



Marco Amann · Joachim Baumann · Marcel Koch

Rust

Konzepte und Praxis
für die sichere Anwendungsentwicklung

dpunkt.verlag

Inhalt

Cover

Über den Autor

Titel

Impressum

Inhaltsübersicht

Inhaltsverzeichnis

Vorwort

Danksagung

1 Rust – Einführung

1.1 Warum Rust?

1.1.1 Rust und der Speicher

1.1.2 Rust und Objektorientierung

1.1.3 Rust und funktionale Programmierung

1.1.4 Rust und Parallelverarbeitung

1.2 Ein Beispielprogramm

1.3 Installation von Rust

1.3.1 Installation von rustup

1.4 IDE-Integration

1.4.1 Rust Language Server und Rust-Analyzer

1.4.2 Visual Studio Code

1.4.3 IntelliJ IDEA

1.4.4 Eclipse

1.4.5 Welche Entwicklungsumgebung ist die beste?

1.5 Unsere erste praktische Erfahrung

1.6 Das Build-System von Rust

- 1.6.1 Die Struktur von Rust-Programmen
- 1.6.2 Die Erzeugung eines Packages
- 1.6.3 Übersetzen und Ausführen eines Packages
- 1.6.4 Verwaltung von Abhängigkeiten
- 1.6.5 Workspaces
- 1.6.6 Weitere nützliche Befehle von Cargo

1.7 Entwicklung der Sprache und Kompatibilität

Teil I Die Sprache

2 Syntax von Rust-Programmen

- 2.1 Programmstruktur
- 2.2 Anweisungsblöcke
- 2.3 Rangfolge von Operatoren
- 2.4 Gängige Kontrollflussstrukturen

2.4.1 Das If-Konstrukt

2.4.2 Das Loop-Konstrukt

2.4.3 Die While-Schleife

2.4.4 Die For-Schleife

3 Variablen

3.1 Veränderbare und nicht veränderbare Variablen

3.2 Weitere Arten der Variablendefinition

3.2.1 Globale Variablen

3.2.2 Konstanten

4 Datentypen

4.1 Skalare Datentypen

4.1.1 Ganzzahlen

4.1.2 Fließkommazahlen

4.1.3 Logische Werte

4.1.4 Zeichen

- 4.1.5 Typkonvertierung
 - 4.2 Tupel und Felder
 - 4.2.1 Tupel
 - 4.2.2 Felder
 - 4.3 Strukturierte Datentypen
 - 4.3.1 Unterstützung bei der Initialisierung
 - 4.4 Tupelstrukturen
 - 4.5 Aufzählungstypen
 - 4.5.1 In Aufzählungen eingebettete Datentypen
- 5 Musterabgleich
 - 5.1 Das Match-Konstrukt
 - 5.1.1 Einfache Verwendung
 - 5.1.2 Rückgabewerte
 - 5.1.3 Zusätzliche Bedingungen für das Muster
 - 5.1.4 Zuweisungen im Muster
 - 5.2 Andere Datentypen und das Match-Konstrukt
 - 5.3 Weitere Musterabgleiche
 - 5.3.1 Das »If Let«-Konstrukt
 - 5.3.2 Das »While Let«-Konstrukt
 - 5.3.3 Das Makro matches!
- 6 Funktionen
 - 6.1 Referenzen auf Funktionen
- 7 Einführung in das Speichermodell
 - 7.1 Stack und Heap
 - 7.2 Rust und der Speicher
 - 7.3 Das Modell für skalare Datentypen
 - 7.3.1 Wechsel von Gültigkeitsbereichen
 - 7.3.2 Aufruf von Funktionen

7.4 Das allgemeinere Modell

7.4.1 Wechsel von Gültigkeitsbereichen

7.4.2 Aufruf von Funktionen

7.5 Referenzen in Rust

7.5.1 Lesereferenzen auf nicht veränderbare Variablen

7.5.2 Lesereferenzen auf veränderbare Variablen

7.5.3 Veränderbare Referenzen

7.6 Verwendung von Variablen und Referenzen

7.7 Vor- und Nachteile des Modells

7.7.1 Nachteile

7.7.2 Vorteile

8 Generische Datentypen

8.1 Typparameter in Datenstrukturen

8.2 Typparameter in Funktionen

8.3 Typparameter in Aufzählungstypen

9 Objektorientierte Konzepte

9.1 Methoden

9.1.1 Die Verwendung von Typparametern

9.2 Module und Sichtbarkeiten

9.2.1 Importieren von Elementen aus anderen Namensräumen

9.2.2 Hierarchische Module

9.2.3 Erweiterte Sichtbarkeiten

9.2.4 Aufteilung in mehrere Dateien

9.3 Traits

9.3.1 Erzeugung und Verwendung

9.3.2 Abhängigkeit von anderen Traits

9.3.3 Verwendung in Funktionen

9.3.4 Verwendung mit generischen Datentypen

9.3.5 Einschränkung von Typparametern mit Traits

9.3.6 Polymorphe Rückgabetypen

9.3.7 Assoziierte Datentypen

9.3.8 Die Größe von Instanzen

9.3.9 Dynamische Trait-Objekte

9.3.10 Traits, die Rust bereitstellt

9.3.11 Der Trait Drop

9.3.12 Das Attributsmakro Derive

10 Problembehandlung in Rust

10.1 Der Datentyp Option

10.2 Der Datentyp Result

10.3 Interoperabilität von Option und Result

10.4 Der ?-Operator

10.5 Nicht behebbare Fehler

10.6 Bewertung

11 Standarddatentypen von Rust

11.1 Kollektionen

11.1.1 Sequenzdatentypen

11.1.2 Map-Datentypen

11.1.3 Mengen

11.1.4 Verschiedene Datentypen in Kollektionen

11.1.5 Der Datentyp Slice

11.1.6 Zeichenketten

11.2 Der Datentyp Range

11.3 Closures

11.3.1 Verwendung als anonyme Funktion

11.3.2 Der umgebende Gültigkeitsbereich

11.4 Iteratoren

- 11.4.1 Erzeugung von Iteratoren
- 11.4.2 Erste Verwendung von Iteratoren
- 11.4.3 Weitere Verarbeitungsmöglichkeiten
- 11.4.4 Erzeugung von Iteratoren aus Iteratoren
- 11.4.5 Erzeugung neuer Kollektionen

12 Makros

- 12.1 Bekannte Makros
- 12.2 Beispiele für weitere Makros

12.2.1 Assertionen

12.2.2 Makros für Zeichenketten

- 12.3 Arten von Makros
- 12.4 Ein eigenes deklaratives Makro

13 Strukturierung von Projekten

- 13.1 Konfiguration des Packages

14 Zusammenfassung

Teil 2 Fortgeschrittene Techniken

15 Ownership im Detail

- 15.1 Näheres zum bekannten Ownership-Modell

15.1.1 Move, Copy, Clone, Borrow

15.1.2 Lifetimes

15.1.3 Die Sicherheit von Rust

- 15.2 Smart Pointer

15.2.1 Box

15.2.2 Rc

15.2.3 RefCell und Cell

15.2.4 Zusammenfassung

- 15.3 Vergleich mit anderen Sprachen ohne Ownership

16 Nebenläufige und parallele Programmierung

- 16.1 Grundlagen
- 16.2 Channels
- 16.3 Shared State
 - 16.3.1 Arc
 - 16.3.2 Send und Sync
 - 16.3.3 Mutex
 - 16.3.4 RwLock
- 16.4 Einfache Parallelisierung mit Rayon
- 16.5 Sicherheit trotz Parallelität
- 16.6 Async/Await
- 16.7 Zusammenfassung
- 17 Testen
 - 17.1 Arten von Tests
 - 17.1.1 Unit-Tests
 - 17.1.2 Integration-Tests
 - 17.1.3 UI-Tests
 - 17.1.4 Testpyramide, Nachwort
 - 17.2 Rust, Cargo und Tests
 - 17.2.1 Platzierung von Testcode
 - 17.3 Ausführung
 - 17.3.1 Erwartungen der Testergebnisse (Assertions)
 - 17.3.2 Benennung der Testfunktionen
 - 17.4 Mocking
 - 17.4.1 Erste Schritte ohne Framework
 - 17.4.2 Einsatz eines Frameworks: Mockall
 - 17.4.3 Abschließendes zu Mockall
 - 17.5 Snapshot-Tests mit insta
 - 17.6 Der Rust-Compiler sieht viel, aber nicht alles

