

Inhalt

Materialien zum Buch	18
1 Über dieses Buch	19
1.1 Was Sie in diesem Buch lernen werden	20
1.2 Was dieses Buch Ihnen zeigen möchte	21
1.3 Noch mehr Informationen und Guides	22
1.4 Danksagung	24
2 Die Installation, die IDE und »Hallo Rust«	25
2.1 Wie Sie Rust installieren	25
2.2 Eine Entwicklungsumgebung wählen	28
2.2.1 Visual Studio Code	28
2.2.2 JetBrains IDEs	30
2.3 Das erste Programm	30
2.3.1 Ein Paket mit »cargo« hinzufügen	31
2.3.2 Die Abhängigkeit einsetzen	32
2.4 Wie es weitergeht	33
3 Variablen und Datentypen	35
3.1 Prelude: Die Standardimporte	35
3.2 Variablen	36
3.2.1 Die Deklaration einer Variable	37
3.2.2 Eine Variable initialisieren	38
3.2.3 Eine Variable mit dem Schlüsselwort »mut« veränderlich machen ..	39
3.2.4 Die Typinferenz bei Variablen	43
3.2.5 Statische Variablen	44
3.2.6 Shadowing: Wenn sich Variablen mit gleichem Namen überdecken ..	50
3.2.7 Praxisbeispiel: Das Alter einlesen und verarbeiten	52

3.3 Konstanten	56
3.3.1 Was sind Konstanten?	56
3.3.2 Konstante Kontexte	59
3.4 Skalare Datentypen	60
3.4.1 bool	61
3.4.2 Integer	62
3.4.3 Fließkommazahlen	72
3.4.4 Character	75
3.4.5 Wie Sie mit »as« den Typ wechseln	78
3.5 Wie Rust mit »Option<T>« auf null verzichtet	81
3.6 Zusammenfassung	84

4 Speichernutzung und Referenzen

87

4.1 Wichtige Speicherbereiche	87
4.1.1 Der Stack	88
4.1.2 Der Heap	88
4.1.3 Statischer Speicher	89
4.2 Eigentumsverhältnisse im Speicher	89
4.2.1 Ein Wert, ein Eigentümer	90
4.2.2 Gültigkeitsbereiche und Blöcke beeinflussen Bindungen	91
4.2.3 Die Lebenszeit des Werts einer Variable	94
4.2.4 Bitweise Kopien mit »Copy« erzeugen	95
4.2.5 Move bindet den Wert an einen neuen Eigentümer	96
4.3 Referenzen und der leihweise Zugriff	98
4.3.1 Adressen, Zeiger und Referenzen: ein Überblick	98
4.3.2 Zugriff auf Werte über geteilte Referenzen	100
4.3.3 Veränderliche Referenzen: Der exklusive Zugriff	105
4.4 Mit Box Objekte im Heap ablegen	111
4.4.1 Klare Eigentumsverhältnisse auch für die Box	111
4.4.2 Auch Smart Pointer wie die Box können nicht alle Fehler verhindern	113
4.4.3 Nur die Box kann den Heap-Speicher freigeben	116
4.4.4 Ein Praxisbeispiel	118
4.5 Zusammenfassung	121

5 Strings

123

5.1	Der String-Slice	123
5.1.1	String-Slices können nur hinter Referenzen auftreten	124
5.1.2	Von wem Sie einen String-Slice erhalten	126
5.1.3	Byte-String-Slices	128
5.1.4	Raw String-Slices	132
5.1.5	String-Slices zusammenfügen	132
5.2	Der String	134
5.2.1	Wie Sie einen »String« anfordern	134
5.2.2	Einen String in andere Datentypen parsen	136
5.2.3	Nützliche Bearbeitungswerzeuge und wie man in einen eigenen Datentyp parst	138
5.3	Wie Sie Strings formatieren	147
5.3.1	Das neue Makro und Parameter	147
5.3.2	Die Ausgabe transformieren	150
5.3.3	Ausgabe-Traits	152
5.3.4	Makros, mit denen Sie Strings formatieren können	152
5.4	Zusammenfassung	154

