



Hochschule **RheinMain**  
University of Applied Sciences  
Wiesbaden Rüsselsheim

# Rust

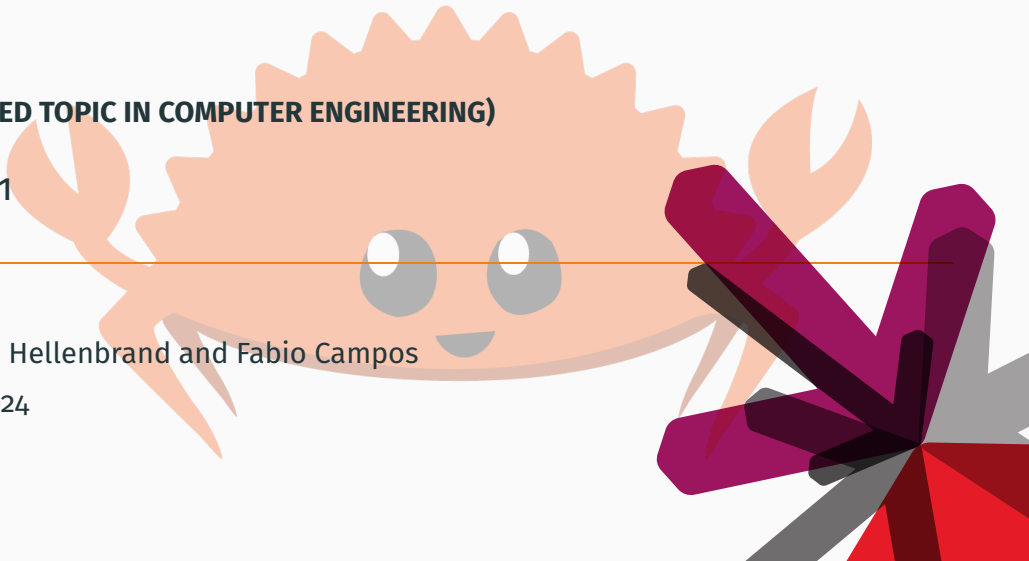
(SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281

---

Andreas Hellenbrand and Fabio Campos

25.04.2024



WICHTIG

Anmeldung QIS bis 06.05

# Ownership

---

# Ownership

- rust's most *unique* feature
- memory management
  - without *malloc* and *free*
  - without garbage collection

# Ownership Rules

1. Each value in Rust has a variable that's called it's owner.

# Ownership Rules

1. Each value in Rust has a variable that's called it's owner.
2. There can only be one owner at a time.

# Ownership Rules

1. Each value in Rust has a variable that's called it's owner.
2. There can only be one owner at a time.
3. Where the owner goes out of scope, the value will be dropped.

# Owner

```
1  fn main() {  
2      let y = 5;      // 5 is owned by "y"  
3      let x = "hello"; // "hello" is owned by "x"  
4  }
```



# Scope

defines where *things* are valid

```
1 fn main() {  
2     {  
3         // s is not valid here, it's not yet declared  
4         let s = "hello"; // s is valid from this point forward  
5  
6         // do stuff with s  
7     }  
8     // this scope is now over, and s is no longer valid  
9 }
```

## easy with primitive types

```
1  fn main() {  
2      let x = 5;  
3      let y = x; // the value 5 gets copied to y  
4      // now x = 5 and y = 5  
5  }
```

# move

```
1  fn main() {  
2      let s1 = String::from("hello");  
3      let s2 = s1; // owner ship gets MOVED from s1 to s2  
4      // s1 can now NOT be used  
5  }
```

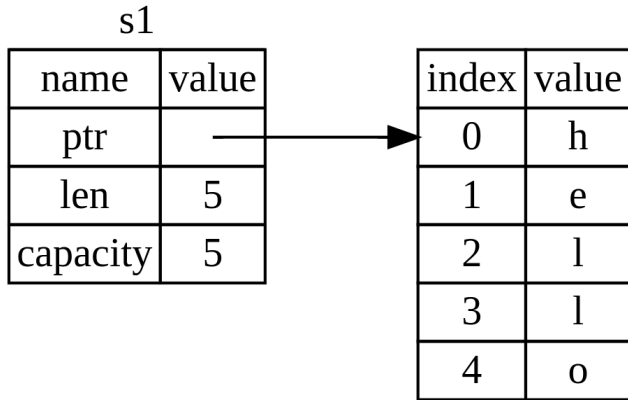
# Memory and Allocation

- *copying* primitive types is cheap
  - primitive = stack-only
    - integers, boolean, floats, ...
  - known size
  - *small-ish*