17.6.1 Überläufe (Overflows)

17.6.2 OutOfBoundsCheck

17.6.3 Stockungen (Deadlocks)

17.7 Fazit

18 Webprogrammierung

18.1 Einführung

18.1.1 Warum Rust für Webprogrammierung?

18.1.2 Warum nicht Rust für Webprogrammierung?

18.1.3 Themen in diesem Kapitel

18.1.4 Eine kleine Warnung vorab

18.2 Grundlagen von Rocket

18.2.1 Handler

18.2.2 Return Types

18.2.3 Ein Blick hinter die Kulissen

18.2.4 Shared State

18.3 Das Kontaktformular

18.3.1 Routen

18.3.2 Formulare

18.3.3 Datenbankverbindung

18.3.4 Was macht Rust bis hierher so besonders?

18.3.5 Middleware

18.3.6 Guards

18.3.7 Fairings oder Guards?

18.3.8 Serverside-Templates

18.3.9 Testen mit Rocket

18.4 Betrieb

18.4.1 Logging

18.4.2 Konfiguration

18.4.3 Deployment

18.5 Fazit

19 Microservices

19.1 Eignet sich Rust für Microservices?

19.2 Aufteilung der Webanwendung in Microservices

19.3 Vorbereitungen

19.3.1 Build mit Docker

19.3.2 Cross Compilation

19.4 Die Microservices

19.4.1 Anfragen annehmen: der »Web«-Service

19.4.2 Gemeinsame Funktionalität

19.4.3 Speichern der Anfragen in der Datenbank

19.4.4 Mail verschicken

19.5 Betrieb

19.5.1 Metriken und Monitoring

19.5.2 Tracing

19.5.3 Skalierung

19.6 Zusammenfassung

20 Systemnahe Programmierung

20.1 Unsafe Rust

20.1.1 Pointer-Grundlagen

20.1.2 Unsafe in std: RefCell als Beispiel

20.2 Systemaufruf

20.2.1 Systemaufruf in Handarbeit

20.2.2 Systemaufruf mit dem Crate libc

20.3 Integration von externen Bibliotheken in Rust

20.3.1 Fallstricke

20.4 Performanceuntersuchung

- 20.4.1 Erste Schritte
- 20.4.2 Benchmarks
- 20.4.3 Untersuchungen
- 20.4.4 Optimierung
 - 20.5 Zusammenfassung
- 21 Spracherweiterungen (Language Bindings)
 - 21.1 Java
 - 21.1.1 Grundsätzliches – Java ruft Rust auf
 - 21.1.2 j4rs
 - 21.1.3 Zusammenfassung
 - 21.2 Node.js
 - 21.3 Fazit
 - 22 WebAssembly
 - 22.1 Aktueller Stand von WebAssembly
 - 22.1.1 Im Browser
 - 22.1.2 Außerhalb des Browsers – ein Anfang
 - 22.2 Rust & WASM
 - 22.2.1 Warum Rust für WASM?
 - 22.2.2 Im Browser: wasm-bindgen & wasm-pack
 - 22.2.3 Auf dem Server
 - 22.3 Fazit
 - 23 Zusammenfassung und Ausblick
 - 23.1 Zusammenfassung
 - 23.2 Ausblick

Index

4 Datentypen

Dieses Kapitel beschäftigt sich mit den verschiedenen, von Rust zur Verfügung gestellten Datentypen. Wir lernen skalare Datentypen, Tupel, Felder aber auch strukturierte Datentypen und Aufzählungstypen kennen.

4.1 Skalare Datentypen

Skalare Datentypen stellen elementare Typen einer Programmiersprache dar, die üblicherweise sehr effizient auf die unterliegende Prozessorarchitektur abgebildet werden können (in Java heißen diese *primitive* Datentypen). Rust bietet uns vier verschiedene skalare Datentypen: Ganzzahlen, Fließkommazahlen, logische Werte und Zeichen. Alle vier betrachten wir im Folgenden.

4.1.1 Ganzzahlen

In den meisten Programmiersprachen gibt es unterschiedliche Typen für Ganzzahlen unterschiedlicher Größe, vorzeichenbehaftet und vorzeichenlos. Dies ist auch in Rust der Fall – anders als in vielen anderen Sprachen enthalten die Ganzzahltypen die Anzahl der Bits, die zur Speicherung verwendet werden, direkt im Namen.

Die folgende Tabelle gibt eine Übersicht der verschiedenen Ganzzahltypen; den Typ `i32` haben wir ja bereits informell kennengelernt.

Ganzzahltypen

Tab. 4-1 Übersicht über die zur Verfügung stehenden Ganzzahltypen

Größe in Bit	vorzeichenbehaftet	vorzeichenlos
8	i8	u8
16	i16	u16
32	i32	u32
64	i64	u64

128	i128	u128
architekturspezifisch	isize	usize

Die Namen der verschiedenen Ganzzahltypen sind sehr logisch aufgebaut und damit einfach erschließbar. Die einzige Besonderheit ist die Angabe der architekturspezifischen Größen `isize` und `usize`. Dies sind die Größen, die von der Prozessorarchitektur, für die das ausführbare Programm erzeugt wird, am besten verarbeitet werden können.

Die maximalen und minimalen Werte, die ein Typ annehmen kann, sind in der Standardbibliothek als Konstanten enthalten. Wir können auf diese zugreifen über `<Typ>::MIN` und `<TYP>::MAX`. Für den Typ `i32` erhalten wir damit `i32::MIN` und `i32::MAX`.

Tipps und Tricks

Ohne explizite Angabe wählt der Rust-Compiler im Normalfall immer den Typ `i32`, weil für diesen garantiert ist, dass auf allen gängigen Architekturen die Verarbeitung mit am schnellsten geschieht. In allen normalen Fällen dürfte dies tatsächlich die beste Wahl sein.

Wenn es hingegen darum geht, die aktuell genutzte Prozessorarchitektur am besten auszureizen, kann es sinnvoll sein, anstelle dessen den Typ `isize` oder `usize` zu verwenden.

Rust bietet uns auch für skalare Basistypen Funktionen an, die in der Standardbibliothek Funktionen für Ganzzahltypen enthalten sind und damit immer zur Verfügung stehen. Hier finden wir beispielsweise viele Funktionen, die uns bei Bitoperationen helfen, zum Beispiel die Bestimmung der Anzahl führender Nullen oder Einsen oder ein Tausch der Bytes innerhalb eines Ganzzahlwertes, aber auch Operationen für den Umgang mit Überlauf bei Berechnungen.