6 Collections

157

6.1	Tupel	157
6.1.1	Tupel und Muster	158
6.1.2	Ein Tupel ist nicht gleich ein Tupel	162
6.1.3	Eigenschaften von Tupeln	163
6.1.4	Ein ganz besonderes Tupel: Der Unit-Typ	165
6.2	Arrays	166
6.2.1	Elemente aus einem Array auslesen	167
6.2.2	Vorsicht bei Datentypen, die nicht Copy sind	168
6.2.3	Die Größe mit konstanten Ausdrücken festlegen	171
6.2.4	Elemente in einem veränderlichen Array neu zuweisen	173
6.3	Elementbereiche	173
6.3.1	Elementbereiche haben einen Iterator	174
6.3.2	Einschränkungen des Iterators	175
6.3.3	Elementbereiche sind nicht Copy	177
6.3.4	Rust besitzt mehrere Elementbereich-Typen	177

6.3.5	Ein gemeinsamer Nenner	178
6.3.6	Elementbereiche als kleine Helferlein	181
6.4	Vektoren	182
6.4.1	Vektoren initialisieren	183
6.4.2	Elemente in einen Vektor einfügen	186
6.4.3	Einen Vektor mit einer anderen Collection zusammenlegen	187
6.4.4	Vektoren auslesen	190
6.4.5	Elemente aus Vektoren entfernen	194
6.4.6	Die Länge und Kapazität eines Vektors	199
6.4.7	Wenn Sie eine Queue benötigen oder auch effizient am Anfang einfügen müssen	204
6.4.8	Verkettete Listen	213
6.5	Slices	214
6.5.1	Einen Slice in Stücke aufteilen	214
6.5.2	Elemente zusammenführen oder wiederholen	218
6.5.3	Eigene Datentypen mit »concat« und »join« verarbeiten	220
6.5.4	Abfragen auf einem Slice ausführen	221
6.5.5	Elemente in einem Slice referenzieren	223
6.5.6	Veränderungen im Slice durchführen	224
6.5.7	Mit ASCII arbeiten	229
6.6	HashMap und BTreeMap	231
6.6.1	HashMap	231
6.6.2	BTreeMap	243
6.7	Hashes	245
6.8	Mengen verwalten	248
6.9	Die Entry API	251
6.9.1	VacantEntry und OccupiedEntry	252
6.9.2	Automatisch einen Eintrag einfügen, wenn er fehlt	254
6.9.3	Veränderungen on-the-fly vornehmen	256
6.10	Elemente verschiedener Datentypen in eine Collection einfügen	257
6.11	Zusammenfassung	260
7	Funktionen	263
7.1	Der Aufbau einer Funktion	264
7.1.1	Den Zugriff über die Sichtbarkeit steuern	264

7.1.2	Bezeichner und die Parameterliste	265
7.1.3	Der Rückgabewert	267
7.2	Funktionszeiger	268
7.3	Referenzen und Lebenszeiten in Funktionen	271
7.3.1	Warum und wann Sie Lebenszeiten angeben müssen	272
7.3.2	Wie Sie generische Lebenszeiten notieren	274
7.3.3	Varianz in Lebenszeiten	276
7.4	Konstante Funktionen	280
7.4.1	Die Ausführung ist zur Kompilierzeit und zur Laufzeit möglich	281
7.4.2	Seiteneffekte sind nicht erlaubt	282
7.4.3	Lebenszeiten in konstanten Funktionen deklarieren	283
7.4.4	Konditionale Ausdrücke und Schleifen	284
7.5	Anonyme Funktionen und Closures	285
7.5.1	Anonyme Funktionen	286
7.5.2	Was der Compiler aus einer anonymen Funktion macht	288
7.5.3	Anonyme Funktionen hinter Funktionszeigern	290
7.5.4	Parameter	291
7.5.5	Closures und die Umgebung einer anonymen Funktion	292
7.5.6	Wie die Speicherverwaltung Closures beeinflusst	300
7.6	Funktions-Traits	302
7.6.1	Die Grenzen des Funktionszeigers	303
7.6.2	Warum es drei verschiedene Funktions-Traits gibt	304
7.6.3	Die Call-Funktionen	310
7.7	Zusammenfassung	311

