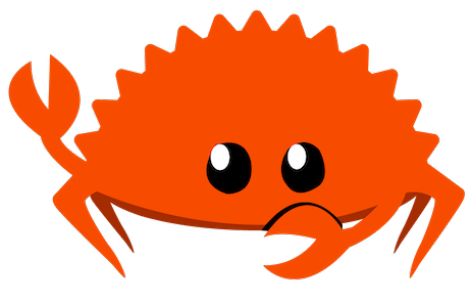
Programmieren mit Rust

Thema 4: Ownership

Robert Baumgartl und Dirk Müller

12. April 2024



Übersicht

- Motivation:
  - Warum `malloc()` und `free()` ungünstig sind
  - Garbage
- Ownership: Regeln
- Strings in Rust
- Wertzuweisung: *Shallow Copy* vs. *Deep Copy* vs. *Move*
- Ownership und Funktionen
- Referenzen
- veränderbare Referenzen

Motivation

Speicherverwaltung in Programmiersprachen:

- explizit: `malloc()`/`free()` in C, `new()`/`delete()` in C++
- `alloca()` für den Stack

Potentielle Fehler bei expliziter Heapverwaltung

- vergessener Test auf Resultat `NULL` bei `malloc()`
- weniger `free()` als `malloc()` („Memory Leak“)
- „Use-after-free“
- „Double free“
- vergessene Initialisierung alloziierten Speichers
- falsche Größe bei `malloc()` („Off-by-One“)
- Annahme, dass `new()` bei Out-of-Memory `NULL` zurückliefert
- Memory Overcommitment durch das System
- `realloc()` bietet weitere interessante Möglichkeiten (vergessene Initialisierung des neu alloziierten Teiles)

Motivation II

Verbesserung: Garbage Collection

- Idee: Programmierer alloziiert explizit, Rückgabe erfolgt automatisch
- Laufzeitumgebung bzw. VM ermittelt, welche Objekte nicht mehr benötigt werden und vernichtet diese „nach Gebrauch“
- Java, Python, Go, ...

Nachteile:

- ineffizienter als explizites Management
- nicht beeinflussbar, wann GC erfolgt ($\rightsquigarrow$ ungeeignet für Echtzeitsysteme)

Stack und Heap

Eigentum (“Ownership”)

3 Regeln:

1. Jeder Wert hat eine Variable, die sein Eigentümer (“Owner”) ist.
2. Jeder Wert hat **genau einen** Eigentümer (es kann nur einen geben . . .).
3. Wenn der Eigentümer seinen Gültigkeitsbereich (*scope*) verlässt, wird der zugehörige Wert vernichtet (“dropped”).

Gültigkeitsbereich (scope) von Variablen

```
{
  // s ist noch nicht definiert, daher ungültig
  let s = "Kaboom!";  // Scope von s beginnt
  println!("{}", s);
}
// Scope von s ist vorbei, s ist ungültig
println!("{}", s);  // unzulässig!
```

Strings

- bislang nur String-Literale, diese sind jedoch nicht modifizierbar (Größe muss zur Übersetzungszeit feststehen)
- für komplexere Operationen bietet Rust den Typ `String`
  - veränderbar, insbesondere verlängerbar
  - Speicherung zweigeteilt auf Stack (Verwaltungsinfo, feste Größe) und auf dem Heap („Nutzdaten“)
  - Zeiger, Länge (des Strings), Kapazität (Größe des zugehörigen Heapsegments)

