Rust Fundamentals

Basics of Rust
Part III

AZZAM S.A

RustCourse
Sep. 29th, 2023

Azzam S.A

OSS devotee, speaker, and teacher.

Open sourceror. Namely Rust, Python, and Emacs.

azzamsa azzamsyawqi azzamsa.com

Course. Not talk!

Follow along!

- Rust Playground
- Exercises

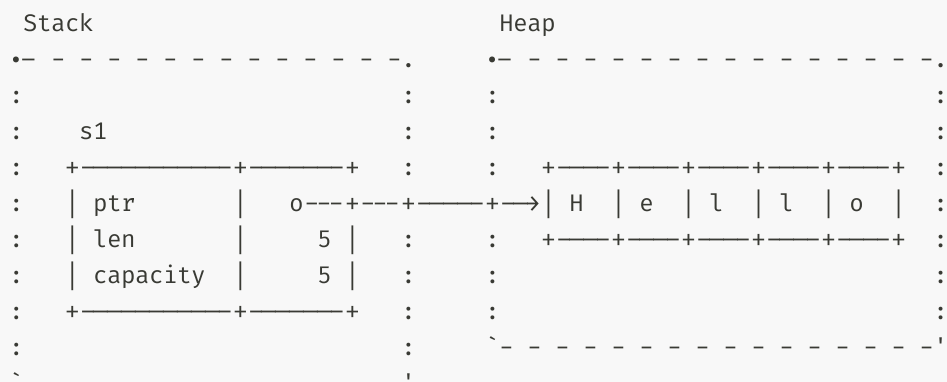
▶ Show hints

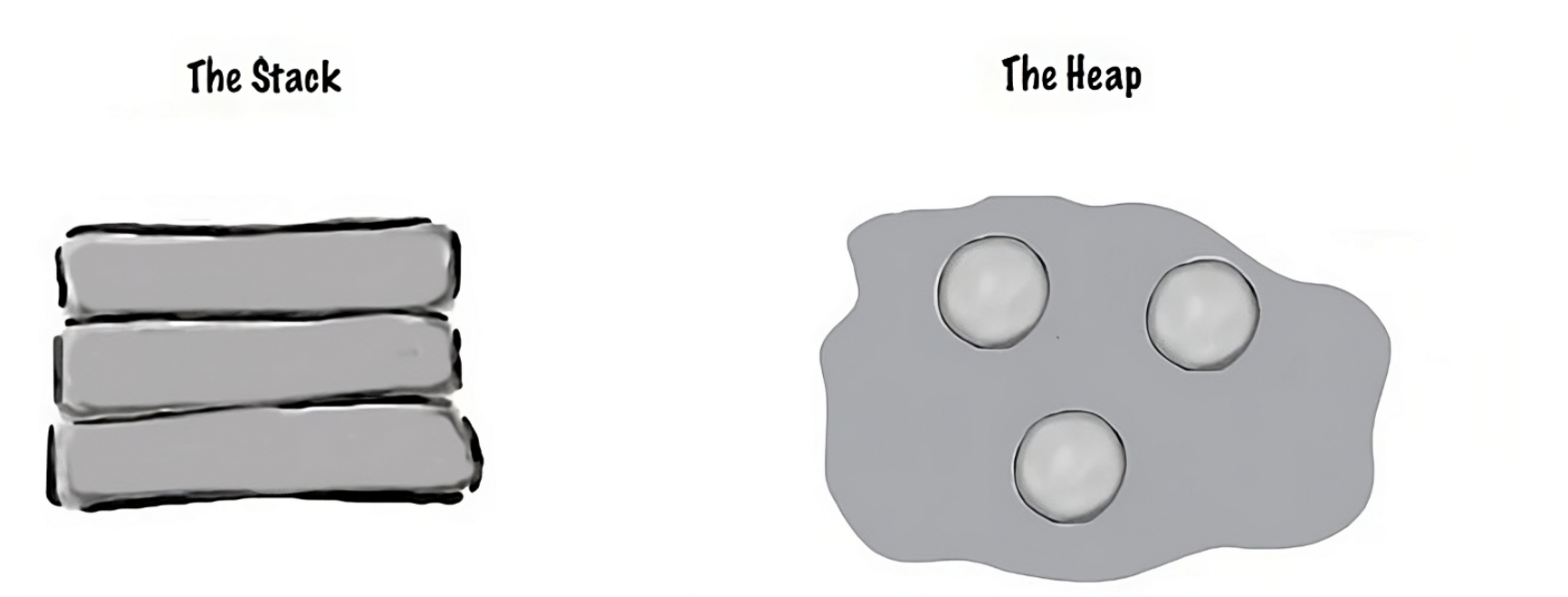
Memory Management

The Stack vs The Heap

- **Stack**: Continuous area of memory for local variables.
  - Values have **fixed sizes** known at compile time.
  - **Extremely fast**: just move a stack pointer.
  - Easy to manage: follows function calls.
- Heap: Storage of values outside of function calls.
  - Values have **dynamic sizes** determined at runtime.
  - **Slower** than the stack: some book-keeping needed.

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





Head First Java by Kathy Sierra

Ownership