8	Anweisungen, Ausdrücke und Muster	313
8.1	Von der Anweisung zum Ausdruck und Muster	313
8.1.1	Die Item-Anweisung	314
8.1.2	Die Ausdruck-Anweisung	314
8.2	Die Zuweisung im Detail	316
8.2.1	Wertausdrücke: Die rechte Seite der let-Anweisung	316
8.2.2	Beispiel match: Ein komplexer Wertausdruck	317
8.2.3	Seiteneffekte, aber kein Rückgabewert	318
8.3	Speicherausdrücke	319
8.3.1	Der Speicherausdruck auf der linken Seite der Zuweisung	320

8.3.2	Der Speicherausdruck im Wertausdruck	321
8.3.3	Der Wertausdruck im Speicherausdruck	323
8.4	Operatoren	325
8.4.1	Arithmetische Operatoren und Vergleichsoperatoren	325
8.4.2	Mit dem Gruppen-Ausdruck in die Auswertungsreihenfolge eingreifen	326
8.4.3	Der Zuweisungsoperator	327
8.4.4	Operatoren auf die eigenen Datentypen anwenden	328
8.4.5	Übersicht zum Vorrang aller Ausdrücke und Operatoren in Rust	329
8.5	Konditionale Ausdrücke	330
8.5.1	Der if-Ausdruck	330
8.5.2	Der if let-Ausdruck	332
8.5.3	Der match-Ausdruck	333
8.6	Schleifen	342
8.6.1	Der Label-Block	342
8.6.2	Bis zur Unendlichkeit mit loop	343
8.6.3	while und while let	346
8.6.4	Die for in-Schleife	348
8.7	Muster	350
8.7.1	Muster in der let-Anweisung	350
8.7.2	Einfache Muster	352
8.7.3	Der Bindungsoperator	359
8.7.4	Die Widerlegbarkeit von Mustern	360
8.8	Zusammenfassung	364

9 Fehlerbehandlung

9.1	Fehler, von denen sich das Programm nicht erholen kann	367
9.1.1	Eine Panic tritt pro Thread auf	368
9.1.2	Wenn Sie Panics für möglich halten, isolieren Sie sie vom Hauptthread	371
9.1.3	Mit »abort« vermeiden, dass der Stack abgewickelt wird	372
9.1.4	Panics abfangen und behandeln	373
9.2	Erwartbare Fehler behandeln	381
9.2.1	Result: Ein Typ, zwei Wege	382
9.2.2	Die Werte in Result verarbeiten und verwenden	386
9.2.3	Der Fragezeichen-Operator	397

9.2.4	Das Trait Error	399
9.2.5	Option<T>: Behälter und Fehlertyp	407
9.3	Zusammenfassung	418

10 Strukturen

10.1	Daten zusammenhängend ablegen	422
10.2	Records: Der Struktur-Urtyp	423
10.3	Strukturen und Instanzen	426
10.3.1	Keine Konstruktoren in Rust	427
10.3.2	Kurzschrifweise und Update-Syntax	428
10.3.3	Der Zugriff auf Felder und die Veränderlichkeit	430
10.3.4	Die automatische Dereferenzierung in der Struktur	433
10.3.5	Datenkapselung in Rust	434
10.4	Lebenszeiten: Wenn Felder Referenzen enthalten	441
10.4.1	Statische und nicht statische Referenzen	441
10.4.2	Generische Lebenszeiten in der Struktur	443
10.4.3	Vermeiden Sie zu viele generische Lebenszeiten in Strukturen	446
10.4.4	Generische Lebenszeiten, die nicht begrenzt werden	448
10.5	Wie Sie dem Compiler mit PhantomData wichtige Typinformationen übergeben	449
10.5.1	Die Speichersicherheit mit »PhantomData« erweitern	454
10.5.2	Einen generischen Datentyp mit PhantomData einsetzen	456
10.6	Eine Datenstruktur ohne feste Größe	460
10.7	Die drei Strukturen	462
10.7.1	Die Tupel-Struktur	462
10.7.2	Das Newtype-Muster	463
10.7.3	Unit-Typ-ähnliche Strukturen	465
10.8	Muster	466
10.9	Daten und Verhalten sind getrennt	468
10.9.1	Der inhärente Implementierungsblock	468
10.9.2	Generische Lebenszeiten im Implementierungsblock	469
10.9.3	Eine generische Lebenszeit im Implementierungsblock einführen	472
10.9.4	Wenn der implementierende Typ eine generische Lebenszeit aufweist	474
10.9.5	Anonyme Lebenszeiten	475

