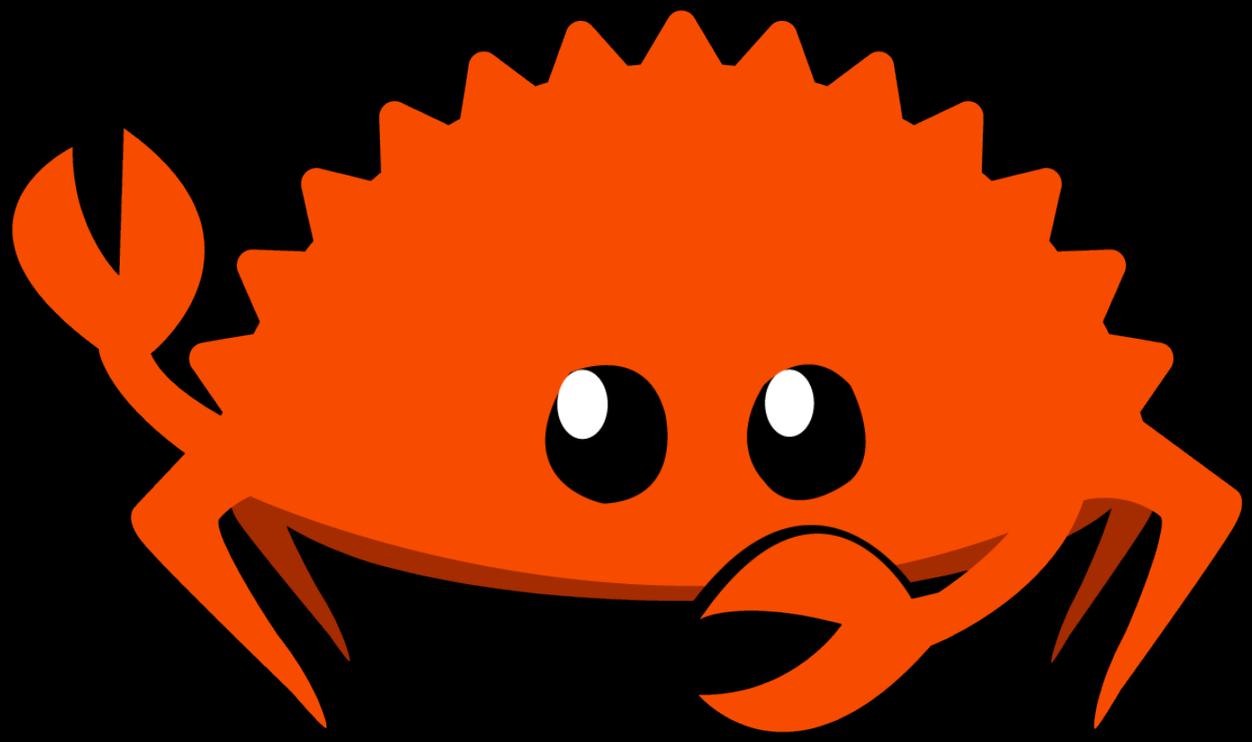


The Beauty of Rust

Structure

Peter Zdankin

Rust is modern

- Many features of modern language design are embedded into the syntax of Rust
- Iterators, map, filter, reduce, type inference
- Code looks more readable than e.g. C/C++
- Implicit what is inferrable
- Explicit what requires specification

```c
#include <stdio.h>
#include <stdint.h>
#define MAX_NUMBERS 500000

int isPrime(uint64_t num){
    for(uint64_t i = 2; i < num; i++){
        if(i*i > num){
            break;
        }
        if(num % i == 0){
            return 0;
        }
    }
    return 1;
}

int main(void){
    uint64_t sum = 0;
    for(uint64_t i = 2; i < MAX_NUMBERS; i++){
        if(isPrime(i)){
            sum += i;
        }
    }
    printf("%llu\n", sum);
}
```

```
➜ time ./a.out

9914236195

./a.out  0,22s user 0,00s system 99% cpu 0,228 total
```

```python
def is_prime(num):
    for i in range(2, num):
        if i*i > num:
            break
        if num % i == 0:
            return False
    return True


print(sum(filter(is_prime, range(2,500000))))
```

```
➜ time python3 foobar.py

9914236195

python3 foobar.py  3,77s user 0,03s system 99% cpu 3,812 total
```

```rust
fn is_prime(num: u64) -> bool {
    for i in 2..num {
        if i * i > num {
            break;
        }
        if num % i == 0 {
            return false;
        }
    }
    true
}

fn main() {
    let sum: u64 = (2..500000).filter(|&x| is_prime(x)).sum();
    println!("{}", sum);
}
```

```
➜ time ./foobar

9914236195

./foobar  0,23s user 0,00s system 99% cpu 0,231 total
```