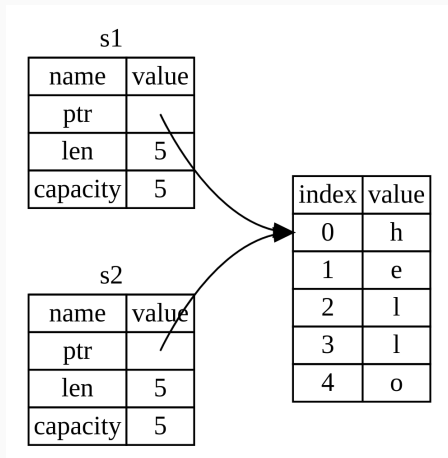
# Memory and Allocation

- *copying* primitive types is cheap
  - primitive = stack-only
    - integers, boolean, floats, ...
  - known size
  - *small*-ish
- but strings and arrays cannot be copied for cheap
  - unknown size at compile time
  - can be large

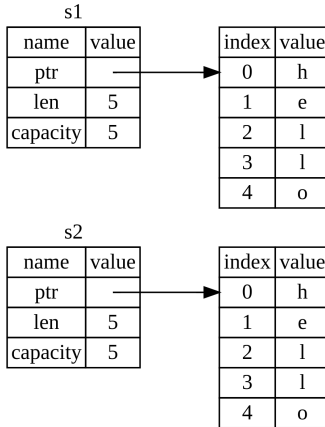
## move - String



## move - copy

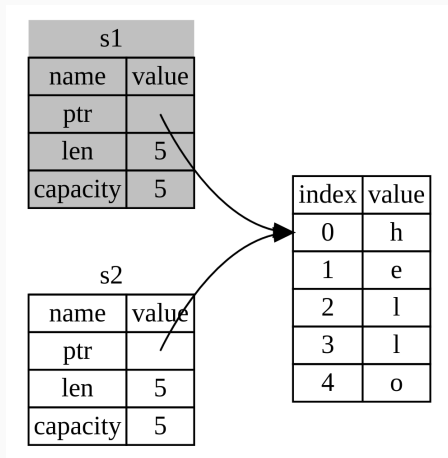


# move - deep copy





# move - the rust way



# clone

```
1  fn main() {  
2      let s1 = String::from("hello");  
3      let s2 = s1.clone();    // deep copy  
4      // now s1 = "hello" and s2 = "hello"  
5  }
```

# clone

```
1  fn main() {  
2      let s1 = String::from("hello");  
3      let s2 = s1.clone();    // deep copy  
4      // now s1 = "hello" and s2 = "hello"  
5  }
```

careful: (could be) expensive!

# Functions

---

# Question 1

```
1  fn main() {  
2      let mut x = 5;  
3      have_fun(x);  
4      println!("{}", x); // what is printed?  
5  }  
6  fn have_fun(mut y: i32) {  
7      y += 15;  
8      println!("{}", y); // what is printed?  
9  }
```

## Question 2

```
1  fn main() {  
2      let x = String::from("Star ");  
3      have_fun(x);  
4      println!("{}", x);  
5  }  
6  fn have_fun(mut y: String) {  
7      y.push_str("{Wars|Trek}");  
8  }
```

# Solutions?

Any Ideas?

