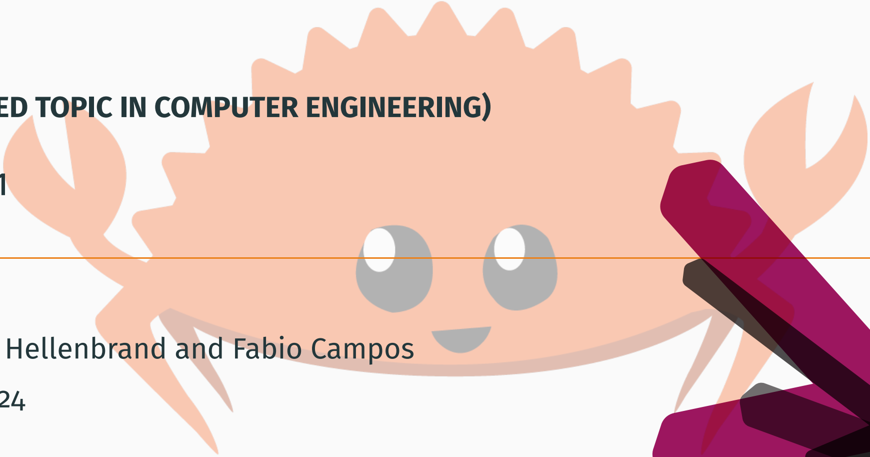
Hochschule RheinMain
University of Applied Sciences
Wiesbaden Rüsselsheim

Rust

(SELECTED TOPIC IN COMPUTER ENGINEERING)

LV 7281



Andreas Hellenbrand and Fabio Campos

25.04.2024

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

```
fn main() {
    let y = 5;         // 5 is owned by "y"
    let x = "hello";   // "hello" is owend by "x"
}
```

Scope

defines where *things* are valid

```
fn main() {
    {
        // 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    // this scope is now over, and s is no longer valid
}
```

copy

easy with primitive types

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

move

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // owner ship gets MOVEd from s1 to s2
    // s2 can now NOT be used
}
```

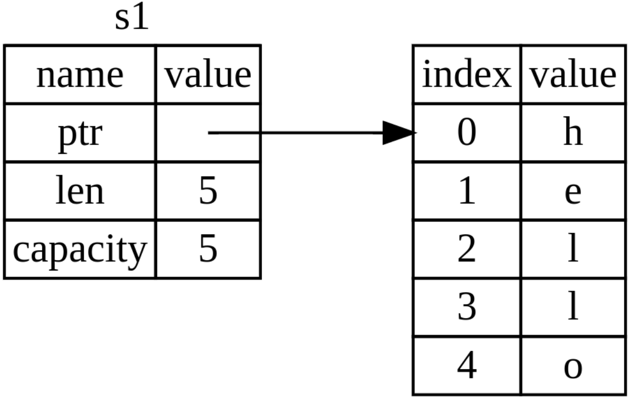
Memory and Allocation

- *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

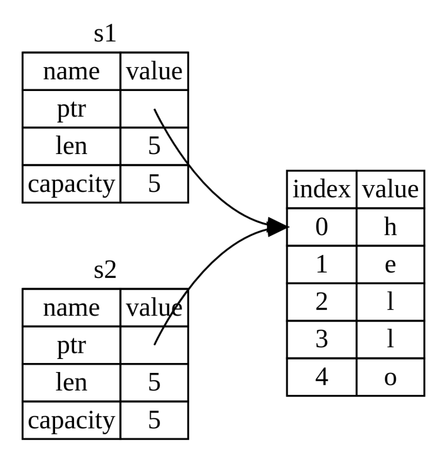


s1

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

move - copy



s1

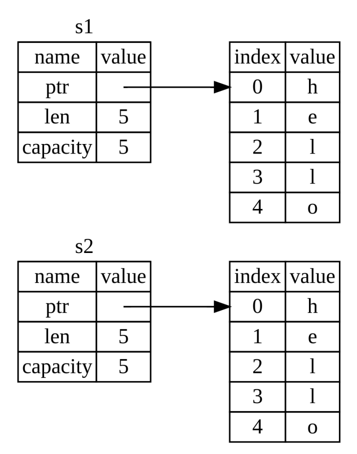
name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

move - deep copy



s1

name	value
ptr	
len	5
capacity	5

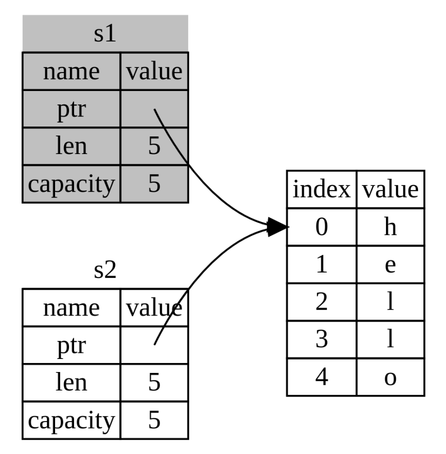
index	value
0	h
1	e
2	l
3	l
4	o

s2

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

move - the rust way



s1

name	value
ptr	
len	5
capacity	5

s2

name	value
ptr	
len	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

clone

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

clone

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

careful: (could be) expensive!

Functions

Question 1

