Rust

Ownership, References & Borrowing, Slices

Marno Janetzky, Gabriel Cmiel

Warum Ownership?

- Alt und doof: selber Speicher verwalten oder Garbage Collector
- Mit Ownership: Speicherverwaltung durch Regeln.
- Verlangsamt nicht
- Man muss nicht selbst allokieren und deallkieren
- Aber: Ist gewöhnungsbedürftig!

Stack und Heap

- Wer weiß nicht was das ist?

Die Regeln

- Jeder Wert hat eine Variable, welche „owner“ genannt wird.
- Es gibt zu jedem Zeitpunkt immer nur einen owner.
- Wenn der owner den Scope verlässt (out of scope), geht der Wert verloren!

Der Scope (Sichtbarkeitsbereich)

```
{                       // s is not valid here, it’s not yet declared
    let s = "hello";    // s is valid from this point forward

    // do stuff with s
}                       // this scope is now over, and s is no longer valid
```

String in Rust

```rust
let s = String::from("hello");
```

- (::) ist ein Operator, um from zu „namespacen“
- String auf dem Heap

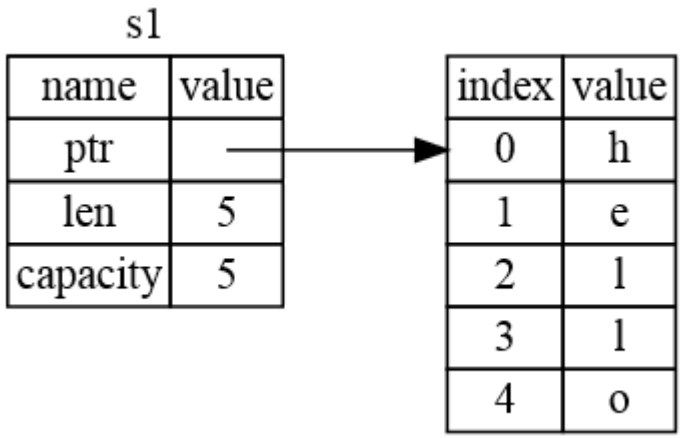
```rust
let mut s = String::from("hello");

s.push_str(", world!"); // push_str() appends a literal to a String

println!("{}", s); // This will print `hello, world!`
```

- „mut“ um den String mutabel zu machen!

Speicherverwaltung

- Weiß der Compiler zur Compilezeit die Größe eines Datentyps → Stack
- Sonst: Kann dynamisch wachsen? → Heap

Pointer(ort?) Wert(Ort?)



Allokieren in Rust

- Zur Laufzeit wird vom OS Speicher angefordert
- Wenn wir fertig sind wird dieser zurück gegeben

- Pro Allokierung jeweils genau ein Free

  → Was wäre wenn nicht?

- Umsetzung in Rust:

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
} // this scope is now over, and s is no
  // longer valid
```

Drop(s) wird automatisch aufgerufen

Move

- Erinnerung: „Es gibt zu jedem Zeitpunkt immer nur einen owner.“

- Funktioniert das?

```
let x = 5;
let y = x;
```

- Und das?

```
let s1 = String::from("hello");
let s2 = s1;
```

Move

- Erinnerung: „Es gibt zu jedem Zeitpunkt immer nur einen owner.“

- Funktioniert das?

```
let x = 5;
let y = x;
```

Ja, x und y sind gültig durch copy

- Und das?