Wann immer wir uns mit Größen von Ganzzahlen beschäftigen, müssen wir uns auch mit dem Überlauf des Typs beschäftigen (*Integer Overflow*). Wir nennen hierbei sowohl das Überschreiten des maximal speicherbaren Wertes als auch das Unterschreiten des minimal speicherbaren Wertes einen Überlauf. Überlauf von Ganzzahlen

In vielen Programmiersprachen wird im Falle einer Kalkulation, die einen Überlauf erzeugt, einfach vom maximal möglichen auf den minimal möglichen Wert und andersherum gewechselt.

Rust wählt hier einen etwas anderen Ansatz und geht anders als viele andere Sprachen davon aus, dass ein Überlauf einen Fehler darstellt. Falls ein solcher

auftritt, wird allerdings verschieden reagiert, je nachdem, ob wir uns im Debug- oder Release-Modus befinden (siehe Abschnitt 1.6.3)

Ein Überlauf kann aber auch exakt der von uns gewünschte Effekt in einer Berechnung sein. Für diesen Fall stellt Rust uns verschiedene Funktionen zur Verfügung, die dem Compiler signalisieren, dass ein Überlauf potenziell von uns gewünscht ist und wie er behandelt werden soll.

Das folgende Beispiel illustriert die Verwendung beispielhaft:

Listing 4-1 *Behandlung des Überlaufs bei Ganzzahlen*

```
fn main() {  
  
    let mut wert : i16 = 32767;  
  
    // wert += 1; // Fehler beim Übersetzen  
  
    let überlauf = wert.wrapping_add(1);  
  
    let gesättigt = wert.saturating_add(1);  
  
    println!("{}", überlauf);  
  
    println!("{}", gesättigt);  
  
}
```

Wir definieren zuerst eine vorzeichenbehaftete Ganzzahlvariable mit 16 Bit und weisen ihr den höchstmöglichen Wert 32767 zu. Die folgende auskommentierte Zeile würde zu einer Fehlermeldung des Compilers führen, da bereits bei der Übersetzung erkennbar ist, dass ein Überlauf stattfindet.

Stattdessen verwenden wir in der folgenden Zeile die Funktion `wrapping_add()`, um diese Berechnung mit einem Überlauf durchzuführen. Rust bietet uns für alle verfügbaren Rechenarten `wrapping_*()`-Funktionen an, die wir nutzen sollten, wann immer es die Möglichkeit für einen Überlauf gibt.

In der nächsten Zeile verwenden wir eine Funktion aus der `saturating_*()`-Gruppe, die bei einem Überlauf einfach beim maximalen beziehungsweise

minimalen Wert verbleibt.

Danach geben wir die Inhalte der beiden Variablen aus, was zum folgenden Ergebnis führt:

-32768

32767

Hintergrund

Es gibt noch weitere Möglichkeiten, Überläufe zu behandeln. Die Funktionen `checked_*()` liefern eine Instanz einer `Option`-Struktur zurück, die entweder den Wert oder bei Überlauf den speziellen Wert `None` enthält (sowohl `Option` als auch `None` lernen wir in Abschnitt 10.1 kennen und nutzen).

Die Funktionen `overflowing_*()` verhalten sich exakt wie die `wrapping_*()`-Funktionen, liefern aber zwei Werte zurück. Der erste ist das Ergebnis der eventuell überlaufenden Berechnung, der zweite ist ein boolescher Wert, der anzeigt, ob ein Überlauf stattgefunden hat. Die Funktion liefert hierfür ein Tupel mit den beiden Werten zurück. Tupel werden wir in Abschnitt 4.2.1 betrachten.

Bei der Notation von Ganzzahlen bietet uns Rust neben der üblichen Notationen von Dezimalzahlen die Möglichkeit, folgende Darstellungen zu verwenden:

Notation von Literalen

Tab. 4-2 Notationen für Literale

Notation	Bedeutung
0x	Präfix für Hexadezimalzahlen
0o	Präfix für Oktalzahlen
0b	Präfix für Binärzahlen
b'<x>'	Spezialnotation für ASCII-Buchstaben <x>, nur für u8
_	Unterstrich, Trennzeichen zwischen Ziffern
i16, i32, ...	Suffix für die Größe

Die möglichen Präfixe sind uns aus anderen Programmiersprachen bekannt. Nur die Notation für einzelne Buchstaben als Ganzzahl ist ungewöhnlich. Das

Trennzeichen `_` kann tatsächlich an beliebigen Stellen verwendet werden, um Zahlen lesbarer zu gestalten. Das folgende simple Beispiel demonstriert dies:

Listing 4-2 *Ganzzahliliterale in Rust*

```
fn main() {  
  
    let wert : i16 = 32_767_i16;  
  
    let r_als_ganzzahl = b'r';  
  
    let binärwert = 0b1000_1001u8;  
  
}
```

4.1.2 Fließkommazahlen

Rust bietet uns zwei verschiedene Arten von Fließkommazahlen an, `f32` und `f64`. Diese entsprechen `float` und `double` in anderen Programmiersprachen. Die Voreinstellung, wenn nicht explizit angegeben, ist hierbei `f64`.

Die maximalen und minimalen Werte, die ein Fließkommatyp annehmen kann, sind genau wie bei den Ganzzahlen in der Standardbibliothek als Konstanten enthalten, und wir können auf diese zugreifen über `<Typ>::MIN` und `<TYP>::MAX`.

Rust erlaubt auch für Fließkommaliterale den Einsatz des Unterstrichs als visuelles Trennzeichen.

Notation von Literalen

Der Nachkommateil wird durch einen Dezimalpunkt abgetrennt. Den Typ eines Fließkommaliterals können wir analog zu den Ganzzahlen mit dem Suffix `f32` oder `f64` festlegen.

Listing 4-3 *Fließkommaliterale in Rust*

```
fn main() {  
  
    let pi_32 = 3.141_592_653_589f32;  
  
    let pi_64 = 3.141_592_653_589;
```

```
println!("{}", pi_32);

println!("{}", pi_64);

}
```

Wir definieren in unserem Beispiel zwei Variablen, die beide einen Näherungswert für Pi enthalten. Die Ausgabe für den ersten Wert zeigt, dass durch die Verwendung des kleineren Fließkommatyps wie erwartet Genauigkeit verloren geht.

```
3.1415927
```

```
3.141592653589
```

4.1.3 Logische Werte

Rust bietet wie viele andere Sprachen auch einen Typ für die logischen Werte `true` und `false`, den Typ `bool`. Dieser wird abgebildet auf ein Byte, und die Werte 1 (für `true`) und 0 (für `false`) werden intern verwendet.

Das folgende simple Beispiel verdeutlicht die Verwendung:

Listing 4-4 Definition von Variablen des Typs `bool`

```
fn main() {

    let wahr = true;

    let falsch = ! true;

}
```

4.1.4 Zeichen

Bei den Ganzzahlen haben wir eine Notation kennengelernt, um ASCII-Buchstaben in einem Byte speichern zu können. Rust bietet jedoch auch einen eigenen Typ `char`, der mit Unicode-Buchstaben umgehen kann und 4 Byte belegt. Jedes Zeichen wird als skalarer Unicode-Wert repräsentiert.