```rust
fn main() {
    let mut x = 5;
    have_fun(x);
    println!("{}", x);  // what is printed?
}
fn have_fun(y: mut i32) {
    y += 15;
    println!("{}", y);  // what is printed?
}
```

Question 2

```rust
fn main() {
    let x = String::from("Star ");
    have_fun(x);
    println!("{}", x);
}
fn have_fun(y: mut String) {
    y.push_str("{Wars||Trek}");
}
```

Solutions?

Any Ideas?

Take and give back

```rust
fn main() {
    let x = String::from("don't do drugs");
    let y = have_fun(x);
}
fn have_fun(y: String) -> String {
    println!("{}", y);
    y
}
```

Take and give back

```
fn main() {
    let s = String::from("rustacean");
    let (s, len) = string_length(s);
}
fn string_length(s: String) -> (String, usize) {
    let len = s.len();
    (s, len)
}
```

Take and give back

```rust
fn main() {
    let s = String::from("rustacean");
    let (s, len) = string_length(s);
}
fn string_length(s: String) -> (String, usize) {
    let len = s.len();
    (s, len)
}
```

kind of anoying

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

&

```
fn main() {
    let s = String::from("cats > dogs");
    let len = string_length(&s);  // ownership stays at "s"
                        //  ^  only a reference
}
fn string_length(y: &String) -> usize {
                //  ^ "y" is only a reference
    y.len()
}
```

&mut

```
fn main() {
    let mut x = String::from("Star ");
    have_fun(&mut x);
    println!("{}", x);
}
fn have_fun(y: &mut String) {
    y.push_str("{Wars||Trek}");
}
```

Question 3

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

Question 3

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

```rust
let r3 = &mut s;
```

Returning References?

```rust
fn main() {
    let ref_to_where = dangle();
}

fn dangle() -> &String {
    let s = String::from("welp");
    &s
}   // s goes out of scope and "s" is dropped
```

Returning References?

```rust
fn main() {
    let ref_to_where = dangle();
}

fn dangle() -> &String {
    let s = String::from("welp");
    &s
}   // 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)

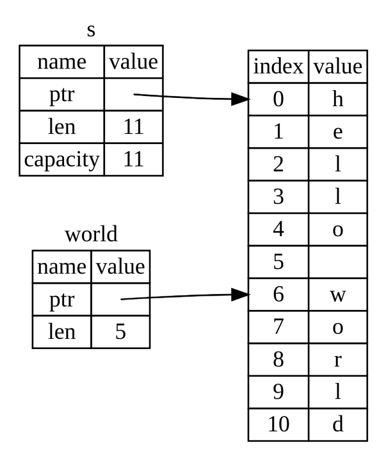
```rust
fn first_word(s: &String) -> usize {
    // return index of first space
}
fn main(){
    let mut s = String::from("we love rust");
    let word = first_word(&s); // word = 2
    s.clear();                 // s = 0
    // word/2 now means nothing!
}

```

Slices

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

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

Slice



s

name	value
ptr	
len	11
capacity	11

world

name	value
ptr	
len	5

index	value
0	h
1	e
2	l
3	l
4	o
5	
6	w
7	o
8	r
9	l
10	d

Example (good)

```
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[..]
}
```

Example (good)

```rust
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[..]
}
```

String has its own slice type: `&str`

Example (good)

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

Other Slices

also works for collections!

```
let a = [1,2,3,4,5,6];
let slice = &a[1..3];   // &[i23]
```

Conclusion

Conclusion

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

all your base are belong to us!

Fragen?