10.10 Strukturen in der Praxis: Das Bestellsystem überarbeiten	475
10.10.1 Die Anforderungen und der Entwurf des Systems	475
10.10.2 Projektaufbau und erste Datenstrukturen	477
10.11 Assoziierte Funktionen und die new-Konvention	480
10.11.1 Öffnungszeiten im Restaurant	481
10.11.2 Die Sichtbarkeit erweitert sich nicht	482
10.11.3 Strukturen mit »new« initialisieren	484
10.12 Methoden	486
10.12.1 Die Methode und der self-Parameter	486
10.12.2 self ist eine Kurzform	488
10.12.3 Mehrere Parameter	490
10.12.4 »&mut self« und andere Seiteneffekte in Methoden	494
10.12.5 »self« und »mut self«: Die eigene Instanz verbrauchen	498
10.13 Referenzen in assoziierten Funktionen und Methoden	501
10.14 Praxisbeispiel: Simulationsfähigkeiten im Restaurant-System	503
10.14.1 Hunger und Durst	503
10.14.2 Die Wahl-Funktion in »BestellungBuilder« anschließen	505
10.15 Rekursion in Strukturen	507
10.16 Typ-Aliasse	510
10.17 Zusammenfassung	512

11 Traits

11.1 Marker-Traits	516
11.2 Trait-Implementierungsblöcke	517
11.2.1 Die Orphan-Regel	518
11.2.2 Elemente, die Sie mit Traits assoziieren können	521
11.2.3 Sichtbarkeiten	538
11.3 Sie können ein Trait jeweils für T und &T implementieren	541
11.3.1 Self kann T oder &T sein	542
11.3.2 Lebenszeiten von Referenzen in der Trait-Implementierung	543
11.3.3 Mehrere Lebenszeiten im Trait-Implementierungsblock	545
11.4 Super-Traits	546
11.5 Trait-Objekte	549
11.5.1 Wie ein Trait-Objekt entsteht	552
11.5.2 Wie eine vTable aussieht	552

11.5.3	Ein Trait-Objekt ist eine Einbahnstraße	555
11.5.4	Der Sized-Marker	556
11.5.5	Nicht jedes Trait kann zu einem Trait-Objekt werden	559
11.5.6	Objektsicherheit: Was ist das?	559
11.5.7	Ein Trait für ein Trait-Objekt implementieren	561
11.5.8	Der inhärente Implementierungsblock eines Trait-Objekts	563
11.6	Beispielprojekt: Trait-Objekte von »Form«	564
11.6.1	Die Komponenten	565
11.6.2	Das Koordinatenfeld zeichnen	566
11.6.3	Die Implementierungen von »Form« für Punkt und Linie	569
11.6.4	Das Rechteck: Mehrere Linien berechnen	571
11.6.5	Der Kreis	572
11.6.6	Trait-Objekte an das Koordinatenfeld übergeben	573
11.7	Undurchsichtige Datentypen zurückgeben	574
11.7.1	Abstrakte Rückgabetypen	575
11.7.2	Anonyme Typparameter	576
11.8	Traits in der Praxis	578
11.8.1	Copy und Clone	579
11.8.2	Any und TypId	585
11.8.3	»drop« – der Rust-Destruktor	595
11.8.4	Default: ein Standardkonstruktor à la Trait	602
11.8.5	Borrow<T> und BorrowMut<T>	605
11.8.6	AsRef<T> und AsMut<T>	613
11.8.7	Typkonvertierungen mit From<T>	615
11.8.8	Into<T> ist das Gegenstück zu From<T>	622
11.8.9	From<T> oder Into<T>: Wann Sie welches Trait implementieren sollten	623
11.8.10	TryFrom<T> und TryInto<T>	625
11.9	Zusammenfassung	627