Hintergrund

Für Zeichenketten, die wir in Abschnitt 11.1.6 kennenlernen, wird eine UTF-8-Codierung verwendet, die im Mittel deutlich weniger Platz benötigt als 4 Bytes pro Zeichen.

Außerdem gibt es weitere Zeichenkettenimplementierungen in den Standardbibliotheken, um effizienter mit durch das Betriebssystem oder durch C-Programme verwendeten Zeichenketten umgehen zu können.

Im folgenden Beispiel definieren wir zwei Variablen `diskette` und `buchstabe`, denen wir jeweils ein Zeichen zuweisen.

Listing 4-5 Verwendung einzelner Zeichen des Typs `char`

```
fn main() {  
  
    let diskette = ' '; // Unicode-Zeichen U+1F4BE  
  
    let buchstabe = 'r';  
  
    println!("{}", diskette, buchstabe);  
  
}
```

Beide Werte geben wir in Folge aus. Wir sehen hierbei, dass wir mehr als einen Platzhalter in der Formatzeichenkette definieren können.

4.1.5 Typkonvertierung

Rust bietet keine implizite Typkonvertierung an, da diese potenziell mit Informationsverlust verbunden ist und damit nicht ohne explizite Entscheidung des Programmierers durchgeführt werden sollte.

Um eine Typkonvertierung durchzuführen, verwenden wir das Schlüsselwort `as`. Bei einer Typkonvertierung in einen größeren Typ, zum Beispiel von `i16` zu `i32`, gibt es keine Einschränkungen. Bei der Konvertierung in einen kleineren Typ, zum Beispiel von `i32` zu `i16`, wird bei einem Überlauf der maximale

beziehungsweise minimale Wert verwendet. Bei Genauigkeitsreduktion wird gegen 0 gerundet. Logische Werte werden zu 0 für `false` und zu 1 für `true`. Es gibt noch eine Liste weiterer Regeln in der Standarddokumentation. Allgemein funktioniert Typkonvertierung aber sehr ähnlich wie in anderen Sprachen. Das folgende Beispiel zeigt ein paar Beispiele:

Listing 4-6 Explizite Typkonvertierung

```
fn main() {  
  
    let ganzzahl = 3_i32;  
  
    // let fp:f64 = ganzzahl; // Fehler  
  
    let fp = ganzzahl as f64;// as f64;  
  
    let c = true as u32;  
  
}
```

Wir definieren zuerst eine Variable `ganzzahl` mit dem Wert 3 und dem Typ `i32`. Die nächste Zeile ist auskommentiert, da eine direkte Zuweisung vom Typ `i32` zum Typ `f64` ohne explizite Typkonvertierung nicht möglich ist. Wenn wir hingegen wie in der dritten Zeile die explizite Typkonvertierung mit `as f64` angeben, funktioniert diese wie gewünscht. Zum Schluss verwenden wir eine explizite Typkonvertierung, um den booleschen Wert `true` in eine Ganzzahlrepräsentation umzuwandeln.

4.2 Tupel und Felder

Rust unterstützt zwei einfache Formen zusammengesetzter Datentypen, Tupel und Felder (engl. *arrays*). Tupel sind hierbei Listen von Elementen potenziell verschiedener Datentypen, während Felder Listen von Elementen gleicher Datentypen enthalten.

In Rust gilt für beide Datentypen, dass die Größe nach der Erzeugung nicht mehr geändert werden kann. Dies erlaubt, beim Zugriff auf die Elemente zu prüfen, ob ein Index innerhalb der Grenzen des Datentyps ist, und im Zweifelsfall den Zugriff zu verhindern. Dies passiert zum einen zum Zeitpunkt der

Übersetzung. Wenn der Compiler dies allerdings aber zur Übersetzungszeit nicht sicherstellen kann, dann wird eine Laufzeitprüfung eingefügt, die dies sicherstellt. Ein Beispiel hierfür wäre, wenn der verwendete Index zur Übersetzungszeit nicht bekannt ist. Durch dieses Vorgehen können Speicherfehler durch fehlerhafte Zugriffe verhindert werden.

4.2.1 Tupel

Tupel werden erzeugt, indem mehrere Werte (oder Variablen), jeweils durch ein Komma getrennt, in runde Klammern gesetzt werden. Auf die einzelnen Werte können wir mit dem Index zugreifen (beginnend bei 0), den wir durch einen Punkt getrennt an den Namen des Tupels anhängen. Zusätzlich können wir ein Tupel entpacken und seine Werte anderen Variablen zuweisen. Hierzu definieren wir auf das Schlüsselwort `let` folgend eine Anzahl von Variablen innerhalb runder Klammern, denen wir das Tupel zuweisen. Anstelle einer Variablen können wir auch den Platzhalter `_` (Unterstrich) verwenden. Dieser signalisiert, dass wir den entsprechenden Wert nicht weiter verwenden. Die Anzahl der Einträge muss der Anzahl der Elemente des Tupels entsprechen.

Tipps und Tricks

Es gibt auch die Möglichkeit, eine beliebige verbleibende Zahl von Werten über zwei Punkte `..` zu ignorieren, sofern der Compiler die Werte eindeutig zuordnen kann. Eine Anweisung `let (a, .., b) = (3, 4, 5, 6)` würde damit `a` den Wert 3 und `b` den Wert 6 zuweisen.

Betrachten wir, wie das praktisch aussieht: Im folgenden Beispiel definieren wir zuerst eine Variable `buchstabe`, die das Zeichen `r` enthält. In der nächsten Zeile wird `buchstabe` zusammen mit anderen Werten verwendet, um in der veränderbaren Variable `tupel` ein Tupel zu erzeugen.

Listing 4-7 Verwendung von Tupeln

```
fn main() {  
  
    let buchstabe = 'r';  
  
    let mut tupel = (buchstabe, 3, 3.1);  
  
    println!("{:?}", tupel);  
  
    tupel.1 = 4;  
}
```

```

let (a, _, mut b) = tuple;

println!("{:#?}, {}", tuple, a);

}

```

In der dritten Zeile geben wir dieses Tupel aus. Hier sehen wir eine neue Art Platzhalterdefinition innerhalb der Formatzeichenkette. Der Doppelpunkt leitet weitergehende Formatierungsanweisungen ein, das Fragezeichen signalisiert, dass der Wert in Debug-Form ausgegeben werden soll.

Nun weisen wir dem zweiten Element des Tupels (mit dem Index 1) einen neuen Wert zu. Dies funktioniert, da wir das Tupel als veränderlich definiert haben.

Im nächsten Schritt entpacken wir das Tupel. Wir weisen das erste Element des Tupels der neuen Variablen `a` zu, ignorieren das zweite Element und weisen das dritte Element einer veränderbaren Variablen `b` zu.

Entpacken von Tupeln

Nun folgt die erneute Ausgabe des Tupels und der Variablen `a`. Hierbei signalisiert die Zeichenfolge `:#?`, dass wir eine Debug-Ausgabe wünschen, die zur besseren Lesbarkeit umformatiert wird (engl. *pretty-printing*).

```
( 'r', 3, 3.1)
```

```
(
    'r',
    4,
    3.1,
), r
```

Wir sehen in der ersten Zeile die Debug-Ausgabe des Tupels mit den Elementen in einer Zeile und runden Klammern, die das Tupel symbolisieren. In den folgenden

Zeilen sehen wir die lesbarer formatierte Version des Tupels gefolgt von dem Wert der Variablen `a`.

