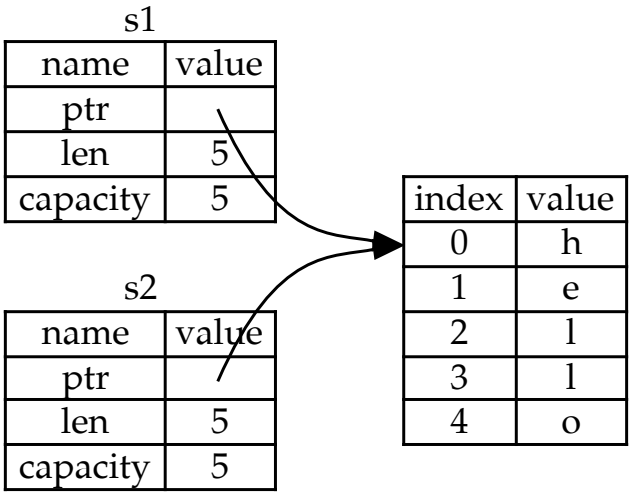

Abb. 3.2: Repräsentation des Speichers eines String: Kopie auf dem Stack

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.

Sobald eine Variable außerhalb des Gültigkeitsbereichs gelangt, wird der Speicher aus dem Heap gelöscht. Aber da Abbildung 3.2 zwei Variablen zeigt, die auf den selben Speicherbereich im Heap zeigen, würde zwei mal der Speicherbereich freigegeben werden. Das ist bekannt als „double free"-Error und kann zu Speicherbeschädigungen und möglicherweise zu Sicherheitsanfälligkeiten führen.

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

Das hat zu 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
```

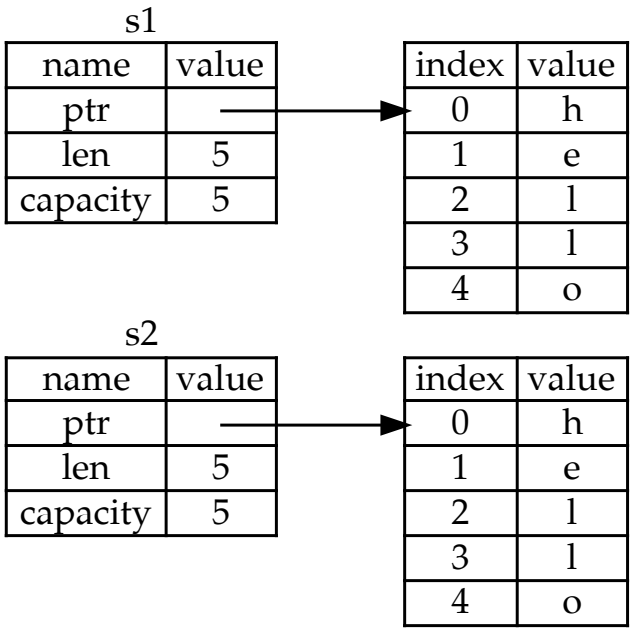

Abb. 3.3: Repräsentation des Speichers eines String: Vollständige Kopie

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.

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

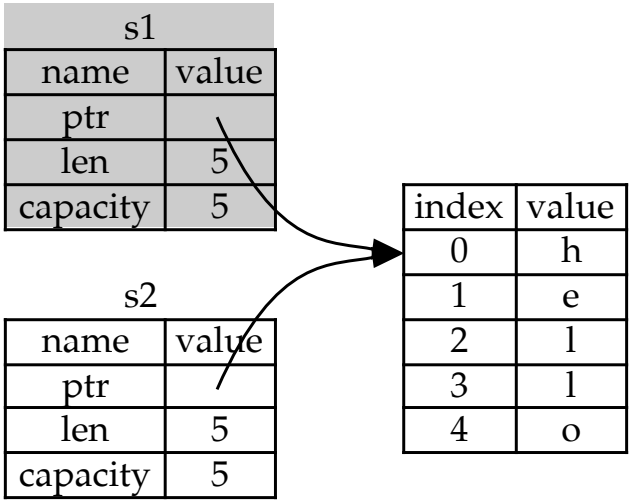

Abb. 3.4: Repräsentation des Speichers eines String: Moved

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.

Stack-Only-Daten kopieren

Primitive Datentypen, die komplett auf dem Stack gespeichert sind, können nicht übergeben werden (Move nicht möglich), sie können nur vollständig kopiert werden. Darum würde folgender Code ohne Fehler übersetzen:

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

println!("x = {}, y = {}", x, y);
```