```
let s1 = String::from("hello");
let s2 = s1;
```

Ja, s2 ist der neue owner

Aber… Achtung

- Ok es ist rot…

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

- Was würde hier in anderen Sprachen passieren?

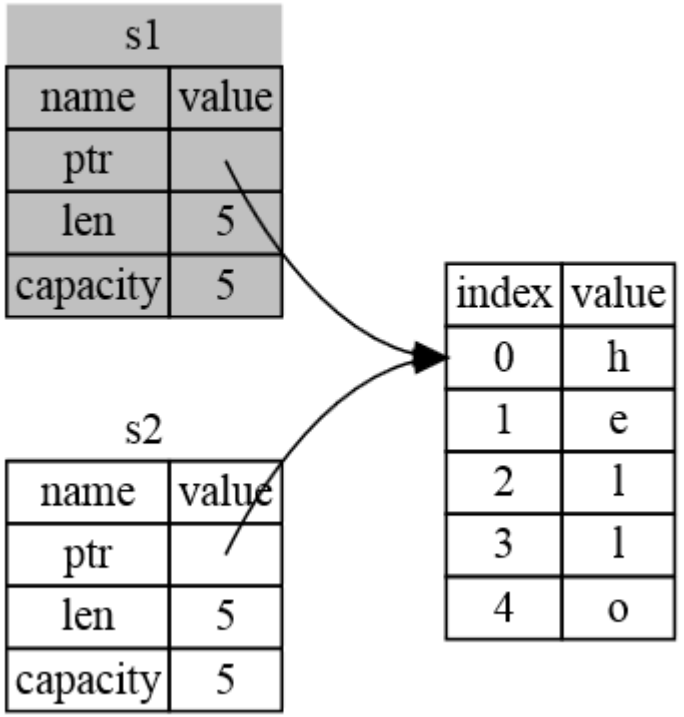
Aber… Achtung

- Ok es ist rot…

```
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```

- Was würde hier in anderen Sprachen passieren? → Shallow Copy, also Pointer kopieren

Was passiert:



Aber… Achtung

- Ok es ist rot… UND wir sind nicht in anderen Sprachen (zum Glück – Warum zum Glück? Vor was werden wir hier bewahrt?)

```rust
let s1 = String::from("hello");
let s2 = s1;

println!("{}, world!", s1);
```


```
error[E0382]: use of moved value: `s1`
 --> src/main.rs:5:28
  |
3 |     let s2 = s1;
  |         -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value used here after move
  |
  = note: move occurs because `s1` has type `std::string::String`, which does
  not implement the `Copy` trait
```

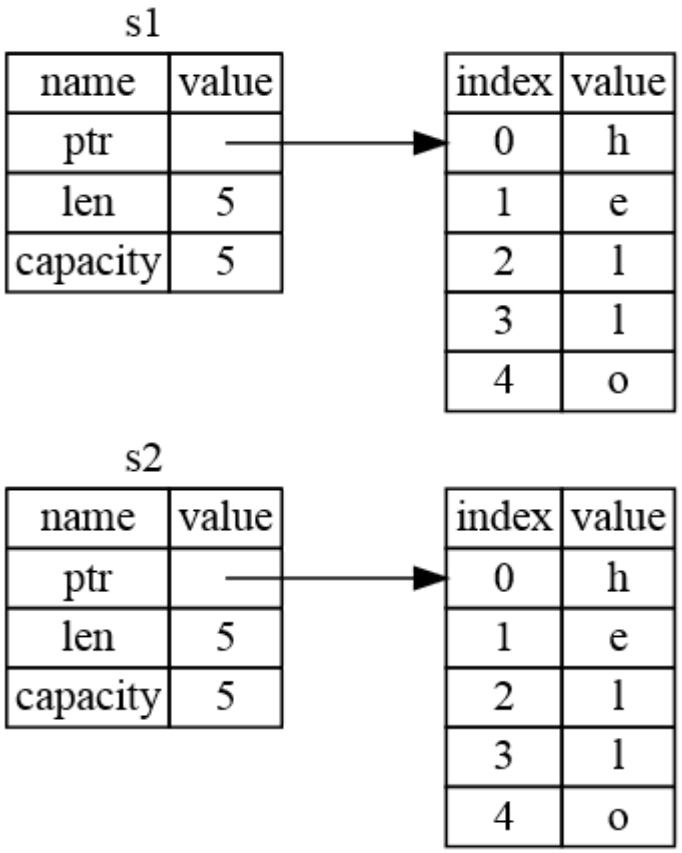
Danke Rust :)

Zurück zum Problem: Lösungsidee?

```
let s1 = String::from("hello");
let s2 = ???

println!("s1 = {}, s2 = {}", s1, s2);
```

Die Lösung: Clone

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```



Aber Moment mal:

```
let x = 5;
let y = x;

println!("x = {}, y = {}", x, y);
```

- Warum geht das jetzt?

