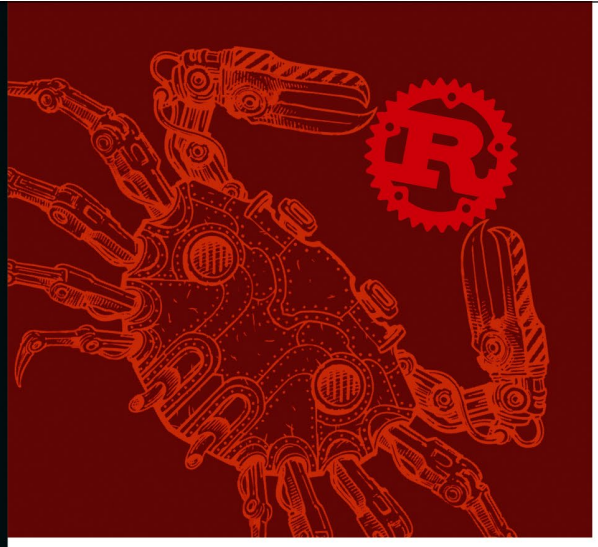


Marc Marburger

struct, Option, I/O, Threading, RefMut, Makro, Unit Tests, Closures, Traits

```
let mut zahl_gruppen = HashMap::new();
let eintrag_vacant = zahl_gruppen.entry("gerade");
println!("{eintrag_vacant:?}"); // Entry(VacantEntry("gerade"))
// Einen Eintrag mit Entry einfügen
eintrag_vacant.or_insert(vec![2, 4, 6]);
println!("{zahl_gruppen:?}");
// {"gerade": [2, 4, 6]}
let eintrag_occupied = zahl_gruppen.entry("gerade");
println!("{eintrag_occupied:?}");
// Entry(OccupiedEntry { key: "gerade", value: [2
```

Rust

Das umfassende Handbuch

- Mit modernen Mitteln performant und sicher programmieren
- Von »Hallo Welt« bis zu komplexen Features und eigenen APIs
- Inkl. Traits und Closures, asynchrone Programmierung, Testautomatisierung, Foreign Functions u. v. m.

Alle Beispielprojekte zum Download

Rheinwerk Computing

Kapitel 4
Speichernutzung und Referenzen

Programme nutzen unterschiedliche Speichersegmente, um vielfältige Anforderungen an die Datenhaltung abzubilden. Dieser Abschnitt beleuchtet den *Stack*, *Heap* und *statischen Speicher*. Sie lernen, wie der Compiler und die Laufzeitumgebung von Rust Werte im Speicher ablegen, bewegen und wann sie die Daten wieder freigeben.

Wir beschäftigen uns insbesondere mit dem Konzept *Eigentum* (engl. *Ownership*). Ein Wert kann in Rust stets nur einen Eigentümer haben. Wenn Sie den Wert einer Variablen an eine andere zuweisen, muss der Compiler daher den Wert bewegen oder eine Kopie erstellen. Alternativ können Sie von einem Wert viele geteilte Referenzen oder eine veränderliche Referenz ausleihen. Auch hier erwartet der Compiler, dass Sie sich an ein Regelwerk halten, das die Speichersicherheit eines jeden Rust-Programms garantiert. Die einzige Ausnahme stellt *Unsafe Rust* dar, das wir in Kapitel 19, »Unsafe Rust und das Foreign Function Interface«, besprechen werden.

4.1 Wichtige Speicherbereiche

Gängige Programmiersprachen und ihre Laufzeitumgebungen setzen auf Speicherallokationen im Heap, um dynamische Speicheranforderungen abzubilden. Ob ein Speicherobjekt noch referenziert wird, prüft dann zumeist ein *Garbage Collector*. So bleibt es der Programmiererin und dem Programmierer erspart, den Speicher selbst anzufordern oder freizugeben. Der Garbage Collector hat jedoch den Nachteil, dass er das Programm für einen unmerklichen Augenblick anhalten muss, um den Speicher zu untersuchen und aufzuräumen. Das kann wichtige Millisekunden kosten, die manch eine Anwendung nicht tolerieren kann.

Rust verzichtet dagegen auf einen Garbage Collector und ermutigt gleichzeitig dazu, den Stack zu verwenden. Trotzdem können Sie uneingeschränkt auf den Heap zurückgreifen. Über das Eigentum und spezielle Eigentumsregeln verfolgt der Compiler die Lebenszeit von Variablen und Referenzen über alle Speicherbereiche hinweg, und zwar schon beim Kompilieren. Ein Garbage Collector ist dazu nicht nötig.

Die Speicherstrukturen Stack, Heap und auch den statischen Speicher zu verstehen, ist somit für den Umgang mit Rust vorteilhaft. Die nächsten Abschnitte bieten einen Überblick über die jeweiligen Strukturen, bevor wir in Abschnitt 4.3 auf die Referenzen in Rust eingehen.

4.1.1 Der Stack

Wenn Sie den Compiler starten, übersetzt er den Quellcode in eine *Objektdatei*. Eine Objektdatei ist in verschiedene *Segmente* unterteilt, in die der Compiler Instruktionen, Variablen, statische Informationen und Konstanten usw. einsortiert. Die Segmente liegen übereinander und belegen unterschiedlich lange Speicheradressbereiche.

Eines dieser Segmente ist der Stack. In den gängigsten Objektdateiformaten ist er an letzter Stelle platziert. Daher weisen die im Stack platzierten Elemente die höchsten Adressen auf. Jeder Aufruf einer Funktion in Ihrem Programm erzeugt einen neuen Eintrag auf dem Stack. Ein Eintrag wird als *Stackframe* oder auch *Activation Record* bezeichnet. Er setzt sich insbesondere aus der Rückkehradresse, von wo die Funktion aufgerufen worden ist, und aus den lokalen Variablen zusammen. Stackframes können daher unterschiedlich groß sein.

Stacks wachsen sowohl nach oben in höhere Adressen als auch nach unten in niedrigere Adressen. Die Richtung kann sich je nach Prozessorarchitektur und *Application Binary Interface* (*ABI*) unterscheiden. Da der Speicher von lokalen Variablen mit dem Stack automatisch angefordert und freigegeben wird, bezeichnet man sie auch als *automatische Variablen*. Je mehr Variablen Sie dementsprechend auf dem Stack anlegen, desto besser können der Compiler und die Laufzeitumgebung auch das Laden der Daten optimieren.

Die Größe eines Stackframes muss beim Kompilieren feststehen, Gleiches gilt für den Speicherbedarf der Variablen. Zur Laufzeit wachsende oder schrumpfende Speicherbereiche können Sie aus diesem Grund nur im Heap abbilden. Stackframes werden übereinandergestapelt. Ergebnisse oder Argumente können deswegen nur durch die Frame-Kette weitergereicht werden. Eine Funktion am Boden des Stacks kann ihre Daten daher nicht mit der Funktion fünf Frames weiter oben austauschen. Das ist ein weiterer Anwendungsfall für den Heap oder statischen Speicher.

4.1.2 Der Heap

Der Heap steht Ihrem Programm zur Verfügung, um Daten nach Bedarf zur Laufzeit im Speicher zu halten. Sie können dort strukturierte Daten als *anonyme Variablen* ablegen. Warum anonym? Statische und lokale Variablen erhalten stets einen Bezeichner. Dagegen greifen Sie auf den Heap-Speicher nur indirekt über *Zeiger* (engl. *Pointer*) zu, die an die Speicheradresse im Heap binden. Die Zeiger-Variablen sind wiederum statisch oder lokal und besitzen einen Bezeichner.

Zur Verwaltung des Heaps existieren verschiedenste Vorgehensweisen: Die Lösungen beginnen bei einer vollständigen Verwaltung durch die Programmiererin oder den Programmierer und enden beim komfortablen *Garbage Collector*, der alle Ob-

jekte des Programms daraufhin untersucht, ob sie noch in Verwendung sind – falls nicht, räumt er sie ab.

Den Speicher manuell zu steuern, führte in der Vergangenheit zu Problemen, etwa zu mehrfach oder gar nicht freigegebenem Speicher. Mit C++ 11 führte die Sprache *Smart Pointer* ein, die selbst für die Freigabe von Speicher im Heap sorgen. Rust setzt gleichermaßen auf Smart-Pointer-Strukturen, die wir in Abschnitt 16.2, »Smart Pointer«, tiefergehend besprechen. Zudem gehören Strukturen im Heap nur einem einzigen Eigentümer. Dieses Design schützt den Programmcode vor der mehrfachen Speicherfreigabe.

Da Rust Speicher zur Laufzeit im Heap anfordert oder aus ihm lädt, steht er einer Funktion nicht unmittelbar zur Verfügung. Stattdessen muss die CPU einige Instruktionen ausführen. Im besten Fall müssen nur Bytes an der angegebenen Adresse nachgeladen werden. Im schlechtesten Fall erfordert eine zusätzliche Speicheranforderung, dass der Allokator sich auf die Suche nach einem leeren Speicherstück machen muss. Wenn er eine entsprechende Lücke findet, erhält die Funktion ihre Daten oder den größeren Puffer. Wie leistungskritisch die zusätzlichen Instruktionen und möglicherweise verpassten CPU-Zyklen sind, entscheidet der Anwendungsfall.

4.1.3 Statischer Speicher

Ein weiteres Speichersegment, auf das ein Programm zurückgreift, ist das statische. Der Compiler legt hier alle mit dem Schlüsselwort `static` eingeführten Variablen ab. Sie erhalten so eine feste, vom Compiler vergebene – deswegen *statische* – Adresse. Statische Variablen sind nicht Teil des Stacks. Das Programm lädt beim Start statische Daten aus diesem Segment. Die Lebenszeit einer statischen Variable beginnt und endet mit der Programmlaufzeit. Dieses Detail ermöglicht daher Sicherheitsgarantien im Zusammenhang mit Referenzen: Das Objekt hinter der Adresse bleibt immer gültig.

4.2 Eigentumsverhältnisse im Speicher

Ein Computerprogramm greift auf zahlreiche Ressourcen eines Computers zu, um ein Problem zu lösen oder Informationen zu modellieren. Darunter fällt auch der Speicher, der verschiedenste Daten und Zusammenhänge aufnimmt. Die Speicherverwaltung umfasst sowohl die Anforderung (*Allokation*) als auch die Freigabe (*Deallokation*) von Speicher. Geht dabei etwas schief, könnten Programmfehler, aufgebrauchter Speicher oder Sicherheitslücken die Folge sein.

Durch die Einhaltung weniger Regeln in Rust, die der Compiler zudem automatisch für Sie prüft, müssen Sie sich über die Vermeidung von Speicherfehlern in den meis-

ten Fällen nicht mehr den Kopf zerbrechen. Der Compiler schafft zwar ein außerordentliches Sicherheitsnetz. Dennoch kann gültiger Rust-Code zu logischen Fehlern oder Race Conditions führen – zu Data Races dagegen nicht.

Dieser Abschnitt führt Sie in die Mechanik ein, die Rust in die Lage versetzt, eine hohe Speichersicherheit zu garantieren. Anschließend besprechen wir die Referenztypen `&T` und `&mut T`, die Sie sich von einem Ausdruck ausleihen können. Zuletzt blicken wir auf `Box<T>`, die Ihnen die Tür öffnet, um Speicher unmittelbar im Heap zu allozieren.

4.2.1 Ein Wert, ein Eigentümer

Eine Variable bindet Bezeichner an Werte, etwa ein `bool` oder eine komplexere Struktur im Speicher. Ob auf dem Stackframe, Heap oder im statischen Speicher: Die Bindung, die Variable und Wert eingehen, ist immer endlich. Während dieser Zeit bleibt der Wert Eigentum der Variable. Listing 4.1 zeigt ein Beispiel mit zwei unterschiedlichen Bindungen:

```
fn main() {
    // Nur auf dem Stack
    let a = 10;

    // "s" bindet an einen String mit Puffer im Heap
    let s = String::from("Dynamischer Speicher");
}
```

Listing 4.1 Die Variablen »a« und »s« binden ihre Werte.

Der Begriff *Eigentum* (engl. *Ownership*) zielt hauptsächlich auf die Verantwortung ab, den Speicher freizugeben, wenn die Variable den Gültigkeitsbereich verlässt. Wenn `main` wie im oberen Beispiel endet und die Laufzeitumgebung den Frame der Funktion vom Stack abräumt, verliert die Variable `a` ihre Gültigkeit. Die im Format `i32` interpretierten Bytes werden damit von `a` – und nur von `a` – für die Wiederverwendung freigegeben.

Rust stellt im Programmablauf ausnahmslos sicher, dass die Frage nach dem Eigentum eine eindeutige Antwort aufweist. Andernfalls könnten mehrfache Deallokationen die Folge sein. Im Beispiel von `a` ist die Speicherfrage trivial: Die Ganzzahl liegt im Stackframe.

Kommt allerdings der Heap ins Spiel, wie im Fall des `String`s, den die Variable `s` bindet, sind die Details zu »Wer gibt den Speicher frei?« und »Wie oft?« von hoher Wichtigkeit! Die lokale Variable `s` liegt zwar auf dem Stack und wird mit dem umgebenden Frame automatisch freigegeben. Die `String`-Instanz führt allerdings daraufhin eine spezielle Routine (*Drop*) aus. Hierdurch weist der `String` das Speichermanagement

von Rust an, den Puffer im Heap freizugeben und erneut zur Verfügung zu stellen. Dazu darf es allerdings nur ein einziges Mal kommen, andernfalls drohen Gefahren wie *Double Frees*!

4.2.2 Gültigkeitsbereiche und Blöcke beeinflussen Bindungen

Eine lokale Variable, die Sie in einer Funktion einführen, ist grundsätzlich vom Anfang der Funktion bis zu deren Ende gültig. Wenn Sie die Variable aber nur über wenige Zeilen hinweg einsetzen und dann wieder fallen lassen möchten, können Sie hierfür einen *Block* einsetzen. Das erweist sich besonders dann als praktisch, wenn Sie eine Variable in einem speziellen Kontext sichtbar machen wollen, etwa in der Nähe einer Abfrage oder Schleife.

Um Gültigkeitsbereiche nach Bedarf einzuführen, umschließen Sie die einzugrenzenden Anweisungen und Ausdrücke mit geschweiften Klammern. Sie können etwa die Berechnung einer Zufallszahl in einen Block auslagern. Dann reduzieren Sie die Sichtbarkeit von Bezeichnern und Variablen auf das Minimum und müssen dafür keine neue Funktion einführen. Ein Beispiel:

```
fn main() {
    let zahl: i32;
    {
        // Rng aus dem Crate "rand" sichtbar machen
        use rand::Rng;
        let mut generator = rand::thread_rng();
        // number ist hier von außen sichtbar
        zahl = generator.gen();
    }
    // Außerhalb des Blocks sind "rand" und
    // "generator" nicht sichtbar!
    // Nur "nachricht" und "zahl" aus dem
    // gleichen oder vorherigen Bereich sind sichtbar.
    let nachricht = "Die nächste zufällige Zahl ist";
    println!("{nachricht}: {zahl}");
}
```

Listing 4.2 Berechnung einer Zufallszahl in einem Block

Sie können im Rumpf des Blocks zwar auf die Umgebung zugreifen (wie auf `zahl`). Aber von außen sind die Typen, die Sie im Block sichtbar gemacht haben (etwa mit `use`), oder lokale Variablen (`generator`) nicht sichtbar. Mit diesem Kniff schaffen Sie mehr Ordnung im Gültigkeitsbereich.

Blöcke verschachteln

Sie können Blöcke ineinander verschachteln. Wenn Ihnen ein weiterer Block zur Abgrenzung in einem anderen Block hilft, führen Sie ihn einfach ein! Rust erlegt Ihnen dahingehend keine Einschränkungen auf.

Neben dem aufgeräumten Kontext nehmen wir außerdem zur Kenntnis, dass der Block die Variable `generator` mit dessen Ende freigibt. Das Speicherobjekt bleibt nicht bis zum Ende der äußeren Funktion am Leben, weil der Eigentümer `generator` seine Gültigkeit verliert und die Bindung auflöst.

Blöcke sind Ausdrücke und liefern demnach wie eine Funktion Werte zurück. Wenn Sie keinen expliziten Rückgabewert bestimmen, gibt ein Block den Unit-Typ `()` zurück – das Rust-Äquivalent zu `void`, wie es C oder C++ verwenden. (Den Unit-Typ behandelt Abschnitt 6.1, »Tupel«.)

Listing 4.2 hat die Rückgabe ignoriert. Stattdessen hat der Block das Ergebnis des Generators an eine lokale Variable zugewiesen, um die Sichtbarkeiten nachzuzeichnen. Das ist jedoch viel umständlicher, als den Rückgabewert des Blocks einzusetzen. Listing 4.3 zeigt den Code, umgeschrieben auf die Rückgabe des Zufallswerts:

```
fn main() {
    let zahl: i32 = {
        // Rng aus dem Crate "rand" sichtbar machen
        use rand::Rng;
        let mut generator = rand::thread_rng();
        generator.gen()
    };

    let nachricht = "Die nächste zufällige Zahl ist";
    println!("{nachricht}: {zahl}");
}
```

Listing 4.3 Ein Block besitzt einen Rückgabewert.

Beachten Sie, wie der Block implizit den Wert des Ausdrucks `generator.gen` zurückgibt. Wenn Sie das Semikolon weglassen, gibt ein Block den Rückgabewert des letzten Ausdrucks an die Umgebung weiter. Das erspart es Ihnen, `return` und das Semikolon einzusetzen.

Block-Bezeichner

Seit Rust 1.65.0 können Sie einem Block einen *Bezeichner* (engl. *Label*) zuweisen. Das ermöglicht Ihnen in Verbindung mit dem Schlüsselwort `break`, den Block an einer ge-

wünschten Stelle zu verlassen. Der Programmfluss muss den Block daher nicht bis zum Ende durchlaufen. Das erspart es Ihnen, den Programmfluss mit anderen Mitteln umzuleiten.

Zuvor war es den Schleifen-Ausdrücken `loop`, `for` und `while` vorbehalten, einen Bezeichner einzusetzen. In Kapitel 8, »Anweisungen, Ausdrücke und Muster«, erfahren Sie dazu mehr. Listing 4.4 zeigt einen Block mit Bezeichner und `break`:

```
'block_label: {
    println!("Bis hierhin");
    break 'block_label;

    // "break" verhindert die Ausführung
    println!("aber nicht weiter");
}
```

Listing 4.4 Einen Bezeichner für einen Block einführen und mit »break« verlassen

Den Block-Bezeichner führen Sie mit einem Hochkomma ein. Der Unterstrich ist als Bezeichner nicht zugelassen und führt daher zu einem Compilerfehler. Sollten allerdings weitere alphanumerische Zeichen folgen, ist der einleitende Unterstrich gültig. Dagegen sind mit Ziffern beginnende Bezeichner stets ungültig. Ein Doppelpunkt schließt den Block-Bezeichner-Ausdruck ab.

Das Hochkomma führt ebenfalls generische Lebenszeiten ein

Mit dem Zeichen ' definieren Sie auch Bezeichner für generische Lebenszeiten. Es begegnet Ihnen in späteren Kapiteln, wenn Sie Referenzen in Strukturen oder Funktionen deklarieren. Die Kapitel 7, »Funktionen«, und 10, »Strukturen«, weisen Sie in den Umgang ein.

Die Ausführung des Blocks können Sie mit dem Schlüsselwort `break` an einer beliebigen Stelle unterbinden. Achten Sie darauf, dass Sie nach `break` stets einen Bezeichner – also ein Ziel – angeben müssen.

Die Sprünge aus einem Block mit `break` erinnern an das *GOTO*-Konzept. Wenn eine Sprache GOTO-Marker unterstützt, können Sie im Code Sprungziele definieren und sie dann ohne weitere Einschränkungen anspringen. Dagegen können Sie in Rust nur zu einem Block-Bezeichner springen, der in derselben Funktion und über dem `break`-Ausdruck liegt. Es handelt sich in Rust daher nicht um eine GOTO-Funktionalität. Beispielsweise führt folgender Code zu einem Fehler:

```
'block_label: {
    println!("Bis hierhin");
```

```
    // Fehler
    break 'bezeichner_gleicher_ebene;

    'bezeichner_gleicher_ebene: {
        println!("Neues Sprungziel");
    }
}
```

Listing 4.5 Mit »break« können Sie nur nach oben heraus springen.

Falls Sie einen `break`-Ausdruck einsetzen, der Compiler jedoch keinen Block mit einem Bezeichner findet, führt das zu einem Fehler. Dann verweist der Compiler darauf, dass `break` nur innerhalb von Schleifen erlaubt ist. Sie müssen also stets einen Block-Bezeichner angeben, damit der Compiler den Kontext richtig einordnen kann.

»return« vs. »break«

Im GitHub-Issue zum Feature »label-break-value« (*https://github.com/rust-lang/rust/issues/48594*, im Jahr 2018 geöffnet) findet sich eine Diskussion dazu, wie die Schreibweise des Block-Bezeichners einmal aussehen soll.

Es wurden Argumente für und wider den Einsatz von `return` oder `break` angeführt. Schließlich kristallisierte sich die Haltung heraus, dass `return` immer als Ende einer Funktion anzusehen ist. Dagegen wirkt `break` bereits in Schleifen und in anderen Sprachen (etwa in `switch`-Abfragen anderer Sprachen) als Sprungbrett.

Besonders einleuchtend ist der Hinweis darauf, dass `return` ein dynamischer Charakter innewohnt: Der Rücksprung aus einer Funktion wird möglicherweise erst zur Laufzeit aufgelöst, weil mehrere Stellen im Code diese Funktion aufrufen. Bei `break` liegt die Sache anders: Den Sprung aus Schleifen oder Blöcken löst bereits der Compiler auf. `break` ist damit statisch, genau wie der angesprungene Block-Bezeichner.

4.2.3 Die Lebenszeit des Werts einer Variable

Wenn Sie eine Variable in einem Block oder einer Funktion einführen, dann ist sie so lange gültig, bis der Programmfluss den Gültigkeitsbereich verlässt. Dieses Verhalten implementiert der Stack. Er fordert den Speicher für die Variablen, die Sie in einer Funktion einführen, automatisch an und gibt ihn frei.

Die Dauer, über die sich eine Variable an einen Wert bindet, nennt man *Lebenszeit* (engl. *Lifetime*). In Listing 4.2 hatte die Variable `zahl` eine größere Lebenszeit als die Variable `generator`, die wir im Block darunter eingeführt hatten. Am Ende gibt der Block `generator` frei, was einen weiteren Zugriff ausschließt. Die Variable hat dann das Ende ihrer Lebenszeit erreicht. Der Begriff Lebenszeit hängt nicht spezifisch mit Rust

zusammen. Die Begriffserklärung trifft ebenfalls auf andere Sprachen zu, etwa auf C. Rust zeichnet jedoch aus, dass der Compiler das Eigentumsverhältnis durch *Move* erzwingt (auch *Move-Semantik* genannt, dazu unten mehr) und generische Lebenszeiten in der Syntax sichtbar macht.

Lebenszeiten spielen im Zusammenhang mit Referenzen eine zentrale Rolle. Der Compiler erkennt und sichert hierdurch die Speicherverwendung. In Abschnitt 4.3.1, »Adressen, Zeiger und Referenzen: ein Überblick«, und Abschnitt 4.3.2, »Zugriff auf Werte über geteilte Referenzen«, nehmen wir das Thema daher erneut auf.

Weil es in Rust immer nur einen Eigentümer eines Werts geben darf, dürfen Sie den Wert einer Variable nicht mehr auslesen, nachdem Sie ihren Wert an ein anderes Ziel gebunden haben. Derselben Variable können Sie jedoch weiterhin einen neuen Wert zuweisen, sofern Sie sie als veränderlich eingeführt haben. Danach wäre auch der lesende Zugriff darauf wieder gültig.

Diese Technik wird als *Move* bezeichnet und ähnelt der *Move-Semantik* von C++. Zur Wahrung der Eigentumsverhältnisse wendet der Compiler daneben auch das Trait `Copy` an, sodass die Begriffe `Copy` und Move in jede Zuweisung involviert sind. Die beiden nächsten Abschnitte beleuchten jeweils die Details.

4.2.4 Bitweise Kopien mit »Copy« erzeugen

Um die bestmögliche Leistung zu garantieren und gleichzeitig das Eigentumsverhältnis zu bewahren, betrachtet der Compiler den gebundenen Wert genau: Wenn er keinerlei Ressource über Referenzen, Zeiger usw. verwaltet, lässt sich gefahrlos eine bitweise Kopie erzeugen. Ein Beispiel:

```
let a = 10;
let b = a;
println!("a = {a}"); // 10
println!("b = {b}"); // 10
```

Listing 4.6 »a« und »b« binden jeweils einen Speicher mit dem Wert 10.

Bei der Zuweisung von `a` an die Variable `b` erkennt der Compiler, dass `i32` (der Typ von `a`) die Eigenschaft `Copy` erfüllt. Daher erstellt er eine bitweise Kopie des Werts und bindet die Variable `b` daran. Beide Variablen besitzen fortan jeweils einen eigenen Wert, den sie am Ende ihres Gültigkeitsbereichs freigeben.

Bitweise Kopien stoßen jedoch bald an Grenzen. Sobald ein Typ intern etwa eine Ressource verwaltet, muss der Compiler Move anwenden. Andernfalls würde mit der bitweisen Kopie auch der Anspruch an das Eigentum vervielfältigt. Dann gäbe es zwei Eigentümer für eine Ressource. Stattdessen geht der Wert durch Move in das Eigentum einer anderen Variable über.

Was ist Copy?

Sie werden im Umfeld von Rust häufig den Satz »`T` ist Copy« (engl. »`T` is Copy«) hören. Was aber bedeutet das? `Copy` ist eine Eigenschaft, die das *Marker-Trait* `std::marker::Copy` ausdrückt. Wenn ein Typ `Copy` implementiert, erkennt der Compiler, dass er gefahrlos eine bitweise Kopie erstellen kann.

Später, wenn Sie eigene Typen entwerfen, steht es Ihnen offen, `Copy` zu implementieren. Diese Entscheidung beeinflusst, wie Nutzer Ihren Typ verwenden können. Versuchen Sie immer, ein für die Rust-Programmiererin oder den Rust-Programmierer intuitives Verhalten zu erreichen – im Rust-Umfeld auch *idiomatisch* genannt.

Das Trait `Copy` besprechen wir in Abschnitt 11.8.1, »Copy und Clone«. Dann betrachten wir insbesondere, warum nicht jeder Typ `Copy` sein kann.

4.2.5 Move bindet den Wert an einen neuen Eigentümer

Das `Copy`-Trait und mit ihm bitweise Kopien eines Werts sind zwar praktisch, viele Typen fallen jedoch aus dem Raster und sind damit nicht kompatibel, so etwa der Datentyp `String`. Seine Struktur liegt auf dem Stack, und seine Nutzdaten hält er im Heap. Das ermöglicht es ihm, dynamisch auf eine unterschiedliche Zeichenanzahl zu reagieren.

Dabei greift `String` intern auf den Vektor `Vec<u8>` zurück, um Zeichen zwischenzuspeichern. Ein `String` besteht aus `Copy`-fähigen Teilen wie der Länge oder Kapazität. Würde jedoch eine bitweise Kopie des Zeigers – der dritten Komponente des Typs – erstellt, könnte dies zu Speicherfehlern führen. In Kapitel 5, »Strings«, werde ich das Thema weiterführen.