Integer besitzen eine feste Speichergröße zur Kompilierzeit und es macht keinen Unterschied, sie mit `clone` zu kopieren. Wenn ein Typ den `Copy`-Trait besitzt, kann

die erste Variable nach der Zuweisung weiter verwendet werden. In der Regel kann jede Gruppe einfacher Skalarwerte über den Stack vollständig kopiert 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 Typen beinhalten, die den `Copy`-Trait implementieren z. B. `(i32, i32)`, jedoch nicht `(i32, String)`

Ownership und Funktionen

Die Semantik für die Übergabe eines Wertes an eine Funktion ähnelt derjenigen, die einer Variablen einen Wert zuweist. Beim Übergeben einer Variablen an eine Funktion wird genauso wie bei einer Zuweisung verschoben oder kopiert. Folgendes Programm enthält ein Beispiel mit einigen Anmerkungen, wann Variablen in den Geltungsbereich gelangen und wieder herauskommen:

```
fn main() {
    let s = String::from("hello");  // s comes
                                    // into scope
    takes_ownership(s);             // s's value moved into
                                    // the function...
    // ... and is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function
                                    // but i32 is Copy, so it's okay
                                    // to still use x afterward

} // Here, x goes out of scope, then s. But because s's
  // value was moved, nothing special happens.
```

```
fn takes_ownership(some_string: String) {
    // some_string comes into scope

    println!("{}", some_string);
} // Here, some_string goes out of scope and 'drop' is
  // called. The backing memory is freed.


fn makes_copy(some_integer: i32) {
    // some_integer comes into scope

    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing
  // special happens.
```

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

Eine Funktion, welche die Länge eines String 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 verwendet 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 und ermöglichen auf einen bestimmten Wert zu verweisen, ohne dessen Ownership zu beanspruchen. Abbildung 3.5 zeigt ein Diagramm hierzu.

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.

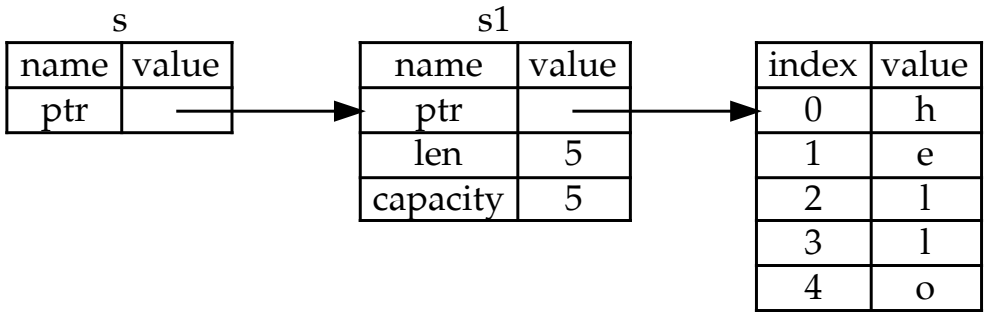

Abb. 3.5: Diagramm von `&String s` Zeiger auf `String s1`

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

Der Gültigkeitsbereich von `s` ist der eines normalen Funktionsparameters, jedoch wird der Wert nicht aus dem Speicher gelöscht, da keine Ownership an `s` übergeben wurde, weshalb auch keine zurückgegeben werden muss. Referenzen als Funktionsparameter werden in Rust „Borrowing“ genannt. Wie in der realen Welt kann Eigentum an andere ausgeliehen werden und 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"); // error
}
```

Veränderbare Referenzen

Kleine Veränderungen beheben den Fehler aus dem obigen Programm:

```
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 veränderbar gemacht werden durch Kennzeichnung mit `mut`. Es wird nun eine veränderbare Referenz übergeben und die Funktion erwartet nun auch eine entsprechende Referenz mit `some_string: &mut String`.