Stack only: Copy

- Nochmal Unterschied: Shallow Copy vs Clone ???

  → Tipp: Heap und Stack

Stack only: Copy

- Nochmal Unterschied: Shallow Copy vs Clone ???

  → Tipp: Heap und Stack

  → Kein Unterschied zwischen copy und clone auf Stack

  → Primitive Datentypen sind automatisch copy in Rust

- Warum wird das nicht auch automatisch auch für den Heap gemacht?

Stack only: Copy

- Nochmal Unterschied: Shallow Copy vs Clone ???

  → Tipp: Heap und Stack

  → Kein Unterschied zwischen copy und clone auf Stack

  → Primitive Datentypen sind automatisch copy in Rust

- Warum wird das nicht auch automatisch auch für den Heap gemacht? → Performancegründe

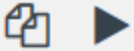
Copy Typen

- All the integer types, such as `u32`.
- The Boolean type, `bool`, with values `true` and `false`.
- All the floating point types, such as `f64`.
- The character type, `char`.
- Tuples, if they only contain types that are also `Copy`. For example, `(i32, i32)` is `Copy`, but `(i32, String)` is not.

Zwischenfragen soweit?

- Das hier ist gerade ein guter Zeitpunkt.

Ownership



```rust
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope. Nothing special happens.
```

Ownership zurück geben

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 goes out of scope but was
  // moved, so nothing happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                             // return value into the function
                                             // that calls it

    let some_string = String::from("hello"); // some_string comes into scope

    some_string                              // some_string is returned and
                                             // moves out to the calling
                                             // function
}

// takes_and_gives_back will take a String and return one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                      // scope

    a_string  // a_string is returned and moves out to the calling function
}
```

Das ist aber nicht immer cool

- Wie nervig… :/

```rust
fn main() {
    let s1 = String::from("hello");

    let (s2, len) = calculate_length(s1);

    println!("The length of '{}' is {}.", s2, len);
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