Wenn zwei lokale `String`-Variablen den gleichen Puffer besitzen, wer würde ihn freigeben? Listing 4.7 zeigt ein Beispiel, wie Rust auf diese Situation reagiert:

```
let str = String::from("Hallo");
let str_2 = str;

println!("str_2: {str_2}"); // Hallo

// Fehler! Das kompiliert so nicht!
println!("str: {str}");
```

Listing 4.7 Der String ist schon einem neuen Eigentümer zugewiesen.

Listing 4.7 initialisiert wie Listing 4.6 einen `String`. Danach kommt es zu einer Zuweisung von `str` an `str_2`. Eine bitweise Kopie kommt nicht Betracht, weshalb der Com-

piler Move anwendet. Die Quelle der Zuweisung (`str`) gibt ihre Bindung und das Eigentum an das Ziel `str_2` ab.

Die erste Ausgabe mit `println!` kompiliert zwar fehlerfrei, doch beim zweiten Aufruf mit `str` schlägt Ihnen ein Fehler entgegen. Die Ausgabe sieht so aus:

```
error[E0382]: borrow of moved value: `str`
  --> src\main.rs:43:25
   |
37 |         let str = String::from("Hallo");
   |             --- move occurs because `str` has type `String`, which does
not implement the `Copy` trait
38 |         let str_2 = str;
   |                     --- value moved here
...
43 |         println!("str: {str}");
   |                         ^^^ value borrowed here after move
   |
   = note: this error originates in the macro `$crate::format_args_nl` (in
Nightly builds, run with -Z macro-backtrace for more info)
```

Die Fehlermeldung des Compilers berichtet von einem *Borrow-after-Move*. Was es mit Ausleihen (engl. *borrow*) auf sich hat, besprechen wir in Abschnitt 4.3, »Referenzen und der leihweise Zugriff«. Die Details sind an dieser Stelle für das Verständnis von Move nicht entscheidend.

Die ersten beiden Meldungen sind dagegen von größerem Interesse: Zunächst fasst der Compiler zusammen, dass `String` die Eigenschaft `Copy` nicht implementiert und es daher zu einem Move kam. Da `str` ab jetzt im aktuellen Kontext ungültig ist – vergleichbar einer Null-Referenz –, dürfen Sie die Variable jetzt nicht mehr verwenden. Wenn Sie die letzte Anweisung auskommentieren, schließt der Compiler den Durchlauf erfolgreich ab.

Wenn Sie nicht »Copy« verwenden können, dann vielleicht »Clone«?

Wenn es darum geht, einen Wert zu duplizieren, ist die bitweise Kopie sehr praktisch. Doch nicht jeder Typ ist `Copy`. Alternativ steht Ihnen das Trait `std::clone::Clone` zur Verfügung, das ein Datentyp alternativ implementieren kann.

`String` implementiert `Clone` und liefert Ihnen nach dem Aufruf ein Duplikat zurück, das einen eigenen Puffer im Heap verwaltet. Gerade beim Einstieg in Rust kann `Clone` sehr praktisch sein, da Sie statt mit ungewohnten Referenzlebenszeiten mit verschiebbaren und duplizierten Daten arbeiten.

4.3 Referenzen und der leihweise Zugriff

Der vorherige Abschnitt thematisierte den Begriff Eigentum und stellte heraus, wie der Compiler `Copy` und Move einsetzt. Wie Sie gesehen haben, haben `Copy` und Move allerdings auch Nachteile: etwa dann, wenn Sie auf dem Original und nicht auf einer bitweisen Kopie arbeiten wollen, wenn der Wert nicht den Eigentümer wechseln soll oder wenn Sie von mehreren Stellen lesend auf den Wert zugreifen möchten. Rust bietet Ihnen deshalb mit `&T` eine *geteilte*, nur lesende sowie mit `&mut T` eine *exklusive*, schreibfähige Referenz. Beide räumen Ihnen leihweise ein Benutzungsrecht ein, das bindet Sie jedoch an ein strenges Regelwerk.

In den folgenden Abschnitten betrachten wir zunächst das Zusammenspiel von Adressen, Zeigern und Referenzen. Dann besprechen wir die geteilten und schreibfähigen Referenztypen, deren Lebenszeiten und welche Regeln Sie bei ihrer Verwendung einhalten müssen.

4.3.1 Adressen, Zeiger und Referenzen: ein Überblick

Der Speicher und andere Ressourcen eines Computers werden mit numerischen Werten *adressiert*. Die Zuweisung einer Zahl an eine Ressource ist genauso wenig magisch wie die Nummerierung von Häusern in einer Straße: Jemand verteilt aufsteigende und eindeutige Werte an Ressourcen (die Häuser), sodass der Zusteller (Speichermanager) seine Ware abliefern, aber auch ein Paket abholen kann.

Ein *Zeiger* (engl. *Pointer*) zeigt – daher der Name – auf eine Adresse und ermöglicht damit *Indirektion*: Sie können *Adressen*, die auf Werte oder Ressourcen zeigen, im Programm speichern und weitergegeben. Zeiger müssen den Typ (`T`) kennen, auf den sie verweisen. Nur so kann der Compiler verstehen, wie viele Bytes er an der angegebenen Speicherstelle zum Lesen oder Schreiben einbeziehen muss. Wichtig ist `T` auch für die Dereferenzierung eines Zeigers. Sie dereferenzieren einen Zeiger mit dem Dereferenzierungsoperator *. (Dazu lesen Sie mehr in Abschnitt 19.2, »Primitive Zeiger«.)

Während der Typ `T` gleich bleibt, kann sich das eigentliche Speicherobjekt hinter dem Zeiger verändern. Es könnte sogar gar nicht (mehr) existieren. Oder der Zeiger wird auf den Zustand »Ich zeige auf kein Objekt« geparkt: ein Null-Zeiger! Ein Beispiel: Sie erstellen einen veränderlichen Zeiger (in Rust: `*mut T`), der zunächst nicht auf ein Speicherobjekt verweist und daher die Adresse 0 erhält). Dann weisen Sie diesem Zeiger die Adresse einer Instanz von T zu, die im Heap liegt. Danach ändern Sie das Ziel des Zeigers erneut. Nun soll der Zeiger kurzzeitig auf einem Speicherobjekt liegen, das auf eine automatische Variable auf dem Stack verweist. Jede Zuweisung einer Adresse zu diesem Zeiger kann nur auf ein Speicherobjekt verweisen, das ein T repräsentiert.

In Rust treffen Sie eher selten auf die primitiven Zeiger-Typen `*const T` (lesend) und `*mut T` (lesend und schreibend). Schon die *Standard C++ Foundation* empfiehlt, Zeiger

nur dann zu nutzen, wenn es nicht anders geht. So hält es auch Rust, das Ihnen die Dereferenzierung eines Zeigers nur in `unsafe`-Blöcken erlaubt (siehe Abschnitt 19.1, »Unsafe Rust«).

Während es sich beim Zeiger um eine eigene Datenstruktur handelt, stellt eine *Referenz* hingegen nur einen *Alias* dar: Das heißt, der durch die Adresse referenzierte Speicherbereich (ein Speicherobjekt) wird von einem oder mehreren verschiedenen Bezeichnern angesprochen – etwa durch Variablen. Wenn Sie einem Ausdruck (etwa Literalen, Variablen, Funktionen) den `&`-Operator voranstellen, erzeugen Sie eine Referenz auf das Speicherobjekt, das auf dieselbe Adresse verweist. Listing 4.8 zeigt dazu ein Beispiel, das eine Referenz anfordert. Wir lesen die dahinterliegende Adresse anschließend über die Referenz aus:

```
let zahl: i32 = 6;
// Die Referenz zeigt auf die Adresse von "zahl"
let referenz_auf_zahl: &i32 = &zahl;

// Referenzen sind Copy
let weitere_referenz: &i32 = referenz_auf_zahl;

// Die Referenz wie das Ziel verwenden
let summe: i32 = zahl + referenz_auf_zahl + weitere_referenz;
// Summe: 18

// Die Speicheradressen sind identisch
println!(
    "Variable: {:p}, Referenz 1: {:p}, Referenz 2: {:p}",
    &zahl, referenz_auf_zahl, weitere_referenz
);
// Alle geben die gleiche Adresse aus, z. B. 0xe4d7b9f85c
```

Listing 4.8 Erste Referenzen

Mit dem `&`-Operator fordern Sie von einem Ausdruck eine Referenz an. Referenzen sind `Copy` (vgl. Abschnitt 4.2.4, »Bitweise Kopien mit »Copy« erzeugen«): Wenn Sie eine Referenz zuweisen, erhält das Ziel eine bitweise Kopie! Referenzen sind mit dem Ziel fest verdrahtet und dürfen ganz im Gegensatz zu einem Zeiger nicht »leer« oder »null« sein.

Listing 4.8 setzt im `println!`-Makro den Formatierer `:p` ein. Mit einem Formatierer können Sie Ausgaben eines Werts verändern, hier etwa, um die Adresse einer Referenz auszulesen und auszugeben. Wir besprechen die Formatierer und Ausgabeparameter in den geschweiften Klammern in Abschnitt 5.3, »Wie Sie Strings formatieren«.

Eine Gefahr im Umgang mit Referenzen oder Zeigern ist der *Dangling Pointer*, also wenn eine Referenz oder der Zeiger auf eine Adresse zeigt, an der sich aber nicht mehr der einst gültige Wert befindet. Gründe dafür könnten abgeräumte Stack-Variablen oder der freigegebene Speicher im Heap sein. Rust begegnet diesem Problem, indem es alle Referenzen vom Anfang bis zum Ende ihrer Lebenszeit verfolgt.

Die Lebenszeit einer Referenz gleicht der Benutzung

Das Ende der *Lebenszeit* fällt nicht unbedingt mit dem Ende eines Gültigkeitsbereichs zusammen, wie Sie später noch sehen werden. Achten Sie vielmehr darauf, ob Sie nachfolgend über eine Referenz lesen oder den Wert neu zuweisen.

Falls Sie die Referenz nicht mehr benutzen, hat der Compiler das automatisch so vermerkt. Sie kann daher nicht mehr mit anderen Referenzen interferieren. Von der Benutzung bleibt allerdings die Tatsache unberührt, dass eine Referenz niemals länger als ihr Ziel leben kann!

Für die Prüfung der Regeln ist der *Borrow-Checker* des Compilers zuständig. Wenn Sie eine Referenz verwenden, die laut Analyse des Borrow-Checkers bereits das Ende ihrer Lebenszeit erreicht hat, meldet der Compiler einen Fehler. Rust garantiert: Ungültige Referenzen werden Ihnen niemals begegnen.

Breite Zeiger

Im späteren Verlauf begegnen wir dem Begriff *breiter Zeiger* (engl. *Fat Pointer* oder *Wide Pointer*). Ein solcher Zeiger besteht aus einer Adresse plus einer zusätzlichen Information. Beispiele hierfür sind etwa *Slices* (siehe Abschnitt 5.1, »Der String-Slice«) oder Trait-Objekte (siehe Abschnitt 11.5, »Trait-Objekte«). Ein breiter Zeiger nimmt im Speicher zwei `usize`s ein und ist damit doppelt so groß wie ein primitiver Zeiger (ein `usize`).

Rust definiert für den sicheren Umgang mit Referenzen strenge Regeln. Das Regelwerk versetzt den Compiler in die Lage, viele typische (und tückische!) Probleme mit Referenzen abzufangen, noch bevor sie für unangenehme Folgen zur Laufzeit sorgen können. Die nächsten beiden Abschnitte machen Sie mit geteilten und veränderlichen Referenzen und den dazugehörigen Regeln vertraut.

4.3.2 Zugriff auf Werte über geteilte Referenzen

Die geteilte Referenz `&T` (`&` ist der Ausleih-Operator oder Referenz-Operator (engl. *Shared Borrow*) und `T` der Typ des referenzierten Werts) dürfen Sie ohne Einschränkungen verteilen, da es jedem Empfänger nur gestattet ist, den Wert zu lesen.

Alle Referenzen unterliegen jedoch einigen (Leih-)Regeln, die der sogenannte Borrow-Checker des Compilers prüft. Besonders wichtig: Die Referenz darf nicht länger

gültig sein, als der Eigentümer der Ressource lebt – die Adresse wäre sonst womöglich schon einem anderen Element zugeordnet. Dazu ein Beispiel:

```
fn main() {
    let a = 4;
    let b = 6;

    let ergebnis =
        {
            let sum = a + b;
            // Die Referenz auf "sum" zurückgeben
            &sum // Fehler
        };
    println!("Das Ergebnis ist: {ergebnis}");
}
```

Listing 4.9 Ein Fehler! Ungültige Referenz auf »sum«

Die Summe aus den Variablen `a` und `b` liegt in der lokalen Variable `sum`, die nur innerhalb des umgebenden Blocks gültig ist. Diese Summe gibt der Block am Ende als Referenz zurück. Damit landen wir schon beim Problem der unterschiedlichen Lebenszeiten: Verlässt der Programmablauf den Block, werden die darin eingeführten Variablen freigegeben und damit auch die dahinterliegenden Adressen.

Die Referenz von `sum` – angefordert durch `&sum` – verweist nicht mehr auf einen gültigen Wert! Diesen Fehler erkennt Rust dank des Borrow-Checkers schon beim Kompilieren. Das vermeidet ungültige Referenzen und damit ungültige Zeiger, die aus diesen Referenzen gewonnen würden (wie die auf `sum` im aktuellen Beispiel). Der Code aus Listing 4.9 hat daher zu einer Fehlermeldung durch den Compiler geführt:

```
error[E0597]: `sum` does not live long enough
  --> src\main.rs:16:13
   |
12 |     let ergebnis =
   |         -------- borrow later stored here
...
16 |             &sum
   |             ^^^^ borrowed value does not live long enough
17 |         };
   |         - `sum` dropped here while still borrowed
```

Nun lässt sich das Problem dadurch lösen, dass eine *Kopie* von `sum` statt der *Referenz* darauf zurückgegeben wird. Das führt bei kleineren Strukturen wie einem Integer nur in den seltensten Fällen zu Leistungseinbußen. Doch bei größeren Strukturen oder

einem sehr umfangreichen `String` (über `Clone`) sähe das schon anders aus! In Abschnitt 4.4 zu `Box<T>` stelle ich Ihnen eine Lösung vor, die zwar eine Kopie erstellt, den kostenintensiven Speicherbereich jedoch an Ort und Stelle belässt.

Eine Referenz dereferenzieren

Weiter oben haben wir die Begriffe *Adresse*, *Zeiger* und *Referenz* eingeordnet. Wir halten fest: Eine Referenz ist ein Alias, also ein anderer oder weiterer Bezeichner für eine Ressource an einer Speicheradresse. Die Adresse und der Zeiger sind Bestandteile einer Referenz. Um den Wert hinter einer Adresse auszulesen, müssen Sie einen Zeiger oder eine Referenz *dereferenzieren*.

Wie C und C++ nutzt Rust dafür den *-Operator. In den bisherigen Beispielen habe ich das *-Zeichen an keiner Stelle verwendet. Dennoch konnten wir stets den Wert hinter einer Referenz benutzen. Das kommt daher, dass der Compiler nach Möglichkeit implizit dereferenziert. Listing 4.10 zeigt dazu ein Beispiel:

```
let zahl = 12;
let referenz = &zahl;
println!("Die Adresse: {:p}", *referenz); // Fehler!

println!("Die Adresse von Referenz: {:p}", referenz);
// Richtig -> 0x93cdbdf39c
println!("Der Wert hinter 'referenz' ist: {}", *referenz);
// Richtig -> 12
println!("(Auto) Der Wert hinter 'referenz' ist: {}", referenz);
// Richtig -> 12
```

Listing 4.10 Eine Referenz dereferenzieren

Der Code initialisiert die Variable `zahl` mit dem Wert 12. Mit dem `&`-Operator erstellen wir darauf den Alias `referenz`. In der dritten Zeile setzen wir den Dereferenzierungsoperator `*` ein, das führt jedoch zu einem Fehler. Der Formatierer `{:p}` erwartet einen Zeiger, bekommt durch die Dereferenzierung mit `*referenz` aber den Wert, auf den der Zeiger verweist.

Der Compiler kann Referenzen selbstständig in Zeiger beugen, das nutzt `:p` aus. Bevor der Compiler jedoch im Aufruf von `println!` die Referenz in einen Zeiger beugen konnte, haben wir durch die Dereferenzierung eine implizite Beugung mit Zugriff auf den Wert vorweggenommen. Die Folge: Statt `&i32` erhält `println!` nun ein `i32`. Hier ist die Fehlermeldung des Compilers:

```
error[E0277]: the trait bound `{integer}: Pointer` is not satisfied
```

Die folgende Zeile ist wiederum korrekt, weil der Compiler die Adresse über die Referenz ermittelt. Dann kommt es zu einer gültigen Dereferenzierung: Der Formatierer erhält mit `*referenz` einen erwarteten Datentyp. Wie der Compiler das genau macht, untersuchen wir unten. Zum Schluss zeigt das Beispiel, wie der Compiler die Referenz automatisch dereferenziert.

Wann Sie »*« auf einem Zeiger anwenden

Der Zeiger besitzt einen eigenen Typ und damit unterscheidbare Eigenschaften und Funktionen. Den Zeigertyp ordnen Sie daher so ein wie die Typen `i32` oder `bool`. Ohne den Dereferenzierungsoperator `*` verwenden Sie den Zeiger selbst oder reichen ihn herum. Sie greifen nicht auf den Wert im Speicher zu, auf den er zeigt. Das heißt, der direkte Wert dieses Zeigers ist die Adresse, auf die er verweist.

Erst wenn Sie den Zeiger dereferenzieren, folgen Sie ihm zu der Speicheradresse und lesen den dahinterliegenden Wert aus. Sie können einen Zeiger daher mit der Nummer eines Bankschließfachs vergleichen: Sie wissen, dass die hinterlegte Nummer ein Schließfach identifiziert, das an einem bestimmten Ort ist. Was darin abgelegt ist, finden Sie aber erst heraus, wenn Sie mit der Nummer zur Bank gehen, sich zum tatsächlichen Kästchen bringen lassen und es öffnen. Nichts anderes übernimmt das Dereferenzieren eines Zeigers für Sie!

Der Compiler kann Referenzen beugen

Greifen wir den Weg von einer Referenz zu einem Zeiger noch einmal auf, womit wir zu einem weiteren Service von Rust kommen: Wenn der Compiler erkennt, dass ein Typ `T` in einen Zieltyp *gebeugt* (engl. *coerced*) werden kann, führt er diesen Schritt automatisch aus. Praktischerweise ist eine solche Beugung vom Typ `&T` (Referenz) auf `*const T` (Zeiger) definiert.

Der Compiler erleichtert Ihnen hierdurch massiv die Arbeit am Code. Er kann aber im Fall mehrerer plausibler Wege nicht immer den richtigen Weg erkennen, etwa bei einer Vergleichsoperation:

```
let zahl = 12;
println!("Sind gleich: {}", &12 == &zahl); // true
```

Der Code funktioniert prima – wenn Sie vorhatten, die Werte hinter den beiden Referenzen `&12` und `&zahl` zu vergleichen. Falls Sie jedoch wissen wollten, ob die beiden Referenzen auf die gleiche Adresse verweisen, hat sich hier ein Fehler eingeschlichen! Der Compiler hat beide Referenzen dereferenziert, um an die Werte dahinter zu gelangen. Wenn Sie Adressen miteinander vergleichen möchten, müssen Sie aber stattdessen explizit aus den Referenzen jeweils Zeiger gewinnen. Das Listing 4.11 zeigt ein Beispiel:

```
let ptr_1 = &12 as *const i32;

let zahl = 12;
let ptr_2 = &zahl as *const i32;
println!(
    "Zeiger ptr_1 ({:p}) und ptr_2 ({:p}) sind nicht gleich: {}",
    ptr_1,
    ptr_2,
    ptr_1 != ptr_2
);
// Beispielausgabe:
// Zeiger ptr_1 (0x7ff77af51458) und
// ptr_2 (0x1cc60ff20c) sind gleich: true
```

Listing 4.11 Referenzen in Zeiger umwandeln

Im Unterschied zu vorher lösen wir für die beiden Ausdrücke `&12` und `&zahl` die dahinterliegenden Zeiger auf und speichern die Adressen in den Variablen `ptr_1` und `ptr_2`. An dieser Stelle machen wir uns die Beugung von `&T` zu `*const T` zunutze. Wenn Sie den Vergleich jetzt erneut durchführen, erhalten Sie das erwartete Resultat: Es handelt sich um unterschiedliche Adressen im Speicher, nämlich um ein Literal (statisch) und um eine lokale Variable (Stack).

Der Compiler hätte Ihnen tatsächlich weiterhelfen können, um zwei Adressen zu vergleichen, ohne dass Sie die davor gelagerten Referenzen zu Zeigern beugen. Der Compiler denkt aber immer nur so weit voraus, wie er muss: Im ersten Versuch fand er heraus, dass er den auf `i32` implementierten Vergleich erreicht, indem er die zwei `&i32` Referenzen dereferenziert. Deswegen sprang er zu diesem Ziel.

Wenn Sie dem Compiler hingegen etwas zum Vergleichen geben, das zwei Zeiger als Argument erwartet, so wird er den von Ihnen erhofften Zeiger-Pfad verfolgen. In der Standardbibliothek findet sich etwa im Zeiger-Modul `std::ptr` die Funktion `eq`, die zwei Zeiger-Argumente erwartet:

```
println!(
    "Haben gleiche Adressen: {}",
    std::ptr::eq(&12, &zahl)
); // false
```

Listing 4.12 Dieses Mal vergleichen wir zwei Zeiger.

Jetzt weiß der Compiler, wo Sie hinwollen, beugt die Referenzen zu Zeigern und liefert das erwartete Ergebnis: Die Zeiger sind ungleich. Damit sparen Sie sich den Aufwand, die Referenzen selbst umzuwandeln.

4.3.3 Veränderliche Referenzen: Der exklusive Zugriff

Wenn Sie eine geteilte Referenz besitzen, können Sie über sie weder einen neuen Wert zuweisen noch aus ihr eine schreibfähige Referenz machen. Um den Wert hinter der Referenz zu überschreiben, müssen Sie immer an den Eigentümer herantreten: Entweder, Sie schreiben direkt über den Eigentümer oder beziehen von ihm eine exklusive schreibfähige Referenz, genannt `&mut T`.

In Abschnitt 3.2, »Variablen«, haben Sie das Schlüsselwort `mut` kennengelernt. Wenn Sie eine Variable als `mut` deklarieren, können Sie auch nach der Initialisierung Werte zuweisen. Analog dazu modifizieren Sie eine Referenz, um Neuzuweisungen vorzunehmen: Statt `&T` schreiben wir dann `&mut T`. Das folgende Beispiel verdeutlicht dies:

```
let mut zahl = 5;
let referenz = &mut zahl;
*referenz = 3;
zahl = 2;
```

Listing 4.13 Zuweisung über eine veränderliche Referenz

Veränderliche Referenzen erhalten Sie nur von veränderlichen Variablen. Daher führt das Beispiel die Variable `zahl` mit `let mut` ein. Die mit `&mut` erfragte Referenz weisen wir der anderen lokalen Variable `referenz` zu.

Warum die Referenz hier nicht veränderlich ist

Die Variable `referenz` wird ohne das Schlüsselwort `mut` eingeführt, obwohl es einen Wert vom Typ `&mut T` erhält. Das kann verwirren, ist aber schnell geklärt.

Der Modifikator `mut` würde sich nur auf `referenz` selbst beziehen. So deklariert, könnten Sie später erneut eine `&mut i32`-Referenz zuweisen. Im Ausdruck `let mut referenz = &mut zahl;` wirken daher zwei unabhängige `mut`-Schalter: `let mut referenz` lässt Sie nach der Definition weitere veränderliche Referenzen zuweisen, und `&mut zahl` fordert eine veränderliche Referenz vom Eigentümer `zahl`.

Den Wert hinter einer veränderlichen Referenz weisen Sie mit dem Dereferenzierungsoperator * neu zu. Der Code im Beispiel ändert damit den Wert von `zahl` auf 3. Zuletzt sehen wir, dass Sie den Wert von `zahl` ebenfalls über den Eigentümer selbst neu zuweisen können. Listing 4.13 zeigt gültigen Code.

Veränderliche Referenzen sind exklusiv! Es kann immer nur eine einzige veränderliche Referenz `&mut T` von ein und demselben Eigentümer geben. Genauso darf ein Eigentümer nicht schreiben, noch während er eine veränderliche Referenz ausgegeben hat. Der folgende Code ist nicht gültig:

```
let mut zahl = 5;
let referenz = &mut zahl;
let weitere_referenz = &mut zahl;
// Fehler! "referenz" und "weitere_referenz" gleichzeitig
*referenz = 2;
*weitere_referenz = 4;
```

Listing 4.14 Er ist nur eine veränderliche Referenz erlaubt.

Exklusivität bedeutet also, dass Sie einen Wert *entweder* über den Eigentümer *oder* über eine ausgeliehene veränderliche Referenz überschreiben, aber niemals über beide gleichzeitig!

Wenn Sie jetzt noch einmal auf Listing 4.13 blicken, stellen Sie allerdings fest, dass dort eine veränderliche Referenz ausgeliehen wurde (`referenz`) und dennoch am Ende in `zahl` selbst geschrieben wird. Ist das korrekt? Ja! Die Referenz `referenz` wird nicht mehr benutzt, wenn über den Eigentümer `zahl` geschrieben wird. Der Compiler ist schlau genug, um das zu erkennen. Die Referenz ist nach der Zuweisung am Ende ihrer Lebenszeit angekommen.

Achten Sie auf ausgeliehene Referenzen

Oft werden Lebenszeiten mit Gültigkeitsbereichen assoziiert. Wenn eine Referenz in einem Block oder einer Funktion eingeführt wird, fällt das Ende ihrer Lebenszeit mit dem Ende des umgebenden Blocks zusammen. Tatsächlich ist der Compiler aber viel genauer und untersucht auch innerhalb eines Blocks den Zugriff auf eine Referenz. Für Listing 4.14 gibt es daher eine gültige Konstellation mit zwei erzeugten Referenzen: Wenn Sie die Zeile `*referenz = 2` auskommentieren, läuft der Compiler durch! Er erkennt, dass Sie `referenz` nicht schreiben oder lesen, sodass die Referenz sofort nach der Zuweisung das Ende ihrer Lebenszeit erreicht – somit gibt es nur eine veränderliche Referenz.

Ein eben noch funktionierender Rust-Code kann dementsprechend schnell ungültig werden. Achten Sie daher genau auf die Meldungen des Compilers! Wenn Sie etwa Listing 4.13 nur minimal anpassen, würden die folgenden beiden Beispiele die Prüfung nicht bestehen. Die Lebenszeiten der schreibenden Referenz und des Eigentümers überschneiden sich jetzt:

```
{
    // Fehler Szenario 1
    let mut zahl = 5;
    let referenz = &mut zahl;
```

```
    // OK
    zahl = 2;

    // Fehler! Hierdurch lebt "referenz" noch
    *referenz = 3;
}
// ...
{
    // Fehler Szenario 2
    let mut zahl = 5;
    let referenz = &mut zahl;

    // OK
    *referenz = 3;

    // OK
    zahl = 2;

    // Fehler! Hierdurch lebt "referenz" noch
    *referenz = 4;
}
```

Listing 4.15 Fehler: Eine veränderliche Referenz ist nicht exklusiv.