Veränderbare Referenzen haben 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 schreiben.
- 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
println!("{}, {}, and {}", r1, r2, r3);
```

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 Pointer 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, sollte 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.

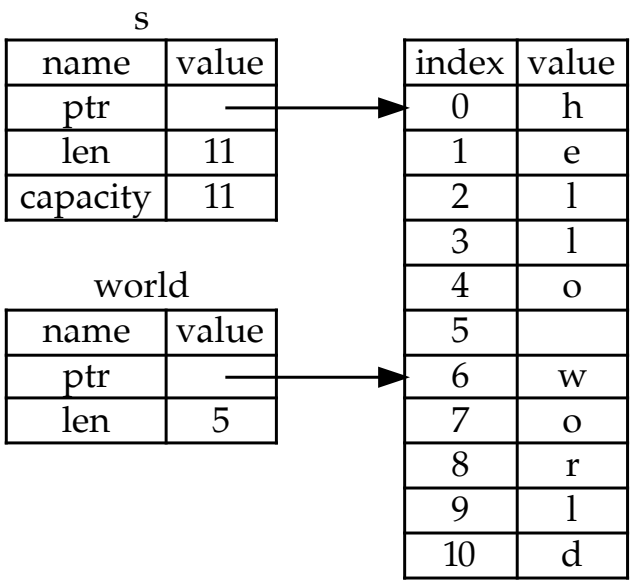

Abb. 3.6: Diagramm eines String Slice

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, 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 Arrays, bei dem wie bei String Slices auch hier Teile von Arrays 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.

3.3 Modulsystem

Dieses System von Rust ermöglicht die Erstellung von Modulen, welche mit Pfaden angesprochen werden können. Hiermit System 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 dem Schlüsselwort `pub` sind privat, wenn der Modulbaum des aktuellen Moduls „nach unten“ betrachtet wird. Elemente ohne `pub` sind jedoch öffentlich, wenn sich diese auf der selben 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 und `impl` Blöcke stellen Methoden für diese bereit.

Eine Struktur in Rust in vergleichbar mit `struct` in C oder C++. Rust kann diese Strukturen jedoch mit Methoden erweitern, sodass diese wie Klassen verwendet werden können.

Beispiel:

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

```
}

impl Employee {
    fn birthday(&mut self) {
        self.age += 1;
    }
}

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

3.4.1 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
    }
}
```

3.4.2 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.

3.4.3 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. [Rusd]

```
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 können nun Funktionen geschrieben werden, die eine beliebige Tiere erwarten:

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

Es können auch Listen von 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 (impl) und sollte daher immer verwendet werden.

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, sodass er nicht weiß, welche Methode für welchen Typ der Aufruf implementiert ist. Stattdessen 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

Rust wurde mit einem hohen Maß an Sorge um die Richtigkeit von Programmen entworfen. Richtigkeit ist jedoch nicht leicht nachzuweisen. Das Typensystem von Rust kann nicht jede Art von Fehler verhindern, daher bietet Rust Unterstützung für das Schreiben automatisierten 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]`. Diese Unit-Tests werden 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 Boolean-Wert; 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. Im letzten Kapitel wurde wurde es bereits erläutert, um Tests fehlschlagen zu lassen. Beim Aufruf dieses Makros gibt das Programm eine Fehlermeldung aus, räumt den Stack auf und beendet sich.

Wenn also eine Panik auftritt, wird das Programm standardmäßig beendet. Das bedeutet, dass Rust den Stack zurücksetzt und die Daten aller Funktionen bereinigt, auf die er stößt. Aber das Zurückgehen des Stacks und Aufräumen ist 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 auf das 100. Element des Listenelements `v` zugegriffen. Dadurch entsteht folgende Fehlermeldung:

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

In C und C++ würden bei Arrays keine Fehlermeldungen erscheinen und es würde ein Zahlenwert ausgegeben werden, der davon abhängig ist, was in diesem Moment im Speicher steht. In C++ können jedoch 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` stellt die Art des Fehlers dar, 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 => /*TODO: create file*/
                               panic!(),
        other_error => panic!("could not open: {:?}",
                              other_error),
    }
};
```

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 der Fehler an den aufrufenden Code zurückgegeben werden, anstatt den Fehler 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

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)
}
```

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 wurde. 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.

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 werden und daraus wird eine Dokumentation mit HTML, CSS und JavaScript erstellt.

