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)

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, Framework in Rust 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, z.B.:

- 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. [Rusf]

2.2.1 Grundlegende Verwendung

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

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

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. Beim Kompilieren werden dadurch Warnungen oder 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. Es besagt, dass eine Variable eingeführt wurde, die nicht im Code verwendet wurde. Dies ist nicht falsch, es könnte jedoch ein Bug sein.

2.3 Cargo

Cargo 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 Author 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. Zum Zeitpunkt der Erstellung dieser Arbeit gibt 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 in 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 gepflegt wird. Mithilfe von Cargo können Tests gestartet werden, genauere Information dazu sind aus Kapitel 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 src/bin/*.rs
- Integrationstests im Ordner tests, Unit-Tests werden in die jeweilige Programmdatei geschrieben
- Beispiele im examples Ordner
- Benchmarks im benches Ordner

Wichtige Kommandozeilenbefehle für Cargo

Zum Kompilieren uns Ausführen:

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

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

# alternative as 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. Jedoch 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. 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 dies nicht bei neu erstellten Crates[3] zu.

Ein Projekt wird meist von mehreren Entwicklern programmiert oder Besitzer des Projekts ändert 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 bzw. entfernen sie aus dem Projekt:

```
# "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 Teams jedoch erlaubt. Ist ein „named owner“ in einem Team, so sind alle Entwickler eines 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. Grobe Struktur:

```
{
    "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“ 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, wie es üblich ist, 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 gibt es zum Zeitpunkt der Erstellung dieses Dokuments bereits über 40 benutzerdefinierte Befehle, die von der Rust-Community erstellt wurden. [Rusg]

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

In diesem Kapitel werden auf Unterschiede bei der Programmierung zwischen den Sprachen Rust und C/C++ eingegangen.

3.1 Grundlagen

Zu den Grundlagen einer jeden Programmiersprache gehört der Umgang mit Variablen und Datentypen, und Kommentarfunktionen. Kontrollstrukturen definieren 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, den Code so zu schreiben, dass die Sicherheit und Parallelität von Rust genutzt werden. [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
```

Das Beispiel zeigt, wie der Compiler dem Programmierer hilft, Fehler im Programm zu finden. Die Fehlermeldung weist darauf hin, dass die Fehlerursache darin liegt, dass auf eine unveränderliche Variable nicht ein zweites Mal zugewiesen werden darf.

Bei C oder C++ ist jede Variablendefinition standardmäßig veränderlich, das heißt bei gleicher Vorangehensweise würde folgendes C-Programm ohne Fehlermeldungen 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 Variablen in C/C++ konstant zu definieren.

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

Wird in C versucht, eine konstant definierte Variable zu verändern, indem ein Wert mit einem nicht konstanten Typ verwendet wird, ist das Verhalten nicht definiert. [ISO, p. 87]

Das Verhalten in Folgendem Beispiel ist somit nicht definiert:

```
#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. Das bedeutet, dass die Typen aller Variablen zur Kompilierzeit bekannt sein müssen. 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 Typanmerkung wie folgt hinzugefügt werden:

```
let guess: u32 = "42".parse().expect("Not a number!");
```

Ohne Angabe des Typs gibt der Rust-Compiler eine Fehlermeldung aus.

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“ statt mit „u“. Due Zahl gibt an, wie viel Speicherplatz sie beansprucht. Folgende Tabelle 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++

Mit „short“ und „long“ sollen 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.

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-Architektur und 32 Bits bei 32-Bit-Architektur.

In Rust wird standardmäßig der Integertyp i32 verwendet. Dieser Typ ist im Allgemein 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, das heißt sie können 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, eine Sammlung mehrerer Werte zu haben, besteht in einem Array. 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, wie Tupel.

In Rust werden die Werte, die in ein Array gehen, als durch Kommata getrennte Liste in eckiggen 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 vergrößert oder verkleinert werden darf.

Das Schreiben des Array-Typs erfolgt mit eckigen Klammern mit dem Typ der Element im Array gefolgt von einem Semikolon und die Anzahl der Elemente im 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 Array angegeben:

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

Das Array mit dem Namen „a“ enthält 5 Elemente, die zunächst alle auf den Wert 3 gesetzt sind. 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 der Indexierung wiefolgt zugegriffen werden:

```
let a = [1, 2, 3, 4, 5];
let first = a[0];
let second = a[1];
```

Wird auf ein Element zugegriffen, das nicht innerhalb 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:

```
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 durch Komma getrennt.

Bei Funktionen mit Rückgabewerten muss am Ende des Funktionskörpers der Rückgabewert als Expression ohne Semikolon stehen. Wenn eine Funktion früh beendet werden soll, kann ein „return“ mit 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
}
```

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 kann, sind if-Ausdrücke und -Schleifen.

if-Anweisung

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 in C oder C++, mit dem Unterschied, dass keine Klammern bei der Expression benötigt werden.

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 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 ist die if-Anweisung eine Expression, somit kann es verwendet werden, um z.B. Werte von Variablen zu definieren:

```
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 heißt, dass bei letzteren if-Anweisung der Typ einheitlich sein muss. Im letzten Beispiel war „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 gestoppt werden mit dem Schlüsselwort „break“. Schleifen können in Rust auch als Expression verwendet werden, wenn an dem „break“ ein Wert 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++. Sie funktionieren auch in Rust entsprechend.

```
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 heißt, dass der Code 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 Sicherheit des Codes erhöht werden, da dadurch Zugriffe außerhalb eines Arrays verhindert werden.

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.

[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. Guide to rustc development. https://rust-lang.github.io/rustc-guide/.

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

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

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

[Rusg] 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