In den Fehlerszenarien sind die beiden Anweisungen markiert, die jeweils einen Fehler verursachen. Der Compiler erkennt anhand der Verwendung von `referenz`, dass die Lebenszeit nicht endet und sich stattdessen über die Schreibanweisung `zahl = 2` ausdehnt. Somit hat `zahl` noch eine schreibfähige Referenz ausgeliehen, während die Variable den Wert selbst überschreiben soll. Eine klare Verletzung der Regel! Eine Fehlermeldung ist die Folge:

```
error[E0506]: cannot assign to `zahl` because it is borrowed
   --> src\main.rs:112:17
    |
110 |                 let referenz = &mut zahl;
    |                               --------- borrow of `zahl` occurs here
111 |
112 |                 zahl = 2;
    |                 ^^^^^^^^ assignment to borrowed `zahl` occurs here
113 |                 // Fehler! Hierdurch lebt "referenz" noch
114 |                 *referenz = 3;
    |                 ------------- borrow later used here
```

Behalten Sie im Gedächtnis, dass der Borrow-Checker des Compilers auch innerhalb von Blöcken auf die Exklusivität von Referenzen achtet. Exklusivität kann durch Änderungen verloren gehen, die zunächst nichts mit der betroffenen Stelle zu tun zu haben scheinen. Listing 4.16 zeigt ein weiteres Beispiel:

```
let mut zahlen = vec![3, 5, 8];
// Index 1 verweist auf die Zahl 5
let letzte_zahl = &zahlen[1];
// Fehler, ein (unscheinbarer) möglicher Dangling Pointer
zahlen.pop();
// Referenz letzte_zahl lebt bis hier
println!("{letzte_zahl}");
```

Listing 4.16 Ein möglicher Dangling Pointer

Wir schreiben die Zahlen 3, 5 und 8 in einen Vektor. Ein Vektor ist eine *Collection* – oft auch *Container* genannt. Rust-Container oder -Collections haben aber nichts mit Container-Software wie Docker zu tun! Ein Vektor legt Elemente hintereinander und zusammenhängend im Speicher ab. Vektoren und ähnliche Strukturen sehen wir uns in Kapitel 6, »Collections«, genauer an.

Die Variable `letzte_zahl` erhält die Referenz auf die zweite Zahl im Vektor. Der Index einer Collection beginnt immer bei 0, sodass der Ausdruck `&zahlen[1]` die Zahl 5 referenziert. Die nächste Zeile führt die Funktion `pop` auf dem Vektor aus, die die Zahl 8 aus der Sammlung entfernt.

Bis zu diesem Punkt ist der Rust-Code gültig. Die nächste Zeile führt jedoch zu einem Fehler! Die Referenz `letzte_zahl`, die wir anschließend auslesen, ist zwar nicht veränderlich und damit ist der Fehler etwas anders gelagert. Das Problem ist allerdings das gleiche wie in den vorherigen Fehlerszenarien: Die Lebenszeiten zweier miteinander inkompatibler Referenzen überschneiden sich. Durch `letzte_zahl` ist bis zum Ende des Listings eine geteilte Referenz aktiv. Wenn wir die Funktion `pop` aufrufen, wird dadurch allerdings implizit eine weitere, veränderliche Referenz aktiv!

Rust schützt Sie in diesem Fall konkret davor, eine möglicherweise ungültige Referenz auf einen Speicherbereich im Vektor zu benutzen, weil der Vektor durch das Einfügen oder Herausnehmen von Elementen wachsen oder schrumpfen könnte. Mehr zum Vektor `Vec<T>` erfahren Sie in Abschnitt 6.4, »Vektoren«. In diesen Situationen wäre nicht auszuschließen, dass der Speichermanager den im Heap angelegten Speicher eines Vektors verlagert oder neu alloziert.

Im Speicher gibt es jedoch keinen Nachsendeauftrag! Ein Zeiger bleibt stur auf der Adresse, die ihm zugewiesen wurde. Ihre Referenz würde somit möglicherweise nicht mehr auf das gewünschte Objekt zeigen, und das kann brandgefährlich werden!

Typen geben ihren Funktionen implizit Referenzen mit

In Kapitel 10, »Strukturen«, werden wir Rusts `struct` und die impliziten Referenzen im Detail besprechen. So viel allerdings schon vorweg: Alle nicht statischen Funktionen eines Typs erhalten eine Referenz auf die Instanz. Das ist `&T` bei lesendem Zugriff und `&mut T`, wenn die Funktion (als Teil einer Struktur wird sie *Methode* genannt) Veränderungen an der Instanz vornehmen soll.

Am Beispiel von `pop` begegnet uns eine Funktion, die ihre Instanz als veränderliche Referenz `&mut self` referenziert. Das bedeutet, dass `pop` die Instanz, auf der Sie sie ausführen, verändern darf. Die Funktion `pop` leiht sich mit `&mut self` eine veränderliche Referenz auf sich selbst, um ein Element aus der eigenen Vektor-Instanz (`self`) zu entfernen.

Die veränderliche Referenz ist auch ein Zeiger

Dass eine Referenz intern als Zeiger arbeitet und dass wir eine Referenz zu einem Zeiger machen können, habe ich weiter oben in Abschnitt 4.3.2 zu den geteilten Referenzen erklärt. Auf die veränderlichen Referenzen trifft dies ebenso zu. Durch das Schlüsselwort `mut` können Sie veränderliche Referenzen auf veränderliche Werte über `*const T` hinaus auch in veränderliche Zeiger `*mut T` beugen. Ein Beispiel:

```
let mut zahl = 12;

let referenz = &mut zahl;
// Zeiger manuell aus Referenz bilden
let zeiger = referenz as *mut i32;

// ... oder automatisch über das std::ptr-Modul
let zeiger = std::ptr::addr_of_mut!(zahl);
println!("Vor dem Schreiben über Zeiger: {}", zahl); // 12

unsafe {
    *zeiger = 4;
    let referenz_aus_zeiger = &(*zeiger);
    println!("Nach dem Schreiben über Zeiger: {}", referenz_aus_zeiger); // 4
}
```

Listing 4.17 Von einer veränderlichen Referenz zum Zeiger

Wir starten erneut mit der Variablen `zahl`, die wir als veränderlich einführen. Dann stehen Ihnen zwei Optionen offen, um einen Zeiger zu gewinnen: Das Beispiel demonstriert zuerst die manuelle Option, bei der wir aus der Variable `zahl` eine Referenz gewinnen und diese dann zu einem Zeiger beugen. Die zweite, automatische Op-

tion ist der Rückgriff auf das Modul `std::ptr`, das wir auch in Listing 4.17 benutzt haben. Das darin definierte Makro `addr_of_mut!` erhält `zahl` als Argument und führt die Schritte durch, die Sie zuvor selbst ausgeführt haben. Das gestaltet den Code übersichtlicher.

Bevor wir eine Veränderung an `zahl` durchführen, gibt `println!` den momentanen Wert 12 aus. Weil der Compiler Sie bei der Arbeit mit Zeigern anders als beim Umgang mit Referenzen nicht mehr vor etwaigen Gefahren schützen kann, muss ein `unsafe`-Block alle unsicheren Anweisungen und Ausdrücke umgeben. `unsafe` ist Ihnen auch bei Neuzuweisungen von statischen Variablen in Abschnitt 3.2.5, »Statische Variablen«, begegnet. In Kapitel 19, »Unsafe Rust und das Foreign Function Interface«, werden wir `unsafe` in Rust weiter diskutieren.

Da `zeiger` auf die Adresse von `zahl` verweist, müssen Sie den Zeiger dereferenzieren. Erst dann kommen Sie an den Wert dahinter. Folglich sehen Sie in der Zuweisung des Werts 4 und in der nächsten Zeile, die aus dem primitiven Zeiger wieder eine Referenz bildet, jeweils den Dereferenzierungsoperator *.

Der Ausdruck `&(*zeiger)` erscheint auf den ersten Blick kompliziert. Wir besprechen ihn daher Stück für Stück. Der Teil `*zeiger` löst den Wert hinter der Zeigeradresse auf, weil wir auf den Wert hinter dem Zeiger zugreifen möchten. Weiter zu `&`: Um die Referenz eines Ausdrucks zu erhalten, stellen wir ihm den Referenz-Operator `&` voran. So auch in diesem Beispiel. Der ganze Ausdruck `&(*zeiger)` ergibt den Typ `&i32` – eine geteilte Referenz! Zuletzt bestätigt eine weitere Ausgabe, dass der Wert von `zahl` nun 4 beträgt.

Wenn Sie `zeiger` nicht dereferenziert hätten, sondern stattdessen `&zeiger` geschrieben hätten, würden Sie der Variable `referenz_aus_zeiger` die Adresse des Zeigers, nicht des Werts zuweisen.

Von einer veränderlichen zur geteilten Referenz

Sie können eine veränderliche Referenz `&mut T` und den Zeiger `*mut T` in geteilte Referenzen `&T` und den konstanten Zeiger `*const T` umwandeln. Der Compiler nimmt die Übersetzung an den erforderlichen Stellen sogar automatisch vor.

Wie immer steht es Ihnen offen, dies manuell zu tun. Von einer geteilten Referenz oder einem `*const T`-Zeiger kommen Sie allerdings nicht zu der jeweils schreibenden Variante. Ob eine Referenz oder ein Zeiger vorher schreibfähig war, ist dabei völlig irrelevant. Listing 4.18 zeigt ein Beispiel:

```
let mut zahl = 5;
let schreibend = &mut zahl;
let zeiger: *mut i32 = schreibend;
```

```
// OK: &mut i32 zu &i32
let nur_lesend: &i32 = schreibend;

// Fehler: &i32 zu &mut i32 ist nicht erlaubt
let wieder_schreibend: &mut i32 = nur_lesend;

// OK: *mut i32 zu *const i32
let konstanter_zeiger: *const i32 = zeiger;

// Fehler: *const i32 zu *mut i32 ist nicht erlaubt
let zeiger_wieder_schreibend: *mut i32 = konstanter_zeiger;
```

Listing 4.18 Die Wandlung geht nur in eine Richtung.

Zeiger werden oft eingesetzt, um dynamisch angelegten Speicher zu referenzieren. Bislang haben wir uns ausschließlich mit Variablen beschäftigt, die wir auf dem Stack oder im statischen Speicher angelegt haben. Den Heap haben wir dabei nur indirekt benutzt, wie im Fall des Vektors. Im nächsten Abschnitt lernen Sie mit `Box<T>` einen Datentyp kennen, der einen Wert aufnimmt und per Move in den Heap verschiebt.

4.4 Mit Box Objekte im Heap ablegen

Aus Abschnitt 4.1.2 wissen Sie bereits, dass dynamische Speicheranforderungen zur Laufzeit im Heap stattfinden. Im Gegensatz zum Stack können Sie Speicherpuffer oder -bereiche im Heap nach Bedarf vergrößern oder verkleinern.

Auch ohne den ungewissen Speicherbedarf kann es notwendig sein, Werte im Heap abzulegen. Etwa weil die Laufzeit den aktuellen Stackframe freigibt, wenn sie die Funktion verlässt, Sie aber Daten mit anderen Komponenten Ihrer Anwendung teilen möchten.

In Rust legen Sie mit dem Typ `Box<T>` – meist einfach *Box* genannt – Werte im Heap ab. In diesem Abschnitt erfahren Sie, wer der Eigentümer des Speichers ist und welche Möglichkeiten die Box Ihnen bietet.

4.4.1 Klare Eigentumsverhältnisse auch für die Box

Wenn Sie eine Box in Ihrer Funktion oder Struktur einführen, wird sie auf dem Stack angelegt. Die Box verwaltet einen Zeiger, der auf das Heap-Segment des Binärprogramms verweist. Listing 4.19 zeigt ein Beispiel:

```
let erster_wert = "Ungeöffnet";
let eine_box: Box<&str> = Box::new(erster_wert);
```

Listing 4.19 Eine Box mit »new« initialisieren

Sie initialisieren eine Box mit der Funktion `Box::new`. Das entspricht dem *new-Idiom* von Rust, da die Sprache keine »regulären« Konstruktoren definiert. Stattdessen bietet ein Datentyp per Konvention die Funktion `new` an. Im Beispiel erhält `Box::new` den String-Slice `"Der erste Wert"`. Die lokale Variable `erster_wert` bindet an die erstellte Box und ist fortan deren Eigentümer. Der Speicher, den die Box im Hintergrund anlegt, ist hingegen das unmittelbare Eigentum der Box selbst: Nur die Box darf den reservierten Speicher über den internen Zeiger freigeben.

Die Box nimmt den Wert, den sie über `new` erhalten hat, und bewegt ihn in den Heap. Abhängig vom Typ des Arguments setzt der Compiler `Copy` oder Move ein, um die Eigentumsverhältnisse zu wahren. Im Fall eines String-Slices `erster_wert` nutzt die Box `Copy`. Daher können Sie die Variable nach der Übergabe des Werts an die Box weiterverwenden. Das ist kein Problem, weil ein String-Slice nur als Referenz `&str` vorkommt, und Referenzen sind ja `Copy`! Die Box erhält damit eine bitweise Kopie der geteilten Referenz `erster_wert` und ist ein weiterer Alias für deren Ziel.

Ganz anders verhält es sich, wenn der Typ des Werts keine bitweise Kopie von sich erlaubt. Dann verschiebt der Compiler den Wert vom ursprünglichen Eigentümer in die Box. Nachfolgende Zugriffe führen dann zu einem Fehler, wie das Beispiel in Listing 4.20 vorführt:

```
let vektor = vec![1, 2, 3];
let vektor_in_box = Box::new(vektor);

// Fehler: vektor schon per Move verschoben!
println!("{}", vektor.len())
```

Listing 4.20 Der Compiler schiebt Werte bei Bedarf in die Box.

Gerade weil man es am Anfang so häufig außer Acht lässt, sage ich es noch mal: Denken Sie daran, dass `Copy` und Move in jede Zuweisung involviert sind und dass es schnell zu einer Fehlermeldung durch den Compiler kommen kann. Etwaige hervorgehobene Probleme sind aber schnell zu lösen.

Mit der Übernahme des Werts ist die Box immer, ganz gleich, ob Sie `Copy` oder Move verwenden, die Eigentümerin des im Heap allozierten Speichers und des darin abgelegten Werts.

Den lesenden oder schreibenden Zugriff auf eine Box steuern Sie wie bei anderen Variablen mit `let` oder `let mut`. Sie können etwa den Wert hinter `eine_box` im Heap ändern, wenn Sie die Variable mit `let mut` einführen:

```
let erster_wert = "Ungeöffnet";
let mut eine_box = Box::new(erster_wert);
*eine_box = "Geöffnet";
println!("{}", *eine_box);
```

Listing 4.21 Der Zugriff auf den Wert in einer Box

Weil die Box intern ohnehin mit einem Zeiger arbeitet und auch Ihnen einen zeigerähnlichen Umgang ermöglichen will, erfolgt die Zuweisung eines neuen Werts über den Dereferenzierungsoperator *. Dass die Box und der dahinterliegende Wert wirklich in verschiedenen Speichersegmenten angelegt wurden, können Sie ganz einfach nachprüfen. Dazu fügen Sie zu Listing 4.21 eine weitere Anweisung hinzu:

```
println!(
    "Box als lokale Variable: {:p} \
    und der Wert im Heap: {:p}",
    eine_box,
    *eine_box
);
// Box als lokale Variable:   0x6000015b0040
// und der Wert im Heap:    0x102362544
```

Listing 4.22 Box und Wert in verschiedenen Speichersegmenten

Die unterschiedlichen Speicheradressen lassen die Segmente erahnen

In Abschnitt 4.1 haben wir die verschiedenen Speicherbereiche und ihre Anordnung in einem Programm besprochen. Der Stack weist demnach die höchsten Adressen auf. Das Speichersegment des Heaps liegt unter dem Stack. Anhand der sehr unterschiedlichen Adresswerte sehen wir, dass die Box als lokale Variable auf dem Stack angelegt wird, während der übergebene Wert in den Heap verschoben wurde.

4.4.2 Auch Smart Pointer wie die Box können nicht alle Fehler verhindern

`Box<T>` ist der erste Vertreter der *Smart Pointer* von Rust, den Sie kennenlernen. Was ist ein Smart Pointer? Diese Datenstrukturen verwalten oder adressieren wie primitive Zeiger einen anderen Speicherbereich. Der Zusatz »Smart« rührt daher, dass man Ihnen die Speicherverwaltung abnimmt. Sie müssen daher weder Speicher allozieren und zuweisen noch ihn freigeben und den Zeiger invalidieren.

Eine Box wird Sie vor Speicherfehlern schützen. Das heißt allerdings nicht, dass Sie keinerlei Fehler oder Bugs einprogrammieren könnten. Um das zu demonstrieren, erweitern wir das Beispiel und fügen in Listing 4.23 absichtlich einen Bug ein:

```
let erster_wert = "Ungeöffnet";
let mut eine_box= Box::new(erster_wert);
*eine_box = "Geöffnet";
println!("{}", *eine_box); // Geöffnet

let referenz_in_box: &&str = eine_box.borrow();
let der_string_in_der_box: &str = *referenz_in_box;
*eine_box = "Geschlossen";

println!("{}", der_string_in_der_box);
// Geöffnet, Fehler: alter Text
println!("{}", *eine_box)
// Geschlossen, richtig
```

Listing 4.23 Ein Bug bei der Benutzung von »Box«

In der Mitte des Listings leihen wir eine geteilte Referenz auf den Wert der Box aus. An dieser Stelle habe ich den zurückgelieferten Typ explizit ausgeschrieben. Es handelt sich um die Referenz auf einen String-Slice: `&&str`. Abbildung 4.1 stellt die Verweise in den Speichersegmenten dar.

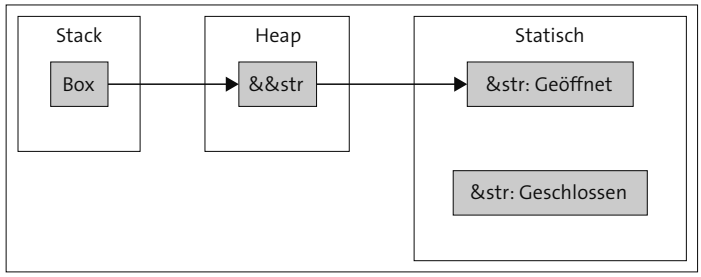


Abbildung 4.1 Die Box, ihr Speicher im Heap und die zwei statischen String-Slices

Mehrere Adress- oder Dereferenzierungsoperatoren hintereinander

Lassen Sie sich von dem doppelten Adressoperator `&` nicht abschrecken. Wenn der Adressoperator mehrmals hintereinander notiert wird, ist das ähnlich wie bei einer Schnitzeljagd: Nicht der Schatz liegt in der ersten Truhe, sondern nur der Hinweis auf die nächste Truhe – mit einem weiteren Hinweis – oder dem Schatz.

Im Fall von `&&str` handelt es sich daher um eine Adresse (erste Truhe), in der eine weitere Adresse liegt (nächste Truhe), an der sich letztlich der Wert (der Schatz) befindet. Ähnlich unspektakulär sind Konstellationen wie `**variable` oder `***variable` usw. Stellen Sie sich einfach vor, dass die Truhen ineinander verschachtelt sind: Sie öffnen mit `*` die äußerste Truhe und sehen dann die nächste ... und die nächste ..., bis Sie schließlich zum Schatz gelangen.

Indem wir `referenz_in_box` dereferenzieren, erhalten wir den String-Slice, den die Box zu diesem Zeitpunkt adressiert. Die Variable `der_string_in_der_box` bindet anschließend den Wert. Damit ändern sich die Verweise im Speicher so, wie in Abbildung 4.2 gezeigt.

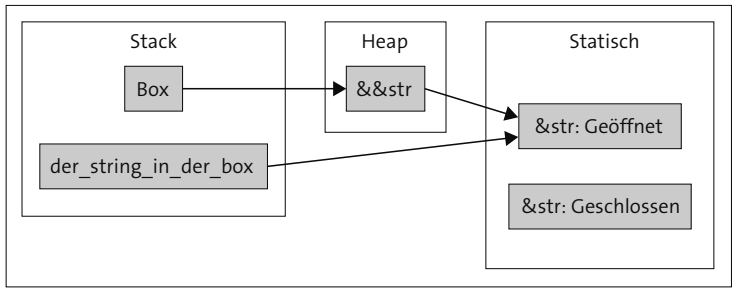


Abbildung 4.2 Die Variable »der_string_in_der_box« verweist nun ebenfalls auf den String-Slice »Geöffnet«.

Wenn Sie den Code kompilieren, meldet der Compiler keinen Fehler. Die Ausgaben zeigen jedoch, dass die Box und die lokale Variable auf unterschiedliche Speicherobjekte verweisen, denn `der_string_in_der_box` und `*eine_box` geben unterschiedliche Inhalte zurück!

Der Fehler hat sich eingeschlichen, als wir `der_string_in_der_box` als neue Variable eingeführt und ihr den Wert in `*referenz_in_box` zugewiesen haben. Das lieferte die Adresse des momentanen String-Slices, den die Box gleich danach schon nicht mehr adressiert. Damit stellte sich im Speicher anschließend die Situation ein, die Sie in Abbildung 4.3 sehen.

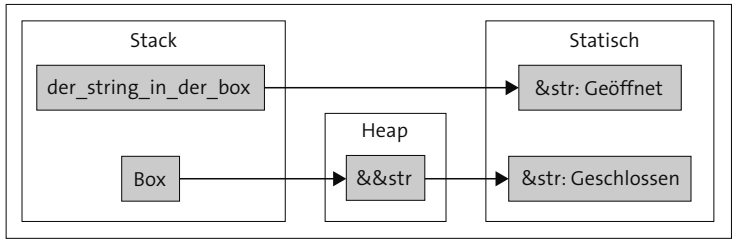


Abbildung 4.3 Die Variable »der_string_in_der_box« zeigt auf das vorherige Speicherobjekt, aber die Box ist weitergewandert.

Eine mit guter Absicht eingeführte Variable führte somit zu einem Bug! Wenn Sie die Zeile mit der Initialisierung von `der_string_in_der_box` entfernen und dafür direkt `*referenz_in_box` verwenden, kommt es nicht mehr zu dem Fehler. In Listing 4.24 sehen Sie den angepassten Code ohne die lokale Variable:

```
let erster_wert = "Ungeöffnet";
let mut eine_box= Box::new(erster_wert);
*eine_box = "Geöffnet";
println!("{}", *eine_box); // Geöffnet
let referenz_in_box: &&str = eine_box.borrow();

*eine_box = "Geschlossen"; // Vorsicht in dieser Zeile

println!("{}", *referenz_in_box); // Geöffnet, Fehler: alter Text
println!("{}", *eine_box) // Geschlossen, richtig!
```

Listing 4.24 Die Version ohne Bug, dafür aber mit Borrow-Checker-Fehler!

Wir lösen zwar den Bug, aber erhalten dafür eine Fehlermeldung. Dieser Fehler weist Sie darauf hin, dass Sie den Wert in der Box überschreiben möchten, noch während eine andere Referenz auf die Speicheradresse verweist. Die alte Referenz würde noch auf den Wert vor der Änderung verweisen, jeder weitere Zugriff auf die Box würde jedoch den neuen Wert ausgeben – inkonsistente Ausgaben wären die Folge.

Genau diese inkonsistenten Ausgaben erhielten wir auch durch den vorherigen Bug! Dort konnte der Borrow-Checker Sie nicht schützen, weil der Code gültig war. Wir haben nicht bedacht, dass durch die Dereferenzierung der Box (`&&str`) eine Kopie der Referenz `&str` erzeugt wurde. Damit haben Sie den Beweis: Auch in Rust-Code begegnen Ihnen nach wie vor Bugs, die mit der Speichernutzung in Zusammenhang stehen. Im Gegensatz zu C oder C++ sind in Rust nur logische, nicht aber kritische Speicherfehler möglich.

4.4.3 Nur die Box kann den Heap-Speicher freigeben

Die Box ist die Eigentümerin des Speichers, den sie im Heap anlegt, und sie ist gleichsam verantwortlich dafür, den Speicher freizugeben. Es macht dabei keinen Unterschied, ob Sie die Box an eine Variable oder an das Feld einer Struktur (Kapitel 10, »Strukturen«) zuweisen. Nur die Box selbst kann den Heap-Speicher zurückgeben, wenn sie das Ende ihrer Lebenszeit erreicht. Listing 4.25 zeigt ein Beispiel:

```
let box_a = Box::new('a');
let referenz_box_b :&char;

{
    let box_b = Box::new('b');
    println!("Box a: {} box b: {}", *box_a, *box_b);
    // Fehler, box_b lebt kürzer als referenz_box_b
    referenz_box_b = box_b.borrow();
```

```
    // <-- Ende der Lebenszeit von box_b
}

// Fehler: Lebenszeit von box_b zu kurz
println!("Box a: {} Box b: {}", *box_a, referenz_box_b);
```

Listing 4.25 Auch für die Box gelten die Regeln des Borrow-Checkers.

Das Beispiel zeigt die unterschiedlichen Lebenszeiten von `box_a` und `box_b` sowie der Referenz auf `box_b` auf. Der Code führt dazu zwei Box-Variablen ein. Eine weitere Variable nimmt die Referenz auf, die von `box_b` ausgeliehen wird.

Ein neuer Block definiert `box_b` und beschränkt neben der Sichtbarkeit auch die Lebenszeit des Werts und möglicher Referenzen auf ihn. Folgerichtig meldet der Borrow-Checker einen Fehler: Wenn der Block endet, wird `box_b` und damit der von der Box verwaltete Heap-Speicher freigegeben. Auch die Referenz, die wir kurz vor Schluss mit `box_b.borrow()` ausleihen, verliert dann ihre Gültigkeit.

Schon zuvor haben wir diskutiert, dass Sie die Referenz auf eine lokale Variable nicht zurückgeben können. Nehmen wir einmal an, dass `box_b` den Block überleben soll. Das können Sie lösen, indem Sie statt einer Referenz `box_b` als Rückgabewert des Blocks einsetzen. Dann wird die Box aus der block-lokalen Variable heraus und in das Ziel bewegt. Wenn Sie Listing 4.25 dementsprechend anpassen, erreichen Sie diesen Stand:

```
// ...
let box_b = {
    let box_b = Box::new('b');
    println!("Box a: {} Box b: {}", *box_a, *box_b);

    box_b
};
// Jetzt kann mit box_b gearbeitet werden
referenz_box_b = box_b.borrow();
// ...
```

Listing 4.26 Eine Box per Move zurückgeben

Der explizite Drop

Mit einem Block können Sie Variablen isolieren, aber auch implizit freigeben. Wenn das Ihr eigentliches Hauptanliegen ist, können Sie den Compiler stattdessen explizit anweisen, den Wert fallen zu lassen. Listing 4.27 zeigt ein Beispiel dazu anhand von `box_b`:

```
let mut box_b = Box::new('b');
// ...

// Fehler, drop darf nicht direkt aufgerufen werden
box_b.drop();

// richtig:
std::mem::drop(box_b);
```

Listing 4.27 Einen Wert explizit per »drop« freigeben

Zwar bietet `Box<T>` die Funktion `drop` an, ein direkter Aufruf ist Ihnen jedoch nicht gestattet. Sie müssen hingegen die Funktion `drop` aus dem Modul `std::mem` einsetzen, die das Argument – in unserem Fall die Box – als neuer Eigentümer entgegennimmt und mit dem Ende der Funktion fallen lässt.

