Wer rastet, der rostet

Ein Rust-Kurs für Anfänger

Hendrik Wolff

Anton Obersteiner

Sommersemester 2023

Ownership

- memory is managed through system of ownership
  - a set of rules checked by the compiler
- memory safety guarantees without a garbage collector
- rule violation ⇒ compile error
- no runtime overhead

Ownership rules

From the original Rust Book:

- Each value in Rust has an **owner**.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be *dropped*.

The String type

- can store data of arbitrary length
- meta data stored on stack
- content allocated on heap

The String type

- can store data of arbitrary length
- meta data stored on stack
- content allocated on heap

```
{
    // allocates memory to store
    // "hello" on the heap
    let s1 = String::from("hello");

    // s1 is valid here
}
// s1 is out of scope
// String will be dropped, memory deallocated
```

The String type

- can store data of arbitrary length
- meta data stored on stack
- content allocated on heap

```
{
    // allocates memory to store
    // "hello" on the heap
    let s1 = String::from("hello");

    // s1 is valid here
}
// s1 is out of scope
// String will be dropped, memory deallocated
```

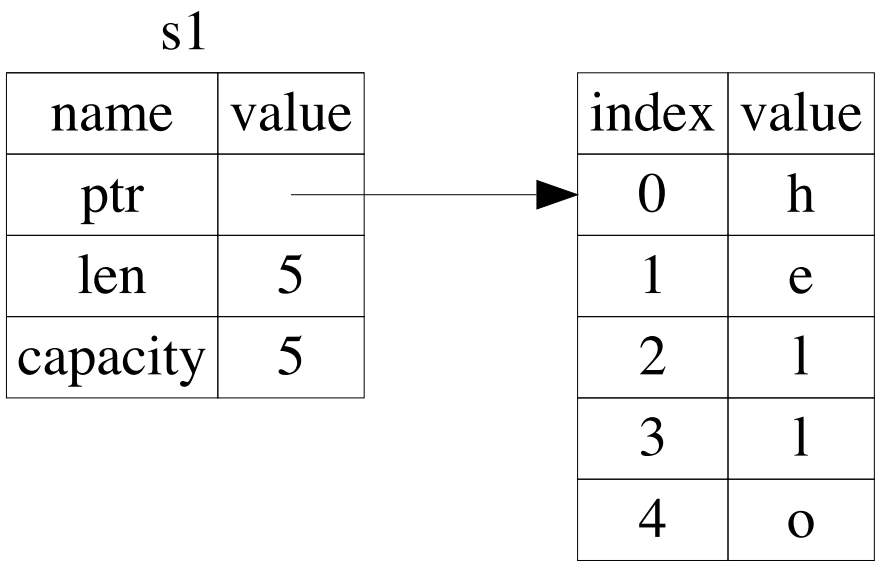


Figure 1: Memory representation of s1.

Move Ownership out of function

```
fn main() {
  let outer_string = foo();
  println!("{}", outer_string);
}

fn foo() -> String {
  let inner_string = String::from("hello world");
  inner_string
}
```

Is this valid code?

```
fn strings() -> String {
  let original = String::from("hello");
  let new_owner = original;

  println!("{}", original);
  new_owner
}
```

Is this valid code?

```
fn strings() -> String {
  let original = String::from("hello");
  let new_owner = original;

  // new_owner takes the ownership from original
  // which means original does not own it anymore
  // „one owner“-Rule

  println!("{}", original);
  new_owner
}
```

Move Ownership into function

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



  println!("|{}| == {}", s, len);
}

fn get_length(some_string: String) -> usize {
  println!("{}", some_string);
  some_string.len()


}
```

Move Ownership into function

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



  println!("|{}| == {}", s, len);
}

fn get_length(some_string: String) -> usize {
  println!("{}", some_string);
  some_string.len()


}
```

in this code:

- where are strings moved?
- where are strings dropped?

Move Ownership into function

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

  // s has been moved into function,
  // is invalid now

  println!("|{}| == {}", s, len);
}

fn get_length(some_string: String) -> usize {
  println!("{}", some_string);
  some_string.len()


}
```

in this code:

- where are strings moved?
- where are strings dropped?

Move Ownership into function

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

  // s has been moved into function,
  // is invalid now

  println!("|{}| == {}", s, len);
}

fn get_length(some_string: String) -> usize {
  println!("{}", some_string);
  some_string.len()

  // some_string is dropped here

}
```

in this code:

- where are strings moved?
- where are strings dropped?