```rust
fn main() {
    {
        let x = 42;
        println!("x: {x}");
    } // variable `x` is dropped, data is freed.

    println!("x: {x}");
}
```

Move Semantics

```
fn main() {
    let s1: String = String::from("Hello!");
    let s2: String = s1;

    println!("s2: {s2}");
    //println!("s1: {s1}");
}
```

- The assignment of `s1` to `s2` transfers ownership.
- There is always *exactly* one variable binding which owns a value.

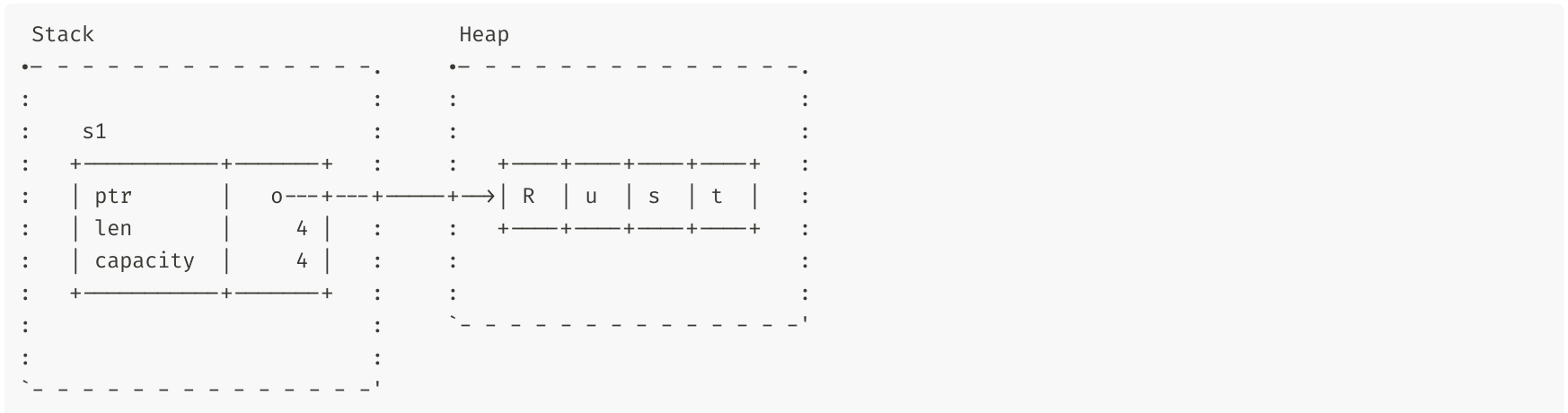
```
error[E0382]: borrow of moved value: `s1`
 ⟶ src/main.rs:6:19
  |
2 |     let s1: String = String::from("Hello!");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2: String = s1;
  |                      -- value moved here
...
6 |     println!("s1: {s1}");
  |                   ^^^^ value borrowed here after move
  |
...
  |
3 |     let s2: String = s1.clone();
  |                      ++++++++
```