Die Funktion `drop` kommt vom gleichnamigen Trait `Drop`, das nicht nur `Box<T>`, sondern alle Datenstrukturen implementieren (sollten), die Ressourcen verwalten. Wann immer Sie explizit Ressourcen freigeben oder die Freigabe eigener Datentypen beeinflussen müssen, implementieren Sie das Trait `Drop`. (Dazu folgt mehr in Abschnitt 11.8.3, »›drop‹ – der Rust-Destruktor«.)

4.4.4 Ein Praxisbeispiel

In Abschnitt 4.3.2 haben wir versucht, die Referenz auf eine Stack-Variable zurückzugeben. Das war allerdings nicht möglich, da deren Lebenszeit zu kurz gewesen ist. Oben haben wir die Box für einen ähnlichen Fall eingesetzt, weshalb wir noch einmal zum Problem zurückkehren. Dieses Mal aber mit einer Lösung! Listing 4.28 zeigt noch einmal den problematischen Code:

```
fn main() {
    let a = 4;
    let b = 6;

    let ergebnis =
        {
            let sum = a + b;
            // Die Referenz auf "sum" zurückgeben
            &sum // Fehler
        };
    println!("Das Ergebnis ist: {ergebnis}");
}
```

Listing 4.28 Ein erneuter Blick auf Abschnitt Listing 4.9

Einen `i32`-Wert könnten Sie einfach über `Copy` zurückgeben. Aber die beiden Mechanismen `Copy` und Move würden bei größeren Speichermengen keine guten Ergebnisse liefern.

Mit der Box könnten Sie einen großen Speicherbereich im Heap anlegen, den der Übergang von Block zu Funktion nicht bewegt. Das würde nur die Box, die auf dem Stack liegt. Wir ändern das Listing entsprechend um und erzeugen dieses Mal einen großen Datensatz. Der Code dazu sieht so aus:

```
fn main() {
    let ergebnis =
    {
        // Wir speichern 65535 (u16::MAX) Datensätze
        let mut daten = vec![
            // Befülle mit Wert
            "Ein Datensatz";
            // u16::MAX (65535) Mal
            usize::from(u16::MAX)
        ];

        let size = std::mem::size_of_val(&*daten);
        println!("Anzahl Bytes des Datensatzes: {size}");
        // Ausgabe: 1048560 auf einem 64-Bit-System

        Box::new(daten)
    };

    let size_of_box = std::mem::size_of_val(&ergebnis);
    println!("Anzahl Bytes der Box: {size_of_box}");
    // Ausgabe: 8

    println!("Anzahl der Datensätze: {}", ergebnis.len());
}
```

Listing 4.29 Die Lösung zum veränderten Abschnitt Listing 4.9

Innerhalb des Blocks ersetzt ein Vektor die Addition der Ganzzahlen und die Variable `sum`. Das `vec!`-Makro haben wir bisher nicht mit zwei Argumenten eingesetzt, das ist aber schnell erklärt: Das erste Argument definiert einen Wert, der mehrmals – und hier kommt das zweite Argument ins Spiel – eingesetzt wird. Der Code erstellt einen 65.535 Einträge langen Vektor, in dem jeder Wert eine Kopie der Adresse auf den String-Slice `"Ein Datensatz"` darstellt.

Wir greifen auf `std::mem:size_of_val` zurück, das wir schon früher eingesetzt haben, um die Byte-Größe hinter einer Referenz zu erfassen. Mit circa einer Million Byte belegen wir nennenswerten Speicher im Heap. Beachten Sie den Ausdruck `&*daten`, den wir `size_of_val` übergeben: Der Vektor ist der Box nicht unähnlich, da er ebenfalls einen Speicherbereich im Heap verwaltet. Er lässt sich ebenfalls wie ein Zeiger dereferenzieren.

Das bringt uns zum Wert im Heap, dem Slice über String-Slices, den wir mit `[&str]` notieren. In Abschnitt 6.5, »Slices«, gehe ich genauer auf den Slice `[T]` ein. Aktuell reicht es zu wissen, dass ein Slice eine Art Liste ist und zwei Informationen trägt: die Speicheradresse, an der das Slice beginnt, und die Zahl der Elemente im Slice. Der Datentyp `T` der Elemente – hier `&str` – muss immer zur Kompilierzeit feststehen.

Weil `size_of_val` eine Referenz erwartet, setzen wir jetzt vor den dereferenzierten Teilausdruck `*daten` den Adressoperator `&`. Den vollständigen Ausdruck führt `size_of_val(&*daten)` aus und liefert die Anzahl 1048560.

Warum die Nutzdaten genau 1.048.560 Bytes groß sind

Wenn Sie die Anzahl durch die Vektorgröße 65.535 teilen, erhalten Sie 16 – eine auffällige Zahl. Um zu klären, warum jeder Eintrag des Vektors 16 Byte groß ist, starten wir mit der Untersuchung am Elementtyp `&str`.

Genau wie ein allgemeiner Slice besteht der String-Slice aus zwei Teilen: aus der Adresse des Slice (in `usize`) und aus der Anzahl der Zeichen im String-Slice. Größen (engl. *Sizes*) werden in Rust auch in `usize` ausgedrückt. Damit nimmt ein Slice-Element zwei `usize` im Speicher ein. Wenn Sie den Code wie ich auf einem 64-Bit-System ausführen, dann weist ein `usize` (das Maschinenwort) acht Byte auf. Damit landen wir bei 16 Byte pro Element beziehungsweise bei `&str`.

Eine Adresse mit einer Zusatzinformation wird auch als *breiter Zeiger* (engl. *Wide* oder *Fat Pointer*) bezeichnet. Der Slice ist ein breiter Zeiger, den Sie häufig antreffen. Ein anderer ist das Trait-Objekt, das Sie in Abschnitt 11.5, »Trait-Objekte«, kennenlernen werden.

Einen bemerkenswerten Kontrast zu der vorherigen Byte-Anzahl bietet der Blick auf `size_of_val(&ergebnis)`: Lediglich acht Bytes wurden per Move aus dem Block an die Funktion `main` übergeben.

`Vec<T>` und `Box<T>` sind sich darin ähnlich, dass sie nur wenig Speicher auf dem Stack einnehmen und stattdessen einen Heap-Speicher verwalten. Eigentlich bedarf es daher im oberen Beispiel keiner Box um den Vektor, da Sie den Vektor ohne Leistungseinbußen per Move hätten zurückgeben können. Der Vektor (24 Bytes) und die Box (8 Bytes) unterscheiden sich in ihrer der Größe auf dem Stack.

4.5 Zusammenfassung

In diesem Kapitel haben wir in den Blick genommen, wie Rust mit dem Speicher umgeht. Mit Hintergrundwissen zu den drei wichtigsten Speichersegmenten (*Stack*, *Heap* und *statischer Speicher*) haben wir den Begriff *Eigentum* besprochen.

Mit den Mechanismen `Copy` und Move sorgt der Compiler zu jedem Zeitpunkt für klare Eigentumsverhältnisse: Jeder Wert hat nur einen einzigen Eigentümer, damit sich nicht mehrere Parteien dafür verantwortlich fühlen, Speicher freizugeben. Das vermeidet Double Frees und ähnlich schwerwiegende Speicherfehler.

Eigentümer können leihweise Referenzen ausgeben und erlauben Abnehmern mit `&T` lesenden oder sogar mit `&mut T` schreibenden Zugriff. Während Sie beliebig viele geteilte Referenzen ausgeben können, kann es nur eine Referenz mit Schreibrechten geben.

Zuletzt haben Sie mit `Box<T>` eine Struktur kennengelernt, mit der Sie Werte in den Heap verschieben können. Dabei handelt es sich um einen sehr schlanken Datentyp auf dem Stack mit einem internen Zeiger in den Heap.

Kapitel 15
Iteratoren

Eine *Collection* legt Elemente zusammengehörig im Speicher ab. Eine *sequenzielle Collection* wie der Vektor bietet Ihnen anschließend eine Schnittstelle an, über die Sie jedes Element in eine Richtung durchlaufen. Während dieses Verhalten heutzutage das *Iterator-Muster* in Programmiersprachen abbildet, standen Programmierern in älteren Zeiten dazu nur *primitive Zeiger*, die Breite des Speicherobjekts und eine (hoffentlich) richtige Länge zur Verfügung.

Wenn Sie in Rust einen primitiven Zeiger auslesen, dann müssen Sie die Codestelle mit dem Schlüsselwort `unsafe` markieren. Daran kann erahnt werden, dass der Zugriff auf das Speicherobjekt in der heutigen Zeit nicht die sicherste Wahl darstellt. Mehr zu *Unsafe Rust* erfahren Sie später in Kapitel 19, »Unsafe Rust und das Foreign Function Interface«.

Stattdessen durchzieht das Iterator-Muster die Collections und sogar den `for`-Ausdruck in Rust. Das gibt Ihnen nicht nur eine vollständige Speichersicherheit und Komfort an die Hand. Sie erhalten außerdem eine einheitliche Schnittstelle, die das Trait `Iterator` vorgibt. Zudem ist es praktisch, dass ein Iterator träge oder faul ist (engl. *lazy*). Das heißt, dass eine `Iterator`-Instanz mit dem Durchlauf wartet, bis Sie mit ihr interagieren. Möglichkeiten dazu bieten die Methode `Iterator::next` und der `for`-Ausdruck.

In diesem Kapitel werden wir `Iterator` diskutieren. Dabei betrachten wir, wie Sie einen `Iterator` beziehen und die Elemente darin über die umfangreichen *Adapter-Methoden* modifizieren. Für das Trait sind konsumierende Methoden implementiert, die die Hauptanwendungsfälle abbilden, etwa `count`. »Konsumierend« bedeutet, dass die Methoden den Iterator mit einem Aufruf verbrauchen.

Der `for`-Ausdruck ist eigentlich nur syntaktischer Zucker. Wir untersuchen daher das Trait `IntoIterator` im Kontext der Schleife. `IntoIterator` kodifiziert, wie Sie einen beliebigen implementierenden Datentyp in einen `Iterator` überführen. Mit `FromIterator` steht ebenfalls der umgekehrte Weg offen, sodass Sie eine Datenstruktur aus einem `Iterator` erzeugen können.

15.1 Wie Sie einen Iterator beziehen

Da `Iterator` ein Trait ist, arbeiten Sie mit konkreten Datenstrukturen, die es implementieren. Ganz gleich, ob Sie `Iterator` für einen Ihrer eigenen Datentypen implementieren oder eine Collection der Standardbibliothek nutzen: Per Konvention bietet eine Datenstruktur, die den `Iterator` implementiert, die Methoden `iter` und `iter_mut` in ihrer Schnittstelle an. In Listing 15.1 dient der Vektor als Beispiel:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    let iterator: std::slice::Iter<i32> = v.iter();

    let mut v = vec![1, 2, 3, 4, 5];
    let iterator_mut: std::slice::IterMut<i32> = v.iter_mut();
}
```

Listing 15.1 Die Methoden »iter« und »iter_mut« von »Vec<T>«

Die Variablendeklarationen führen die Datentypen explizit an. Damit möchte ich Ihnen zeigen, dass hinter `iter` und `iter_mut` kein Trait, sondern eine Konvention steht. Die Iterator-Datenstrukturen `Iter<T>` und `IterMut<T>` im Modul `std::slice` stellen nur ein Implementierungspaar dar. Vektoren setzen Slices ein, um den Zugriff auf den internen Element-Puffer über ihre Schnittstelle zu erlauben. Infolgedessen kann `Vec<T>` ebenfalls für seine `Iterator`-Schnittstelle auf `std::slice` zurückgreifen. Das Gleiche gilt für das primitive Array, wie Sie in Listing 15.2 sehen:

```
fn main() {
    let mut a = [1, 2, 3, 4, 5];
    let iterator: std::slice::Iter<i32> = a.iter();
    let iterator_mut: std::slice::IterMut<i32> = a.iter_mut();
}
```

Listing 15.2 Auch das Array setzt »std::slice« ein.

Ein Beispiel für eine Collection, die ein anderes Implementierungspaar für `iter` und `iter_mut` anbietet, ist `HashMap<K, V>`. Der assoziative Zugriff auf die Elementdaten fordert von der `Iterator`-Datenstruktur ein anderes Vorgehen als beim sequenziellen Vorgehen in `std::slice`. Daher liefern die Methoden `iter` und `iter_mut` auf `HashMap`-Instanzen andere Datenstrukturen zurück. In Listing 15.3 sehen Sie ein Beispiel:

```
use std::collections::{
    HashMap,
    hash_map
};
```

```
fn main() {
    let h = HashMap::from([(1, "Hallo")]);
    let iterator: hash_map::Iter<i32, &str> = h.iter();

    let mut h = HashMap::from([(1, "Rust")]);
    let iterator: hash_map::IterMut<i32, &str> = h.iter_mut();
}
```

Listing 15.3 Die »HashMap« implementiert eine eigene Iterator-Datenstruktur.

Im Rust-Alltag werden Sie die Iterator-Datenstrukturen jedoch praktisch nie explizit deklariert sehen. Das liegt daran, dass die Strukturen das Trait `Iterator` implementieren, das eine verlässliche und standardisierte Schnittstelle herstellt. Dort greifen Sie auf Adapter-Methoden (siehe Abschnitt 15.2, »Iterator-Adapter«) und Iterator-Methoden (siehe Abschnitt 15.3, »Einen Iterator konsumieren«) zu.

15.1.1 Das Trait »IntoIterator«

Zuvor möchte ich Ihnen aber noch den dritten Weg zum `Iterator` vorstellen: `IntoIterator`. Damit lassen Sie die Konvention hinter sich und betreten den festen Boden einer Trait-Implementierung. Dass eine `Iterator`-Datenstruktur `IntoIterator` implementiert, ist sehr wahrscheinlich. Verantwortlich dafür ist der `for`-Ausdruck, der auf dem Trait aufbaut. Er ist syntaktischer Zucker für den `IntoIterator`-Aufruf.

`IntoIterator` verbraucht die eigene Instanz und unterscheidet sich darin deutlich von den Referenz-Iteratoren `iter` und `iter_mut`. In Situationen, in denen Sie gewisse Elemente eines Iterators an einen neuen Eigentümer überführen müssen, sollten Sie demzufolge die Methode `into_iterator` aufrufen.

Wenn `IntoIterator::into_iterator` die eigene Instanz verbraucht, der `for`-Ausdruck aber genau diese Methode aufruft, wird dann eine `for`-Schleife die Collection verbrauchen? Ja! Hier versteckt sich also ein Fallstrick, den man gerade zu Beginn im Umgang mit dem `for`-Ausdruck schnell übersieht: Collections implementieren `IntoIterator`, weshalb Sie unter anderem den Vektor direkt an `for` übergeben können. Listing 15.4 zeigt ein Beispiel:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5];
    for e in v {
        // ...verbraucht "v" und die Elemente (Move)
    }
```

```
    // Fehler, "v" wurde zuvor bewegt
    println!("Vektor: {v:?}");
}
```

Listing 15.4 Der implizite Aufruf von »IntoIterator::into_iterator« auf einem Vektor

Sie vermeiden das Problem mit Referenzen. Dazu übergeben Sie entweder `&v` respektive `&mut v` an den Schleifenkopf. Oder Sie nutzen den Rückgabewert von `iter` oder `iter_mut`. In Listing 15.5 sehen Sie Beispiele dazu:

```
fn main() {
    let v_ref = &v;
    let iterator = v.iter();

    for e in iterator {
        // ...
    }

    for e in v_ref {
        // ...
    }

    // Ok
    println!("Vektor: {v:?}");

    // Fehler, "iterator" wurde
    // durch Move in for verbraucht
    println!("Vektor: {iterator:?}");
}
```

Listing 15.5 Die Collection davor schützen, verbraucht zu werden

Die Indirektion über die Referenz bewahrt das Original. Der Aufruf von `into_iterator` im `for-Ausdruck` verbraucht zwar weiterhin den übergebenen Wert. Der besteht jetzt aber nur aus einer Referenz, die der Compiler an den Eigentümer zurückgibt.

15.1.2 »Iterator« implementieren

Im Folgenden möchte ich Ihnen vorführen, wie Sie `Iterator` für einen beliebigen Datentyp implementieren. Dann können Sie etwa über dessen Felder iterieren. Als Implementierungsziel dient die Datenstruktur `Person`, die aus den Feldern besteht, die Sie in Listing 15.6 sehen:

```
struct Person {
    alter: i32,
    name: String,
    beruf: String,
}
```

Listing 15.6 Die Struktur »Person«

Bei der Betrachtung der Iteratoren von `Vec<T>`, dem Array und der `HashMap<K, V>` haben wir gesehen, dass die Datenstrukturen jeweils eine `Iterator`-Datenstruktur vorgeschoben und nicht selbst `Iterator` implementiert haben. Das trennt die Aufgabenbereiche, die sonst weniger übersichtlich in eine einzige Implementierung gefüllt würden. Wir halten es mit `Person` genauso. Listing 15.7 zeigt die `Iterator`-Strukturen für `iter` und `iter_mut` auf `Person`:

```
struct PersonIterator<'a> {
    // Der Iterator soll die Instanz nicht verbrauchen können.
    person: &'a Person,
    // "Cell", um Interior Mutability auszunutzen
    // Dazu mehr in Abschnitt 16.2.2
    _printed_elements: std::cell::Cell<i32>,
}

struct PersonIteratorMut<'a> {
    // Der Iterator soll die Instanz nicht verbrauchen können.
    person: &'a mut Person,
    // "Cell", um Interior Mutability auszunutzen
    // Dazu mehr in Abschnitt 16.2.2
    printed_elements: std::cell::Cell<i32>,
}
```

Listing 15.7 Person-Iteratoren

Der einzige Unterschied liegt in der Veränderlichkeit, die `PersonIteratorMut` einführt. Die Strukturen `std::slice::Iter` und `std::slice::IterMut` sind ebenfalls bis auf den verwalteten Zeiger ähnlich (`Iter: *const T`, `IterMut: *mut T`).

»Cell« und die Interior Mutability

Die `Person`-Iteratoren sollen in Listing 15.7 für jeden Durchlauf einen Index inkrementieren, den das Feld `_printed_elements` hält. Der Datentyp des Felds ist `std::cell::Cell<T>`. `Cell` ist ein *Smart Pointer* und implementiert die *Interior Mutability*, durch die Sie Werte sogar hinter einer geteilten Referenz verändern können.

Ohne dieses Konzept müsste eine `Person`-Instanz sich selbst als veränderlich ausleihen, um den Zähler zu erhöhen. Die Interior Mutability erlaubt Ihnen stattdessen, den Zähler hinter `&self._printed_elements` zu verändern. (Dazu lesen Sie später mehr in Abschnitt 16.2.2, »Cells und die Interior Mutability«.)

Im nächsten Schritt verbinden Sie die Implementierung von `Person` über die Methoden `iter` und `iter_mut` mit den `Iterator`-Strukturen. In Listing 15.8 sehen Sie, wie das geht:

```
impl PersonIterator<'_> {
    pub fn new(p: &Person) -> PersonIterator {
        PersonIterator {
            person: p,
            _printed_elements: Default::default(),
        }
    }
}

impl PersonIteratorMut<'_> {
    pub fn new(p: &mut Person) -> PersonIteratorMut {
        PersonIteratorMut {
            person: p,
            printed_elements: Default::default(),
        }
    }
}

impl Person {
    fn iter(&self) -> PersonIterator {
        PersonIterator::new(&self)
    }
    fn iter_mut(&mut self) -> PersonIteratorMut {
        PersonIteratorMut::new(self)
    }
}
```

Listing 15.8 Die Iterator-Strukturen mit »iter« und »iter_mut« verbinden

Damit sind die für die Lösung essenziellen Strukturen eingeführt. Bevor Sie Instanzen von `PersonIterator` und `PersonIteratorMut` als Iteratoren einsetzen können, müssen Sie jeweils das Trait `Iterator` implementieren. Um den Rahmen nicht zu sprengen, betrachten wir fortan nur die Implementierung von `Iterator` für `PersonIterator`, die Sie in Listing 15.9 sehen:

```
impl Iterator for PersonIterator<'_> {
    type Item = String;

    fn next(&mut self) -> Option<Self::Item> {
        let result = match self._printed_elements.get() {
            0 => Some(self.person.alter.to_string()),
            1 => Some(self.person.name.clone()),
            2 => Some(self.person.beruf.clone()),
            _ => {
                self._printed_elements.set(0);
                None
            }
        };
        // Den Wert in der Cell austauschen
        self._printed_elements.set(
            self._printed_elements.get() + 1
        );
        result
    }
}
```

Listing 15.9 Die Iterator-Implementierung

Jeder Durchlauf eines `Iterator`s, etwa durch den `for`-Ausdruck, ruft die Methode `Iterator::next` auf. Solange ein Container-`Iterator` ein gültiges Element zurückgeben kann, erhält der Aufrufer ein `Some(T)`. Das ist im Listing dreimal der Fall. Dann sind alle Felder von `Person` durchlaufen, und infolgedessen ist das Ergebnis im vierten Aufruf von `next` ein `None`.

Ein Iterator ist immer nur für einen Elementdatentyp `T` implementiert. Den Elementdatentyp eines Iterators geben Sie mit dem assoziierten Datentyp `Item` vor, im Beispiel etwa als `type Item = String`. Wir konvertieren die Felder der Datenstruktur `Person` daher zu `String`.

Das letzte Bindeglied, das Ihren Iterator mit einem `for`-Ausdruck verbindet, ist `IntoIterator`. Wie weiter oben erwähnt, übersetzt der Compiler eine `for`-Schleife in den Aufruf einer Implementierung von `IntoIterator`. Praktischerweise generiert der Compiler automatisch eine Implementierung von `IntoIterator` für einen Iterator. Das erlaubt Ihnen, die Rückgabewerte der Methoden `iter` und `iter_mut` in `for`-Ausdrücken einzusetzen.

Das nächste Beispiel zeigt in Listing 15.10 die Implementierung von `IntoIterator` für `&Person`, die eine Referenz in einen `PersonIterator` übersetzt:

```
impl<'a> IntoIterator for &'a Person {
    type Item = String;
    type IntoIter = PersonIterator<'a>;

    fn into_iter(self) -> Self::IntoIter {
        PersonIterator::new(self)
    }
}
```

Listing 15.10 »IntoIterator« für »Person«

Der assoziierte Datentyp `Item` bestimmt den Wert-Datentyp, den die Aufrufe von `Iterator::next` zurückliefern. Damit entspricht er dem assoziierten Datentyp `Item` in `Iterator`. `IntoIter` definiert den Datentyp des anzuwendenden Iterators, hier `PersonIterator`. Das Beispiel implementiert `IntoIterator` nicht für `Person`, sondern nur für `&Person`, damit eine Schleife die Instanz nicht bewegt.

Die Implementierung ist abgeschlossen. Rufen Sie zum Test die Methode `iter` auf `Person` auf und übergeben Sie die `Iterator`-Struktur an einen `for`-Ausdruck. Wie Sie in Listing 15.11 sehen, durchlaufen Sie jedes Feld in einer `String`-Repräsentation:

```
// ...

fn main() {
    let mut person = Person {
        alter: 3,
        name: "Thalea".to_string(),
        beruf: "Familienmanagerin".to_string(),
    };

    // Ok
    for eigenschaft in person.iter() {
        print!("{eigenschaft }");
    }
    println!(); // Ausgabe: 3 Thalea Familienmanagerin

    // Ok, IntoIterator für &Person
    for eigenschaft in &person {
        // ...
    }
}
```

Listing 15.11 Schleifendurchläufe über die Felder von »Person«

Das Gegenstück zu »IntoIterator«: »FromIterator«

Listing 15.10 hat gezeigt, wie Sie eine Datenstruktur wie `Person` in einen `Iterator` überführen. Wo Sie in Rust `Into` lesen, da ist üblicherweise auch `From`. Wenn Sie das Trait `FromIterator` für eine Datenstruktur implementieren, bieten Sie dem Nutzer Ihrer Schnittstellen eine Möglichkeit, um die Instanz einer Datenstruktur aus der Auflistung von Werten zu generieren. Ein Beispiel dafür ist `HashMap`. Eine `HashMap`-Instanz können Sie mit einem `Iterator` über Tupel definieren:

```
use std::collections::HashMap;
let h: HashMap<i32, &str> = HashMap::from_iter([
    (1, "Hallo"),
    (2, "Rust")
]);
```

15.2 Iterator-Adapter

Der Iterator besteht hauptsächlich aus zwei Methoden-Arten: den *Adaptern*, die wir in diesem Abschnitt behandeln, und den Methoden, die einen Iterator *konsumieren*. Letzteres besprechen wir in Abschnitt 15.3, »Einen Iterator konsumieren«.

Manche der Methoden werden als Adapter bezeichnet, weil sie das *Adapter-Entwurfsmuster* erfüllen. Dabei formt der Aufruf der Adapter-Methode einen `Iterator` in einen anderen `Iterator` um. Durch diesen Adapter passen der ursprüngliche `Iterator` und die Zielschnittstelle oder Zielvorstellung zusammen. Ein Adapter nimmt also einen `Iterator` entgegen und gibt seinerseits einen `Iterator` zurück. So können Sie ganze *Iterator-Ketten* zusammensetzen, die Elemente immer weiterverarbeiten.

Das Trait `Iterator` implementiert alle Methoden auf der Grundlage von `Iterator::next`. Das ist der Grund dafür, dass Sie im implementierenden Datentyp von `Iterator` nur den assoziierten Datentyp `Item` und die Methode `next` definieren müssen, dafür aber einen erheblichen Funktionsumfang zurückerhalten.