References and Borrowing

```
fn main() {
  let s = String::from("hello");
  let len = get_length(&s);
  println!("|{}| == {}", s, len);
}
```

```
fn get_length(some_string: &String) -> usize {
  some_string.len()
}
```

References and Borrowing

```rust
fn main() {
  let s = String::from("hello");
  let len = get_length(&s);
  println!("|{}| == {}", s, len);
}

// here, get_length borrows the string,
// it doesn't move it and doesn't own it

fn get_length(some_string: &String) -> usize {
  some_string.len()
}
```

References and Borrowing: mutably

```
fn main() {
  let s = String::from("hello");
  append_world(&mut s);
  println!("|{}|", s);
}
```

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

References and Borrowing: mutably

```
fn main() {
  let s = String::from("hello");
  append_world(&mut s);
  println!("|{}|", s);
}

// here, append_world gets a mutable reference
// that it changes

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

Scope of References

```
fn main() {
  let s = String::from("hello");
  let ref1 = &s;  // why is this not allowed?
  append_world(&mut s);
  println!("|{}|", ref1);
  println!("|{}|", s);
}
```

Scope of References

```
fn main() {
  let s = String::from("hello");
  let ref1 = &s;  // why is this not allowed?
  append_world(&mut s);
  println!("|{}|", ref1);
  println!("|{}|", s);
}

// references exist until their last use
// an immutable reference is not allowed while there‘s also a &mut
```

Dereferencing

- the * operator follows the reference to its original location

```
let a = 13; // a has type `i32`
let b = &a; // b has type `&i32`
let c = *b; // c has type `i32`
```

Dereferencing

- the * operator follows the reference to its original location

```
let a = 13; // a has type `i32`
let b = &a; // b has type `&i32`
let c = *b; // c has type `i32`
```

```
let mut value = 12;
let value_ref = &mut value;
*value_ref = 6;

println!("{value}");
// prints 6, because we changed the
// content of `value` with `value_ref`
```

Dangling References

- returning a reference to a value that will be dropped is an error