    (s, length)
}
```

References und Borrowing

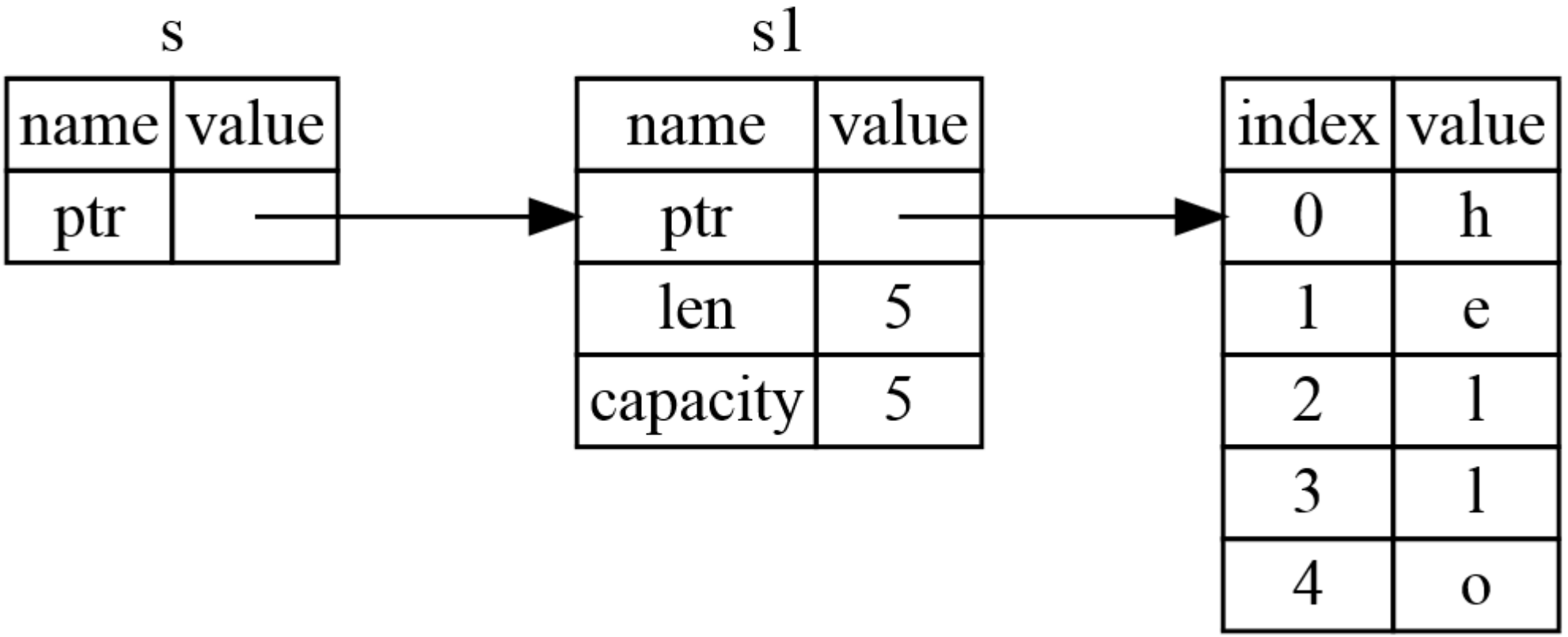
```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}
```

```rust
fn calculate_length(s: &String) -> usize { // s is a reference to a String
    s.len()
} // Here, s goes out of scope. But because it does not have ownership of what
  // it refers to, nothing happens.
```

- Referenz gekennzeichnet durch & (ist einfach ein Pointer)
- Nimmt nicht die Ownership weg

References und Borrowing

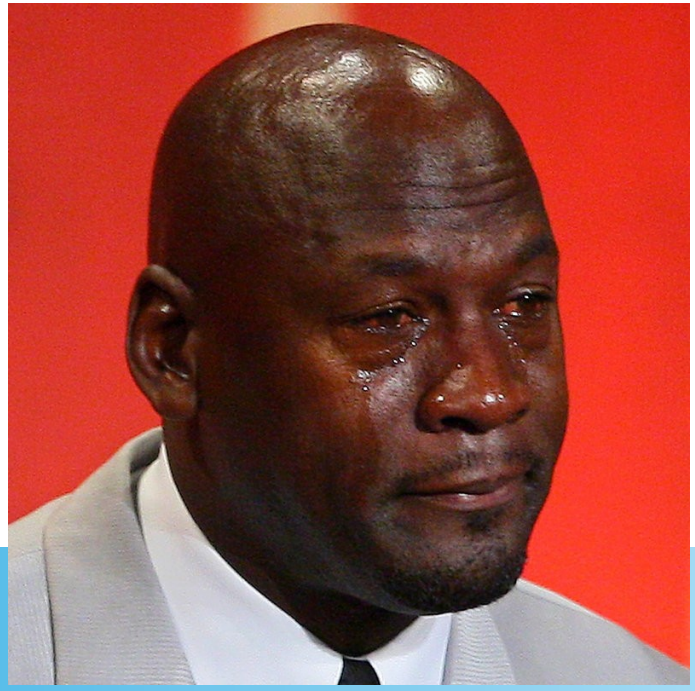
These ampersands are *references*, and they allow you to refer to some value without taking ownership of it. Figure 4-5 shows a diagram.



Spoiler: Es funktioniert nicht

```rust
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

- Warum nicht?