Mit dem `Iterator` kommen daher zwei Dinge zusammen: eine große Anzahl von Methoden, mit denen Sie Daten filtern oder transformieren können, sowie eine praktisch unendliche Kombinierbarkeit. Listing 15.12 zeigt ein Beispiel:

```
fn main() {
    let v = vec![1, 2, 3, 4, 5, 6, 7];

    let positive_zahlen = v
        // Iterator aus Vektor erstellen
        .into_iter()
```

```
        // Zwei am Anfang überspringen
        .skip(2)
        // Dann nimm vier Zahlen
        .take(4)
        // Zwischenstand überprüfen
        .inspect(|x| println!("Elemente: {x}"))
        // Verarbeite nur die positiven Zahlen weiter.
        // Dereferenzierung, da n: &i32
        .filter(|n| *n % 2 == 0)
        // Kehre die Reihenfolge um
        .rev()
        // Führe den Iterator aus
        .collect::<Vec<_>>();

    println!("{positive_zahlen:?}");
    // [6, 4]
}
```

Listing 15.12 Verschiedene Iterator-Adapter in Kombination

Alle Aufrufe bis zu `collect` sind Adapter, weil sie den `Iterator` in einen neuen Zustand überführen, auf dem Sie sie aufrufen. `collect` ist dagegen eine konsumierende Methode: Sie verbraucht den `Iterator`, um dessen Inhalt in einen Vektor zu überführen. Eingangs hatte ich erwähnt, dass Iteratoren träge oder faul (engl. *lazy*) sind. Ohne die Methode `collect` würden Sie in Listing 15.12 nur eine Beschreibung des Iterators zurückerhalten, nicht aber das Ergebnis wie hier in Listing 15.13:

```
    let positive_zahlen = v
        // Iterator aus Vektor erstellen
        .into_iter()
        // Zwei am Anfang überspringen
        .skip(2)
        // Dann nimm vier Zahlen
        .take(4)
        // Zwischenstand überprüfen
        .inspect(|x| println!("Elemente: {x}"))
        // Verarbeite nur die positiven Zahlen weiter
        // Dereferenzierung, da n: &i32
        .filter(|n| *n % 2 == 0)
        // Kehre die Reihenfolge um
        .rev();

    println!("{positive_zahlen:?}");
```

```
    // Rev { iter: Filter {
    //         iter: Inspect {
    //             iter: Take {
    //                 iter: Skip {
    //                     iter: IntoIter(
    //                         [1, 2, 3, 4, 5, 6, 7]
    //                     ),
    //                     n: 2
    //                 },
    //                 n: 4
    //             }
    //         }
    //     }
    // }
}
```

Listing 15.13 Iteratoren sind faul.

Zu `skip` und `take` existieren zudem die Varianten `skip_while` und `take_while`. Dort hinterlegen Sie eine Closure, die zu `true` oder `false` auswertet. Der `Iterator` wird jeweils Elemente hinzufügen, bis die Closure das erste Mal `false` liefert. Ein Beispiel dafür sehen Sie in Listing 15.14:

```
fn main() {
    let zahlen = vec![1, 2, 3, 4, 5, 6, 7]
        .into_iter()
        .skip_while(|n| *n < 3)
        .take_while(|n| *n < 6)
        .collect::<Vec<_>>();

    // [3, 4, 5]
}
```

Listing 15.14 »skip_while« und »take_while«

15.2.1 Eigentum im Iterator

Die Methode `iter` gewährt Ihnen den Zugriff auf die Elemente eines Iterators über geteilte Referenzen. Iterator-Adapter und die konsumierenden Methoden erhalten somit Werte vom Typ `&T`. Das schränkt demzufolge Ihre Möglichkeiten auf Szenarien ein, in denen Sie den Iterator nur auslesen.

Für den exklusiven und veränderlichen Zugriff nutzen Sie stattdessen `iter_mut`. Adapter und Methoden arbeiten infolgedessen mit `&mut T`. Die veränderliche Referenz

verhindert jedoch, dass Sie Werte aus dem Iterator bewegen – das bleibt dem Eigentümer vorbehalten. Das Eigentum übertragen Sie mit `into_iter`. Das verbraucht allerdings die Collection oder Container-Datenstruktur, auf der Sie die Methode aufrufen.

Die drei Methoden beschreiben allesamt Wege, wie Sie zu einem `Iterator` gelangen. Dabei verändert sich nur der Datentyp des assoziierten Datentyps `Iterator::Item`, etwa `&T`, `&mut T` oder `T`. Wenn Sie hingegen den Zugriff auf die Werte in einem bereits existierenden `Iterator` steuern möchten, setzen Sie einen der Adapter `by_ref`, `cloned` oder `copied` ein.

»copied« und »cloned«

Bei `cloned` und `copied` geht es darum, Klone von `&T`, `&mut T` oder `T` zu erstellen. Datentypen, die durch `Copy` eine bitweise Kopie von sich erstellen können, erlauben den Adapter `copied`. Alternativ setzen Sie `cloned` ein, wenn der Elementdatentyp im `Iterator` lediglich das Trait `Clone` implementiert. In Listing 15.15 sehen Sie ein Beispiel zu `copied`:

```
fn main() {
    let zahlen = vec![1, 2, 3, 4, 5, 6, 7];
    let iterator = zahlen
        .iter()
        // Jedes Element kopieren
        .copied()
        // Transformiert Typ "T" in "U"
        // oder verändert den Wert im Iterator
        .map(|n| n * n)
        .collect::<Vec<i32>>();

    // Zahlen:   [1, 2, 3, 4, 5, 6, 7]
    // Iterator: [1, 4, 9, 16, 25, 36, 49]
}
```

Listing 15.15 Der Adapter »copied«

Der Datentyp `i32` ist `Copy`, weshalb Sie auf dem `Iterator` über `zahlen` den Adapter `copied` aufrufen dürfen. Alle folgenden Adapter erhalten nun die Kopien. Die Originale im Vektor `zahlen` bleiben damit unverändert, wenn der Adapter `map` ausführt.

`map` ist eine der beliebtesten `Iterator`-Methoden, weil er so vielseitig ist. Die Closure von `map` erhält in jedem Durchlauf das aktuelle Element des Iterators und darf es dank einer veränderlichen Referenz ändern. Während wir in diesem Beispiel nur eine Multiplikation ausführen, sehen Sie gleich, wie Sie mit `map` sogar den Datentyp von `T` auf `U` ändern. Der veränderte Datentyp muss jedoch für jedes Element der gleiche sein. Sie können daher nicht ein Element `T` zu `U` und das nächste zu `V` ändern.

Datentypen wie `String` implementieren zwar nicht `Copy`, dafür aber `Clone`. In diesen Fällen rufen Sie den Adapter `cloned` so wie in Listing 15.16 auf:

```
let worte = ["Hallo", ",", "Rust", "!"]
    .iter()
    // &str zu String transformieren
    .map(|s| s.to_string())
    .collect::<Vec<_>>();

let iterator = worte
    .iter()
    // Jedes Element klonen
    .cloned()
    // Der Iterator ist jetzt Eigentümer der Klone
    .map(|s| s.to_ascii_uppercase())
    .collect::<Vec<_>>();

// ...
```

Listing 15.16 Klone mit »cloned« erstellen

Im ersten Iterator-Block arbeitet erneut die Methode `map`. Dieser Aufruf überführt jeden String-Slice des Iterators in einen `String`. Das Ergebnis ist ein Iterator mit dem assoziierten Datentyp `String`, den `collect` ausführt und in einen Vektor verwandelt.

Im zweiten Block erstellt der Adapter `cloned` Kopien von den Strings, die `map` anschließend in Großschrift abändert. Davon bleiben die `String`-Elemente in `worte` aber unberührt. Um die Ergebnisse von `worte` und `iterator` nebeneinander zu sehen, fügen wir einen dritten Iterator-Block ein, der den Adapter `zip` einsetzt.

`zip` arbeitet wie ein Reißverschluss und fügt jeweils die Elemente zweier Iteratoren zu einem Tupel zusammen. Das Resultat ist ein einzelner Iterator. Listing 15.17 zeigt den Code dazu:

```
fn main() {
    // ...
    let zip_iterator = worte
        .iter()
        .cloned()
        .zip(iterator)
        .collect::<Vec<_>>();

    println!("Wort und Klon: {zip_iterator:?}");
    // Wort und Klon:
    // [
```

```
    //     ("Hallo", "HALLO"),
    //     (",", ","),
    //     ("Rust", "RUST"),
    //     ("!", "!")
    // ]
}
```

Listing 15.17 Der »zip«-Adapter

Die Gegenoperation stellt der `unzip`-Adapter dar, mit dem Sie aus den Tupeln erneut zwei einzelne Vektoren herstellen. Listing 15.18 zeigt, wie das geht:

```
fn main() {
    // ...

    let (originale, klone): (Vec<_>, Vec<_>) = zip_iterator
        .iter()
        .cloned()
        .unzip();

    println!("Originale: {originale:?}, Klone: {klone:?}")
    // Originale:
    //  ["Hallo", ",", "Rust", "!"],
    // Klone:
    //  ["HALLO", ",", "RUST", "!"]
}
```

Listing 15.18 Die konsumierende Funktion »unzip«

Beachten Sie, dass nach `unzip` kein Aufruf von `collect` notwendig ist. `unzip` ist kein Iterator-Adapter, weil sein Ergebnis keinen `Iterator` darstellt. Sie können also nach `unzip` nicht mehr mit der Iterator-Kette weiterarbeiten.

`unzip` hat die mit dem Reißverschluss verbundenen Elemente aufgetrennt und in zwei separate Vektoren aufgeteilt. Das Tupel-Muster in der `let`-Anweisung führt zwei Variablen ein, sodass Sie das Ergebnis in der Standardausgabe einsehen können.

Der Iterator hinter einer Referenz: »by_ref«

Iterator-Adapter verbrauchen ihre Instanz. Nachdem Sie einen Iterator in einen `for`-Ausdruck oder in eine Adapter-Kette eingesetzt haben, müssen Sie also für Ersatz sorgen. Wenn Sie den Iterator mit `iter` oder `iter_mut` bezogen haben, rufen Sie die jeweilige Methode einfach noch einmal auf. Haben Sie den Iterator stattdessen über `into_iter` bezogen, ist sowohl die Quelle als auch der Iterator mit der letzten Verwendung verpufft. Listing 15.19 zeigt ein Beispiel:

```
fn main() {
    let satzteile = ["Hallo", ",", "Rust", "!"]
        .iter()
        // &str zu String transformieren
        .map(|s| s.to_string())
        .collect::<Vec<_>>();

    let mut iterator = satzteile.into_iter();

    let worte = iterator
        .filter(|s| s.len() > 1)
        .collect::<Vec<_>>();

    // Fehler, die Instanz ist verbraucht!
    let worte_gross = iterator
        .map(|s| s.to_ascii_uppercase())
        .collect::<Vec<_>>();

    // ...
}
```

Listing 15.19 Der Iterator wird mit der ersten Adapter-Kette verbraucht.

Das Programm möchte zwei Aktionen auf demselben Iterator ausführen. Da ist zunächst eine Zählung der Worte, die vereinfacht annimmt, dass ein Wort größer als ein Zeichen sein muss. Dazu verwendet das Programm den Adapter `filter`, der in Suchrichtung das erste Element zurückliefert, das die Bedingung in der Closure erfüllt.

Sie benötigen den Iterator also für zwei Iterator-Ketten. Die erste Kette verbraucht jedoch die Instanz. Was können Sie nun tun? `Iterator` stellt hierfür den Adapter `by_ref` zur Verfügung. Er nimmt die eigene Iterator-Instanz als `&mut self` anstatt als `self` entgegen. Sie können demnach Veränderungen durchführen, der Iterator bleibt aber weiterhin gültig. In Listing 15.20 sehen Sie die Anpassung von Listing 15.19:

```
fn main() {
    let satzteile = ["Hallo", ",", "Rust", "!"]
        .iter()
        // &str zu String transformieren
        .map(|s| s.to_string())
        .collect::<Vec<_>>();

    let mut iterator = satzteile.into_iter();

    let worte = iterator
```

```
        .by_ref()
        .filter(|s| s.len() > 1)
        .collect::<Vec<_>>();

    let worte_gross = iterator
        .by_ref()
        .map(|s| s.to_ascii_uppercase())
        .collect::<Vec<_>>();

    // ... mit "iterator" weiterarbeiten
}
```

Listing 15.20 Der Adapter »by_ref« schützt den Iterator.

Einen Iterator klonen

Sie können einen Klon von einem Iterator erzeugen, indem Sie `clone` aufrufen. Weil der Iterator bis zur Ausführung bloß die Beschreibung eines Verhaltens darstellt, ist `clone` eine kostengünstige Lösung für den mehrfachen Zugriff auf denselben Iterator. Das gilt freilich nicht, wenn nachfolgende Aufrufe auf den veränderten Zustand durch einen vorherigen Iterator-Adapter zählen. In diesem Fall bleiben Sie bei `by_ref`.

15.2.2 Iteratoren zusammenfügen

Sie könnten sich in einer Situation befinden, in der Sie mehrere Iterator-Strukturen mit dem gleichen Elementdatentyp bearbeiten. Das Trait `Iterator` bietet Ihnen Adapter, um Iteratoren über Iteratoren oder viele separate Iteratoren zusammenzufügen. Verschachtelte Iteratoren behandeln Sie mit `flatten`. Der Adapter `chain` kettet zwei Iteratoren zusammen. In Listing 15.21 sehen Sie zwei Beispiele:

```
fn main() {
    let erster_teil = [1, 2, 3, 4, 5]
        .into_iter();
    let zweiter_teil = [6, 7, 8, 9, 10]
        .into_iter();
    let mut alle_zahlen = [erster_teil, zweiter_teil]
        .into_iter();

    let alle_flach = alle_zahlen
        .flatten()
        .collect::<Vec<i32>>();
```

```
    println!("Flach: {alle_flach:?}");
    // Flach: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

    // ...

    let erster_teil = [1, 2, 3, 4, 5]
        .into_iter();
    let zweiter_teil = [6, 7, 8, 9, 10]
        .into_iter();

    let gekettet = erster_teil
        .chain(zweiter_teil)
        .collect::<Vec<_>>();

    println!("Kette: {gekettet:?}");
    // Kette: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
}
```

Listing 15.21 Die Adapter »flatten« und »chain« fügen Iteratoren zusammen.

15.2.3 Transformationen in der Iterator-Kette

Der Iterator definiert einige Adapter, mit denen Sie den Elementdatentyp für nachfolgende Adapter verändern können. Ein Beispiel haben Sie oben bereits kennengelernt: `map` (siehe Listing 15.19). In diesem Abschnitt möchte ich Ihnen weitere Adapter vorstellen, die wie `map` arbeiten oder darauf aufbauen.

Index und Element in »enumerate«

Nehmen wir an, Sie benötigen den Index eines Elements, das Sie mithilfe eines Iterators verarbeiten. Benutzen Sie in diesem Fall den Adapter `enumerate`, der Ihnen ein Tupel aus Index und Element liefert, wie Sie in Listing 15.22 sehen:

```
fn main() {
    ["Hallo", ",", "Rust", "!"]
        .iter()
        // Die nächsten Adapter erhalten (Index, Element)
        .enumerate()
        .for_each(
            |paar| {
                println!(
                    "Index: {}, Element: {}",
                    paar.0,
                    paar.1
```

```
                );
            }
        );
    // Index: 0, Element: Hallo
    // Index: 1, Element: ,
    // Index: 2, Element: Rust
    // Index: 3, Element: !
}
```

Listing 15.22 Index und Element mit dem Adapter »enumerate« auslesen

Das Interessante an `enumerate` ist, dass er den Elementtyp in der Iterator-Kette ändert. Das haben Sie zuvor in Beispielen gesehen, die den `map`-Adapter eingesetzt haben. Die konsumierende Methode `for_each` erhält demzufolge nicht `&&str`, sondern `(usize, &&str)`. Die doppelte Referenz rührt daher, dass `iter` eine Referenz `&` auf das String-Slice `&str` ausleiht.

Mehr »map«

Der `map`-Adapter durchkämmt den Iterator von Anfang bis Ende. Soll der Adapter hingegen aufhören, sobald eine Bedingung nicht mehr erfüllt ist, dann rufen Sie `map_while` auf. Das ist praktisch, wenn Werte keine sinnvollen Ergebnisse mehr liefern, falls sie die Bedingung nicht erfüllen. Listing 15.23 zeigt ein Beispiel:

```
fn main() {
    let zahlen = [1, 2, 3, 4, 5]
        .into_iter()
        .map_while(|n| {
            if n <= 3 {
                Some(n.to_string())
            } else {
                None
            }
        }).collect::<Vec<_>>();
    // ["1", "2", "3"]
}
```

Listing 15.23 Die bedingte Transformation mit »map_while«

Sobald `map_while` als Ergebnis der Closure den Wert `None` ausliest, endet die Ausführung auf dem Iterator. So landen nur die Werte in der Ergebnismenge, die Sie mit der Bedingung gefiltert haben.

Mit dem `filter`-Adapter durchlaufen Sie alle Elemente im Iterator, aber nehmen nur jene mit, die eine Bedingung erfüllen. Der Adapter `filter_map` verbindet `filter` und

`map` miteinander, sodass Sie nur eine Methode aufrufen müssen, wenn Sie sowohl filtern als auch transformieren möchten – etwa so wie in Listing 15.24:

```
fn main() {
    let zahlen = [1, 2, 3, 4, 5]
        .into_iter()
        // Die Zahlen 2 bis 4 in Strings umwandeln
        .filter_map(|n| {
            if n >= 2 && n <= 4 {
                Some(n.to_string())
            } else {
                None
            }
        })
        .collect::<Vec<_>>();
    // ["2", "3", "4"]
}
```

Listing 15.24 Der Adapter »filter_map«

Dann wäre da noch `flat_map`. Dieser spezifische `map`-Adapter arbeitet mit Datentypen, die `IntoIterator` implementieren. Oder anders ausgedrückt: Sie rufen `flat_map` auf einem Iterator über Iteratoren auf. Ein weiterer Unterschied zu `map` ist, dass jeder Aufruf der Closure in `flat_map` einen Iterator zurückgibt. Sobald `flat_map` alle Ergebnisse gesammelt hat, flacht es die einzelnen Iteratoren wie der `flatten`-Adapter ab (siehe Abschnitt 15.2.2, »Iteratoren zusammenfügen«). In Listing 15.25 sehen Sie ein Beispiel:

```
fn main() {
    let erster_teil = ["Hallo", ","]
        .into_iter();
    let zweiter_teil = [" Rust", "!"]
        .into_iter();

    let alle_zahlen = [erster_teil, zweiter_teil]
        .into_iter()
        // Nimm IntoIterator entgegen, gib abgeflacht zurück
        .flat_map(|n| n.enumerate())
        .collect::<Vec<_>>();
    // [(0, "Hallo"), (1, ","), (0, " Rust"), (1, "!")]
}
```

Listing 15.25 »map« mit Iteratoren: »flat_map«

15.2.4 Weitere Iterator-Adapter

In den vorherigen Beispiele habe ich Ihnen die häufig angewendeten Adapter vorgestellt. Damit ist die Bandbreite jedoch noch nicht abgedeckt. Daher finden Sie in Tabelle 15.1 eine Zusammenstellung derjenigen Adapter, die wir bislang nicht diskutiert haben.

Adapter	Beispiel	Beschreibung
`step_by`	`step_by(3)`	Gibt das erste Element zurück. Springt dann nicht zum nächsten, sondern macht einen Schritt der Größe `usize`. Der Adapter setzt diese Schritte bis zum Ende fort.
`cycle`	`cycle()`	Sobald der Iterator das Ende erreicht (`None`), fängt er selbstständig wieder am Anfang an.
`fuse`	`fuse()`	Invalidiert den Iterator. Der nächste Aufruf gibt einen Wert zurück. Alle folgenden geben nur noch `None` zurück.
`peekable`	`peekable()`	Die nächsten Elemente mit `peek` und `peek_mut` ansehen. Der Iterator schreitet dabei nicht voran.

`scan`	`scan(`
	`  status,`
	`  | status, &element |`
	`  { … }`
	`)`
		Wie `fold` (siehe Abschnitt 15.3, »Einen Iterator konsumieren«). Gibt jedoch einen Iterator zurück, anstatt einen akkumulierten Wert.

Tabelle 15.1 Übersicht über weitere Adapter

15.3 Einen Iterator konsumieren

Iteratoren lassen sich auf vielerlei Art mit Adaptern konfigurieren. Allerdings kommt der Zeitpunkt, an dem Sie einen Iterator *konsumieren* müssen. Im vorherigen Abschnitt passierte das durch die Methode `collect`.

Der Iterator definiert eine umfangreiche Methodenpalette, mit der Sie nach Elementen suchen, sie zählen oder miteinander verrechnen. Einige dieser Methoden präsentiere ich Ihnen anhand von Beispielen. Viele weitere finden Sie am Ende des Abschnitts in Tabelle 15.2.

15.3.1 Eine Zusammenstellung einfacher Konsumenten

Einige der Methoden, die einen Iterator konsumieren, werden zwar häufig eingesetzt, sind jedoch äußerst trivial. Wir betrachten diese Aufrufe zusammengefasst in Listing 15.26:

```
fn main() {
    let zahlen = [1, 2, 3, 4, 5]
        .into_iter();

    // Anzahl der Elemente
    assert_eq!(zahlen.clone().count(), 5);

    // Erfüllt mindestens eines der Elemente die Bedingung?
    assert_eq!(zahlen.clone().any(|n| n == 3), true);

    // Erfüllen alle Elemente die Bedingung?
    assert_eq!(zahlen.clone().all(|n: i32| n % 2 == 0), false);

    // Gib das letzte Element des Iterators zurück
    assert_eq!(zahlen.clone().last(), Some(5));

    // Der kleinste Wert im Iterator
    assert_eq!(zahlen.clone().min(), Some(1));

    // Der größte Wert im Iterator
    assert_eq!(zahlen.clone().max(), Some(5));

    // Führe die Closure für jedes Element aus
    zahlen.clone().for_each(|n| println!("Element: {n}"));

    // Rufe "FromIterator" für collect::<T> auf
    let kopie = zahlen.clone().collect::<Vec<_>>();

    // Lese das n-te Element aus
    assert_eq!(zahlen.clone().nth(3), Some(4));
}
```

Listing 15.26 Einige konsumierende Methoden

15.3.2 Finden, falten und reduzieren

Häufig suchen Sie ein spezifisches Element und möchten herausfinden, ob es sich im Iterator befindet. Dazu rufen Sie in Rust die Methode `find` auf und übergeben eine

Closure als Suchvorschrift. Wenn der Iterator das Element findet, entnimmt er es aus dem Iterator. `find` ist demnach eine Methode, die einen veränderlichen Zugriff auf die eigene Instanz verlangt! Listing 15.27 zeigt ein Beispiel:

```
fn main() {
    let mut zahlen = [1, 2, 3, 4, 5]
        .into_iter();
    // Findet und entnimmt das Element. Ansonsten "None"
    assert_eq!(
        zahlen.find(|n| n == &4 || n % 2 == 1 ),
        Some(1)
    );
    // [2, 3, 4, 5]
}
```

Listing 15.27 Eine spezifische Zahl im Iterator finden

Neben `find` steht Ihnen zusätzlich `find_map` zur Verfügung. Diese Methode erlaubt es Ihnen, jedes Element `T` im Closure-Durchlauf zunächst zu `U` zu transformieren (`map`) und anschließend das erste Element darauf zu finden (`find`), das `Some(U)` zurückgibt. Diese Funktionalität geht über `find` hinaus, weil `find` ausschließlich mit dem Elementtyp `T` des Iterators arbeitet und diesen zurückgibt.

In Rust-Code erweisen sich auch `fold` und `reduce` als äußerst hilfreich. Diese Methoden ersetzen in vielen Fällen klobige `for`-Ausdrücke. Mit `fold` und `reduce` verrechnen Sie alle Elemente eines Vektors miteinander und produzieren daraus einen Wert. In Listing 15.28 sehen wir uns zunächst `fold` an:

```
fn main() {
    let zahlen = [1, 2, 3, 4, 5]
        .into_iter();

    // Startwert 10, dann addiere weitere Elemente hinzu
    assert_eq!(
        zahlen.fold(10, |mut summe, n| {
            summe += n;
            summe
        }),
        25 // 10 + 2 + 3 + 4 + 5
    );

    let v = vec!["Hallo, ", "Rust!"]
        .iter()
        .map(|s| s.to_string())
```

```
        .collect::<Vec<_>>();

    // Starte mit leerem String, füge die Teile zusammen
    assert_eq!(
        v.iter().fold(String::new(), |mut text, s| {
            use std::ops::Add;

            text.add(&s)
        }),
        "Hallo, Rust!".to_string()
    );
}
```

Listing 15.28 Die Methode »fold«

`reduce` ist der Methode `fold` ähnlich, verzichtet aber auf den Startwert. Das Ergebnis von `reduce` basiert also ausschließlich auf den Elementen des Iterators. Ein Unterschied liegt ferner im Rückgabetyp, der von `T` zu `Option<T>` wechselt. In Listing 15.29 sehen Sie die Beispiele aus Listing 15.28, angepasst auf `reduce`:

```
fn main() {
    let zahlen = [1, 2, 3, 4, 5]
        .into_iter();

    assert_eq!(
        zahlen
            .reduce(|mut summe, n| {
                summe += n;
                summe
            }),
        Some(15) // 2 + 3 + 4 + 5
    );

    let v = vec!["Hallo, ", "Rust!"]
        .iter()
        .map(|s| s.to_string())
        .collect::<Vec<_>>();

    assert_eq!(
        v.into_iter().reduce(|text, s| {
            use std::ops::Add;

            text.add(&s)
        }),
```

```
        Some("Hallo, Rust!".to_string())
    );
}
```

Listing 15.29 Alle Elemente mit »reduce« verrechnen

15.3.3 Sonstige Methoden, die einen Iterator konsumieren

Alle konsumierenden Methoden des Iterators in größerem Detail zu betrachten, würde den Rahmen dieses Kapitels sprengen. Daher fasse ich in Tabelle 15.2 die übrigen Methoden zusammen.

Mehr Informationen zum Iterator und seinen Methoden finden Sie in der Dokumentation der Standardbibliothek unter *https://doc.rust-lang.org/std/iter/trait.Iterator.html#provided-methods*.

Methode	Beispiel	Beschreibung
`eq`	`eq(anderer_iterator)`	Vergleicht, ob die Elemente der Vektoren gleich sind.
`ne`	`ne(anderer_iterator)`	Vergleicht, ob die Elemente der Vektoren ungleich sind.
`cmp`	`cmp(anderer_iterator)`	Setzt zwei Iteratoren in `Ordering`-Relation: `Equal`, `Less` oder `Greater`.
`partial_cmp`	`partial_cmp(anderer_iterator)`	Prüft die Implementierungen von `PartialEq` der Elemente in beiden Iteratoren.
`ge`	`ge(anderer_iterator)`	Wahr, wenn dieser Iterator `Ordering::Greater` als der andere oder `Ordering::Equal` ist.
`gt`	`gt(anderer_iterator)`	Wahr, wenn dieser Iterator `Ordering::Greater` ist als der andere.
`le`	`le(anderer_iterator)`	Wahr, wenn dieser Iterator `Ordering::Less` als der andere oder `Ordering::Equal` ist.
`lt`	`lt(anderer_iterator)`	Wahr, wenn dieser Iterator `Ordering::Less` ist als der andere.

Tabelle 15.2 Übersicht über weitere Konsumenten-Methoden

Methode	Beispiel	Beschreibung
`max_by`	`max_by(| x, y | { … })`	Sucht das Maximum-Element im Vektor und vergleicht es mit der Closure.
`max_by_key`	`max_by_key(| x | { … })`	Sucht das Maximum-Element im Vektor. Nutzt die Closure, um die Elemente in Schlüsselwerte einzuordnen. Wenn mehrere Elemente den höchsten Schlüsselwert aufweisen, nimmt die Methode das erste Element dieser Menge.
`min_by`	`min_by(| x, y | { … })`	Sucht das Minimum-Element im Vektor und vergleicht es mit der Closure.
`min_by_key`	`max_by_key(| x | { … })`	Wie `max_by_key`, aber als Minimum-Schlüsselwert.
`partition`	`partition(| x | x % 2 == 0)`	Trennt den Iterator in zwei Partitionshälften (siehe auch den Partitionspunkt im Vektor in Abschnitt 6.4.7).
`position`	`position(| x | { x == 5 })`	Gibt den `Some(Index)` des Elements zurück, das im Vergleich in der Closure zu `true` auswertet. Andernfalls `None`.
`product`	`product()`	Multipliziert alle Elemente des Vektors. Der Elementtyp muss dazu das Trait `Product` implementieren.
`sum`	`sum()`	Addiert alle Elemente des Vektors. Der Elementtyp muss dazu das Trait `Sum` implementieren.
`unzip`	`unzip()`	Trennt zweiwertige Tupel-Elemente in zwei Collections auf.