3.8.1 Grundlegende Verwendung

Folgendes Beispiel zeigt die grundsätzliche Verwendung der Kommentare, sie werden mit drei 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. [Rusc]

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 3.4.3.

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 von 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.

Dabei 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

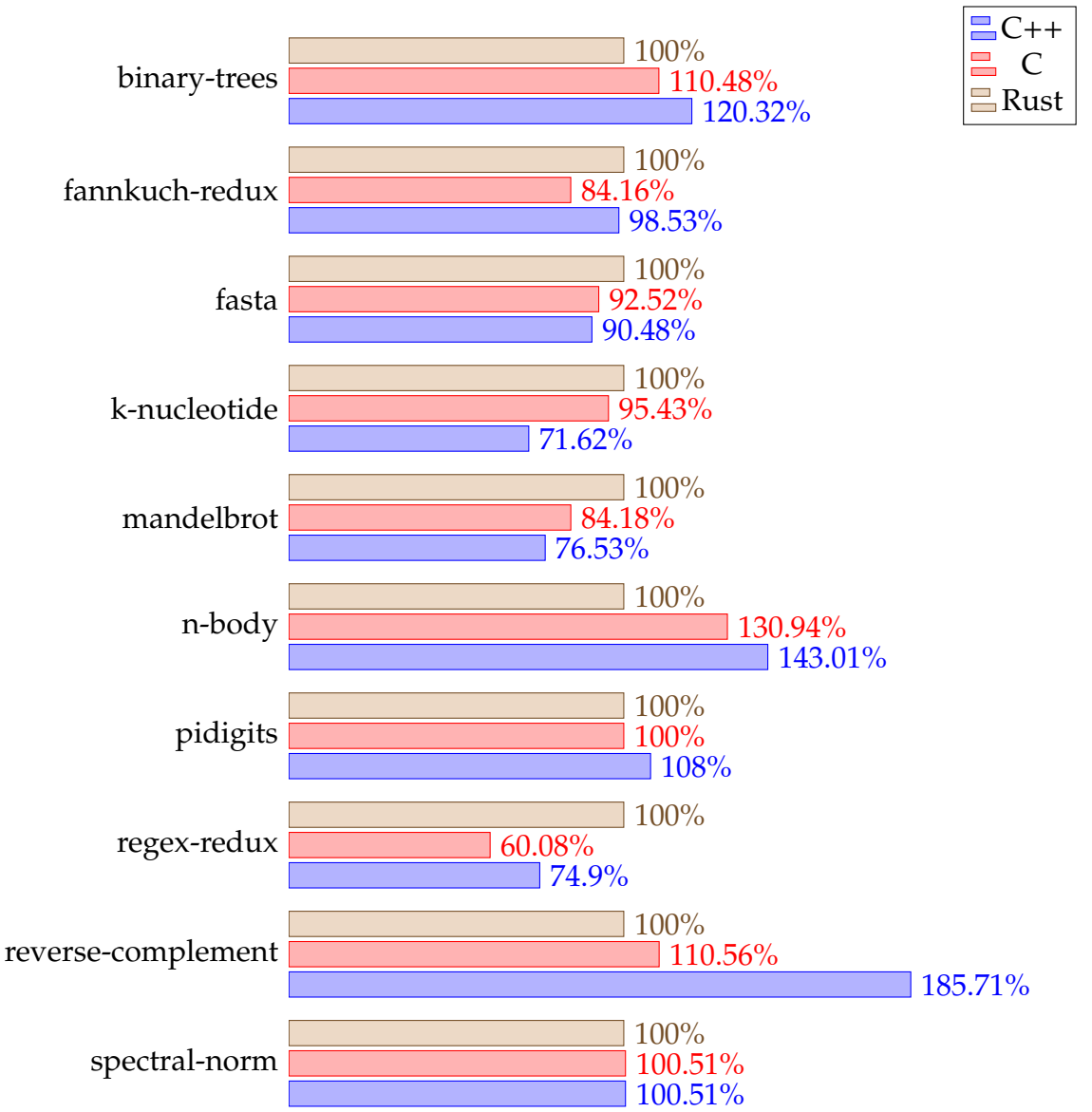

Abb. 4.1: Ausführungszeit: Rust im Verhältnis zu C und C++

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, welche durchschnittlich 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.