Spoiler: Es funktioniert nicht

```rust
fn main() {
    let s = String::from("hello");

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```


```
error[E0596]: cannot borrow immutable borrowed content `*some_string` as mutable
 --> error.rs:8:5
  |
7 | fn change(some_string: &String) {
  |                        ------- use `&mut String` here to make mutable
8 |     some_string.push_str(", world");
  |     ^^^^^^^^^^^ cannot borrow as mutable
```

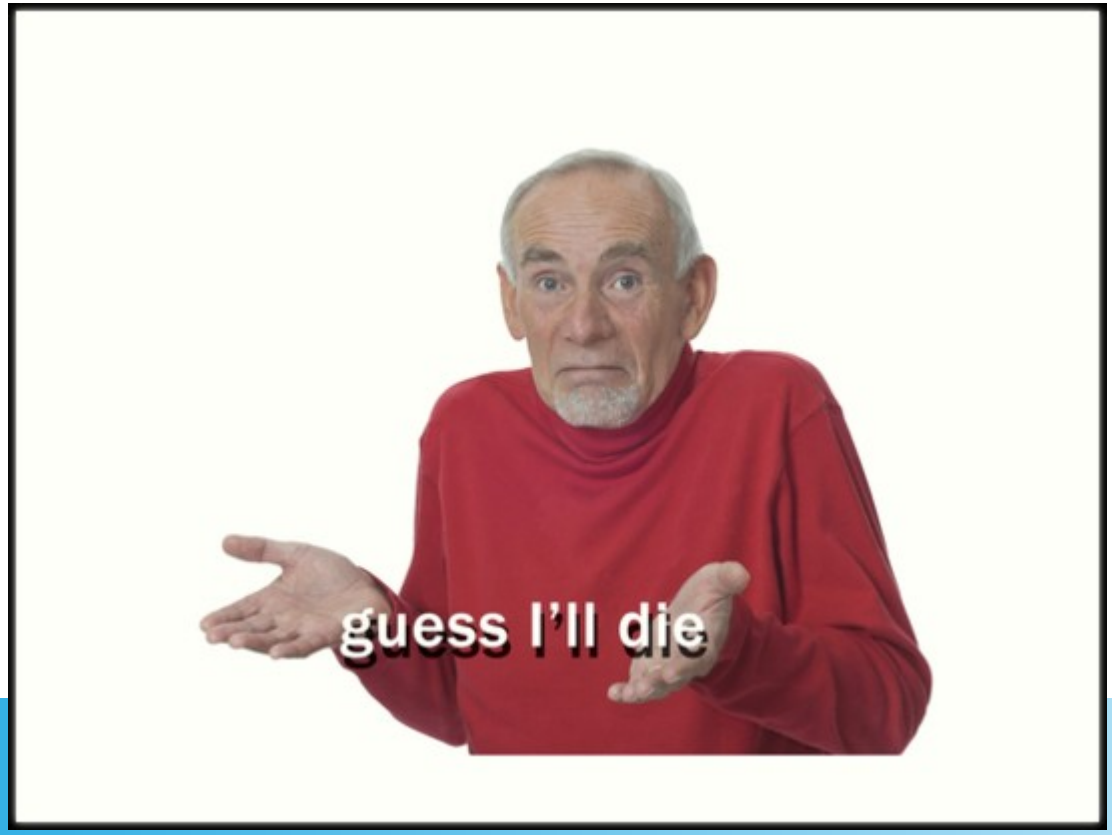
- Was muss man ändern?

Mutable Reference

```
fn main() {
    let mut s = String::from("hello");

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Aber mal wieder: Achtung!

```rust
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```



Aber mal wieder: Achtung!