Common programming techniques

- „Repeat this code if condition holds“
- „Branch into either of these blocks, depending on this condition“
- „Extract this code into a unit“
- „Give me some memory that I can interpret as a type“

„Repeat this code if condition holds“

While loop


```rust
let x = 5;
let mut counter = 0;
while counter < x {
    println!("Brrrt");
    counter += 1;
}
```

„Repeat this code if condition holds“

For range

```
for i in 0..10{
    println!("{i}");
}
```

„Repeat this code if condition holds“

Iterate over elements

```rust
let v = vec![10,20,30,40,50];
for element in v{
    println!("{element}");
}
```

INFINITE LOOP?

```
while true{
    println!("SOY");
}
```

```
loop{
    println!("BRRRT");
}
```

Branch into either of these blocks, depending on this condition

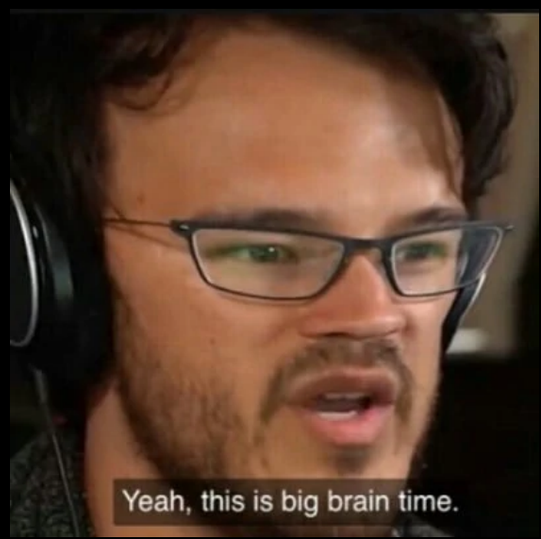
IF/ELSE

```
let a = 5;
let b = 3;
if a > b{
    println!("BIG");
} else{
    println!("SMOL");
}
```

Branch into either of these blocks, depending on this condition

Match

```
for i in 1..200{
    match i % 5{
        0 => println!("5!"),
        1..=3 => println!("lame"),
        _ => println!("else")
    }
}
```



„Extract this code into a unit“

Functions

```rust
fn add_3(number: i32) -> i32{
    number + 3
}
```

„Extract this code into a unit“

Structs and methods

```
struct Adder{
    number: i32
}

impl Adder{

    fn new(number: i32) -> Adder{
        Adder{number: number}
    }

    fn add_3(&self) -> i32{
        self.number + 3
    }

    fn add_5(&self) -> i32{
        self.number + 5
    }
}
```

```
let adder = Adder::new(5);
println!("{}", adder.add_3());
println!("{}", adder.add_5());
```

Primer

Functions vs Methods

- Functions are pure, have a fixed set of input parameters and a fixed set of output parameters
- Methods are associated with a type and operate on it
- Methods implicitly have all member variables of an object as additional parameters

BEACH BREAK

Give me some memory that I can interpret as a type

- In most programming languages a lot of this is abstracted absurdly
- You say „a = [1,2,3,4]“ and have a list that contains 4 numbers and you can refer to the list as „a“.
- How big are the numbers in memory? 8 Bit? 16 Bit? 32 Bit?
- Are they signed/unsigned?
- How does the list grow/shrink?
- When will the memory be freed?
- What happens when I say „b = a“?