Tabelle 15.2 Übersicht über weitere Konsumenten-Methoden (Forts.)

15.4 Zusammenfassung

In diesem Kapitel haben wir den `Iterator` diskutiert. Damit sind Ihnen die drei Wege bekannt, mit denen Sie einen `Iterator` erfragen: `iter`, `iter_mut` oder `into_iter`. Die ersten beiden Bezeichner folgen einer Konvention, denn jeder implementierende Datentyp von `Iterator` sollte diese Methoden oder zumindest eine von ihnen implementieren. `into_iter` ist hingegen eine Methode des `IntoIterator`-Traits. Anhand eines Beispiels haben Sie alle drei Varianten in einer eigenen Datenstruktur implementiert.

Der `for`-Ausdruck ist syntaktischer Zucker. Der Compiler übersetzt eine `for`-Schleife in den Aufruf `IntoIterator::into_iterator`, der den Iterator verbraucht. Das ist für `iter` und `iter_mut` kein Problem, weil diese Methoden jeweils eine Referenz auf den Iterator zurückliefern. Eine *Blanket-Implementierung* (siehe Abschnitt 14.4.4, »Blanket-Implementierungen«) sorgt dafür, dass ein `Iterator` automatisch `IntoIterator` implementiert und sich selbst zurückgibt. Daher können Sie das Ergebnis von `iter` und `iter_mut` unmittelbar an einen `for`-Ausdruck übergeben.

Einen Iterator verwenden Sie zum einen mit Adaptern, die den Iterator in einen neuen Zustand überführen. Dabei können Sie Iteratoren filtern, transformieren, trennen oder zusammenstecken. Außerdem implementiert der Iterator eine große Anzahl an Methoden, die ihn konsumieren. Diese Methoden nutzen Sie, um etwa die Länge eines Iterators zu berechnen, Abfragen wie `all` oder `any` zu starten oder ein spezifisches Element darin zu finden.

Kapitel 17
Makros

Was hat es mit Makros auf sich? Makros sind in Rust allgegenwärtig. Schon ein Hallo-Welt-Programm greift auf das Makro `println!` zurück. Ein Makro erkennen Sie stets an dem Ausrufezeichen nach dem Bezeichner. Wenn Sie an Ihre Codeexperimente zurückdenken, fallen Ihnen sicherlich weitere Beispiele ein, in denen Sie Makros eingesetzt haben.

Die Verbreitung von Makros hängt damit zusammen, dass sie ein sicherer Teil der Sprache sind und sich darin von C- oder C++-Makros unterscheiden. Der Compiler verarbeitet ein Makro, bevor er mit dem Kompilieren beginnt. Das bedeutet, dass die Code-Änderung eines Makros den gleichen Regeln unterliegt wie regulärer Code.

Wir beginnen die Untersuchung mit den *deklarativen Makros*, die einige komplexere Aspekte der Makro-Programmierung abstrahieren und Ihnen dadurch den Zugang zur Meta-Programmierung erleichtern. Diese Erleichterungen bringen jedoch Einschränkungen mit sich. Die *prozeduralen Makros* bieten Ihnen hingegen auf Kosten der Zugänglichkeit mehr Möglichkeiten, zum Beispiel die Implementierung eigener Attribute.

Beide Makros verschaffen Ihnen im Rahmen der Meta-Programmierung die Fähigkeit, Code zu untersuchen, zu erweitern oder zu entfernen. Das vermeidet Code-Duplikate oder sogenannten Boilerplate-Code, da Sie stattdessen auf eine Codestelle einfach ein Makro anwenden.

17.1 Deklarative Makros

Das *deklarative Makro* ist eine Form der *Meta-Programmierung* und daher nicht Teil der syntaktischen Elemente, die Sie von Rust kennen. *Meta-Elemente* bilden hingegen eine eigene Form und Sprache. Ihr Zweck besteht darin, über Rust-Elemente sprechen und mit ihnen in abstrakter Form arbeiten zu können.

Über Rust-Elemente sprechen? Ein Beispiel: Sie drücken mit dieser Makro-Sprache aus, dass Sie einen Element-Bezeichner als Eingabe erwarten, müssen den allerdings nicht konkretisieren. Das könnte der Bezeichner einer Datenstruktur sein, wie `Person` in `struct Person`. Im Makro fügen Sie das Meta-Element in eine Vorlage aus Rust-Code ein. Dann übersetzt der Compiler: An die Stelle des deklarativen Makros tritt jetzt regulärer Rust-Code.

Mit bloßer Rust-Syntax können Sie den Vorgang dagegen nicht beschreiben, da der Compiler konkrete Datentypen *als Ganzes* entweder selbst inferieren muss oder Ihnen eine Typangabe abverlangt. Der Bezeichner einer Datenstruktur ist hingegen nur ein *Fragment* eines Datentyps. (Dazu folgt mehr in Tabelle 17.1.)

Der Begriff *deklarativ* hat in der Gestalt der deklarativen Programmierung besonders im Bereich der Oberflächenprogrammierung an Bedeutung gewonnen. »Deklarativ« meint, dass Sie eine Funktion beschreiben, die eine Eingabe in eine Ausgabe übersetzt. Die Durchführung produziert den Zustand des Systems, kennt ihn aber selbst nicht.

17.1.1 Warum Makros?

Doch wozu sollten Sie Rust-Code auf einer Meta-Ebene behandeln? Sie deklarieren damit Code-Vorlagen, die Sie mit Meta-Elementen anreichern. Die Vorlagen müssen hierbei keinen Bezug zum Einsatzort aufweisen und sind vielseitig einsetzbar. Das Makro erweist sich vor diesem Hintergrund als nützlich, um syntaktische oder konzeptionelle Wiederholungen zu vermeiden.

Denken Sie beispielsweise an `vec!`. Das Makro haben Sie gelegentlich eingesetzt, um einen Vektor zu initialisieren und gleichzeitig die als Argument übergebenen Werte einzufügen. Das komprimiert `Vec<T>::new()` sowie vielfache Aufrufe von `push` in ein einziges Makro, etwa `vec![1, 2, 3, 4, 5, 6]`, das Sie dann in einer Zuweisung einsetzen.

Konzeptionelle Wiederholungen treten auf, wenn Sie zwar verschiedene konkrete Datenstrukturen deklarieren und einsetzen, diese jedoch immer dem gleichen Aufbau oder der gleichen Funktion folgen. Solcherlei Wiederholungen vermeiden etwa Strukturen oder Enumerationen im Rust-Code: Sie führen Datentypen ein, um logisch zusammenhängende Daten gemeinsam zu verwalten.

Das trifft auch auf Funktionen zu: Wenn ein Verhalten immer wieder mit verschiedenen Eingangswerten auftritt, ziehen Sie die Stellen zu einer Funktion zusammen. Mit der generischen Programmierung (vgl. Kapitel 14, »Generische Programmierung«) definieren Sie einen Ablauf oder eine strukturelle Zusammensetzung sogar für Datentypen, die zum Zeitpunkt der Deklaration unbekannt sind.

Wenn Sie diese Lösungsansätze auf Makros übertragen, sprechen wir nicht über Datenfelder, die sich wiederholen, oder über ein Verhalten, das an verschiedenen Stellen auftritt. Stattdessen entfernen Sie sich von konkreten Datentypen und betrachten eine Kombination von Rust-Elementen von der *Meta-Ebene* aus. Hier bewegt sich auch der Compiler.

Makros haben allerdings eine Schattenseite: Die Kompilierzeiten *können* massiv steigen. Der Compiler verarbeitet Meta-Elemente, indem er sie an den Verwendungs-

orten schrittweise *expandiert*. Je tiefer Sie Makros ineinander verschachteln, desto mehr Schritte führt der Compiler aus.

Die Serialisierung und Deserialisierung als Anwendungsgebiet

Ein Anwendungsbeispiel für Makros: Sie führen ein fachliches Objekt in ihre *Domäne* ein. Daten dieses Objekts betreten Ihre Anwendung an einer Schnittstelle, etwa durch die Response einer Web-API. Dort liegen die Daten im Schnittstellen-Format vor, beispielsweise *JSON*.

Sie müssen das Objekt in eine Rust-Struktur *deserialisieren*. Dazu schreiben Sie einigen Code, der lediglich Datenpunkte verbindet und primitive Werte verschiebt. Das *Domänenobjekt* wird Ihre Anwendung an dieser Schnittstelle ebenfalls verlassen. Sie müssen es also später erneut ins Schnittstellen-Format *serialisieren*. Am Ende haben Sie eine Datenstruktur und zwei Abläufe eingeführt. Einige Trait-Implementierungen stoßen hinzu.

In diesem Szenario setzen Sie Makros ein, sodass Sie nur noch das Domänenobjekt als Datenstruktur implementieren müssen. Die Traits soll der Compiler übernehmen. Dafür ist das `derive`-Attribut zuständig, das Sie bereits in einigen Beispielen gesehen oder verwendet haben – dahinter stehen ebenfalls Makros. Die Serialisierung und Deserialisierung des Domänenobjekts müssen Sie in Rust daher nicht selbst übernehmen. Dafür ziehen Sie das Paket `serde` heran, das Rust-Elemente mithilfe von Makros automatisch übersetzt.

Makros bilden somit die Grundlage für derlei Automatisierungen. Gleichzeitig ersparen Makros es Ihnen, umfassenden Boilerplate-Code zu schreiben. Die Lösung ist in diesem Fall, ein Makro zu definieren, das jedes Feld einer Datenstruktur auf der Meta-Ebene durchläuft und dabei die Verdrahtungen zur Serialisierung und Deserialisierung generiert.

Genau hier liegt der Unterschied zu Rust-Code! Mithilfe des Makros geben Sie dem Compiler den Ablauf vor – etwa so: »Betrachte eine beliebige Datenstruktur und verarbeite jedes Feld. Implementiere außerdem selbstständig die Traits `X` und `Y`.« Das heißt: Sie lösen sich vom Konkreten und bleiben beim Generellen, also demjenigen, das hinter einer spezifischen Datenstruktur steht – dem Meta-Element.

Aus der Sicht des Rust-Codes sind Sie dagegen auf die Benennung *konkreter* Datentypen beschränkt. Sie können dem Compiler demnach mitteilen, dass Sie den *konkreten* Typ `Person` verarbeiten und die *konkreten* Felder `name` und `alter` in Funktionen serialisieren oder deserialisieren wollen. Zusätzlich implementieren Sie die Traits `X` und `Y` *konkret* für den Datentyp `Person`.

Wie Makros Ihren Testcode vereinfachen

Ein weiteres Beispiel stellt Testcode dar. Auch hier tritt eine Kombination aus Datenstrukturen und Verhalten auf: zum Beispiel Testklassen in Form von Test-Cases oder *Setup*- und *Teardown*-Funktionen, die einen Test begleiten. Diese Kombination aus syntaktischen Elementen generiert etwa das `test`-Attribut für Sie (dazu lesen Sie mehr in Kapitel 18, »Automatische Tests und Dokumentation«).

Denken Sie ferner an Testfälle: Nehmen wir an, Sie haben eine Testfunktion geschrieben und möchten nun überprüfen, wie sich das zu testende System in Grenzfällen oder bei sinnlosen Eingangswerten verhält. Jeder Testfall wird eine konzeptionelle Ähnlichkeit aufweisen. Sie müssen die gleichen Vorbedingungen herbeiführen, dieselben Variablen zuweisen usw. Was sich jedes Mal ändert, sind die Testdaten. Den notwendigen Code könnten Sie imperativ ausformulieren und immer wieder mit anderen Werten duplizieren (konzeptionelle Wiederholungen). Oder Sie schreiben ein Makro, das diesen Testablauf einmal beschreibt und das Sie anschließend mit abweichenden Eingangswerten einsetzen.

Während Sie mit einer Funktion einen Teil des Verhaltens imperativ abbilden, bietet hingegen nur das Makro die Fähigkeit, Testdaten als Datentypen deklarativ einzuführen und für den Test zu arrangieren.

17.1.2 Ein Beispiel-Makro

Das deklarative Makro ist einem `match`-Ausdruck nicht unähnlich. Sie geben in beiden Fällen ein Paar aus Muster und Ausführungsblock an. Statt von `match`-Armen sprechen wir allerdings von *Regeln*. Um ein deklaratives Makro einzuführen, benutzen Sie das Makro `macro_rules!`. In Listing 17.1 sehen Sie das deklarative Makro `ausgeben` mit einer einzelnen Regel:

```
macro_rules! ausgeben {
    () => {
        let str = "Leer!";
        println!("{str}");
    };
}
```

Listing 17.1 Das Makro »ausgeben«

Der Code könnte fast als unspektakulär durchgehen. Sonderbar an ihm ist jedoch das Makro zu Beginn: `macro_rules!`. Diese Makro-Form kommt nur vor, wenn Sie deklarative Makros einführen. Man nennt sie auch *exemplarische Makros* (engl. *Macro by Example*), weil der Compiler durch die *Matcher* Beispiele erhält, für die er die jeweiligen *Transcriber* einsetzt. Für einen Aufruf ohne Argumente soll das Makro die Variable `str` deklarieren und deren Wert mit dem Makro `println!` ausgeben.

`macro_rules!` verbindet die Regeln mit einem Bezeichner, hier `ausgeben`. Den Bezeichner setzen Sie beim Aufruf im Rust-Code ein. Listing 17.2 zeigt ein Beispiel in `main`:

```
fn main() {
    ausgeben!();
}
```

Listing 17.2 Das Makro »ausgeben« aufrufen

Auf den Bezeichner eines Makros folgt immer ein Ausrufezeichen, um es als solches im Quellcode hervorzuheben. Sie können Makros, insbesondere die *prozeduralen Makros* (dazu folgt mehr in Abschnitt 17.2, »Prozedurale Makros«), dazu einsetzen, Rust-Code zu generieren und zu verändern. Das Makro ist ungleich mächtiger als die Funktion. Daher ist es wichtig, das Makro auch optisch von einem bloßen Funktionsaufruf zu unterscheiden.

Die Sichtbarkeit des Makros

Wenn Sie das Makro und den Aufrufort in der gleichen Datei, das heißt im gleichen impliziten Modul, einführen, dann müssen Sie auf die Reihenfolge der Deklaration achten. Deklarieren Sie das Makro in diesem Fall stets vor der Benutzung.

Falls Sie das Makro in einem anderen Modul Ihres Crates einführen, müssen Sie es mit dem äußeren Attribut `#[macro_export]` modifizieren. Dieses Attribut hakt das Makro auf der Ebene des Wurzelmoduls ein. Zum Gültigkeitsbereich lesen Sie mehr in Abschnitt 17.1.4, »Der Gültigkeitsbereich«.

17.1.3 Die Meta-Syntax

Makros setzen eine eigene Beschreibungssprache ein. Sie erinnert zwar in manchen Aspekten an *reguläre Ausdrücke*, zum Beispiel dadurch, dass Sie mit den Wiederholungs-*Token* `?`, `+` und `*` arbeiten. Jedoch weicht die Meta-Syntax deutlich von regulären Ausdrücken ab.

Die Klammerung

Zudem geht der Matcher sehr flexibel mit der eigenen Klammerung um. Listing 17.1 zeigt den Matcher als `() => ...`, syntaktisch richtig wären auch `[] => ...` oder `{} => ...`. Als idiomatisch gelten im Matcher jedoch die runden Klammern. Der Matcher hätte den Aufruf in Listing 17.2 in jeder Variante erkannt und dafür den Transcriber eingesetzt. Somit passen `()`, `[]` oder `{}` im Matcher zu `()` im Aufruf.

Diese Flexibilität gilt andersherum genauso: Wenn Sie die idiomatische runde Klammerung im Matcher einsetzen und das Makro mit anderen Klammern aufrufen wollen, übersetzt der Compiler den Code weiterhin. Die Wahlmöglichkeit ist darin

begründet, dass ein Makro sich an den mutmaßlichen Anwendungsbereich anpasst, ohne dort als Fremdkörper aufzufallen. Für das Makro `ausgeben!` sind somit die Formen gültig, die Sie in Listing 17.3 sehen:

```
fn main() {
    ausgeben!(
        // Argumente
    );
    ausgeben! [
        // Argumente
    ];
    ausgeben! {
        // Argumente
    };
}
```

Listing 17.3 Verschiedene Arten der Klammerung beim Makro-Aufruf

Ein Beispiel für eckige Klammern im Aufruf haben Sie bereits mit `vec![…]` kennengelernt. Die Wahl der Klammerung beeinflusst, wie vertraut sich eine Nutzerin oder ein Nutzer mit einem Makro fühlt. Bei `vec!` erinnert die eckige Klammerung etwa an den Indizierungsausdruck `[]`, den eine Datenstruktur durch die Trait-Implementierung von `Index` oder `IndexMut` anbietet. Die Syntax des Makros fügt sich also geschmeidig in den Kontext ein.

Die geschweiften Klammern umschließen Argumente häufig dann, wenn ein Makro wie ein Modul oder ein Implementierungsblock auftritt, also syntaktische Elemente entgegennimmt. Hierbei übersteigt die Verarbeitungskomplexität jedoch eine Grenze, sodass Sie diese Form eher bei prozeduralen Makros sehen. (Dazu folgt mehr in Abschnitt 17.2, »Prozedurale Makros«.)

Die runden Klammern im Makro-Aufruf erinnern an den Aufruf einer Funktion. Verwenden Sie daher diese Klammerung, wenn Ihr Makro wie eine Funktion wirken soll. Beispiele dafür sind `println!()`, `todo!()` oder `unimplemented!()`.

Meta-Variablen

Regeln können Werte über den Matcher aus der Aufrufumgebung entgegennehmen. Dazu müssen Sie sogenannte *Meta-Variablen* deklarieren. Wir machen uns damit schrittweise vertraut und fügen eine weitere Regel zu `ausgeben!` hinzu, die ein einzelnes Argument erwartet.

Dazu deklarieren wir die Meta-Variable `$g` und verweisen im Transcriber-Block darauf. In Listing 17.4 sehen Sie den Code dazu:

```
macro_rules! ausgeben {
    () => {
        let str = "Leer!";
        println!("{str}");
    };
    ($g:expr) => {
        println!("{}", $g);
    };
}
```

Listing 17.4 Meta-Variable und -Fragment in »ausgeben«

Dieses Mal ist der Matcher nicht leer. Die Syntax erinnert an die Deklaration einer Variable, etwa `let g: expr = <…>`. Eine Variable deklariert `$g:expr` zwar auch, das jedoch auf der Meta-Ebene. `$g` wird infolgedessen als Meta-Variable bezeichnet.

Ein deklaratives Makro vergleicht jeden Aufruf mit seinen Regeln. Dabei setzt es den Mustervergleich ein, daher auch der Name *Matcher* (von engl. *to match*). In einem Matcher können Sie nach Belieben Literale einsetzen (siehe Eintrag zum Fragment Literal in der Tabelle 17.1). Eine einfache Regel von `ausgeben` könnte daher auch so aussehen wie in Listing 17.5:

```
macro_rules! ausgeben {
    // ...
    (H R:) => {
        println!("Hallo, Rust");
    };
}

fn main() {
    // Literale passen
    ausgeben!(H R:);
}
```

Listing 17.5 Ein deklaratives Makro vergleicht jeden Matcher mit dem Aufruf.

Meta-Fragmente

Daran schließt sich hinter dem : das *Meta-Fragment* an. `expr` steht für *Expression*, also für einen Ausdruck. Ausdrücke, das wissen Sie aus Kapitel 8, »Anweisungen, Ausdrücke und Muster«, bilden die größte Gruppe syntaktischer Elemente in Rust. In Tabelle 17.1 sehen Sie eine Übersicht der Fragmente.

Fragment	Beschreibung
`item`	Ein syntaktisches Element, etwa eine Struktur, Enumeration, `use`-Direktive oder ein Trait
`block`	Ein Block ({...})
`stmt`	Eine Anweisung
`pat_param`	Abwärtskompatibles Muster-Fragment. Vor der Edition Rust 2021 wurden *Oder-Muster* nicht unterstützt (siehe unten). Dieses Fragment schließt Oder-Muster im Matcher aus.
`pat`	Ein Muster; es unterstützt seit Rust 2021 *Oder-Muster*, etwa `if let Some(a | b) = ...`.
`expr`	Ein Ausdruck
`ty`	Ein Datentyp
`ident`	Der Bezeichner eines syntaktischen Elements oder Musters
`path`	Ein Pfad
`tt`	Kurz für *TokenTree*, zu Deutsch *Token-Baum*. Das ist eine Abfolge von relevanten Token, die Sie in einem Makro interpretieren können. Rust-Code setzt sich aus diesen Token zusammen.
`meta`	Ein Attribut
`lifetime`	Eine Lebenszeit
`vis`	Ein Modifikator der Sichtbarkeit (`pub`)
`literal`	Ein Literal-Ausdruck, etwa `2`, `5.4` oder `"Hallo"`

Tabelle 17.1 Die Meta-Fragmente

Innerhalb des Transcribers greifen Sie mit dem `$`-Zeichen auf ein Meta-Element zu, hier eine Meta-Variable. Weiter unten lernen Sie, wie Sie mit `$(...)` sogar einen Code-Bereich mit einer Meta-Variable verknüpfen, etwa um ihn für jedes Argument zu wiederholen.

Einen Hinweis darauf, dass `$g` kein Rust-Bezeichner ist, liefert `println!` in Listing 17.4. In Rust-Code kann der Format-String `"{}"` Bezeichner aus der Umgebung auflösen. Mit einer Meta-Variable ist das allerdings nicht kompatibel: `"{$g}"` verursacht einen Fehler.

Im Matcher trennen Sie Meta-Variablen mit Kommas voneinander. Die nächste Regel fordert etwa neben dem Ausdruck auch einen Datentyp (Fragment `ty`), wie Sie in Listing 17.6 sehen:

```
macro_rules! ausgeben {
    () => {
        let str = "Leer!";
        println!("{str}");
    };
    ($g:expr) => {
        println!("{}", $g);
    };
    ($g:expr, $t:ty) => {
        println!(
            "{}, Typ: {}",
            $g,
            core::any::type_name::<$t>()
        );
    };
}
```

Listing 17.6 Mehrere Argumente durch Kommas trennen

Die zweite Meta-Variable ist das Datentyp-Fragment `ty` (siehe Tabelle 17.1). Wir nutzen die Funktion `type_name`, um eine textuelle Repräsentation des Typs auszugeben. Beachten Sie, wie `$t` in der Position eines generischen Parameters auftritt.

»core« statt »std«

Der vollständig qualifizierte Pfad setzt nicht in der `std` an, also der Standardbibliothek, sondern durch `core` am Rust-Core. Andernfalls könnten Sie das Makro nur in jenen Kontexten einsetzen, die die Standardbibliothek einbinden – ein Rust-Projekt kann darauf allerdings mit dem inneren Attribut `#![no_std]` verzichten.

Nun sehen wir uns den Wechsel zur Funktion `main` an. Wie in Listing 17.7 rufen Sie Ihr Makro `ausgeben!` jeweils mit unterschiedlichen Argumenten auf:

```
fn main() {
    ausgeben!();
    ausgeben!("Hallo");

    ausgeben!("Hallo", i32);
    // Hallo, Typ: i32
```

```
    ausgeben!("Hallo", String);
    // Hallo, Typ: alloc::string::String
}
```

Listing 17.7 Ein Makro mit zwei Argumenten aufrufen

Mehrere Argumente und die Typdeklaration durch Makros

Dieses Verhalten hätten Sie ebenfalls mit einer generischen Funktion abbilden können. Lassen Sie uns daher einen Versuch wagen, der auch die generische Programmierung hinter sich zurücklässt.

Dazu fügen Sie neben dem `macro_rules!`-Eintrag von `ausgeben!` ein weiteres deklaratives Makro hinzu. Das soll mit jedem Aufruf Datenstrukturen einführen. Dabei konfigurieren Sie den Bezeichner sowie den Datentyp und den Bezeichner eines Felds mit Makro-Argumenten. Listing 17.8 zeigt die Regel:

```
macro_rules! ausgeben {
    …
}

macro_rules! neue_struktur {
    (
         $name:ident,    // Name der Struktur
         $feld:ident,    // Name des Felds
         $feld_typ:ty,   // Typ des Felds
         $feld_init:expr // Wert des Felds
     ) => {{
        #[derive(Debug)]
        struct $name {
            $feld: $feld_typ,
        }

        impl $name {
            pub fn new() -> $name {
                $name { $feld: $feld_init }
            }
        }

        $name::new()
    }};
}
```

Listing 17.8 Eine Datenstruktur einführen und initialisieren

Das Beispiel demonstriert eine Vielzahl neuer Punkte. Zum einen, dass Sie pro deklarativem Makro einen neuen `macro_rules!`-Block einführen. `ausgeben!` und `neue_struktur!` stehen nicht in Relation zueinander, sondern teilen nur das Modul.

Im Matcher der einzigen Regel kommen drei verschiedene Fragmente vor. `ident` haben Sie bislang bislang nicht eingesetzt. Die Meta-Variablen `$name` und `$feld` treten an die Stelle der konkreten Bezeichner von Struktur und Feld. Den Datentyp der Felddeklaration bestimmt `$feld_typ`.

Der Struktur-Bezeichner erscheint im inhärenten Implementierungsblock von Listing 17.8 erneut, insbesondere in der `new`-Funktion. Der Ausdruck `$feld_init` ist ein `String` und wird in das Feld bewegt.

Die Fragmente »ident« und »ty« im Vergleich

Im Matcher des Makros `neue_struktur!` in Listing 17.8 wäre statt `$feld_typ:ty` auch `$feld_typ:ident` kein Fehler gewesen. Sie können `ident` daher an allen Stellen einsetzen, wo Sie ein `ty` benötigen – Typ-Fragmente sind eine Teilmenge der Bezeichner-Fragmente. Bezeichner schließen aber zusätzliche syntaktische Elemente ein, unter anderem Pfadsegmente.

Nehmen wir an, Sie ändern das Fragment von `feld_typ` in Listing 17.8 von `ty` auf `ident`. Dieser Aufruf ist gültig:

```
let p = neue_struktur!(Person, name, String, "Freda".into());
```

Der nächste Aufruf nutzt zwar gültige Argumente, die kann der Compiler im Makro jedoch nicht mehr expandieren:

```
let p = neue_struktur!(Person, name, core, "Freda".into());
```

Mit dem Fragment `ty` stellen Sie explizit klar, dass Sie den Bezeichner eines Datentyps und nicht etwa den einer Variable oder eines Pfadsegments einfordern.