```rust
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```

```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:10
  |
4 | let r1 = &mut s;
  |          ------ first mutable borrow occurs here
5 | let r2 = &mut s;
  |          ^^^^^^ second mutable borrow occurs here
6 | println!("{}, {}", r1, r2);
  |                    -- borrow later used here
```

- Kann man das umgehen?

Aber mal wieder: Achtung!

```
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2);
```


```
error[E0499]: cannot borrow `s` as mutable more than once at a time
 --> src/main.rs:5:10
  |
4 | let r1 = &mut s;
  |          ------ first mutable borrow occurs here
5 | let r2 = &mut s;
  |          ^^^^^^ second mutable borrow occurs here
6 | println!("{}, {}", r1, r2);
  |                    -- borrow later used here
```

- Kann man das umgehen? Nein
- Warum wäre es schlimm, wenn es ginge?

Antwort: Data Race

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There’s no mechanism being used to synchronize access to the data.
- „Rust prevents this problem from happening because it won’t even compile code with data races!“

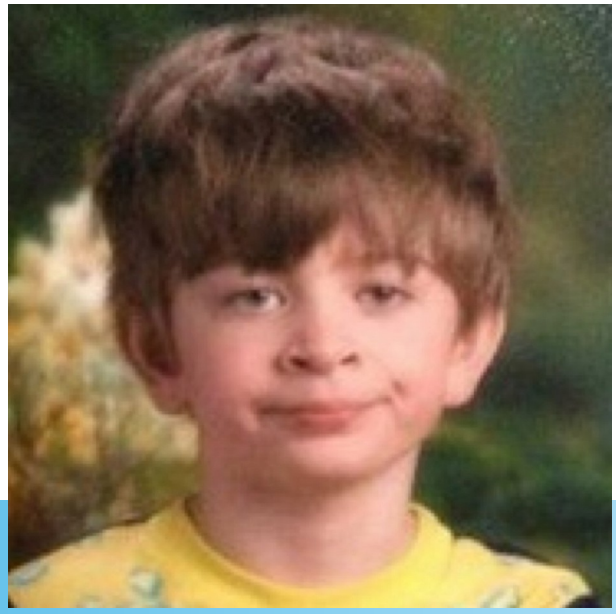
Sowas wäre aber OK

```
let mut s = String::from("hello");

{
    let r1 = &mut s;

} // r1 goes out of scope here, so we can make a new reference with no problems.

let r2 = &mut s;
```

Was ist damit?

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```

- Warum geht das jetzt nicht?



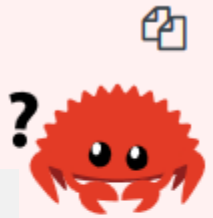
Was ist damit?

```
let mut s = String::from("hello");

let r1 = &s; // no problem
let r2 = &s; // no problem
let r3 = &mut s; // BIG PROBLEM

println!("{}, {}, and {}", r1, r2, r3);
```


```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:10
  |
4 | let r1 = &s; // no problem
  |          -- immutable borrow occurs here
5 | let r2 = &s; // no problem
6 | let r3 = &mut s; // BIG PROBLEM
  |          ^^^^^^ mutable borrow occurs here
7 |
8 | println!("{}, {}, and {}", r1, r2, r3);
  |                            -- borrow later used here
```

Dangling References

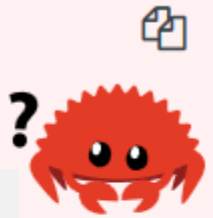
```
fn main() {
    let reference_to_nothing = dangle();
}
```




```
fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away.
  // Danger!
```

Dangling References

```
fn main() {
    let reference_to_nothing = dangle();
}
```




```
fn dangle() -> &String { // dangle returns a reference to a String

    let s = String::from("hello"); // s is a new String

    &s // we return a reference to the String, s
} // Here, s goes out of scope, and is dropped. Its memory goes away.
  // Danger!
```

```
error[E0106]: missing lifetime specifier
 --> main.rs:5:16
  |
5 | fn dangle() -> &String {
  |                ^ expected lifetime parameter
  |
  = help: this function's return type contains a borrowed value, but there is
  no value for it to be borrowed from
  = help: consider giving it a 'static lifetime
```

- Lösung?