Es gibt ein spezielles Tupel ohne Elemente, das durch `()` notiert wird. Dies verwenden wir in Rust immer dann, wenn es keinen »sinnvollen« Wert als Ergebnis gibt. Tatsächlich ist uns dieser Wert schon als Rückgabewert einer `While`-Schleife begegnet. Da es in diesem Anweisungsblock keinen konzeptuell sinnvollen Rückgabewert gibt, wird dort das Einheitstupel als Rückgabewert benutzt.

Unit- oder Einheitstupel

4.2.2 Felder

Felder erzeugen wir, indem wir mehrere Werte (oder Variablen) durch Komma getrennt in eckige Klammern setzen. Dies ist ein klarer Unterschied zu Tupeln, bei denen wir runde Klammern verwenden. Alle Werte innerhalb des Feldes müssen den gleichen Typ haben. Auf einzelne Werte können wir zugreifen, indem wir den Index (beginnend bei 0) in eckigen Klammern an den Namen des Feldes anhängen.

Es gibt noch eine zweite Möglichkeit, um Felder zu erzeugen. Indem wir in den eckigen Klammern der Feldinitialisierung einen Wert `x` gefolgt von einem Semikolon und eine Wiederholungsanzahl `n` angeben, erzeugen wir ein Feld mit `n` Elementen, die alle den Wert `x` haben.

Initialisierung mit einem wiederholten Wert

Hintergrund

Felder haben grundsätzlich eine feste Größe, die zur Übersetzungszeit bekannt sein muss. Wir lernen später noch andere Datentypen kennen, die dynamische Größenänderungen erlauben, und Konzepte, die es uns ermöglichen, auf Teilfeldern zu operieren.

Auch Felder können wir entpacken. Wir verwenden das Schlüsselwort `let` gefolgt von Variablen innerhalb eckiger Klammern. Damit zeigen wir an, dass wir ein Feld entpacken wollen. Der Platzhalter `_` (Unterstrich) anstelle einer Variablen erlaubt uns, einen Feldeintrag zu ignorieren. Bei Feldern gilt genau wie bei Tupeln, dass die Anzahl der Einträge der Anzahl der Elemente des Feldes entsprechen muss.

Ein Beispiel illustriert die Verwendung:

Listing 4-8 Die Verwendung von Feldern

```
fn main() {  
  
    let wiederholung = [3.141; 10];  
  
}
```

```

let ganzzahl = 2;

let mut feld = [ganzzahl, 3, 5];

println!("{:?}", feld);

feld[1] = 4;

let [mut a, _, b] = feld;

println!("{:#?}", {}, feld, a);

}

```

Wir definieren zuerst eine Variable `wiederholung`, die ein Feld mit 10 Elementen enthält, von denen jedes den Wert 3.141 hat.

Nun definieren wir eine Variable `ganzzahl`, die wir in der nächsten Zeile verwenden, um einen Feldeintrag des veränderbaren Feldes `feld` zu definieren. Die folgende Ausgabe verwendet wieder die Formatierungsanweisung für die Debug-Ausgabe.

Im nächsten Schritt verändern wir einen Wert unseres Feldes mit einer Zuweisung. Darauffolgend entpacken wir das Feld und weisen den ersten Feldeintrag der veränderbaren Variablen `a` zu, ignorieren den zweiten und weisen den dritten der Variable `b` zu.

Zum Schluss geben wir das Feld und die Variable `a` aus, wobei wir das Feld zur besseren Lesbarkeit umformatieren lassen.

```
[2, 3, 5]
```

```
[
```

```
  2,
```

```
  4,
```

5,

], 2

In der Ausgabe sehen wir diesmal eckige Klammern, die anzeigen, dass wir ein Feld ausgeben, gefolgt von der zur besseren Lesbarkeit umformatierten Version des Felds.

Mehrdimensionale Felder legen wir durch verschachtelte Folgen eckiger Klammern an. Dabei verlangt Rust, dass die Dimensionen aller Teilfelder konsistent sein müssen. Dies sorgt dafür, dass wir keine »Löcher« in unserem mehrdimensionalen Feld haben.

Mehrdimensionale Felder

Listing 4-9 Die Verwendung von mehrdimensionalen Feldern

```
fn main() {  
  
    let feld2 = [[1,2,3], [11,12,13], [21,22,23]];  
  
    println!("{}", feld2[1][2]);  
  
    println!("{:?}", feld2[1]);  
  
}
```

Im Beispiel definieren wir ein zweidimensionales Feld mit der Ausdehnung 3×3. Wenn wir bei dieser Definition zum Beispiel den Wert 3 oder 23 weglassen, dann haben wir eine Inkonsistenz, die der Rust-Compiler mit einer Fehlermeldung quittiert. Damit können wir zur Laufzeit sicher sein, dass kein Adressierungsfehler auftritt (der im schlimmsten Fall mit einem Speicherfehler enden könnte).

Im nächsten Schritt geben wir einen Wert dieses Feldes aus. Wir sehen, dass der Zugriff auf die Elemente des zweidimensionalen Feldes wie erwartet durch aufeinanderfolgende Indizes funktioniert. Zum Schluss greifen wir auf ein (eindimensionales) Unterfeld des Gesamtfeldes zu und geben es über die Debug-Ausgabe aus.

4.3 Strukturierte Datentypen

Tupel erlauben uns, Elemente unterschiedlicher Datentypen zusammenzufassen. Tupel haben aber zwei Nachteile. Zum einen werden die einzelnen Elemente nur durch ihren Index identifiziert und nicht durch einen Namen. Zum anderen ist ein Tupel (i32, i32), das Koordinaten repräsentiert, nicht von einem Tupel (i32, i32) unterscheidbar, das Länge und Breite geometrischer Formen darstellt. Hierfür brauchen wir eigene Datentypen.

Rust bietet uns die Möglichkeit, eigene strukturierte Datentypen zu definieren. Dies geschieht, indem wir eingeleitet durch das Schlüsselwort `struct` und einen Namen eine Auflistung von Elementen mit Namen und assoziiertem Typ, getrennt durch Kommata, innerhalb von geschweiften Klammern definieren. Da Strukturen ganz normale Datentypen darstellen, können sie auch in anderen Strukturen als Elemente auftauchen. Dies erlaubt uns, eine beliebig hohe Komplexität unserer Daten abzubilden. Die Definition von strukturierten Datentypen kann in einem beliebigen Geltungsbereich passieren, innerhalb von Anweisungsblöcken, Funktionen, aber auch global.

Die Erzeugung von neuen Instanzen geschieht durch simple Benennung des Typs gefolgt von geschweiften Klammern und den Initialisierungswerten für alle Elemente der Struktur getrennt durch Kommata. Ein finales Komma vor der schließenden Klammer kann zur Lesbarkeit beitragen. Der Hauptunterschied zu anderen Sprachen ist hierbei, dass wir nicht explizit Speicher anlegen wie in Java durch `new()` oder in C durch `malloc()`.

Zugriff auf die einzelnen Elemente der Instanz erfolgt über den Instanznamen gefolgt von einem Punkt und den Elementnamen.

Listing 4-10 *Unser erster strukturierter Datentyp*

```
struct Point {  
  
    x: i32,  
  
    y: i32,  
  
}  
  
fn main() {  
  
    let origin = Point {
```

```

        x: 0,

        y: 0,

};

println!("Point({}, {})", origin.x, origin.y);

}

```

In unserem Beispiel definieren wir eine Struktur namens `Point`, die zwei Elemente namens `x` und `y` des Typs `i32` enthält (inklusive finale Komma). In der `main()`-Funktion erzeugen wir eine Variable `origin` dieses Typs und initialisieren `x` und `y` jeweils mit `0`.