Es empfiehlt sich daher, mit Fragmenten so eng wie möglich umzugehen, um die Möglichkeiten einzuschränken – wenn Sie den Spielraum nicht explizit ausnutzen. Ansonsten kann es dazu kommen, dass ein Makro in ersten Tests oder Verwendungen zwar funktioniert, später aber Fehler verursacht, obwohl die Nutzerin oder der Nutzer syntaktisch korrekte Argumente an das Makro übergeben hat.

Eine zunächst ebenfalls gültige Version wäre `$feld_typ:path`. Sie kann jedoch ebenso schnell zu einem Fehler führen:

```
// Ok, Pfad verweist auf Datentyp
let p = neue_struktur!(Person, name, std::string::String, "Freda".into());
// Fehler:
// Ein Pfad, der allerdings keinen Datentyp,
// sondern ein Modul adressiert
let p = neue_struktur!(Person, name, std::string, "Freda".into());
```

Mit geschweiften Klammern zum Block expandieren

Zuletzt soll der Aufruf der `new`-Funktion für die Struktur mit dem Bezeichner `$name` eine Instanz an den Aufrufer des Makros zurückgeben. Das erfordert die zusätzlichen geschweiften Klammern, die innerhalb des Transcribers auftauchen. Hierdurch wird der Compiler das Makro in einen Block expandieren und Blöcke weisen Rückgabewerte auf! Ohne den Block landet die Rückgabe eines Werts zunächst in einer temporären Variable im expandierenden Makro, um unmittelbar danach vom Stack zu verschwinden.

Denken Sie daher daran, Ihren Transcriber-Inhalt in einen zusätzlichen Block zu bewegen, damit das Makro einen Wert produziert. Die geschweiften Klammern weisen allerdings einen weiteren Effekt auf – dazu gleich mehr. In Listing 17.9 sehen Sie einen Aufruf des neuen Makros:

```
fn main() {
    // ...
    let p = neue_struktur!(Person, name, String, "Freda".into());
    println!("Instanz von Person: {p:?}");
    // Instanz von Person: Person { name: "Freda" }
}
```

Listing 17.9 Der Aufruf von »neue_struktur!«

Sie haben in nur einem einzigen Aufruf eine neue Struktur deklariert, initialisiert und die Instanz zurückgegeben. An die soeben eingeführte Datenstruktur `Person` kommen Sie aber nicht heran, wenn Sie von außen darauf zugreifen möchten! Das ist eine Nebenwirkung der zusätzlichen Klammerung. Wir erstellen in Listing 17.10 ein weiteres Makro, das dieses Mal auf den Rückgabewert verzichtet:

```
// ...
macro_rules! struktur {
    (
        $name:ident,
        $feld:ident,
        $feld_typ:ty,
        $feld_init:expr
    ) => {
        #[derive(Debug)]
        struct $name {
            $feld: $feld_typ,
        }

        impl $name {
            pub fn new() -> $name {
```

```
                $name { $feld: $feld_init }
            }
        }
    };
}

fn main() {
    // ...
    let p = neue_struktur!(Person, name, String, "Freda".into());
    println!("Instanz von Person: {p:?}");
    // Instanz von Person: Person { name: "Freda" }

    // Fehler, Person ist nicht bekannt
    println!("Typ: {}", any::type_name::<Person>());

    struktur!(PersonNeu, name, String, "Fred".into());

    // Ok, Struktur wurde im Gültigkeitsbereich eingefügt
    let p = PersonNeu::new();
    println!("PersonNeu: {:?}", p);
    // PersonNeu: PersonNeu { name: "Fred" }
}
```

Listing 17.10 Datentypen ohne zusätzliche geschweifte Klammern einführen

Ohne die doppelten geschweiften Klammern erreicht die neue Struktur tatsächlich den Gültigkeitsbereich, in dem Sie sie aufrufen. Nach dem Aufruf von `struktur!` sind Zugriffe auf den Datentyp `PersonNeu` gültig, etwa auf die assoziierte Funktion `new`.

Bezeichner und Elemente auf das Makro beschränken

Ob Sie die doppelt geschweiften Klammern einsetzen oder nicht, haben wir von einem Rückgabewert abhängig gemacht. Dieses Merkmal können Sie jedoch unabhängig von diesem einsetzen, um jegliche Elemente, die Sie im Transcriber eingeführt haben, auf ebendiesen Transcriber zu beschränken. So vermeiden Sie mögliche Überdeckungen oder Namenskonflikte! Den Punkt greifen wir in Abschnitt 17.1.5, »Makro-Hygiene«, erneut auf.

Makro, Move und Referenzen

Der Zugriff aus dem Makro auf die Umgebung bleibt nicht ohne Folgen. Wie für Rust-Code üblich, wendet der Compiler entweder Move an oder prüft die Einhaltung der Borrow-Regeln. Dazu blicken wir erneut auf das Makro `ausgeben!`, insbesondere auf die zweite Regel (siehe Listing 17.11):

```
macro_rules! ausgeben {
    () => {
        let str = "Leer!";
        println!("{str}");
    };
    ($g:expr) => {
        println!("{}", $g);
    };
    // ...
}

// ...
fn main() {
    let str = String::from("Hallo, Rust");

    ausgeben!(str);
    // Ok oder Fehler?
    ausgeben!(str);
}
```

Listing 17.11 Nutzt die Meta-Variable den Move oder eine Referenz?

Wird der Compiler einen Fehler aufzeigen, da `str` mit der ersten Nutzung bewegt wurde? In *diesem* Fall: nein! Doch sobald wir die zweite Regel nur ein kleines bisschen anpassen, wendet sich das Blatt, wie Sie in Listing 17.12 sehen:

```
macro_rules! ausgeben {
    () => {
        let str = "Leer!";
        println!("{str}");
    };
    ($g:expr) => {
        println!("{}", $g);
        let s: String = $g;
    };
    // ...
}
// ...

fn main() {
    let str = String::from("Hallo, Rust");

    ausgeben!(str);
```

```
    // Fehler! Der Wert wurde schon bewegt!
    ausgeben!(str);
}
```

Listing 17.12 Zuweisung des eingefangenen Strings an eine lokale Variable

Der Transcriber des Makros unterliegt den üblichen Regeln

Ein Fehler besteht auch darin, `$g` zweimal zuzuweisen, wie Sie folgend an Listing 17.13 erkennen:

```
($g:expr) => {
    println!("{}", $g);
    let s: String = $g;
    // Fehler, schon bewegt! - Aber nur wenn verwendet!
    let s: String = $g;
};
```

Listing 17.13 Doppelter Move im Transcriber

Da der Compiler ein Makro am Aufrufort expandiert, sehen Sie den Fehler erst bei der Benutzung. Das birgt die Gefahr, dass der Fehler unbemerkt bleibt, wenn Sie die Regel nicht im Quellcode verwenden! Das ist insbesondere für Autorinnen oder Autoren einer Bibliothek kritisch. Achten Sie aus diesem Grund auf eine hinreichende Testabdeckung.

Tatsächlich entscheidet der Transcriber darüber, ob es ausreicht, den Wert als Referenz zu beziehen oder ob der Compiler das Eigentum übertragen muss. Die geteilte Referenz wird so lange ausreichen, bis damit eine Nutzung des Werts nicht mehr möglich ist. Das gilt ebenso für exklusive Referenzen! Ein weiteres Beispiel sehen Sie in Listing 17.14:

```
macro_rules! ausgeben {
    // ...
    ($g:expr) => {
        println!("{}", $g);
        // Muss veränderliche Referenz anfordern
        $g.make_ascii_uppercase();
    };
    // ...
}
// ...
fn main() {
    let mut str = String::from("Hallo, Rust");
    let str_ref = &str; // geteilte Referenz aktiv
```

```
    // Fehler!
    ausgeben!(str);     // Bezieht veränderliche Referenz!
    // Fehler!
    ausgeben!(str);

    println!("Noch aktiv: {str_ref}");
}
```

Listing 17.14 Auch Makros müssen sich den Borrow-Regeln beugen.

Typsicherheit und Seiteneffekte

In Listing 17.14 verwendet der Transcriber die Methode `make_ascii_uppercase`. Doch woher weiß die Regel eigentlich, dass `$g` ein `String` ist? Den Datentyp von `$g` findet der Compiler erst heraus, wenn Sie das Makro am Aufrufort expandieren. Die Verwendung mit `String` macht `$g` zu einem `String`, bis dahin ist die Meta-Variable nur *ein* Ausdruck.

Diese Spezialisierung bleibt zunächst unproblematisch. Aber sobald Sie das gleiche Makro mit einem abweichenden Datentyp als Ausdruck verwenden, kommen Sie nicht mehr weiter. Listing 17.15 zeigt dazu ein Beispiel:

```
fn main() {
     ausgeben!("Hallo".to_string());

     // Fehler, nur ein String ist kompatibel
     ausgeben!(42_i32);
}
```

Listing 17.15 Die Regel in »ausgeben!« erwartet einen String.

Der Compiler wird den Typ immer auf den kleinsten gemeinsamen Nenner bringen, um das Makro so vielseitig nutzbar wie möglich zu machen. Das Verhalten ist analog zur Präferenz für Referenzen, die wir im vorherigen Abschnitt diskutiert haben.

Wenn Sie die Zeile `$g.make_ascii_uppercase()` aus Listing 17.14 entfernen, kompiliert Listing 17.15 wieder. Dann erlegt der Compiler dem Makro nur die Forderung auf, dass `$g` das Trait `Display` implementiert. Beachten Sie im Umgang mit Makros daher, dass sowohl die Verwendung als auch der Transcriber den Typ eines Ausdrucks einschränken können. Das gilt ebenso für andere Meta-Fragmente, bei denen eine Mehrzahl an konkreten Möglichkeiten denkbar ist.

Da auch Funktionen, Schleifen oder anderes Verhalten zu den Ausdrücken zählen, müssen Sie im Umgang mit Meta-Variablen dieses Fragments auf Seiteneffekte achten. Betrachten Sie dazu das Makro aus Listing 17.16 und dessen Verwendung:

```
macro_rules! funktion {
    ($f: expr) => {
        let a = $f;
        let b = $f;

        assert_eq!(a, b);
    };
}
fn main() {
    let mut v = vec![1, 2, 3];
    // Ok
    funktion!(v[0]);

    // Fehler! Ruft pop zweimal auf
    funktion!(v.pop());
}
```

Listing 17.16 Jeder Zugriff auf »$f« führt die Funktion abermals aus.

Der erste Aufruf von `funktion!` ist zwar unschön, weil er das erste Element des Vektors doppelt abruft. Die Methode ist jedoch *idempotent*, das heißt, dass sie bei jeder Ausführung das gleiche Ergebnis liefert und dabei den Zustand des Aufrufziels nicht verändert. Dagegen verändert `pop` den Vektor, indem die Methode bei jedem Aufruf das oberste Element vom Vektor-Stack entfernt.

Achten Sie daher darauf, dass der Compiler den Rückgabewert eines Methoden- oder Funktionsaufrufs nicht einer temporären Variable zuweist. Stattdessen greift das Makro sich die Methode oder Funktion und führt sie im Zweifelsfall mehrmals aus!

Vermeiden Sie diese Fehlerquelle im Vorhinein, indem Sie eine Meta-Variable des Ausdruck-Fragments nur einmal einsetzen und fortan referenzieren. Demnach ist die Version des Makros `funktion!`, die Sie in Listing 17.17 sehen, vor Seiteneffekten sicher:

```
macro_rules! funktion {
    ($f: expr) => {
        let a = $f;
        let b = &a;
        assert_eq!(a, *b);
    };
}

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

```
    // Ok
    funktion!(v[0]);

    // Ok
    funktion!(v.pop());
}
```

Listing 17.17 Die Meta-Variable »$f« nur einmal verwenden

Wiederholung von Code im Makro

Wenn Sie ein Makro entwerfen, können Sie sich dafür entscheiden, eine festgelegte Anzahl Argumente zu erwarten. Stattdessen ist es Ihnen auch möglich, eine beliebige Anzahl zu verarbeiten. So geht etwa das `println!`-Makro vor, das je nach Aufruf eine Vielzahl Argumente zur Ausgabe übernimmt.

Sie unterstützen die Wiederholung einer Meta-Variable, indem Sie sie mit der Klammerung `$( ... )` umschließen. Optional können Sie ein Trennzeichen angeben, das zwischen den einzelnen Argumenten auftauchen soll. Der Matcher wird dann nur jene Makro-Aufrufe mit der Regel verbinden, die exakt dieses Trennzeichen einsetzen. Verzichten Sie stattdessen darauf, können Sie Argumente direkt aufeinanderfolgen lassen (das ist problematisch bei Literal-Fragmenten wie `ii` statt `i i` oder `42` statt `4 2`) oder Sie setzen Leerzeichen ein.

Zusätzlich stellt die Meta-Syntax Ihnen spezielle *Wiederholungs-Token* bereit, mit denen Sie die Pluralität eines Arguments steuern. Eine Übersicht zu diesen Token sehen Sie in Tabelle 17.2.

Token	Beschreibung
`?`	Das vorherige Element ist optional, darf aber maximal einmal auftreten.
`+`	Mehrere Elemente sind möglich, aber mindestens eins muss vorhanden sein.
`*`	Ein Element, mehrere oder keins dürfen vorhanden sein.

Tabelle 17.2 Die Wiederholungs-Token der Meta-Syntax

Die drei Elemente setzen sich zu dieser Wiederholungssyntax zusammen:

```
$( <Variable> : <Fragment> ) <Trennzeichen> <Wiederholungs-Token>
```

Wir probieren diese Verarbeitung mit `ausgeben!` aus und führen hierzu im Makro eine neue Regel ein. Die neue Regel wird keinen Ausdruck (leer), einen oder mehrere Ausdrücke (*-Token) entgegennehmen. Bereitgestellte Argumente müssen implizit das

Trait `Display` implementieren. In Listing 17.18 sehen Sie die Erweiterung von `macro_rules!`:

```
macro_rules! ausgeben {
    // ...
    (
        // $g:expr in $(...) einschließen
        $( $g:expr )* // Kein Trennzeichen, *-Token
    ) => {

        // Diesen Codeblock in Abhängigkeit von &g wiederholen
        $(
            println!("Ausgabe: {}", $g );
        )*
    };
}

fn main() {
    // Mögliche Verwendungen
    ausgeben_!();
    ausgeben_!("Hallo, Rust!");
    ausgeben_!(2 "Hallo" "," "Rust" "!");
}
```

Listing 17.18 Die Wiederholung eines Ausdrucks in »ausgeben«

Die Wiederholung eines Makro-Teils im Transcriber dirigiert der Matcher. So hängt die Anzahl der Vervielfachung davon ab, wie viele Argumente Sie übergeben. Daher müssen Sie eine oder mehrere Meta-Variablen entsprechend kennzeichnen. Die Syntax besteht wie erwähnt aus der Klammerung mit `$(…)`, einem optionalen Separator (wie dem Komma zwischen Argumenten im Funktionsaufruf) und dem Wiederholungs-Token.

In Listing 17.18 deklariert das Makro `ausgeben!` eine Wiederholung der Meta-Variable `$g` mit dem Meta-Fragment `expr`. Auf einen Separator verzichtet das Beispiel und fügt stattdessen direkt das Wiederholungs-Token `*` an. Einige Beispiele in der Funktion `main` zeigen, wie flexibel die Regel mit den Argumenten umgeht. Beachten Sie insbesondere, wie die Durchmischung verschiedener Datentypen möglich ist (`2 "Hallo" …`).

Die spezielle Syntax für Wiederholungen (`$(…)`) findet sich auch im Transcriber wieder. Wenn Sie dagegen nur `$g` verwenden, würde der Compiler einen Fehler melden (siehe Listing 17.19). Die Wiederholungen müssen in Matcher und Transcriber übereinstimmen!

```
( $( $g:expr )* ) => {
    // Fehler, behandelt nicht alle Wiederholungen
    println!("Ausgabe: {}"
, $g );
};
```

Listing 17.19 Ein Fehler tritt auf, falls der Transcriber die Wiederholungen nicht berücksichtigt.

Achten Sie ferner darauf, dass die Wiederholungs-Token in Matcher und Transcriber *übereinstimmen*. Sie könnten zwar * im Matcher und + im Transcriber einsetzen, doch das führt zu einem Fehler! Ein Makro-Aufruf ohne Argument würde auf den Transcriber treffen, der mindestens eine Wiederholung eines Codeblocks expandieren möchte. Listing 17.20 zeigt ein Beispiel:

```
(  $( $g:expr )*  ) => {
    $(
        println!("Ausgabe: {}", $g );
    )+ // Fehler, Wiederholungs-Token stimmt nicht überein

    // ... mit $i arbeiten
};
```

Listing 17.20 Sie müssen Wiederholungs-Token in Matcher und Transcriber aufeinander abstimmen.

Falls Sie mehrere Meta-Variablen im Matcher derselben Regel einsetzen wollen und eine davon Wiederholungen ermöglichen soll, dann müssen Sie entweder auf die Reihenfolge achten. Dann führen Sie die wiederholte Meta-Variable am Ende des Matchers auf. Oder Sie müssen nach der wiederholenden Meta-Variable ein Trennzeichen einfügen. Listing 17.21 zeigt beide Varianten im Vergleich:

```
( $i:ident $( $g:expr )* ) => {
    $(
        println!("Ausgabe: {}", $g );
    )*

    // ... mit $i arbeiten
};

// Alternative: Mit Trennzeichen
( $( $g:expr )*, $i:ident ) => {
    $(
        println!("Ausgabe: {}", $g );
```

```
    )*

    // ... mit $i arbeiten
};
```

Listing 17.21 Eine wiederholte Meta-Variable am Schluss des Matchers einfügen

Makros unterstützen die Wiederholung mehrerer Meta-Variablen in einer Regel. Sie können die Variablen anschließend getrennt in eigenständigen Wiederholungsblöcken im Transcriber einsetzen – etwa so wie in Listing 17.22:

```
macro_rules! ausgeben {
    ( $( $t:ty )*, $( $g:expr )*  ) => {
        $(
            println!("Ausgabe: {}", $g );
        )*

        $(
            println!(
                "Typname: {}",
                core::any::type_name::<$t>()
            );
        )*
        // ... mit $t arbeiten
    };
}
// ...

fn main() {
    ausgeben!(i32 String, "Ein Ausdruck");
}
```

Listing 17.22 Zwei unterschiedliche Wiederholungen

Falls Sie die Variablen im gleichen Wiederholungsblock verwenden möchten, müssen Sie beachten, dass beide Meta-Variablen die gleiche Argumentanzahl aufweisen. Wenn etwa `$t` mehr Durchläufe als `$g` verlangt, würde der Zugriff auf `$g` nicht mehr sicher sein, da der Compiler mit leeren Händen dasteht. In Listing 17.23 sehen Sie dazu ein Beispiel:

```
macro_rules! gemeinsam {
    ( $( $t:ty )*, $( $g:expr )*  ) => {
        $(
            println!("Ausgabe: {}", $g );
```

```
            println!(
                "Typname: {}",
                core::any::type_name::<$t>()
            );
        )*
    };
}
// ...

fn main() {
    // Ok, gleiche Anzahl der Argumente
    gemeinsam!(i32 String, "Zwei" "Ausdrücke");

    // Fehler, abweichende Anzahl der Argumente
    gemeinsam!(String String, "Ein Ausdruck"); // 2x $t, 1x $g
}
```

Listing 17.23 Achten Sie auf die gleiche Anzahl der Argumente in gemeinsamen Wiederholungsblöcken

17.1.4 Der Gültigkeitsbereich

Der Compiler löst ein Makro wie jedes andere syntaktische Element im lokalen Modul oder Gültigkeitsbereich auf. Achten Sie darauf, dass Sie ein Makro stets vor der ersten Verwendung deklarieren müssen. Andere Elemente wie Strukturen können Sie dagegen auch dann einsetzen, wenn die Deklaration später erfolgt. Sollte der Compiler das Makro auf diesem Weg nicht finden, untersucht er die Pfade, die Sie im Modul eingebunden haben. Sie können Makros also stets über einen Pfad aufrufen.

Untermodule sehen die Makros aller Elternmodule, einschließlich des Wurzelmoduls. Ein Makro, das Sie im gesamten Crate verwenden möchten, können Sie demzufolge bequem im Wurzelmodul einhängen. Dazu setzen Sie das Attribut `#[macro_export]` ein, durch das der Compiler ein Makro unabhängig vom eigentlichen Ort der Deklaration in das Wurzelmodul verlegt.

Die Technik der Überschattung (engl. *Shadowing*) kennen Sie von Variablen. Auch Makros überschatten einander, sodass der Compiler, vom Aufrufort gesehen, immer die nächstliegende oder letzte Deklaration referenziert. Das folgende Listing 17.24 zeigt ein Beispiel:

```
macro_rules! todo {
    ($prio:literal) => {
        println!("Noch zu erledigen, Priorität {}", $prio);
        // Auf Original verweisen
```

```
        core::todo!();
    };
}

fn main() {
    // Mit Angabe der Priorität
    todo!(1);
}
```

Listing 17.24 Makros überdecken einander.

17.1.5 Makro-Hygiene

Was in deklarativen Makros passiert, bleibt in deklarativen Makros – jedenfalls zum Teil. Ein expandiertes Makro bildet einen Kokon. Nur Deklarationen oder Änderungen an Datenstrukturen durchdringen diese Isolationsschicht, während etwa Variablen und andere syntaktische Elemente ausschließlich im Inneren gültig sind. Rust bezeichnet diese Isolation als *Hygiene*. Wenn der Compiler ein Makro in einen Gültigkeitsbereich expandiert, bleiben dessen Variablen, Funktionen usw. unberührt oder unbeeinflusst.

Die Isolationsschicht des Makros ist, auf das Bild der Hygiene übertragen, wie der Latexhandschuh in der medizinischen Behandlung. Sie interagieren mit dem Objekt, Verunreinigungen (durch Überschattungen oder Namenskonflikte im Gültigkeitsbereich) sind jedoch ausgeschlossen.

Dass neue Datenstrukturen oder Veränderungen an existierenden Datentypen in den Gültigkeitsbereich der Expansion hineinreichen, ist ein Entwurfsmerkmal der Makros. Denken Sie etwa an die Attribute `derive` oder `test` (für Testfunktionen), deren Zweck darin besteht, Boilerplate-Code zu vermeiden, wenn Sie Datenstrukturen um gewisse Fähigkeiten erweitern möchten. Weiter oben in Abschnitt 17.1.3, »Die Meta-Syntax«, und zwar im Abschnitt »Mit geschweiften Klammern zum Block expandieren«, haben Sie allerdings mit den doppelten geschweiften Klammern im Transcriber eine Möglichkeit kennengelernt, mit der Sie den gedanklichen Kokon abdichten. Dann können ihn auch Deklarationen von Datenstrukturen nicht durchdringen.

17.2 Prozedurale Makros

Der vorherige Abschnitt hat Sie Stück für Stück durch die Meta-Syntax und die Eigenschaften der deklarativen Makros geführt. Das deklarative Makro schöpft jedoch nicht alle Möglichkeiten aus, die das Konzept umfasst. Diese Makro-Art ist selbst eine

Abstraktion, weshalb sie in der Erscheinung einer Funktion auftaucht (`println!(…)`, `ausgeben!(…)` usw.) und nicht zuletzt hinsichtlich der Hygiene einigen Einschränkungen unterliegt.

Im Gegenzug erhalten Sie ein Werkzeug, das Sie mit wenig Aufwand dazu einzusetzen, zur Kompilierzeit mithilfe eines Makro-Aufrufs weiteren Code zu erzeugen. Das Makro `vec!` der Standardbibliothek ist dafür ein anschauliches Beispiel.

Unterhalb dieser Abstraktionsschicht liegen die *prozeduralen Makros*. »Prozedural« daher, weil Sie reguläre Rust-Funktionen schreiben, die nicht in Datenstrukturen gebunden sind. Daher erinnert dieser Entwurf an die *prozedurale Programmierung*, die vor der objektorientierten vorherrschte.

Diese Funktionen verarbeiten *Token-Ströme* als Eingabe und geben einen Token-Strom zurück. Ein Token-Strom besteht aus Rust-Code, den der Compiler bislang nicht interpretiert hat. Der Begriff *Token* steht in der *lexikalischen Analyse* für eine Zeichenkette, der eine Bedeutung in diesem Kontext zugeteilt wird. Die lexikalische Analyse ist wiederum der Prozessschritt, in dem der Compiler Ihren Quellcode, den Sie im Editor als bloße Zeichenketten eingetippt haben, in Token übersetzt.

Ein Beispiel: Anhand der Buchstaben »fn« erkennt der Compiler in Rust das Funktions-Token. C++ würde die Zeichenkette hingegen als bloßen Bezeichner behandeln. Wenn eine Programmiersprache ein Token definiert, dann weist sie einer syntaktischen Einheit (dem Token `fn`) eine semantische Bedeutung zu (eine Funktionsdeklaration).

Ein prozedurales Makro verarbeitet demnach eine Token-Kette und ist darin einem Compiler nicht unähnlich. Die Hauptaufgabe dabei ist das Parsen der Token. Sobald Sie den Token-Strom geparst haben, können Sie Bezeichner oder ganze syntaktische Elemente wie Strukturen oder Funktionen erkennen und bei Bedarf sogar manipulieren. Ein Beispiel: Sie nehmen den Token-Strom `pub fn schreibe() {}` entgegen. Die ersten drei Token sind der Sichtbarkeit-Modifikator, die Funktion mit `fn` und der Bezeichner der Funktion. Im prozeduralen Makro könnten Sie die Sichtbarkeit entfernen oder ein Suffix an den Bezeichner hängen.

Das Parsen ist allerdings nicht einfach, da Sie die *Sprach-Grammatik* von Rust voll oder zum Teil abbilden müssen. Deshalb greifen Sie zu Paketen wie `syn`, `proc-macro2` und `quote`. Diese Abhängigkeiten nehmen Ihnen das Parsen ab. Dann können Sie direkt mit Token arbeiten, die den Meta-Fragmenten der deklarativen Makros nicht unähnlich sind. Mit `quote` schreiben Sie sogar Rust-Code, den Sie als Zeichenkette und damit als Token-Strom im Programm weiterverarbeiten – Code im Code. Das Paket `quote` erleichtert es Ihnen daher, einen Ausgabe-Token-Strom zu erzeugen. Diese drei Pakete gehören zu den am häufigsten heruntergeladenen Artefakten auf *crates.io*.

Der Fokus liegt auf den Makro-Funktionen

In diesem Abschnitt besprechen wir drei Arten von prozeduralen Makros. Weil die Bearbeitung der `TokenStream`s ein Thema für sich ist, fokussieren wir uns stattdessen auf die Einführung der Makros und deren Aufruf. Da Sie ohnehin nicht ohne Pakete wie `syn` arbeiten sollten, verweise ich auf die umfangreiche Dokumentation des Pakets unter *https://docs.rs/crate/syn/latest*. Spannend und daher erwähnenswert sind außerdem die Übungen im Workshop *https://github.com/dtolnay/proc-macro-workshop*.

17.2.1 Für prozedurale Makros müssen Sie eine Bibliothek anlegen

Prozedurale Makros arbeiten enger mit dem Compiler zusammen als regulärer Rust-Code. Das spiegelt sich nicht zuletzt darin wider, dass Sie ein neues Bibliotheks-Crate anlegen müssen, das explizit und *ausschließlich* prozedurale Makros aufnimmt.

Erstellen Sie daher neben Ihrem Beispiel-Projekt eine neue Bibliothek oder nutzen Sie den Unterordner *examples* der Bibliothek. Wir entscheiden uns in diesem Abschnitt für Letzteres. Erstellen Sie die Bibliothek mit diesem Kommando:

```
cargo new prozedural --lib
```

Dann erstellen Sie den Ordner *examples* im Projektverzeichnis. An dieser Stelle legen wir für jede Art des prozeduralen Makros ein neues Beispiel ab. Jede Rust-Datei auf der ersten Dateiebene unterhalb von *examples* stellt ein ausführbares Crate dar. Anders ausgedrückt: Wenn Sie eine Datei erstellen, führen Sie implizit einen eigenständigen Modulbaum ein, und somit müssen Sie im Wurzelmodul die Funktion `main` definieren.

Wir probieren diesen Aufbau zunächst mit einem Beispiel-Crate aus, das nicht mit Makros zusammenhängt. Führen Sie unterhalb von *examples* die Datei *hallo_beispiel.rs* ein. Dann schreiben Sie die Funktion `main` aus Listing 17.25:

```
// examples/hallo_beispiel.rs

fn main() {
    println!("Hallo, Rust im Beispiel!");
}
```

Listing 17.25 Ein Wurzelmodul im Unterverzeichnis »examples«

Der Ordner *examples* ist dafür gedacht, beispielhafte Verwendungen Ihres Programms oder Ihrer Bibliothek zu präsentieren. Ein Beispielprogramm rufen Sie spezifisch mit dem Kommando `cargo run --example <Name>` auf. Der Name entspricht dem Dateinamen ohne die Endung *rs*, hier etwa `cargo run --example hallo_beispiel`.

Bevor wir mit der Diskussion der prozeduralen Makro-Arten beginnen können, müssen Sie noch Ihre Bibliothek für die Makros vorbereiten. Im Unterschied zu deklarativen Makros sind die prozeduralen enger mit dem Compiler verzahnt – darauf habe ich schon zuvor hingewiesen. Die Standardbibliothek liefert mit `proc_macro` sogar ein eigenes Crate aus, das Sie in Ihrem Bibliotheks-Crate aufgreifen müssen. Dazu fügen Sie der Manifestdatei *Cargo.toml* (vergleiche Abschnitt 13.3.3, »Das Paket«) den Eintrag hinzu, den Sie in Listing 17.26 sehen:

```
[package]
# ...