Moved Strings in Rust

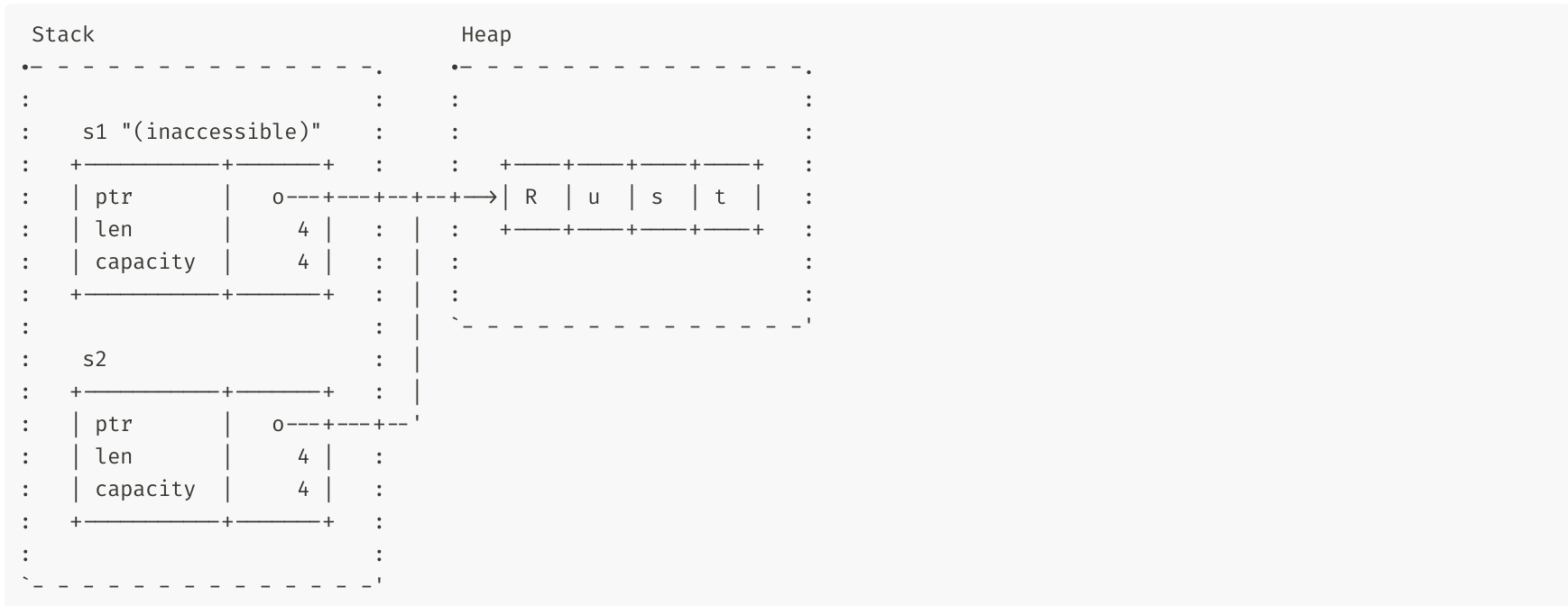
```
fn main() {
    let s1: String = String::from("Rust");
    let s2: String = s1;
}
```

- The heap data from `s1` is reused for `s2`.
- When `s1` goes out of scope, nothing happens (it has been moved from).