Lösung!

```
fn no_dangle() -> String {
    let s = String::from("hello");

    s
}
```

- Ownership is moved out. Nothing deallocated.

Referenzen Zusammenfassung

- Regeln:
  - EINE mutable oder mehrere immutable Referenzen
  - Referenzen müssen IMMER gültig sein.

Slices

- Haben kein Ownership!
- Sind im Prinzip auch einfach nur Referenzen, aber Referenzen auf eine Teilmenge einer Collection.

Warum ist das schlecht?

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

Warum ist das schlecht?

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return i;
        }
    }

    s.len()
}
```

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s); // word will get the value 5

    s.clear(); // this empties the String, making it equal to ""

    // word still has the value 5 here, but there's no more string that
    // we could meaningfully use the value 5 with. word is now totally invalid!
}
```

- Und es wird nicht besser:

```
fn second_word(s: &String) -> (usize, usize) {
```

String Slices als Lösung

- Ein String Slice hat eine Referenz zu einem bestimmten String

Slice Syntax

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

```
let s = String::from("hello world");

let hello = &s[0..=4];
let world = &s[6..=10];
```

- Von bis
- Von bis einschließlich

```
let s = String::from("hello");

let slice = &s[0..2];
let slice = &s[..2];
```

By the same token, if your slice includes the last byte of the `String`, you can drop the trailing number. That means these are equal:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[3..len];
let slice = &s[3..];
```

You can also drop both values to take a slice of the entire string. So these are equal:

```
let s = String::from("hello");

let len = s.len();

let slice = &s[0..len];
let slice = &s[..];
```

Zurück zum Problem (und Lösung):

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();

    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }

    &s[..]
}
```

- Typ String Slice wird &str geschrieben

```
fn second_word(s: &String) -> &str {
```

Ein letztes mal Achtung!

```
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```

- Wer hat aufgepasst?

Ein letztes mal Achtung!

```rust
fn main() {
    let mut s = String::from("hello world");

    let word = first_word(&s);

    s.clear(); // error!

    println!("the first word is: {}", word);
}
```


```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
  --> src/main.rs:10:5
   |
8  |     let word = first_word(&s);
   |                           -- immutable borrow occurs here
9  |
10 |     s.clear(); // error!
   |     ^^^^^^^^^ mutable borrow occurs here
11 |
12 |     println!("the first word is: {}", word);
   |                                       ---- borrow later used here
```

String Literals are Slices

```
let s = "Hello, world!";
```

- S hat den Typ &str
- &str ist eine immuatble Referenz

String Slices as Parameters

```
fn first_word(s: &String) -> &str {
```

- Das geht, aber…

```
fn first_word(s: &str) -> &str {
```

- Das ist besser. Warum?

String Slices as Parameters

```
fn first_word(s: &String) -> &str {
```

- Das geht, aber…

```
fn first_word(s: &str) -> &str {
```

- Das ist besser. Warum?

→ Wir können die Funktion für String und &str benutzen

```
fn main() {
    let my_string = String::from("hello world");

    // first_word works on slices of `String`s
    let word = first_word(&my_string[..]);

    let my_string_literal = "hello world";

    // first_word works on slices of string literals
    let word = first_word(&my_string_literal[..]);

    // Because string literals *are* string slices already,
    // this works too, without the slice syntax!
    let word = first_word(my_string_literal);
}
```

Other Slices

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

let slice = &a[1..3];
```

- Hat den Typ &[i32]

Auf zum Live Coding!

Quellen

- https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html
- Und die Unterpunkte

- Zuletzt aufgerufen am 15. April 2019