4.2.3 Speicherverbrauch

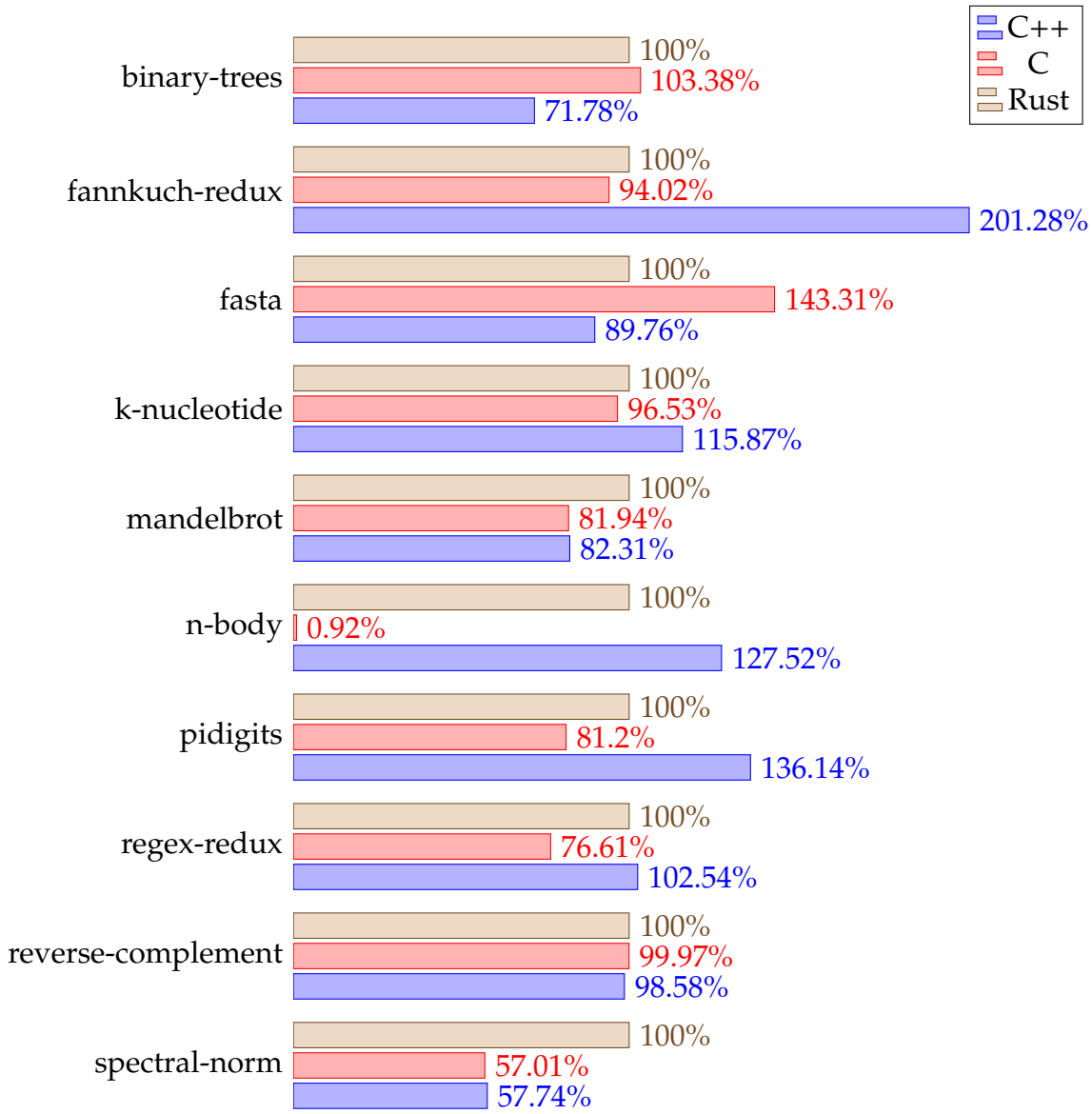

Abb. 4.2: Speicherverbrauch: Rust im Verhältnis zu C und C++

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. Rust and webassembly. https://rustwasm.github.io/docs/book/.

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

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

[Rusf] 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