Before move to `s2`:



After move to `s2`:



Moves in Function Calls

```
fn say_hello(name: String) {
    println!("Hello {name}")
} // The heap memory allocated for `name` will be freed here


fn main() {
    let name = String::from("Alice");
    say_hello(name); // Ownership transferred to say_hello
    // say_hello(name); // You can't use `name` here anymore because it was moved
}
```

```rust
fn say_hello(name: &String) {
    println!("Hello, {}", name);
}

fn main() {
    let name = String::from("Alice");
    say_hello(&name); // Borrowing `name`, ownership remains with `main`
    // You can still use `name` here
}
```

```rust
fn say_hello(name: String) {
    println!("Hello {}", name);
}

fn main() {
    let name = String::from("Alice");
    let cloned_name = name.clone();
    say_hello(cloned_name);
    // You can still use `name` here because you cloned it.
```

Copying and Cloning

Move semantics are the default, but certain types are copied by default.

```
fn main() {
    let x = 42;
    let y = x;

    println!("x: {x}");
    println!("y: {y}");
}
```

These types implement the `Copy` trait.

- Primitive numeric types
- Tuples (if all their elements implement Copy)
- Fixed-size arrays (if their elements implement Copy)
- Some built-in types
- User-Defined types

```
#[derive(Debug)]
struct Point(i32, i32);

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;

    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

```
error[E0382]: borrow of moved value: `p1`
 ⟶ src/main.rs:8:19
  |
5 |     let p1 = Point(3, 4);
  |         -- move occurs because `p1` has type `Point`, which does not implement the `Copy` trait
6 |     let p2 = p1;
  |              -- value moved here
7 |
8 |     println!("p1: {p1:?}");
  |                    ^^^^^^ value borrowed here after move
  |
```

```
#[derive(Copy, Clone, Debug)]
struct Point(i32, i32);

fn main() {
    let p1 = Point(3, 4);
    let p2 = p1;

    println!("p1: {p1:?}");
    println!("p2: {p2:?}");
}
```

Shared and Unique Borrows

Rust puts constraints on the ways you can borrow values:

- You can have one or more `&T` values at any given time, *or*
- You can have exactly one `&mut T` value.

Lifetimes

A borrowed value has a *lifetime*:

- The lifetime can be implicit: `` `add(p1: &Point, p2: &Point) → Point` ``.
- Lifetimes can also be explicit: `` `&'a Point` ``, `` `&'document str` ``.

Lifetimes in Function Calls

```
#[derive(Debug)]
struct Point(i32, i32);

fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
    if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
    let p1: Point = Point(10, 10);
    let p2: Point = Point(20, 20);
    let p_ref: &Point = left_most(&p1, &p2);
    println!("left-most point: {:?}", p_ref);
}
```

```rust
#[derive(Debug)]
struct Point(i32, i32);

fn left_most<'a>(p1: &'a Point, p2: &'a Point) -> &'a Point {
    if p1.0 < p2.0 { p1 } else { p2 }
}

fn main() {
    let p1: Point = Point(10, 10);
    let p2: Point = Point(20, 20);
    let p_ref: &Point = left_most(&p1, &p2);
    println!("left-most point: {:?}", p_ref);
}
```

Lifetimes in Data Structures

```
#[derive(Debug)]
struct Highlight<'doc>(&'doc str);

fn main() {
    let text = String::from("The quick brown fox jumps over the lazy dog.");
    let fox = Highlight(&text[4..19]);
    let dog = Highlight(&text[35..43]);
    println!("{fox:?}");
    println!("{dog:?}");
}
```

```rust
#[derive(Debug)]
struct Highlight<'doc>(&'doc str);

fn main() {
    let text = String::from("The quick brown fox jumps over the lazy dog.");
    let fox = Highlight(&text[4..19]);
    let dog = Highlight(&text[35..43]);
    println!("{fox:?}");
    println!("{dog:?}");
}
```

Exercises

- `move_semantics1`
- `move_semantics2`
- `move_semantics3`
- `lifetimes1`
- `lifetimes2`
- `lifetimes3`

Credits 🌟

- Mo's (mo8it) Comprehensive Rust 🦀
- rustlings 🦀