Repräsentation eines Strings

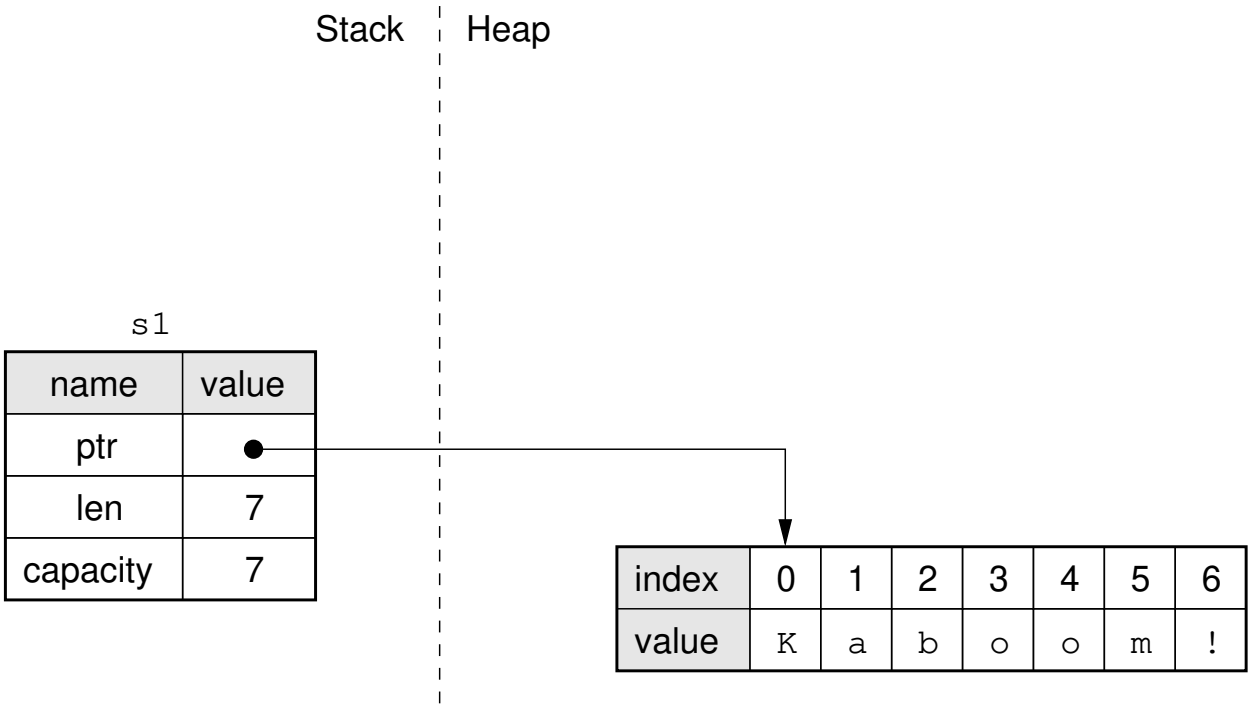


Abbildung: Repräsentation des Strings `s1`

Kopieren: *Shallow Copy*

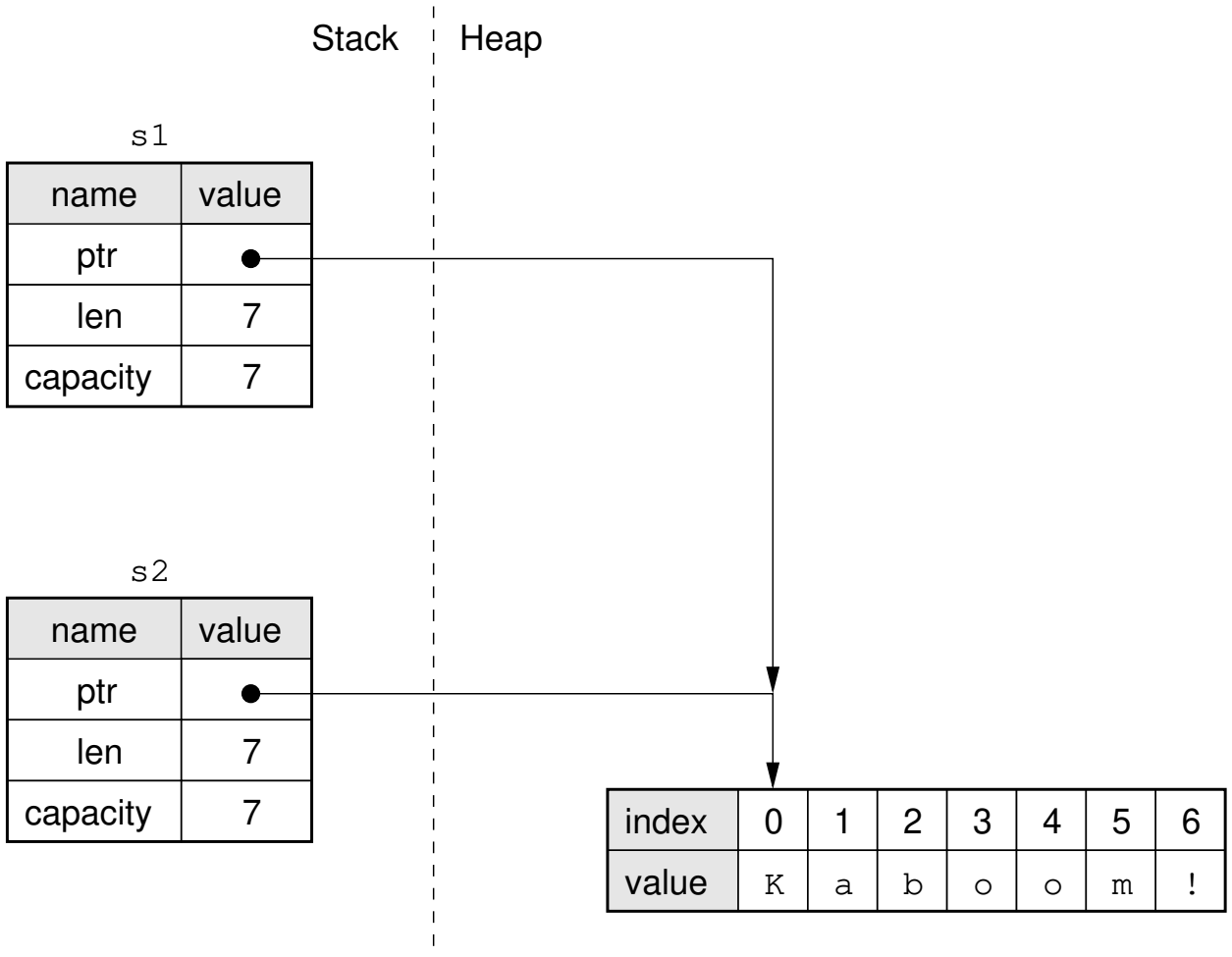


Abbildung: `s2` ist eine *shallow copy* von `s1`

Kopieren: *Deep Copy*

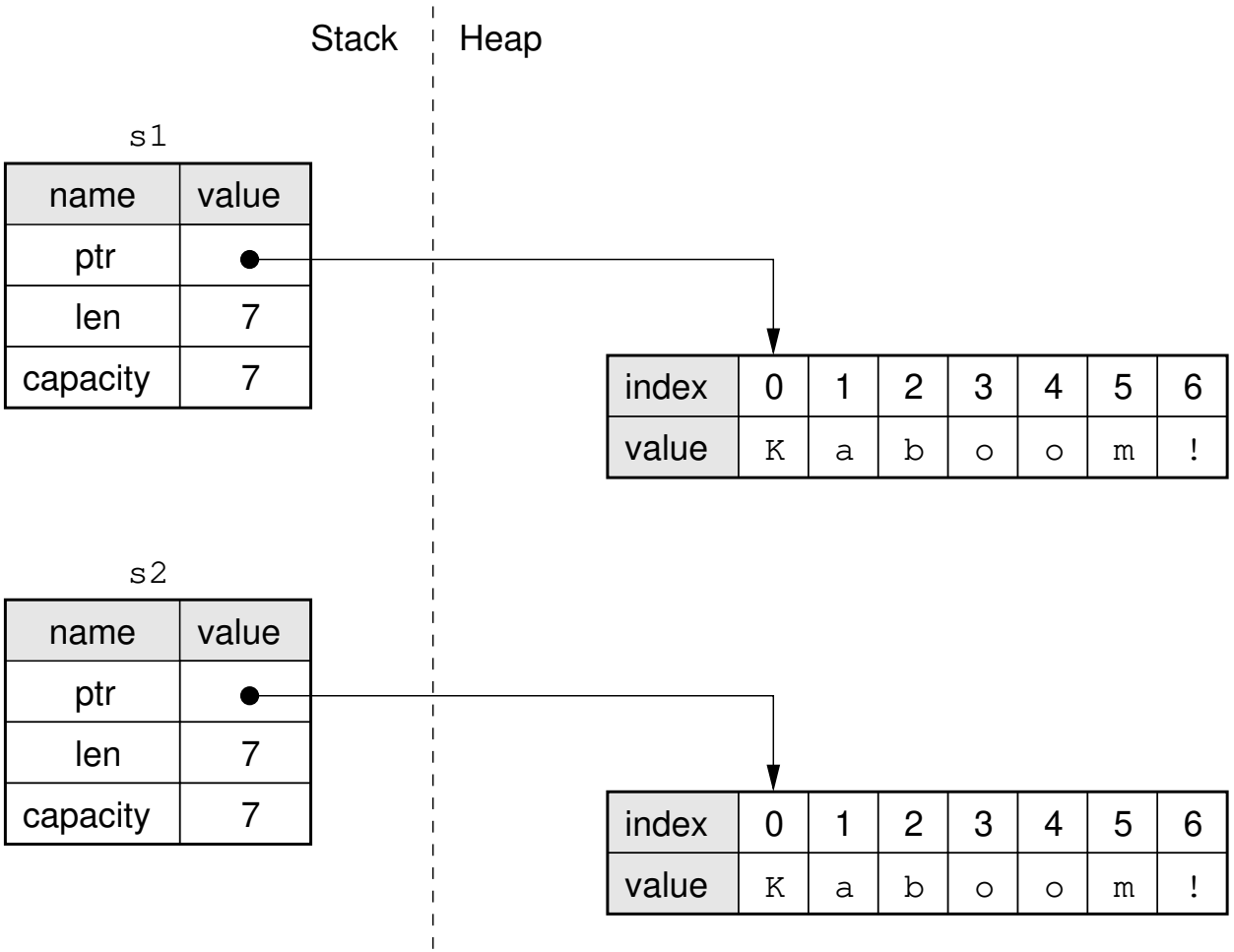


Abbildung: `s2` ist eine *deep copy* von `s1`

Kopieren: *Move* anstatt *Copy*

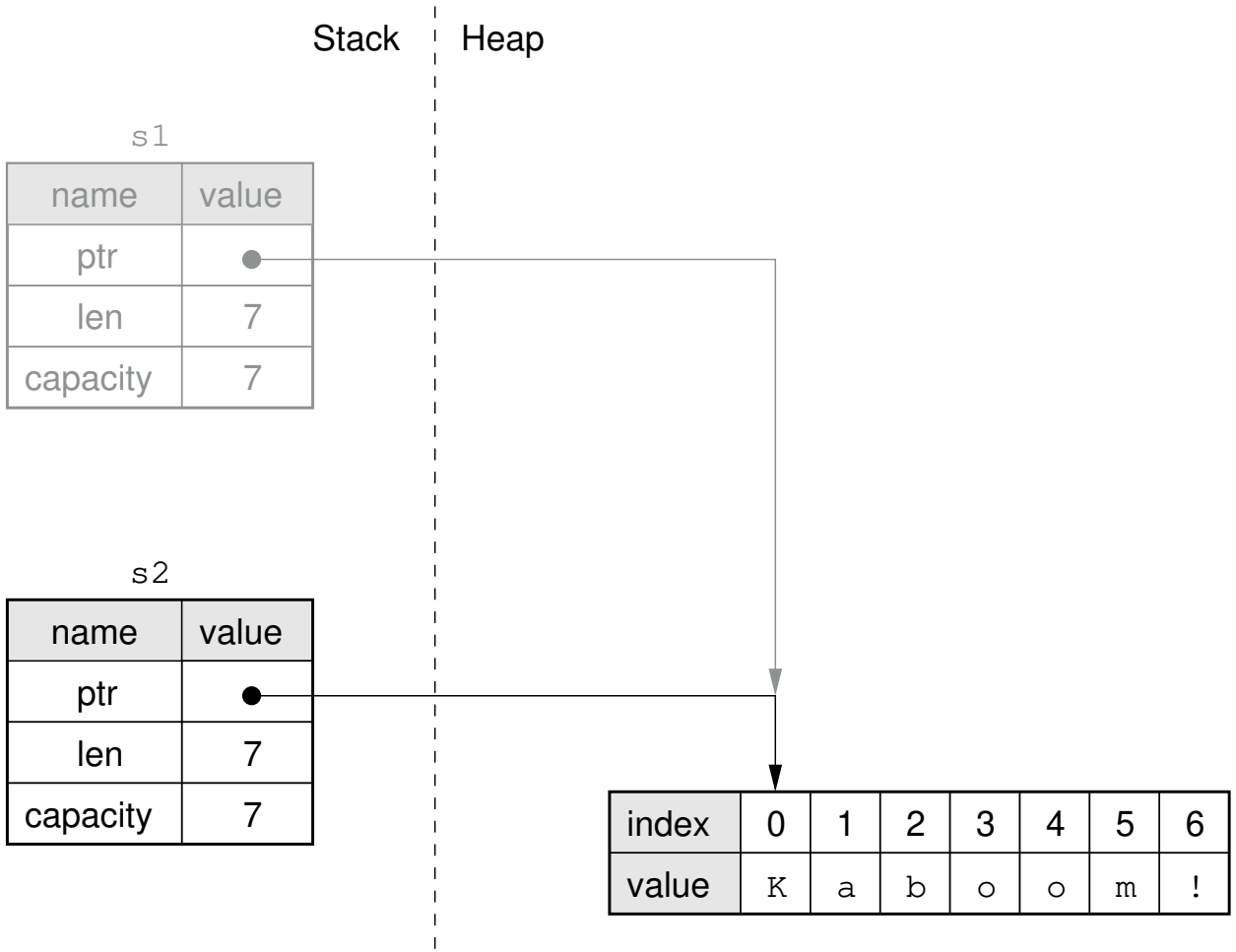


Abbildung: `s2` wird bewegt, `s1` verliert seine Gültigkeit

Bemerkungen

- eine Zuweisung in Rust **kopiert oder bewegt** das Datum, je nachdem