Numbers in Rust

- In most C-style languages, you work with int, double, float, long, etc.
- Difficult to say how many bytes in memory
- Difficult to say if it’s consistent across platforms

- In Rust you say {u|i}{8|16|32|64}
- e.g. u8 for an unsigned 8 bit number
- e.g. i32 for a signed 32 bit number
- usize/isize for address size (e.g. 32 Bit on 32 Bit architecture)

Floating point numbers and the rest

- f32/f64 are your usual suspects
- boolean (duh)
- char/String (utf-8 encoded)

How does a buffer like type look like?

E.g. String/Vector/List

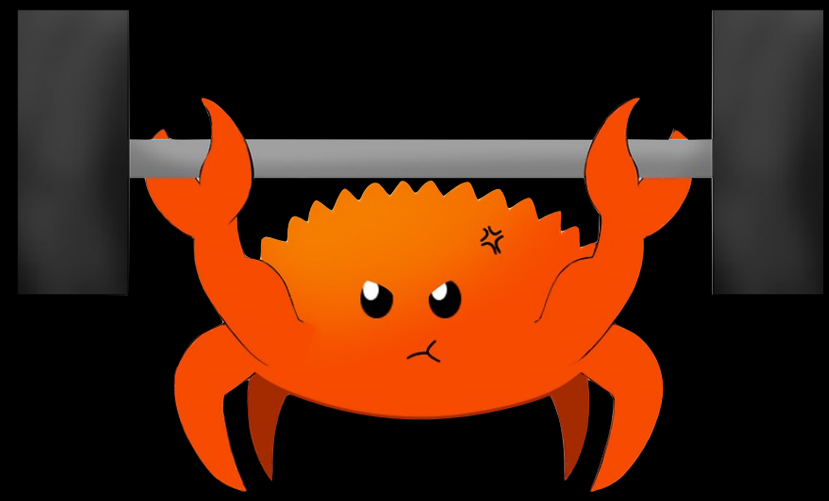
- In C: You have an address and have to know its bounds
- In Rust: Pointer + Length + Capacity
- Writing/Accessing is always bounds-checked
- Runtime error if you screw up

OWNERSHIP

How does Rust know when to drop memory?

- No Garbage collection, as it is too slow, inefficient and stops the world until it is finished
- No manual memory management, as programmers have problems doing it right EVERY time

The Holy Rust Mantra



- „Each value in Rust has a variable that’s called its *owner*.“
- „There can only be one owner at a time.“
- "When the owner goes out of scope, the value will be dropped.“

The role of the owner

- There is only one owner of each piece of memory
- When the owner is out of scope, its associated memory is freed!
- Hence, all references must live shorter than the owner!

Let’s do an example

Sum the numbers in a vector

```rust
fn sum_of_vector(vec: Vec<i32>) -> i32{
    let mut sum = 0;
    for elem in vec{
        sum += elem;
    }
    sum
}

fn main() {

    let v = vec![1,2,3,4];
    println!("{}", sum_of_vector(v));
    println!("{}", sum_of_vector(v));
```

WHY??

Ownership can be moved

- We move the ownership of the vector into the function
- Once the function is finished, the value is dropped
- Hence, the next call fails as the vector is dropped already..

```
error[E0382]: use of moved value: `v`
  --> src/main.rs:37:34
   |
35 |     let v = vec![1,2,3,4];
   |         - move occurs because `v` has type `Vec<i32>`, which does not implement the `Copy` trait
36 |     println!("{}", sum_of_vector(v));
   |                                   - value moved here
37 |     println!("{}", sum_of_vector(v));
   |                                   ^ value used here after move
```

Ok lets fix

```rust
fn sum_of_vector(vec: Vec<i32>) -> (Vec<i32>,i32){
    let mut sum = 0;
    for elem in vec{
        sum += elem;
    }
    (vec,sum)
}
```

```
error[E0382]: use of moved value: `vec`
   --> src/main.rs:30:6
    |
25  | fn sum_of_vector(vec: Vec<i32>) -> (Vec<i32>,i32){
    |                  --- move occurs because `vec` has type `Vec<i32>`, which does not implement the `Copy` trait
26  |     let mut sum = 0;
27  |     for elem in vec{
    |                 ---
    |                 |
    |                 `vec` moved due to this implicit call to `.into_iter()`
    |                 help: consider borrowing to avoid moving into the for loop: `&vec`
...
30  |     (vec,sum)
    |      ^^^ value used here after move
```