[lib]
proc_macro = true

[dependencies]
# ...
```

Listing 17.26 Der Konfigurationsschalter »proc_macro«

Zusätzlich tragen wir wie in Listing 17.27 die Abhängigkeiten `syn` und `quote` ein, da wir auf sie im Verlauf unserer Betrachtungen noch zurückkommen werden:

```
# ...
[dependencies]
syn = { version = "2.0.41", features = ["full"] }
quote = "1.0.33"
```

Listing 17.27 Abhängigkeiten für die Entwicklung prozeduraler Makros

17.2.2 Die drei prozeduralen Makro-Arten

Manche Teile von Rust wirken zum Teil wie Magie, zum Beispiel die Attribute. Abschnitt 13.3.9, »Attribute«, hat Ihnen zwar aufgezeigt, was Attribute sind und wie Sie sie einsetzen – doch was dahintersteckt, ist bislang offen. Indem Sie die prozeduralen Makros untersuchen, lüftet sich nicht nur dieser Schleier.

Wir gehen der Reihe nach durch *Funktions-*, *Attributs-* und eigene *derive-Makros*. Mit Letzterem definieren Sie zusätzliche und eigene Implementierungen des `derive`-Makros, sodass Sie etwa `#[derive(IhrBezeichner)]` einsetzen können.

Funktion-ähnliche prozedurale Makros

Diese Art Makros können Sie am ehesten mit den deklarativen Makros vergleichen, die Sie in Abschnitt 17.1, »Deklarative Makros«, kennengelernt haben. Die Funktion-

ähnliche prozedurale Variante erlegt Ihnen jedoch keine Einschränkungen der *Hygiene* auf (vergleiche Abschnitt 17.1.5, »Makro-Hygiene«).

Erstellen Sie zunächst ein neues Beispielprogramm im Unterordner *examples* (siehe den vorherigen Abschnitt), zum Beispiel, indem Sie dort die Datei *funktions_makro.rs* anlegen und die Funktion `main` als Eintrittspunkt definieren. Zur Implementierung des Makros wenden wir uns aber zuerst der Datei *lib.rs* zu. Listing 17.28 zeigt eine Übersicht zum Stand des Projekts, nachdem Sie *funktions_makro.rs* erstellt haben:

```
├── Cargo.lock
├── Cargo.toml
├── examples
│   ├── funktions_makro.rs
│   └── hallo_beispiel.rs
├── src
│   └── lib.rs
└── target
    └── ...
```

Listing 17.28 Projektübersicht zum Funktion-ähnlichen Makro

Sie verarbeiten im Funktion-ähnlichen Makro einen Token-Strom, den die Datenstruktur `TokenStream` aus dem `std`-Crate `proc_macro` bereitstellt. Innerhalb des Funktionsrumpfs nehmen Sie nach Bedarf Anpassungen vor, um schließlich den neuen `TokenStream` an den Aufrufer zurückzugeben. Dies stellt sich wie folgt in der Datei *lib.rs* dar:

```
// lib.rs
use proc_macro::TokenStream;
use quote::quote;

#[proc_macro]
pub fn einfuegen(_: TokenStream) -> TokenStream {
    quote!(
        println!("Hallo, aus dem Makro");
    )
    .into()
}
```

Listing 17.29 Das Funktion-ähnliche prozedurale Makro

Die prozeduralen Makros definieren Sie mit `pub` als öffentlich zugänglich, damit Nutzerinnen und Nutzer ein Makro überhaupt auflösen können. Jede prozedurale Makro-Art nutzt ein Attribut, das die Hintergrundmechanik abstrahiert.

Das Paket »quote«

Der Funktionsblock setzt das Makro `quote::quote!` ein, das wir in diesem Abschnitt noch häufiger verwenden. Aber was bewirkt dieses Makro? Der Compiler erstellt Token-Ströme aus Rust-Code. Wenn Sie in oder mit prozeduralen Makros arbeiten, befinden Sie sich jedoch nicht im *eigentlichen* Code. Sie erhalten durch den `TokenStream` den Teil-Code, der im Makro-Aufruf steht. Wenn Sie diese Eingabe nutzen und in einem erweiterten Code-Fragment einsetzen möchten, manipulieren Sie den ausgehenden `TokenStream`.

In Token zu denken, ist freilich aufwendig und fehleranfällig. Hier schafft `quote` Abhilfe. Das Makro `quote::quote!` lässt Sie Rust-Code schreiben, den es nicht kompiliert, sondern in einen `TokenStream` übersetzt. Diesen Strom können Sie durch Interpolation mit Werten und Berechnungen anreichern – wie eine Zeichenkette mit Werten in `println!` oder `format!`.

Der `println!`-Aufruf in Listing 17.29 ist demnach bloßer Text, der von `quote` geparst wird. Die `Into`- bzw. `From`-Implementierung erlaubt die Konvertierung in den `TokenStream`, den der Compiler anstatt des Makro-Aufrufs im Rust-Code einsetzt.

Wechseln Sie jetzt zur Datei *funktions_makro.rs*, und probieren Sie das Makro wie in Listing 17.29 aus:

```
use prozedural::einfuegen;
use quote::quote;

fn main() {
    einfuegen!("Hallo Rust!");

    einfuegen!(quote! {});

    einfuegen! {
        #[derive(Debug)]
        struct Person {
            alter: i32,
            name: String
        }
    };
}
```

Listing 17.30 Aufrufe des Funktion-ähnlichen Makros

Der zweite Aufruf von `einfuegen` zeigt, wie `quote!` implizit einen `TokenStream` an das Makro übergibt. Der dritte Aufruf führt innerhalb der geschweiften Klammern eine Elementdeklaration mitsamt Attribut durch. Auch dies produziert einen `TokenStream`,

den das prozedurale Makro verarbeitet. Die Implementierung von `einfuegen` wird jeden der Ströme verwerfen und stattdessen eine Ausgabe mit `println!` einsetzen.

Prozedurale Makros verzichten ohne weitere Konfiguration auf Hygiene. Daher werden im Makro eingeführte Elemente in dem Gültigkeitsbereich erscheinen, in dem der Aufruf erfolgte. Listing 17.30 zeigt ein Beispiel:

```
// lib.rs
#[proc_macro]
pub fn einfuegen(_: TokenStream) -> TokenStream {
    quote!(
        let mut ausgabe = String::new();
        println!("Hallo, aus dem Makro");
    )
    .into()
}

// examples/funktions_makro.rs
fn main() {
    einfuegen!("Hallo Rust!"); // Führt "ausgabe" ein

    ausgabe = "Variable aus Makro".into();
    println!("{ausgabe}"); // Ok
    // ...
}
```

Listing 17.31 Standardmäßig keine Hygiene

Der Compiler verknüpft prozedurale Makros durch die Datenstruktur `Span` mit ihrem Kontext. Sie können `Span` dafür einsetzen, um `Debug`-Informationen oder Fehlermeldungen anzureichern oder um die Hygiene zu bestimmen. Dazu stellt das Paket `quote` das Makro `quote_spanned` bereit, das jedoch ein weiteres Paket verlangt. Führen Sie daher den Befehl `cargo add proc-macro2` aus. `proc-macro2` ist eine alternative Implementierung zu `proc_macro`, das Rust als Teil der Standardbibliothek ausliefert.

Warum »proc-macro2«?

Wie ich in Abschnitt 17.2, »Prozedurale Makros«, erklärt habe, müssen Sie für prozedurale Makros ein eigenständiges Bibliothek-Crate anlegen und dort einen Schalter in der Manifestdatei umlegen. Erst dann dürfen Sie die Makros einführen.

Dagegen abstrahiert `proc-macro2` das Konzept der Meta-Programmierung von den prozeduralen Makros. Sie können dann ohne die Makro-Infrastruktur mit Token-Strömen und Meta-Elementen arbeiten. Das erlaubt nicht zuletzt, Unit-Tests einzusetzen, um die Logik in den prozeduralen Makros zu testen.

Listing 17.31 zeigt die *saubere* Variante von `einfuegen!`:

```
// lib.rs

use proc_macro::{TokenStream};
use proc_macro2::Span;
use quote::quote_spanned;

#[proc_macro]
pub fn sauber_einfuegen(_: TokenStream) -> TokenStream {
    // Die Hygiene auf mixed setzen, das entspricht dem
    // Verhalten von deklarativen Makros.
    let span :Span = Span::mixed_site();

    quote_spanned!( span =>
        let mut ausgabe = String::new();

        println!("Hallo, aus dem Makro");
    ).into()
}

// examples/funktions_makro.rs
use prozedural::sauber_einfuegen;

fn main() {
    sauber_einfuegen!("Hallo Rust!");

    // Fehler, Hygiene isoliert die Variable
    ausgabe = "Variable aus Makro".into();
    println!("{ausgabe}"); // Fehler
}
```

Listing 17.32 Die Hygiene isoliert jetzt die Deklaration von Variablen.

Das soll als Vorgeschmack auf Funktion-ähnliche Makros genügen. Wenn Sie mehr Flexibilität benötigen, als die deklarativen Makros Ihnen bereitstellen, dann greifen Sie zu dieser Makro-Art. Nutzen Sie hierzu unbedingt die vorgestellten Pakete `syn`, `quote` und `proc-macro2`. Diese Abhängigkeiten erleichtern Ihnen erheblich den Umgang mit den Token-Strömen.

Attribut-Makros

Die prozeduralen Makros öffnen eine Ebene, auf der Sie Rust-Code in seiner primitivsten Form verarbeiten und manipulieren können – nämlich als Token. Im vorigen

Abschnitt haben Sie die Token-Ströme am Beispiel von Funktion-ähnlichen Makros kennengelernt.

Gleich zwei dieser Ströme verarbeiten die prozeduralen *Attribut-Makros*. In Abschnitt 13.3.9, »Attribute«, habe ich darauf hingewiesen, dass die Attribut-Inhalte, die Sie einem Attribut mitgeben können, keinem festen Format folgen, dafür aber Konventionen. So halten die meisten Attribut-Inhalte sich an die Meta-Typen, die Sie in Tabelle 13.2, »Übersicht zur Meta-Typ-Syntax«, gesehen haben.

Der Begriff *Meta* – das wissen Sie nach der vorangegangenen Diskussion in Abschnitt 17.1.3, »Die Meta-Syntax« – erinnert an den Sprachgebrauch der Meta-Syntax von deklarativen Makros. Tatsächlich sprechen wir bei Attribut-Inhalten von Token-Strömen. Diesbezügliche Konventionen stellen sicher, dass Sie ein Attribut-Makro einfacher implementieren können.

Wir erstellen ein neues Beispielprogramm, um Attribut-Makros auszuprobieren. Legen Sie dazu im Verzeichnis *examples* die Datei *attribut_makro.rs* inklusive dortiger `main`-Funktion an. Das Beispiel rufen Sie danach mit `cargo run --example attribut_makro` auf.

Nehmen wir an, dass Sie ein Attribut einführen wollen, mit dem Sie ein syntaktisches Element nur für eine Plattform kompilieren. Falls Sie das Programm für diese Plattform übersetzen, soll der Compiler außerdem bestimmte Traits implementieren. Diese Anforderung simuliert das `derive`-Makro mit dem Attribut-Inhalt `Debug`. Das folgende Listing 17.33 zeigt das Attribut-Makro:

```
//lib.rs

#[proc_macro_attribute]
pub fn nur_auf(
    attribut: TokenStream,
    element: TokenStream
) -> TokenStream {
    let item = syn::parse_macro_input!(element as syn::Item);

    // Lit = Literal
    let attribut = syn::parse_macro_input!(attribut as syn::Lit);

    TokenStream::from(quote! {
        #[cfg(target_os = #attribut)]
        #[derive(Debug)]
        #item
    })
}
```

Listing 17.33 Ein prozedurales Attribut-Makro

Das Attribut-Makro erwartet zwei `TokenStream`s. Der erste bewegt den Attribut-Inhalt in die Funktion, der zweite bewegt den `TokenStream` des Elements, auf das Sie ein Attribut anwenden. `nur_auf` wird ein Literal erhalten, das die Funktion an das Attribut `#[cfg(target_os = …)]` weitergibt.

Aus den Token-Strömen müssen Sie zuvor allerdings syntaktische Elemente parsen, damit Sie sie im Rust-Code innerhalb von `quote!` verwenden können. Sowohl für Argumente als auch Elemente verwenden wir dafür `syn::parse_macro_input!`. Der `as`-Operator ist in diesem Kontext erlaubt, um die `TokenStream`s in die gewünschte Typ-Richtung zu lenken. An dieser Stelle zeigt sich, wie wichtig das Paket `syn` für die Entwicklung eines prozeduralen Makros ist.

Der Compiler verlangt vom Attribut-Makro einen `TokenStream` als Ausgabe. Daher setzt `nur_auf` die Funktion `TokenStream::from` ein. Hier dient im Aufruf das `quote!`-Makro als Argument. Es kommt zu zwei Interpolationen: `#attribut` und `#item`. Das `#`-Zeichen leitet die Interpolation innerhalb von `quote!` ein. Praktischerweise erkennt es automatisch den Unterschied zwischen dem Rust-Attribut und der Interpolationsvariable.

Die Attribute `cfg` und `derive` werden dafür sorgen, dass das syntaktische Element nur kompiliert und das Trait `Debug` implementiert, falls die Zielplattform mit dem Wert in `attribut` übereinstimmt. Das probieren wir im Beispielprogramm von Listing 17.34 aus:

```
use prozedural::nur_auf;

fn main() {
    #[nur_auf("macos")]
    struct Person;

    // Kontrolle: Ausgabe nur, wenn das
    // eigene Makro funktioniert, ansonsten Fehler!
    #[cfg(target_os = "macos")]
    println!("{:?}", Person);
}
```

Listing 17.34 Das Attribut-Makro »nur_auf« zur konditionalen Kompilierung einsetzen

Falls Sie den Beispielen auf dem System *macOS* folgen, sehen Sie die Debug-Ausgabe in der Konsole, sobald Sie das Programm ausgeführt haben. Nutzen Sie dazu den Befehl `cargo run --example attribut_makro`. Auf *Windows* oder *Linux* geben Sie entsprechend die Literale `windows` oder `linux` anstatt `macos` ein.

Eigene derive-Makros

Die Liste derjenigen Traits, die das `derive`-Makro unterstützen, können wir nicht als umfangreich bezeichnen. Die prozeduralen Makros erlauben Ihnen allerdings, Ihre eigenen Erweiterungen in die `derive`-Mechanik einzuhängen. Dann können Sie `#[derive(IhrBezeichner)]` auf eine Datenstruktur (`struct`, `enum` oder `union`) anwenden.

Die Signatur der Makro-Funktion nimmt einen `TokenStream` entgegen und gibt einen zurück. Listing 17.35 zeigt ein Beispiel, das wir in die Datei *lib.rs* einfügen:

```
// lib.rs

use proc_macro::TokenStream;
use quote::quote;

#[proc_macro_derive(PersonIdentifikation)]
pub fn person_identifikation(stream: TokenStream) -> TokenStream {
    let person = syn::parse_macro_input!(
        stream as syn::DeriveInput
    );

    let fn_name = format!(
        "id_{}", person.ident.to_string().to_lowercase()
    );
    let fn_ident = proc_macro2::Ident::new(
        &fn_name, Span::mixed_site()
    );
    TokenStream::from(quote! {
        fn #fn_ident() -> String {
            // ...
            format!("ID: {}", rand::random::<usize>())
        }
    })
}
```

Listing 17.35 Ein eigenes »derive«-Makro

Das Programm greift auf das Paket `rand` zurück. Dafür müssen Sie es mit dem Kommando `cargo add rand` im Projekt als Abhängigkeit eintragen. Einen `derive`-Inhalt können Sie mit `syn` als `DeriveInput` parsen. Darüber erhalten Sie unter anderem den Bezeichner, wie es im Zugriff auf `person.ident` zu sehen ist.

Den Bezeichner der Datenstruktur konvertieren Sie dann in einen Identifikator, den `proc_macro2::Ident` repräsentiert. Auch dieses Beispiel nutzt also die alternative Implementierung zum `proc_macro`-Crate der `std`.

Sobald Sie das `derive`-Attribut mit dem Inhalt `PersonIdentifikation` auf eine Datenstruktur anwenden, generiert der Compiler eine Funktion `id_<Datenstruktur>`. Wir erweitern die `main`-Funktion aus Listing 17.34 so wie in Listing 17.36:

```
fn main() {


    #[nur_auf("macos")]
    #[derive(PersonIdentifikation)]
    struct Person;

    // Kontrolle: Ausgabe nur, wenn das
    // eigene Makro funktioniert, ansonsten Fehler!
    #[cfg(target_os = "macos")]
    {
        println!("{:?}", Person);
        println!("{}", id_person());
    }
}
```

Listing 17.36 Das eigene »derive«-Makro aufrufen

Die kleinen Helferlein des derive-Makros

In `#[proc_macro_derive(PersonIdentifikation)]` können Sie weitere Unter-Attribute (auch engl. *Helper* genannt) angeben, etwa `#[proc_macro_derive(PersonIdentifikation, attributes(helfer, alter, name))]`.

Diese Attribute setzen Sie in der Datenstruktur ein, die Sie mit dem `derive`-Attribut versehen. Jedes dieser Attribute wird an die `derive`-Makro-Funktion gesendet. Die Rust-Referenz weist dazu ein Beispiel auf, das Sie unter *https://doc.rust-lang.org/reference/procedural-macros.html#derive-macro-helper-attributes* finden.

Die Möglichkeiten, die Ihnen die prozeduralen Makro-Arten bieten, sind ausgesprochen umfangreich. In diesem Buch können wir den Themenbereich daher nur notdürftig anschneiden. Die drei Beispielprogramme konnten Ihnen aber dennoch einen Eindruck davon vermitteln, wie Sie im jeweiligen Fall vorgehen. Die Dokumentationen der Pakete `syn`, `quote` und `proc_macro2` helfen Ihnen bei Detailfragen weiter.

17.3 Zusammenfassung

Makros öffnen Ihnen den Weg zur Meta-Programmierung in Rust. Das heißt, dass Sie anstatt mit Rust-Code mit Token arbeiten, die auch Bestandteil der lexikalischen Ana-

lyse im Compiler sind. Makros sind sicher. Der Compiler liest Ihre Makros vor dem Kompilieren aus und expandiert die Token-Ströme in Rust-Code. Fehler löst er daher frühzeitig auf.

Man unterscheidet deklarative und prozedurale Makros. Die *deklarativen Makros* sind einfacher einzusetzen und decken bereits viele Anwendungsfälle ab. Deklarative Makros sind außerdem *hygienisch*. Mit diesem Begriff meint Rust, dass ein Makro nur Datentypen und deren Anpassungen in denjenigen Gültigkeitsbereich expandiert, in dem Sie das Makro aufgerufen haben. Deklarationen von Variablen bleiben auf das Makro beschränkt. Zudem greift das Makro ausschließlich auf jene Werte des Gültigkeitsbereichs zu, die Sie übergeben haben.

Die *prozeduralen Makros* sind etwas komplexer, dafür aber sehr wirkmächtig. Sie teilen sich in Funktion-ähnliche Makros, Attribut-Makros und `derive`-Makros auf. Mit diesen Werkzeugen generieren Sie zusätzlichen Code, nehmen Transformationen vor oder entfernen Token aus den Eingabeströmen, bevor Sie die gefilterte Ausgabe an den Compiler weiterleiten. Die *Funktion-ähnlichen Makros* sind wie deklarative Makros, schränken Sie aber nicht in der Hygiene ein. Das heißt, dass auch Variablendeklarationen in den Gültigkeitsbereich des Aufrufs expandieren. Das können Sie jedoch über die `Span`-Datenstruktur konfigurieren. *Attribut-Makros* erlauben es Ihnen, die Meta-Programmierung mit einem Attribut zu verbinden, das Sie auf syntaktische Elemente anwenden. Das eigene `derive`-*Makro* lässt Sie Regeln definieren, nach denen der Compiler ein `#[derive(…)]` nach Ihren Vorstellungen behandelt.

Die Komplexität der prozeduralen Makros beherrschen Sie mit den zusätzlichen Paketen `syn`, `quote` und `proc-macro2`. `syn` nimmt Ihnen etwa das Parsen ab und definiert zahlreiche Datenstrukturen, um die Token-Ströme in Meta-Repräsentationen abzubilden.

Mit `quote` schreiben Sie Rust-Code, den Sie mit Meta-Repräsentationen (Bezeichner, Eigenschaften wie Sichtbarkeit usw.) interpolieren können. Das Ergebnis ist ein Token-Strom, den Sie an den Compiler oder andere Werkzeuge weiterleiten können.

`proc-macro2` ist eine alternative Implementierung zu `proc_macro`, einem Crate der Standardbibliothek. Auch auf dieser Ebene liegen schon viele Abstraktionen und Hilfswerkzeuge vor. Die Alternative ermöglicht es, die Token-Ströme als Konzept anstatt als Implementierungsdetail zu behandeln und so vom prozeduralen Makro zu abstrahieren. Das ist eine Vorbedingung, um die Logik der prozeduralen Makros mit automatischen Tests zu prüfen oder sie außerhalb einer expliziten Crate-Bibliothek zu verwenden.

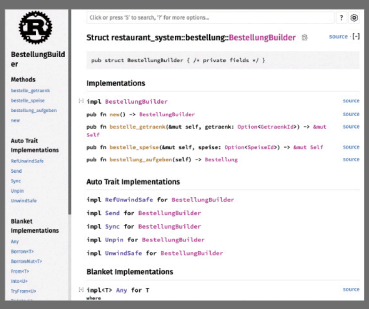
Inhalt

Rust

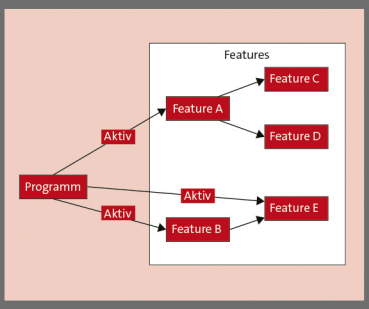
»Ideal zum Selbststudium und zum Nachschlagen«

Geballtes Know-how für moderne, sichere Software

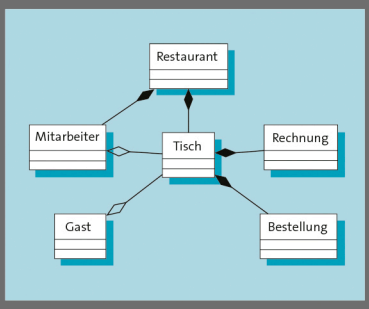
Hier erfahren Sie alles, was Sie für die Programmierung mit Rust brauchen – ganz egal, ob Sie neu einsteigen oder Ihr Wissen vertiefen und professionalisieren möchten. Von der Installation über das Speichermanagement bis hin zu Unsafe Rust und zum Foreign Function Interface führt der Autor Sie Schritt für Schritt an die Sprache heran.



Professionelle Werkzeuge



Design und Architektur



Anwendungsentwicklung

So gelingt der Einstieg!

Setzen Sie Ihre Toolchain auf und erstellen Sie Ihr erstes Rust-Programm. Hier widmen Sie sich zunächst ausführlich den Abhängigkeiten, Variablen und Datentypen in Rust und lernen den Compiler und den Paketmanager kennen.

Features und Tools spielend beherrschen

Speichermanagement und Safety sind zwei der größten Vorzüge von Rust. Lernen Sie dafür wichtige Tools kennen und binden Sie sie in Ihren Programmierstil ein – ob systemnah, asynchron, objektorientiert oder funktional.

Über den Tellerrand hinaus

Asynchrone und generische Programmierung, Makros und viele weitere Themen ermöglichen den gezielten Einstieg in fortgeschrittene Bereiche der Programmierung mit Rust – lernen Sie vom Profi!

Alle Codebeispiele zum Download

Marc Marburger ist freiberuflicher Softwareentwickler mit einem Portfolio von Rust über Kotlin bis hin zu Flutter. Er begeistert sich für moderne Sprachfeatures und zeigt in seinen Büchern, wie man sie effizient verwendet. Rust setzt er bei seiner Arbeit täglich ein.

Aus dem Inhalt

- Rust installieren
- Abhängigkeiten, Variablen und Datentypen
- Speichermanagement und Referenzen
- Collections und Iteratoren
- Multithreading
- Fehlerbehandlung
- Strukturen, Muster und Methoden
- Generische Programmierung
- Nebenläufige und asynchrone Programmierung
- Makros
- Doku und Tests automatisieren
- Unsafe Rust
- Foreign Functions

Rheinwerk Computing

Gedruckt in Deutschland
Papier aus nachhaltiger Waldwirtschaft
Mineralölfreie Druckfarben

€ 49,90 [D] € 51,30 [A]

Für Windows, macOS und Linux
Programmierung

ISBN 978-3-8362-9018-0