- Daten, die nur auf dem Stack liegen, werden bei Zuweisungen kopiert (weil dies schnell geht) – sie implementieren das „Copy Trait“: alle Integer-Typen, `bool`, alle Gleitkomma-Typen, `char`, Tupel-Typen, die nur aus den vorigen Typen bestehen
- Daten die Alloziierung auf dem Heap benötigen, dürfen das Copy Trait nicht implementieren, sie werden bewegt
- Rust legt niemals automatisch Kopien von (komplexen) Daten an, die auf dem Heap liegen
- bei Funktionsparametern ist es ganz genauso (die Variable wird entweder kopiert oder in die Funktion bewegt)

Beispiel: Strings

```
fn main() {
    let s1 = String::from("Kaboom!");
    println!("s1 = {s1}");
    let s2 = s1;
    println!("s2 = {s2}");
    println!("s1 = {s1}");   // not allowed; s1 has moved
    let s3 = s2.clone();
    println!("s3 = {s3}");   // this _is_ allowed
}
```

- Ist (am Programmende) `s1.clone()` erlaubt?
- Ist (am Programmende) `s3.clone()` erlaubt?

Beispiel: Funktionsparameter

```
fn printint (i : usize) {
    println!("{i}");
}

fn printstring(s : String) {
    println!("{s}");
}

fn main() {
    let mut x = 42;
    printint(x);
    x += 23;    // legal
    printint(x);
    let s1 = String::from("Kalte");
    printstring(s1);    // transfers ownership to printstring
    s1.push_str("Ente"); // illegal, because s1 has moved
    println!("{s1}");  // also illegal
}
```

Beispiel: Resultatwerte von Funktionen

- Funktionen transferieren Ownership der Resultate an die rufende Instanz

```
fn roll_dice() -> usize {
    6
}
fn newstr() -> String {
    String::from("You are indeed brave, Sir Knight, but the
    ↪  fight is mine.")
}

fn main() {
    let r = roll_dice();
    println!("{r}");
    let s = newstr();
    println!("{s}");
}
```

Weiterverwendung der Funktionsargumente?

Was, wenn das Argument einer Funktion (in der rufenden Instanz) weiterverwendet werden soll?