```
fn create_dangling_ref() -> &String {
    let s = String::from("hello");
    &s // `s` will be dropped, so any reference to `s` will be invalid
}
```

Summary: Rules of References

- you can have *either*
  one mutable reference *XOR*
  any number of immutable references
- references must always be valid
  ⇒ no references to dropped memory

The Slice type

- a **slice is a reference** to a contiguous sequence of elements in a collection
  ⇒ does not take ownership

The Slice type

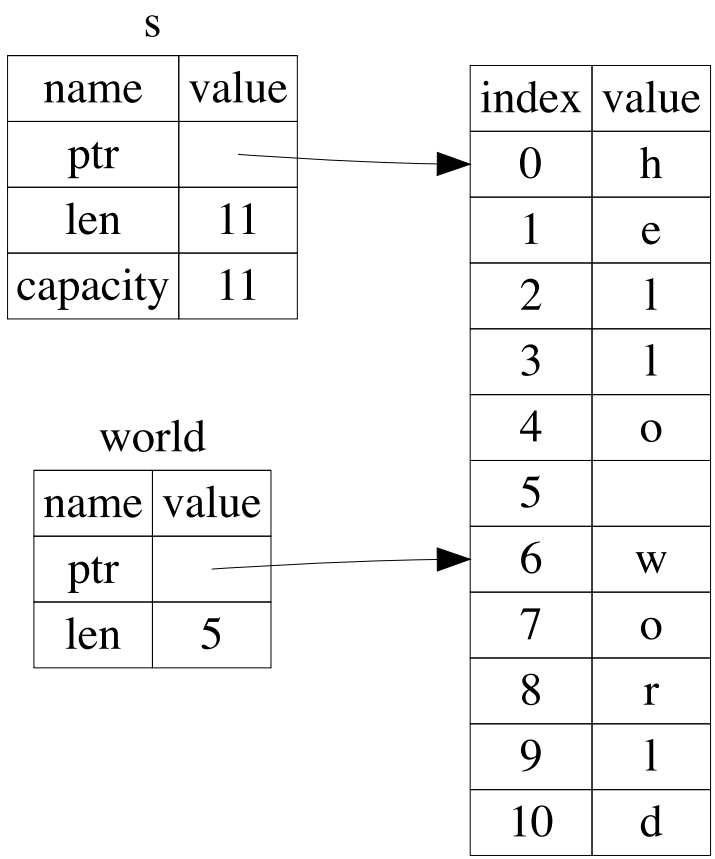
- a **slice is a reference** to a contiguous sequence of elements in a collection
  ⇒ does not take ownership
- `&str` is a *string slice*

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

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

The Slice type

- a **slice is a reference** to a contiguous sequence of elements in a collection
  ⇒ does not take ownership
- `&str` is a *string slice*

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

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

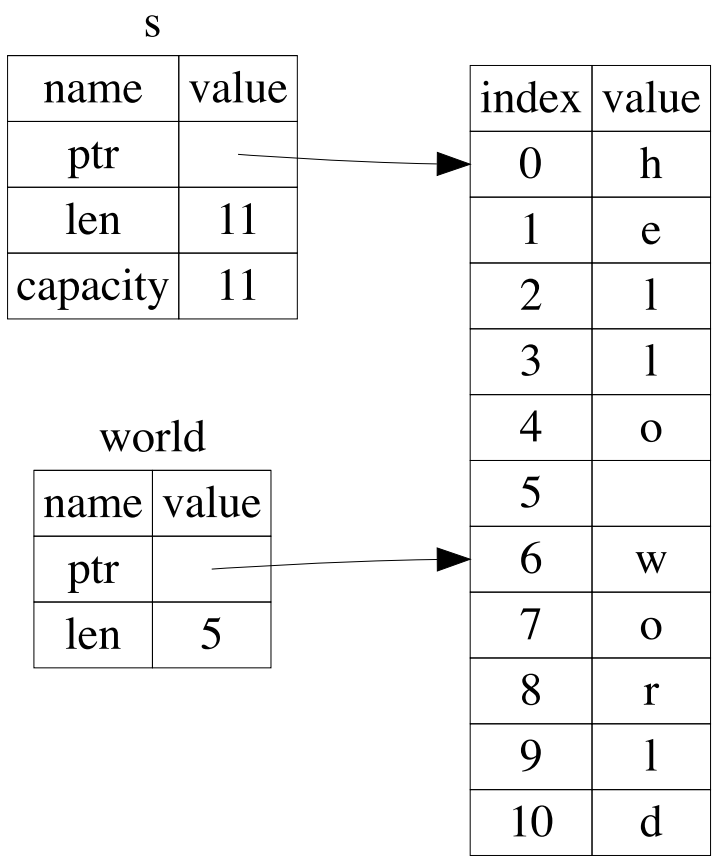


The Slice type

- a **slice is a reference** to a contiguous sequence of elements in a collection
  ⇒ does not take ownership
- `&str` is a *string slice*

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

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


let string_literals = "are slices too!";
```

- they point to a place in the binary (data segment), not in the heap



Range Syntax

- [a, b) ⟺ [inclusive..exclusive]
- [a, b] ⟺ [inclusive..=inclusive]

```
let s = String::from("range fun");
//                   012345678

let range = &s[..5];    // same as [0..5]
let fun = &s[6..];      // same as [6..(s.len())]
let range_fun = &s[..]; // same as [0..=(s.len()-1)]
```

String slices as parameters

- converting a `String` to a `&str` is easy
- converting a `&str` to a `String` requires heap allocation

```
fn compute_string(s: &String) -> &str {...}
```

- takes a reference to a `String`
- can only be called with a reference to a `String`

```
fn compute_str(s: &str) -> &str {...}
```

- takes a *string slice*
- can easily be called with any `String`, *string slice*, *string literal*, reference to a `String`
- generally preferred

Slicing arrays

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

let slice: &[i32] = &array[1..4];

assert_eq!(slice, &[2, 3, 4]);
```

Exercise

Write a program that reads a sentence from standard input. Print each word from that sentence in a new line. Hint: Try to separate the words by whitespace.

You can use this code to read a single line from standard input.

```
let mut sentence = String::new();
std::io::stdin().read_line(&mut sentence).expect("Failed to read sentence");
```