Abschließend geben wir diese Werte aus, indem wir mit der Punktnotation auf die Elemente von `origin` zugreifen.

4.3.1 Unterstützung bei der Initialisierung

Um die Initialisierung von strukturierten Datentypen zu vereinfachen, gibt es eine verkürzte Notation. Diese können wir verwenden, wenn der Elementname innerhalb einer Instanz gleich dem Variablennamen im umgebenden Geltungsbereich ist. In diesem Fall kann auf die Doppelnennung und den Doppelpunkt verzichtet werden.

Zusätzlich können wir bereits existierende Instanzen unseres Datentyps als Vorlage verwenden. Aus dieser Vorlage können alle Elementwerte kopiert werden, die wir nicht explizit angeben. Hierzu geben wir nach allen Elementen, die wir explizit initialisieren die als Vorlage zu verwendende Instanz eingeleitet durch zwei Punkte `..` an.

Das folgende Beispiel illustriert neben diesen Arten der Initialisierung auch verschachtelte Strukturen:

Listing 4-11 Verwendung verschiedener Initialisierungsarten

```

#[derive(Debug)]

struct Point {x: i32, y: i32}

```

```
#[derive(Debug)]

struct Rect {p0: Point, p1: Point}

fn main() {

    let (x, y) = (0, 0);

    let origin = Point {x, y};

    let p1 = Point {x: 10, ..origin};

    let rectangle = Rect {p0: origin, p1};

    println!("{:#?}", rectangle);

}
```

Die erste Anweisung `#[derive(Debug)]` (genauer ein Compiler-Attribut) fügt eine einfache Implementierung für die formatierte Ausgabe zur Struktur hinzu. Damit müssen wir uns in unseren Ausgabeanweisungen keine weitergehenden Gedanken über die Formatierung unserer Strukturen machen. Wie dies unter der Haube funktioniert, werden wir in Abschnitt 9.3.12 kennenlernen.

Hintergrund

Tatsächlich funktioniert dies, indem eine Default-Implementierung des Traits `Debug` zu der Strukturdefinition als Attribut hinzugefügt wird. Diese Default-Implementierung wird dann zum Zeitpunkt der Ausgabe zur Formatierung verwendet. Default-Implementierungen gibt es für viele Funktionalitäten, und sie sind in vielen Fällen vollständig ausreichend. Weitere Beispiele sind Vergleichs- und Hash- oder Kopierfunktionalität. In Abschnitt 9.3 zu Traits finden Sie weitergehende Informationen.

Wir definieren zwei strukturierte Datentypen `Point` und `Rect` in einem globalen Kontext, beide mit Funktionalität zur Debug-Ausgabe. Der Typ `Rect` besteht hierbei aus zwei Instanzen des Typs `Point`.

In der `main()`-Funktion beginnen wir mit der Erzeugung zweier Variablen `x` und `y`, die wir jeweils auf den Wert 0 setzen. Diese verwenden wir im nächsten Schritt zur Initialisierung einer Instanz von `Point` namens `origin`. Dann

erzeugen wir eine `Point`-Instanz namens `p1`, bei der wir den `x`-Wert explizit setzen und alle anderen Elemente aus der `origin`-Instanz kopieren (in diesem Fall nur den Wert für das Element `y`).

In der vorletzten Zeile erzeugen wir eine neue `Rect`-Instanz. Das Element `p0` wird mit der Variable `origin` initialisiert. Für das Element `p1` verwenden wir die verkürzte Initialisierung, um dieses Element auf die Instanz `p1` im Geltungsbereich der `main()`-Funktion zu setzen.

Hintergrund

Brauchen wir die verkürzte Initialisierung und Instanzvorlagen? Nicht unbedingt – außerdem können sie für Neulinge beim ersten Lesen zu leichter Überraschung führen. Sie sind aber in vielen Fällen so elegant und geschickt, dass man sie nach kurzer Zeit nicht mehr missen mag.

Ähnlich wie Tupel können wir auch Strukturen entpacken. Da wir wegen der Verwendung von Namen unabhängig von Reihenfolgen sind, geben wir hierbei die Namen der Elemente explizit an. Alle Elemente müssen benannt sein, müssen aber nicht unbedingt Variablen zugewiesen werden.

Entpacken von Strukturen

Listing 4-12 Entpacken von Strukturen

```
struct Point {x: i32, y: i32}

fn main() {

    let p1 = Point {x: 10, y: 10};

    let Point{x: breite, y} = p1; // let breite = p1.x;

    println!("{}", breite);

}
```

Im Beispiel definieren wir einen Datentyp `Point`, den wir in der `main()`-Funktion in der Variable `p1` instanziierten. Im nächsten Schritt entpacken wir die Struktur und weisen der Variablen `breite` den Wert des Elements `x` zu. Die hierfür notwendige Syntax führt auf der linken Seite den Typ und in geschweiften Klammern alle Elemente auf. In unserem einfachen Fall entspricht dies der Anweisung:

```
let breite = p1.x;
```

In komplexeren Fällen empfehlen wir jedoch das Entpacken, da es eine deutlich klarere Syntax sein kann.

4.4 Tupelstrukturen

Es gibt Situationen, in denen wir die Definition eines Typs kombinieren möchten mit der einfachen Notation von Tupeln, ohne jedes Element explizit benennen zu müssen. Dies unterstützt Rust mit den Tupelstrukturen, die die Syntax von Strukturen und Tupeln kombinieren.

Während sich trefflich über den Sinn von Tupelstrukturen mit mehreren Elementen streiten lässt (sprich, es ist eine Geschmacksache), haben Tupelstrukturen mit einem Element einen direkten Anwendungsfall. Wir können durch das Verpacken von (vor allem skalaren) Datentypen in eine Tupelstruktur eine semantische Information ausdrücken, zum Beispiel in Form einer Maßeinheit.

Im folgenden Beispiel erzeugen wir zwei Tupelstrukturen namens `Kilogramm` und `Celsius`. Hierbei folgt dem Schlüsselwort `struct` der Name des Typs und am Ende in runden Klammern (wie beim Tupel) die Aufzählung der Elementtypen.

Listing 4-13 Verwendung von Tupelstrukturen

```
fn main() {  
  
    struct Kilogramm(f64);  
  
    struct Celsius(f64);  
  
    let value = 42.1;  
  
    let kg = Kilogramm(value);  
  
    let temp = Celsius (value);  
  
    println!("{}", {}, {}, {}, value, kg.0, temp.0);  
}
```

```
}
```

Im nächsten Schritt definieren wir eine Variable `value` vom (durch den Compiler inferierten) Typ `f64`, die wir zur Initialisierung zweier Variablen `kg` und `temp` vom Typ `Kilogramm` und `Celsius` verwenden.

Damit erhalten wir die drei Variablen `value`, `kg` und `temp` unterschiedlicher Typen, obwohl der jeweils ausgedrückte skalare Wert bei allen der gleiche ist.

In der finalen Ausgabe der Werte der drei Variablen sehen wir, dass auch der Zugriff auf die Elemente einer Tupelstruktur genau wie bei einem Tupel durch Anhängen des Index getrennt durch einen Punkt funktioniert. Auch das Entpacken von Tupelstrukturen funktioniert exakt wie bei Tupeln.