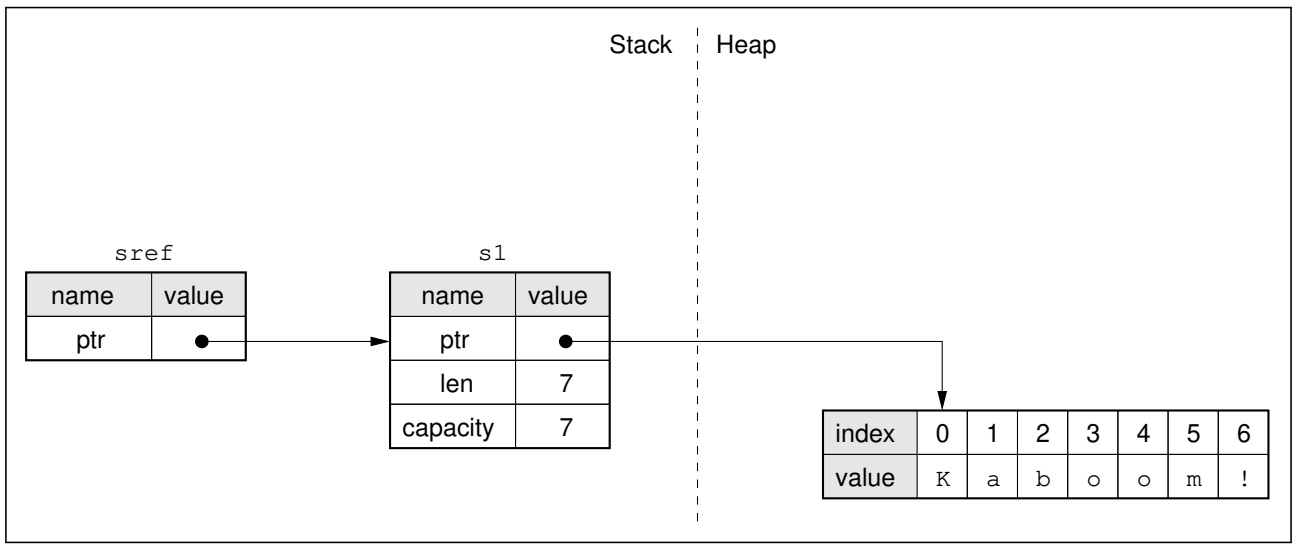
- Parameter als Resultat wieder zurückliefern
- per Tupel möglich

```
fn length(s1 : String) -> (usize, String) {
    (s1.len(), s1)
}

fn main() {
    let s = String::from("I am Roger the Shrubber");
    let (l,s) = length(s);
    println!("The string \"{s}\" has length {l}.");
}
```

- unelegant, unhandlich

Referenzen

```
fn main() {
    let s1 = String::from("Kaboom!");
    let sref = &s1;

    println!("{s1} oder {sref}");
}
```



- ähnlich Zeiger (C) bzw. Referenz (C++), jedoch immer auf gültiges Datum verweisend
- zum Zugriff auf Wert, **ohne** Eigentümer zu werden

Referenzen: Beispiel 2

```
fn length(sref : &String) -> usize {
    sref.len()
}

fn main() {
    let s = String::from("I am Roger the Shrubber");
    let l = length(&s);
    println!("The string \"{s}\" has length {l}.");
}
```

- Wenn die Referenz `sref` den Scope verlässt, wird sie vernichtet, aber nicht das Datum, auf das sie verweist
- Anlegen einer Referenz wird **Borgen** (*Borrowing*) genannt

Veränderbare Referenzen

- Um geborgtes Datum ändern zu können, benötigt man eine *veränderbare Referenz* (**mutable reference**)

```
fn add_something(sref: &mut String) {
    sref.push_str(", the Brave")
}

fn main() {
    let mut s1 = String::from("Sir Lancelot");
    add_something(&mut s1);
    println!("s1 is '{s1}'");
}
```

1. Funktionskopf muss `&mut` enthalten
2. zu veränderndes Datum muss `mut` definiert sein
3. Funktionsaufruf muss ebenfalls `&mut` enthalten, denn dort wird die veränderbare Referenz zur Laufzeit angelegt