12 Enumerationen

		631
--	--	-----

12.1	Die Eigenschaften einer Enumeration	632
12.1.1	Eine Enumeration ohne Varianten	633
12.1.2	Implizite und explizite Diskriminanten	635
12.1.3	Explizite Diskriminanten sind konstante Werte	638
12.1.4	Instanzen einer Enumeration	639

12.2 Verschiedene Variant-Typen	644
12.2.1 Tupel-Varianten	644
12.2.2 Struktur-Varianten	647
12.2.3 Varianten als Datentypen	648
12.2.4 Speicherbedarf	653
12.3 Enumerationen und Muster	656
12.4 Implementierungsblöcke und Verhalten	660
12.4.1 »Gewicht« und der inhärente Implementierungsblock	660
12.4.2 Das Trait »Add« für »Gewicht«	662
12.4.3 Das Trait »Display« für »Gewicht«	665
12.5 Zusammenfassung	667

13 Module, Pfade und Crates 669

13.1 Das Modul	669
13.1.1 Das implizite Modul	670
13.1.2 Ein explizites Modul definieren	671
13.1.3 Namensraum und Sichtbarkeit im Modul	672
13.1.4 Der Modulbaum	678
13.1.5 Denken Sie in Modulen, nicht in Dateien oder Verzeichnissen	687
13.2 Pfade	697
13.2.1 Die Anatomie eines Pfads	697
13.2.2 Welche Elemente bekommen einen Pfad?	699
13.2.3 Wohin ein Pfad führen kann	701
13.2.4 Wie Sie einen Pfad »umbiegen«	710
13.2.5 Die »use«-Deklaration	712
13.3 Vom Crate zum Paket, vom Paket zum Workspace	721
13.3.1 Am Anfang war das Crate	721
13.3.2 Das kleinste Crate ist eine Rust-Datei	722
13.3.3 Das Paket	723
13.3.4 Paketabhängigkeiten hinzufügen: Ein Paket kommt selten allein	731
13.3.5 Paketabhängigkeiten konfigurieren	734
13.3.6 »dev-dependencies«: Abhängigkeiten nur für die Entwicklung	735
13.3.7 »Build-Skripte« und die »build-dependencies«	736
13.3.8 Paketversionen anfordern	737
13.3.9 Attribute	742
13.3.10 Konditionale Kompilierung	753

13.3.11 Features	762
13.3.12 Workspaces	772
13.4 Zusammenfassung	777

14 Generische Programmierung

14.1 Von der Vorlage zur Konkretisierung: Monomorphisierung	781
14.2 Typparameter, generische Konstanten und Lebenszeiten	783
14.3 Syntaktische Elemente, die generisch sein können	785
14.3.1 Implementierungsblöcke reichen Typparameter weiter	785
14.3.2 Der assoziierte Datentyp eines Datentyps ist generisch	786
14.3.3 Generische und assoziierte Datentypen verbinden	787
14.3.4 Assoziierte Datentypen und Trait-Grenzen	788
14.4 Mehr zu Trait-Grenzen	789
14.4.1 Trait-Grenzen kombinieren	789
14.4.2 Anonyme Typparameter: »impl Trait«	791
14.4.3 Trait Grenzen mit »where«	791
14.4.4 Blanket-Implementierungen	792
14.5 Zusammenfassung	794