Ok lets fix (second try)

```rust
fn sum_of_vector(vec: Vec<i32>) -> (Vec<i32>,i32){
    let mut sum = 0;
    for elem in &vec{
        sum += elem;
    }
    (vec,sum)
}

fn main() {

    let v = vec![1,2,3,4];
    let (v, res) = sum_of_vector(v);
    println!("{}", res);
    let (v, res) = sum_of_vector(v);
    println!("{}", res);
```

BORROWING

- If you want to use something, you mustn’t own it
- You can borrow and return it

- You can either borrow and just read it, or you can borrow and do something with it
- Logically, either one person can do something with the borrowed thing, or many people can read the borrowed thing

There can be either exactly one mutable borrow (&mut T)

OR

An infinite number of read-only borrows (&T)

```rust
fn sum_of_vector(vec: &Vec<i32>) -> i32{
    let mut sum = 0;
    for elem in vec{
        sum += elem;
    }
    sum
}


fn main() {

    let v = vec![1,2,3,4];
    println!("{}", sum_of_vector(&v));
    println!("{}", sum_of_vector(&v));
```

Mutable borrow

```rust
fn add_to_vector(vec: &mut Vec<i32>, value: i32){
    vec.push(value);
}

fn main() {

    let v = vec![1,2,3,4];
    add_to_vector(&mut v, 12);
    println!("{}", sum_of_vector(&v));
    println!("{}", sum_of_vector(&v));
```

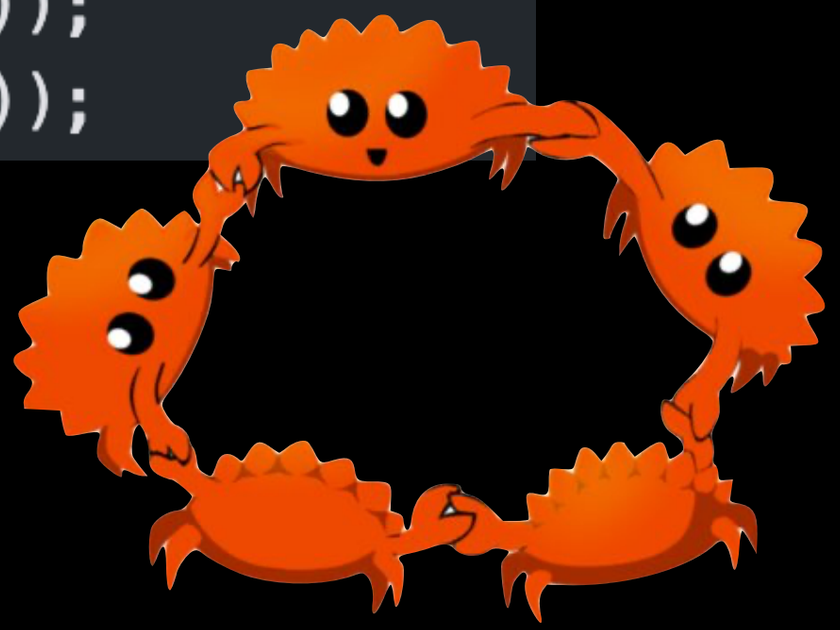
```
error[E0596]: cannot borrow `v` as mutable, as it is not declared as mutable
  --> src/main.rs:40:19
   |
39 |     let v = vec![1,2,3,4];
   |         - help: consider changing this to be mutable: `mut v`
40 |     add_to_vector(&mut v, 12);
   |                   ^^^^^^ cannot borrow as mutable
```

Mutable borrow

```rust
fn add_to_vector(vec: &mut Vec<i32>, value: i32){
    vec.push(value);
}

fn main() {

    let mut v = vec![1,2,3,4];
    add_to_vector(&mut v, 12);
    println!("{}", sum_of_vector(&v));
    println!("{}", sum_of_vector(&v));
```

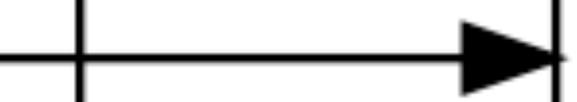


Stack and Heap Management

- Variables with a known size get stored on the Stack
- Integers, static Strings, constant Strings, etc.
- Variables with a variable Size (mutable Strings, Vectors) have a Stack and a Heap part
- On the Stack is known sized metadata (Pointer, Length, Capacity, etc.)
- Heap stores actual Data