# Take and give back

```
1  fn main() {  
2      let x = String::from("don't do drugs");  
3      let y = have_fun(x);  
4  }  
5  fn have_fun(y: String) -> String {  
6      println!("{}", y);  
7      y  
8  }
```



# Take and give back

```
1  fn main() {  
2      let s = String::from("rustacean");  
3      let (s, len) = string_length(s);  
4  }  
5  fn string_length(s: String) -> (String, usize) {  
6      let len = s.len();  
7      (s, len)  
8  }
```

# Take and give back

```
1  fn main() {  
2      let s = String::from("rustacean");  
3      let (s, len) = string_length(s);  
4  }  
5  fn string_length(s: String) -> (String, usize) {  
6      let len = s.len();  
7      (s, len)  
8  }
```

kind of annoying

**Borrowing FTW!**

---

# Reference

Idea:

- we don't *move* *s* into the function
- only give it a *reference*

# Reference

Idea:

- we don't *move* *s* into the function
- only give it a *reference*

*compare: pointers*

```
1 fn main() {
2     let s = String::from("cats > dogs");
3     let len = string_length(&s); // ownership stays at "s"
4         // ^ only a reference
5 }
6 fn string_length(y: &String) -> usize {
7     // ^ "y" is only a reference
8     y.len()
9 }
```

# &mut

```
1 fn main() {  
2     let mut x = String::from("Star ");  
3     have_fun(&mut x);  
4     println!("{}", x);  
5 }  
6 fn have_fun(y: &mut String) {  
7     y.push_str("{Wars|Trek}");  
8 }
```

## Question 3

```
1      let mut s = String::from("<3");  
2      let r1 = &s;  
3      let r2 = &s;
```



## Question 3

```
1      let mut s = String::from("<3");
2      let r1 = &s;
3      let r2 = &s;

1      let r3 = &mut s;
```

# Returning References?

```
1  fn main() {  
2      let ref_to_where = dangle();  
3  }  
4  
5  fn dangle() -> &String {  
6      let s = String::from("welp");  
7      &s  
8  } // s goes out of scope and "s" is dropped
```

# Returning References?

```
1  fn main() {  
2      let ref_to_where = dangle();  
3  }  
4  
5  fn dangle() -> &String {  
6      let s = String::from("welp");  
7      &s  
8  } // s goes out of scope and "s" is dropped
```

Compiler says no!

# Rules of References

1. At any given time, you can have *either* but not both of:
  - One mutable reference
  - Any number of immutable references
2. References must always be valid

# Slices

---

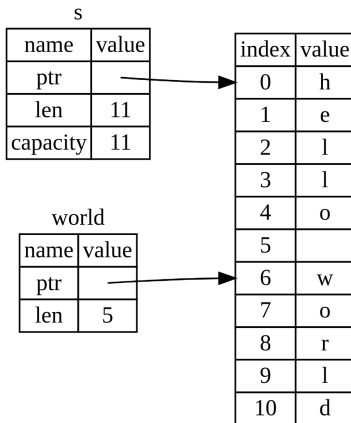
# Example (bad)

```
1  fn first_word(s: &String) -> usize {
2      // return index of first space
3  }
4  fn main(){
5      let mut s = String::from("we love rust");
6      let word = first_word(&s); // word = 2
7      s.clear();                // s = 0
8      // word/2 now means nothing!
9  }
10
```

Slices let us reference a continuous sequence of elements in a collection!

```
1      let s = String::from("hello world");  
2      let hello = &s[0..5];  
3      let world = &s[6..11];
```

# Slice





## Example (good)

```
1  fn first_word(s: &String) -> &str {
2      let bytes = s.as_bytes();
3      for (i, &item) in bytes.iter().enumerate() {
4          if item == b' '; {
5              return &s[0..i];
6          }
7      }
8      &s[..]
9  }
```

## Example (good)

```
1  fn first_word(s: &String) -> &str {
2      let bytes = s.as_bytes();
3      for (i, &item) in bytes.iter().enumerate() {
4          if item == b' '; {
5              return &s[0..i];
6          }
7      }
8      &s[..]
9  }
```

String has its own slice type: *&str*

## Example (good)

```
1  fn first_word(s: &String) -> &str {  
2      // return string slice of first word  
3  }  
4  fn main(){  
5      let mut s = String::from("bald geschafft");  
6      let word = first_word(&s);  
7      s.clear();    // Error!  
8  }
```

# Other Slices

also works for collections!

```
1      let a = [1,2,3,4,5,6];  
2      let slice = &a[1..3];    // &i32]
```

# Conclusion

---

# Conclusion

- *strict* ownership rules
- ownership, borrowing and slices ensure memory safety
- checked at compile time

all your base are belong to us!  
Fragen?