Analog zum Einheitstupel gibt es auch einen `Unit- oder Einheitstyp` Einheitstyp, gleichfalls notiert durch leere runde Klammern `()`, der keine Elemente hat. Dieser findet allerdings in Rust keine weitergehende Verwendung.

4.5 Aufzählungstypen

Rust unterstützt ähnlich wie Java typisierte Aufzählungstypen (in C und in C++ vor v11 hingegen sind Aufzählungstypen typlos).

Ein Aufzählungstyp enthält eine Menge von (unterschiedlichen) Namen, die die Wertemenge des Typs definieren. Da die Namen spezifisch für den Typ und damit im Namensraum des Typs sind, hat der gleiche Name in einem anderen Aufzählungstyp auch eine andere Bedeutung. Eine entsprechende Verwendung wird vom Rust-Compiler bemerkt und als Fehler gemeldet.

Wir definieren einen Aufzählungstyp mit dem Schlüsselwort `enum` gefolgt vom Namen des Typen, darauf in geschweiften Klammern die Namen, die die Wertemenge ausmachen, durch Kommata getrennt.

Listing 4-14 Die Definition und Verwendung von Aufzählungstypen

```
fn main() {  
  
    #[derive(Debug)]  
  
    enum Ampel { Rot, Gelb, Grün }
```

```

let ampel = Ampel::Grün;

println!("Die Ampel steht auf {:?}", ampel);

}

```

In unserem Beispiel definieren wir zuerst einen Aufzählungstyp `Ampel` mit drei Werten `Rot`, `Gelb`, `Grün` im Wertebereich. Wie schon vorher fügen wir eine einfache Implementierung der Debug-Ausgabe zu dem Typ mit dem Compiler-Attribut `#[derive(Debug)]` hinzu. Dann weisen wir einer neuen Variable den Wert `Ampel::Grün` zu. Die zwei aufeinanderfolgenden Doppelpunkte haben wir ganz zu Beginn des Buches bereits kennengelernt. Diese spezifizieren einen Namensraum, in unserem Fall den Namensraum des Typs `Ampel`, aus dem wir den Wert `Grün` wählen.

Zum Schluss geben wir den Wert unserer Variablen als Debug-Ausgabe aus (deshalb vorher das Compiler-Attribut `derive`):

```
Die Ampel steht auf Grün
```

4.5.1 In Aufzählungen eingebettete Datentypen

Während schon einfache Aufzählungstypen für uns bei der Programmierung sehr hilfreich sind, bietet uns Rust noch zusätzlich die Möglichkeit, in jeden der Namen einen beliebigen anderen Datentyp zu integrieren. Dies kann sowohl ein skalarer Datentyp als auch ein beliebig komplexer zusammengesetzter Datentyp sein. Diese Funktionalität bietet elegante Ausdrucksmöglichkeiten an, die wir tatsächlich schon kennengelernt haben.

Unser Beispiel im ersten Kapitel enthielt unter anderem das Öffnen einer Datei:

```

let file = Datei::open("hallo.txt")

    .expect("Konnte Datei nicht öffnen");

```

Worauf wir dort nicht näher eingegangen sind, ist der Rückgabetyt der `open()`-Methode. Es handelt sich dabei um einen Aufzählungstyp `Result`, der zwei

Namen `Ok` und `Err` im Wertebereich enthält. Die Methode `expect()` dieses Typs prüft jetzt nur, ob der Name `Ok` zurückgeliefert wurde oder der Name `Err`. Im ersten Fall wird der Inhalt des zugehörigen Datentyps zurückgeliefert, im zweiten Fall ein nicht behandelbarer Fehler inklusive der übergebenen Fehlermeldung. Wir werden die Implementierung dieses Datentyps in Abschnitt 10.2 genauer betrachten.

Hintergrund

Ein solcher Aufzählungstyp ist auch auf der Implementierungsseite sehr elegant. Nur der Platz für den größten im Aufzählungstyp verwendeten Datentyp und der Platz für die Information des aktuell verwendeten Datentyps werden benötigt. Damit haben wir eine typsichere Variante des in C verwendeten Typs `union`.

Tatsächlich gibt es auch in Rust einen Datentyp `union`, der auch noch auf den zusätzlichen Aufwand zur Speicherung der Datentypinformation verzichtet. Dieser ist aber inhärent unsicher und damit nur in einem speziellen Modus von Rust, dem `unsafe`-Modus, verwendbar.

Um einen Datentyp hinzuzufügen, geben wir diesen in runden Klammern nach dem Namen an. Werfen wir einen Blick auf das folgende Beispiel:

Listing 4-15 Die Verwendung von Datentypen in Aufzählungstypen

```
fn main() {  
  
    #[derive(Debug)]  
  
    enum Wert {  
  
        Ganzzahl(i32),  
  
        Fließkomma(f64),  
  
        Tupel((i32, f64))  
  
    }  
  
    let wert = Wert::Ganzzahl(3);  
  
    println!("Der Wert ist {:?}", wert);  
}
```

```
}
```

Wir definieren einen Aufzählungstyp `Wert` mit drei Namen `Ganzzahl`, `Fließkomma` und `Tupel`. Jeder dieser Namen bekommt einen anderen Datentyp zugeordnet. Wie auch in den vorherigen Beispielen verwenden wir wieder die von Rust bereitgestellte einfache Implementierung zur Debug-Ausgabe.

Wir definieren eine Variable von unserem gerade definierten Aufzählungstyp und geben diese in Folge aus:

```
Der Wert ist Ganzzahl(3)
```

Die offensichtliche Frage ist jetzt, wie wir auf den im Aufzählungstyp enthaltenen Wert zugreifen können. Hier nutzen wir das `Match`-Konstrukt, dem wir uns jetzt zuwenden.

Index

A

Abhängigkeiten 17
Akzeptanztest 269
Anweisungsblöcke 30
Arc 251
AssemblyScript 414
Assertionen 202, 276
Assoziierte Datentypen 129
Async 261
Attributsmakro Derive 143
Aufteilung in mehrere Dateien 116
Aufzählungstypen 56
 im Match-Konstrukt 64
Ausführen 15
Await 261

B

Backpressure 347
Box 228
Browser 415
Build-Konfigurationen 16
Build-System 14
ByteCode Alliance 424

C

call_from_java 401
Cargo 14
 build 15

- check 16
- fix 24
- fmt 24
- help 24
- init 15
- install 17
- new 15
- run 16
- search 18
- test 276
- tree 20
- uninstall 17
- update 21
- Cargo.lock 21
- Cargo.toml 15–16, 19–20, 22, 25
- cdylib 389
- Cell 241
- Channel 250
- Chrome 415
- Clone 216
- Closures 174
 - als anonyme Funktion 176
 - als Parameter und Rückgabewert 176
 - der umgebende Gültigkeitsbereich 177
 - die verschiedenen Traits 178
 - Erzwungener Eigentumsübergang mit move 181
 - Komplexere Interaktionen mit 183
- Container 339
- Cookie 321
- Copy 217
- Copy in Rust 79
- criterion 374
- Cross Compilation 342

D

- Das allgemeinere Speichermodell 81

Datenbanken 313
Datentypen 43
 BTreeMap 161
 Ganzzahltypen 43
 HashMap 160
 In Aufzählungen eingebettete 57
 LinkedList 159
 Option 147
 Range 172
 Result 149
 Slice 166
 str 169
 String 169
 Vec 156
 VecDeque 157
Deadlocks 295
Default-Werte für eigene Datentypen 141
Deklarative Makros 204
Deno 403
Deserialize 303
Diesel 313
Docker 337, 339, 418
Dynamische Trait-Objekte 134