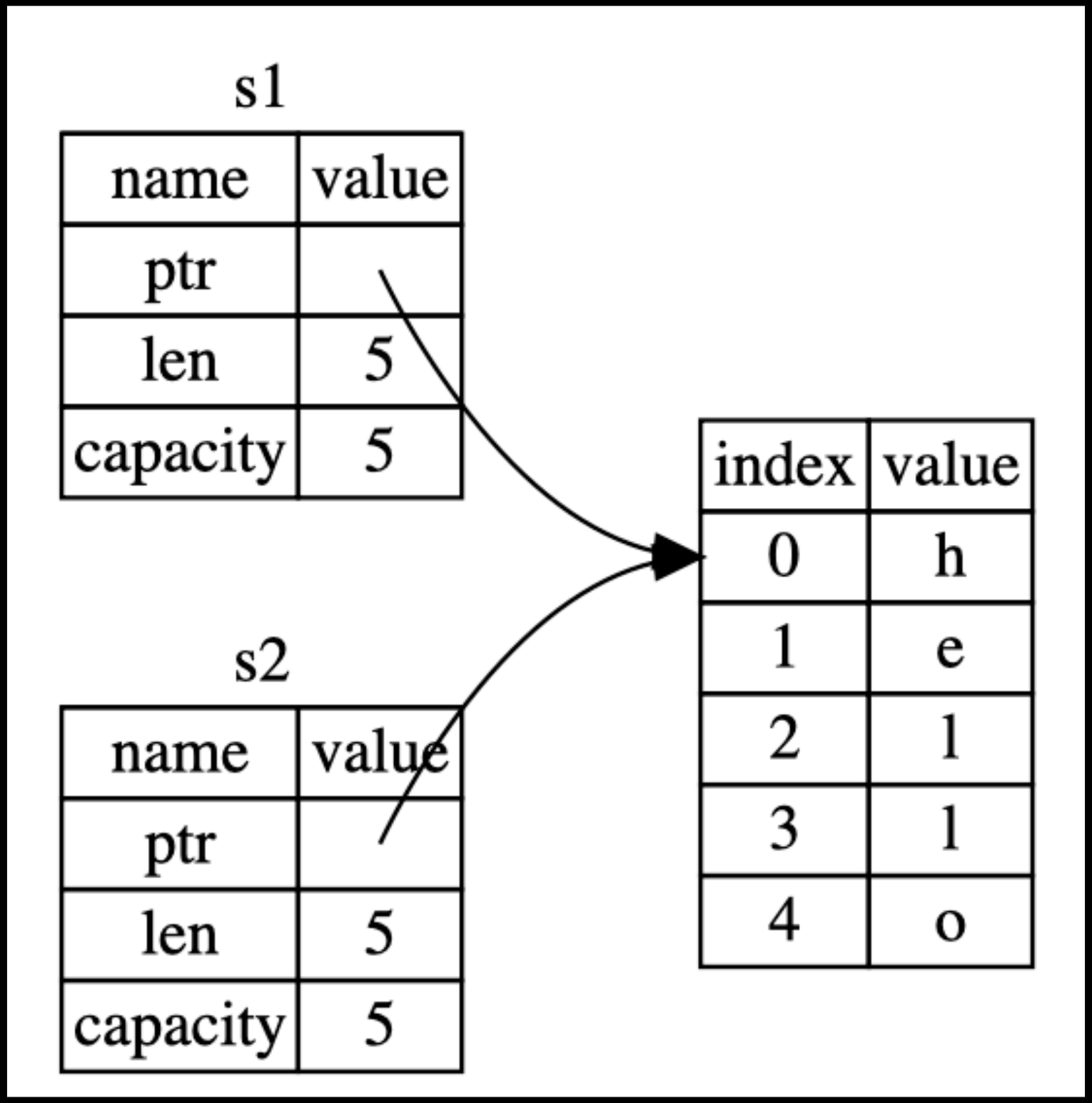
s1

name	value
ptr	
len	5
capacity	5

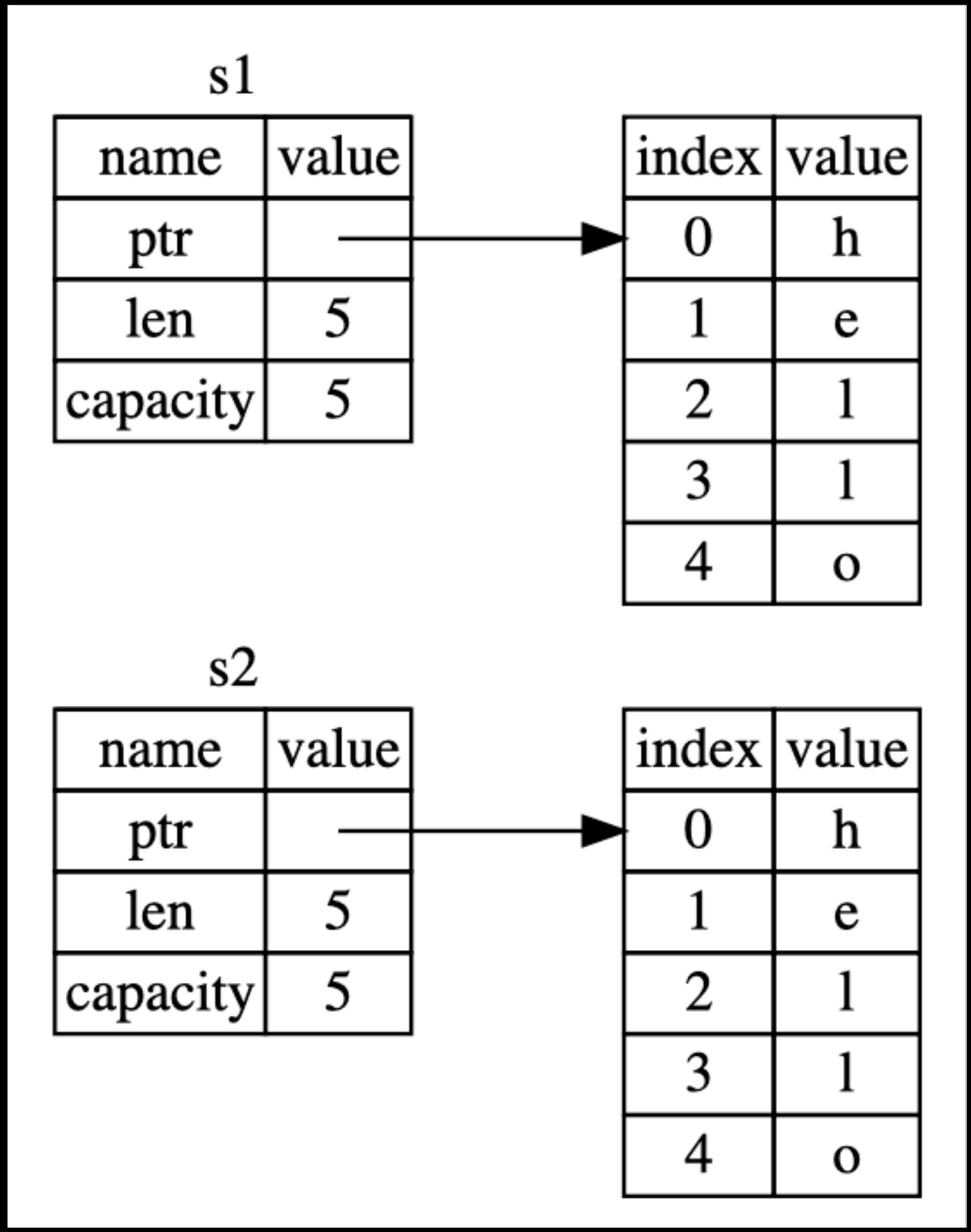


index	value
0	h
1	e
2	l
3	l
4	o

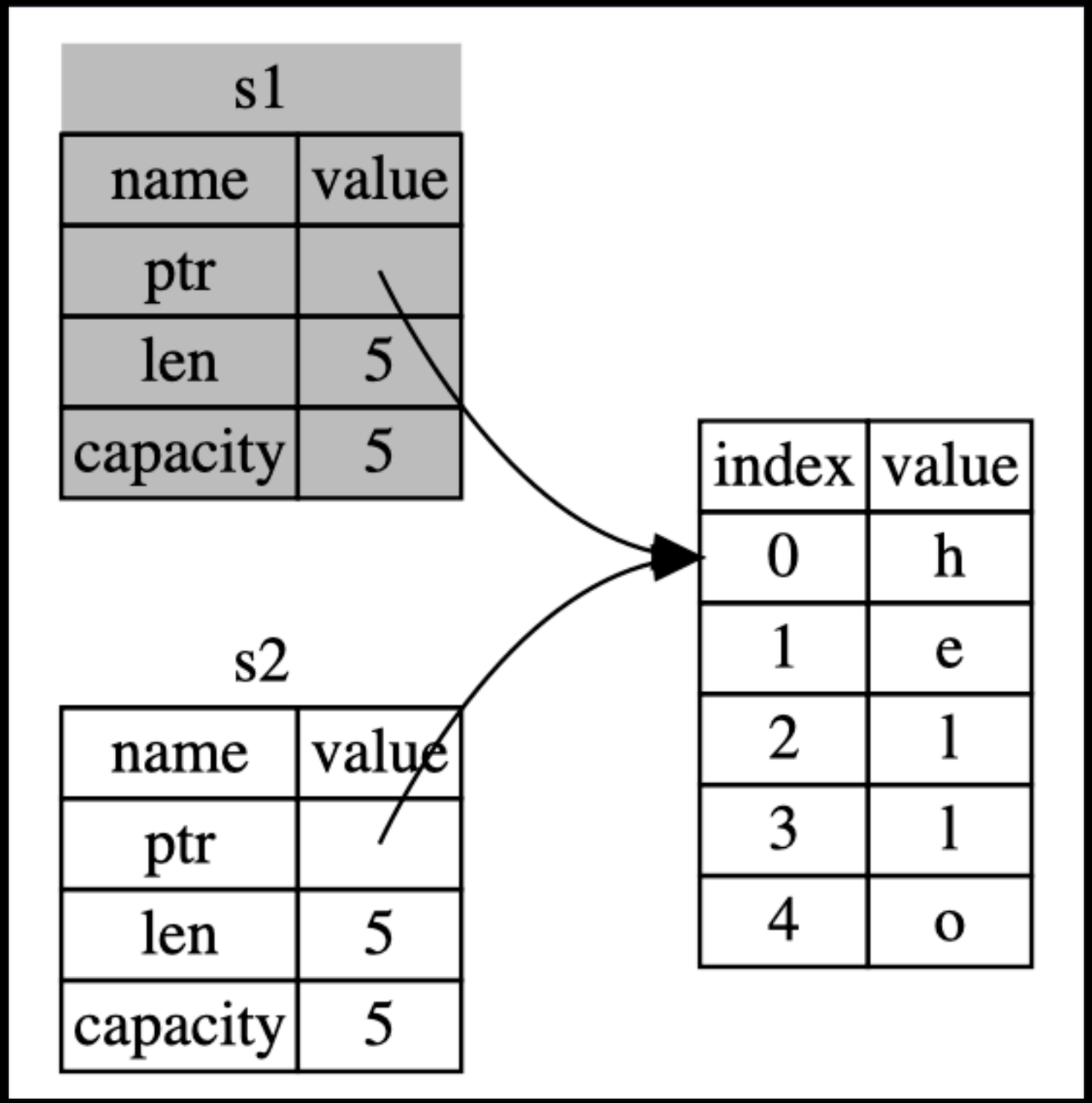
s2 = s1


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

s2 = s1


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

s2 = s1


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

Exercises!

- Write a program that prints all odd numbers between 1 and 100
- Write a program that prints the cube of all numbers in a vector
- Write a function, that returns the number divided by 2 if it is even, else returns 3*number + 1 (try looping it)
- Write a function that calculates the product of a vector
- Write a struct for rectangles and circles that each have an get_area method
- Write a 3D Point struct and create methods to get the length of a vector and the normalised vector