Restriktionen

Regel: Es darf maximal eine veränderbare Referenz auf ein und dasselbe Datum geben.

```
fn main() {
    let mut s1 = String::from("Brother Maynard");
    let sref1 = &mut s1;
    let sref2 = &mut s1;

    // not allowed
    println!("sref1 contains \"{sref1}\", sref2 contains
    ↪  \"{sref2}\"");
}
```

Zweck: Verhinderung so genannter *Data Races* =

- mehrere Zeiger referenzieren ein- und dasselbe Datum,
- mindestens eine der assoziierten Operationen ist ein *Write*,
- es findet keine explizite Synchronisation statt

Restriktionen

Mehrere unveränderliche Referenzen sind hingegen erlaubt.

```
fn main() {
    let mut s1 = String::from("Brother Maynard");
    {
        let sref1 = &s1;
        let sref2 = &s1;
        // this _is_ allowed
        println!("s1 contains \"{s1}\"");
        println!("sref1 points to \"{sref1}\"");
        println!("sref2 points to \"{sref2}\"");
    }
    s1 = String::from("Rabbit from Caerbannog");
    println!("s1 contains \"{s1}\"");
}
```

Gültigkeitsbereich (Scope) einer Referenz

Start: Definition der Referenz

Ende: Stelle der letztmaligen Nutzung (!)

- Restriktionen gelten nur, wenn sich die Gültigkeitsbereiche der beteiligten Referenzen überschneiden!

Gültigkeitsbereich (Scope) einer Referenz

Beispiel

```
fn change_string(sr: &mut String) {
    *sr = String::from("Dingo");
}

fn main() {
    let mut s = String::from("Zoot");
    let sref1 = &s; // immutable
    let sref2 = &s; // immutable
    println!("sref1 points to '{sref1}' and sref2 points to
    ↪  '{sref2}'");
    // last time sref1 and sref2 were used

    let sref3 = &mut s; // this is allowed
    change_string(sref3);
    println!("s is now {s}");
}
```

- `sref1` und `sref2` sind beide unveränderbar
- Scope von `sref3` beginnt erst **nach** Ende des Scopes von `sref1` und `sref2` $\rightsquigarrow$ zulässig

Hängende Referenzen („Dangling References“)

- . . . verweisen auf ein Datum, das nicht (mehr) existiert

```c
#include <stdio.h>

char *return_something(void)
{
  char string[] = "Swamp castle";
  return string;
}

int main(int argc, char *argv[])
{
  printf("%s sank into the swamp.\n", return_something());
  return 0;
}
```

```
robge@ilpro122:~/txt/job/htw/rust/src/dangling-ref-in-c$ gcc -o dangling-ref dangling-ref.c
dangling-ref.c: In function ‘return_something’:
dangling-ref.c:7:10: warning: function returns address of local variable [-Wreturn-local-addr]
    7 |   return string;
      |          ^~~~~~
robge@ilpro122:~/txt/job/htw/rust/src/dangling-ref-in-c$ ./dangling-ref
(null) sank into the swamp.
robge@ilpro122:~/txt/job/htw/rust/src/dangling-ref-in-c$
```

Hängende Referenzen in Rust

- . . . verbietet der Compiler.

```
fn return_something() -> &String {
    let s = String::from("Swamp castle");
    &s
}

fn main() {
    let dangling_ref = return_something();
}
```

- beim Verlassen von `return_something()` endet der Scope von `s`, das Datum wird vernichtet
- damit wird der Resultatwert `&s` ungültig

Lösung: `String` anstelle `&String` zurückliefern

Zusammenfassung

- Rust versucht, durch das Konzept der *Ownership* speicherbasierte Fehler zur Übersetzungszeit zu eliminieren.
- Elementare Datentypen werden auf dem Stack angelegt, komplexe Datentypen auf dem Heap.
- Elementare Datentypen werden bei Zuweisungen kopiert, komplexe Datentypen unterliegen der Ownership (werden bewegt).
- Verlässt ein Datum seinen Gültigkeitsbereich, so wird der zugehörige Wert vernichtet („dropped“).
- Referenzen gelten von ihrer Definition bis zur letztmaligen Nutzung.
- Es kann entweder genau eine veränderbare Referenz oder beliebig viele unveränderbare Referenzen zu einem Zeitpunkt geben.
- Dangling References sind in Rust unmöglich.