E

Eclipse 11, 274
Editionen 25
Eingeschränkte Typparameter und impl 127
Entpacken von Strukturen 55
Entpacken von Tupeln 50
Erweiterte Sichtbarkeiten 116
Erzeugung von Iteratoren aus Iteratoren 193
Event-Loop 410
Exceptions 317–318

F

Fairings 318
Felder 48, 50
 Initialisierung mit einem wiederholten Wert 50
Firefox 415
Flamegraphs 373
Fließkommazahlen 46
Formulare 312
For-Schleife 35
FromRequest 303
From-Trait 140
Funktionen 69
 Tupel als Rückgabewerte 71
Future 416
Fuzzy Testing 295

G

Ganzzahlentypen
 Funktionen auf 44
Generische Datentypen 97
Globale Variablen 41
Grafana 353
Graphen 233
Größe von Instanzen 133
Guards 320
Gültigkeit einer Variablen 39

H

Handler 302
Header-Dateien 387
Heap 75
Hierarchische Module 114

I

If Let-Konstrukt 66
If-Konstrukt 32
Importieren von Elementen aus anderen Namensräumen 112
insta 291

- Installation 6
- instantiateStreaming 416
- Instruments 355
- Integration-Tests 271, 275
- IntelliJ 274
- IntelliJ IDEA 10
- Interface Types 414
- Interne Veränderlichkeit 241
- Interoperabilität von Option und Result 151
- Iteratoren 186
 - Erzeugung neuer Kollektionen 198
 - Transformation der Elemente 195
 - Transformation der Menge der Elemente 194

J

- Java 385
- JavaScript 415
- JNA 389
- JNI 386
- JoinHandle 253
- Json 304
- j4rs 389

K

- Kafka 338
- Kollektionen 155
- Kommandozeilenparameter 329
- Komponententest 269
- Konfiguration 329
- Konfigurationsdateien 331
- Konstanten 41
- Kontaktformular 311
- Kontrollflussstrukturen 31
- Konvertierung zwischen Datentypen 140
- Krustlet 418
- K9 277

L

Language Server 8
lazy_static 225, 259
Lesereferenzen 87
let 39
let mut 39
lettre 351
Lifetimes 220
Lin Clark 415
Logging 328
Logische Werte 47
Loop-Konstrukt 33

M

Mail 350
main()-Funktion 69
Makros 201

- eigenes deklaratives 205
- für Zeichenketten 203

Managed State 309
Map-Datentypen 160
Marker-Trait Sized 134
matches! 67
Match-Konstrukt 59

- Aufzählungstypen 64
- Der @-Operator 62
- komplexe Datentypen 63
- Rückgabewerte 60
- Strukturierte Datentypen im 64
- Zusätzliche Bedingungen 61
- Zuweisungen im Muster 62

Mehrdimensionale Felder 52
Mengen 163
Mengenoperationen 164
Methoden 105

- zur Erzeugung eines Iterators 187
- zur Identifikation einzelner Elemente im Iterator 189

Metriken 353

Middleware 318

mio 266

Mockall 282

Mocking 280

mock! 288

Module und Sichtbarkeiten 110

Move 219

Move in Rust 83

mpsc 250

Musterabgleich 59

Mutex 256

N

Namensräume 111

Neon 403

Newtype 315

Nicht behebbare Fehler 153

Node.js 403

Notation von Literalen 46

no_mangle 388

NPM 404

O

Oberflächentest 269

Objektorientierte Konzepte 105

OpenTelemetry 353

Operationen auf allen Werten des Iterators 190

Operatorüberladung 137

Ordering 255

ORM 313

OutOfBoundsCheck 294

Overflows 294

P

panic! 153
Parameter 303
Pointer 364
Poison 258
Polymorphe Rückgabetypen 128
Postgres 337
PostgreSQL 313
Problembehandlung 147
Programmstruktur 29
Prometheus 353
Promise 416
Prozedurale Makros 204

R

Rayon 259
Rc 235
Reborrowing 91
RefCell 239
Referenzen auf Funktionen 72
Referenzen in Rust 86
Referenzzyklen 238
Regressionstest 269
Rocket 301
Runtime 264
Rust Language Server 8
Rust-Analyzer 8
rustfmt 24
rustup 7
RwLock 258

S

Safari 415
Schlüsselwörter
 const 41
 let 39
 static 41, 225

- seccomp 418
- Send 256
- Sequenzdatentypen 155
- shared memory 248
- Shared State 251
- Skalare Datentypen 43
- Skalierung 361
- Smart Pointer 228
- Snapshot-Tests 291
- SPA 311
- Speichermodell 75
 - Aufruf von Funktionen 80
 - für skalare Datentypen 79
- SQLite 313
- Stack 76
- Standarddatentypen 155
- Statuscodes 306
- Stockungen 295
- Streams 416
- Strukturierte Datentypen 52
 - Instanzvorlagen 53
 - Unit- oder Einheitstyp 56
 - Unterstützung bei der Initialisierung 53
 - Verkürzte Initialisierung 53
- Sync 256
- systemd 332
- Systemtest 269

T

- Task 265
- Templates 323
- Testcode 273
- Testpyramide 269, 273
- thread 247
- Tokio 262, 400, 408
- Toolchain 343

- Tracing 356
- Trait Drop 142
- Traits 118
 - Abhängigkeit von 122
 - die mit derive generierbar sind 144
 - die Rust bereitstellt 137
 - Erzeugung und Verwendung 120
 - Verwendung in Funktionen 123
 - Verwendung mit generischen Datentypen 124
- Tupel 48–49
 - Unit- oder Einheitstupel 50
- Tupel im Match-Konstrukt 63
- Tupelstrukturen 55
- Typ- 97
- Typinferenz 39
- Typkonvertierung 48
- Typparameter 97
 - Einschränkung mit Traits 126
 - in Aufzählungstypen 101
 - in Datenstrukturen 99
 - in Funktionen 100

U

- Überlappende Slices 168
- Überlauf von Ganzzahlen 44
- Überläufe 294
- Übersetzen 15
- Udo Jürgens 430
- UI-Tests 269, 272
- Umgebungsvariablen 330
- Unit-Tests 270, 273
- unsafe 363

V

- Variablen 39
- Variablen vs. Referenzen 94

Veränderbare Referenzen 90
Veränderbare Referenzen mit Reborrow-Semantik 92
Verdeckung von Variablen 40
Vergleiche von Iteratoren 192
Verknüpfung von Iteratoren 197
Verschiedene Datentypen in Kollektionen 165
Verwendung von Iteratoren 187
Verwendung von Typparametern 108
Visual Studio Code 9
Vorteile von assoziierten Datentypen 132

W

WASI 417, 423
wasm-bindgen 414
wasm-pack 420
Wasmtime 417
Wasm3 417, 427
WebAssembly 413

- Interface Types 414
- System Interface 417

Webprogrammierung 298
Wechsel von Gültigkeitsbereichen 80
weiteren Artikel von Lin Clark 417
While Let-Konstrukt 67
While-Schleife 34
Workspaces 22

Z

Zeichen 47
Zeichenketten 168
Zipkin 357

Sonderzeichen

?-Operator 151