15 Iteratoren

15.1 Wie Sie einen Iterator beziehen	798
15.1.1 Das Trait »Intoliterator«	799
15.1.2 »Iterator« implementieren	800
15.2 Iterator-Adapter	805
15.2.1 Eigentum im Iterator	807
15.2.2 Iteratoren zusammenfügen	812
15.2.3 Transformationen in der Iterator-Kette	813
15.2.4 Weitere Iterator-Adapter	816
15.3 Einen Iterator konsumieren	816
15.3.1 Eine Zusammenstellung einfacher Konsumenten	817
15.3.2 Finden, falten und reduzieren	817
15.3.3 Sonstige Methoden, die einen Iterator konsumieren	820
15.4 Zusammenfassung	822

16 Nebenläufige und asynchrone Programmierung 823

16.1 Nebenläufige Programmierung	824
16.1.1 Threads	826
16.1.2 »Send« und »Sync«: sicherer nebenläufiger Code	838
16.1.3 Channels: Die Kommunikationsinfrastruktur zwischen Threads	851
16.1.4 Synchronisierung in Konkurrenz: Der wechselseitige Ausschluss	858
16.1.5 Einen oder mehrere Threads per Signal aufwecken mit »Condvar«	864
16.1.6 RwLock: Mehrere Leser oder ein exklusiver Zugriff	868
16.1.7 Atomare Datentypen und Operationen	869
16.2 Smart Pointer	876
16.2.1 »Cow«: Referenz und Klonanleitung zugleich	877
16.2.2 Cells und die Interior Mutability	880
16.2.3 Mit Referenzzählern zum geteilten Eigentum	888
16.3 Asynchrone Programmierung	893
16.3.1 Der Unterschied zwischen Thread und Task	894
16.3.2 »async« und »await«	895
16.3.3 Die asynchrone Laufzeitumgebung	897
16.3.4 Asynchrone Funktionen, Parallelität und Join	899
16.3.5 Die Future ist ein Zustandsautomat	902
16.4 Zusammenfassung	915

17 Makros 917

17.1 Deklarative Makros	917
17.1.1 Warum Makros?	918
17.1.2 Ein Beispiel-Makro	920
17.1.3 Die Meta-Syntax	921
17.1.4 Der Gültigkeitsbereich	938
17.1.5 Makro-Hygiene	939
17.2 Prozedurale Makros	939
17.2.1 Für prozedurale Makros müssen Sie eine Bibliothek anlegen	941
17.2.2 Die drei prozeduralen Makro-Arten	942
17.3 Zusammenfassung	950

18 Automatische Tests und Dokumentation	953
18.1 Tests	954
18.1.1 Das Untermodul »tests«	954
18.1.2 Testfunktionen, Result<T, E> und der Fragezeichen-Operator	956
18.1.3 Die Attribute »ignore« und »should_panic«	957
18.1.4 Teststrukturierung	960
18.1.5 Dynamische und statische »asserts«	963
18.1.6 Integrationstests	965
18.2 Rust-Projekte dokumentieren	966
18.2.1 Die Dokumentation erzeugen	967
18.2.2 Wie Sie Ihr Projekt dokumentieren	970
18.2.3 »Doc-Tests«: Codebeispiele in Kommentaren	975
18.3 Zusammenfassung	979
19 Unsafe Rust und das Foreign Function Interface	981
19.1 Unsafe Rust	981
19.1.1 »unsafe« in Blöcken und Funktionen	982
19.1.2 Unsichere Traits und Trait-Implementierungen	985
19.1.3 Statisch-globale Variablen verändern	986
19.2 Primitive Zeiger	987
19.2.1 Wie Sie gültige Zeiger erhalten	987
19.2.2 Null-Zeiger	989
19.2.3 Operationen	990
19.2.4 Der Fallstrick Move	996
19.3 Union	998
19.4 Foreign Function Interface	1001
19.4.1 Von Rust zu C oder C++	1002
19.4.2 Von C oder C++ zu Rust	1003
19.5 Zusammenfassung	1005
Index	1007