ECE 459
Programming for Performance

Lecture 3
Rust: Borrowing, Slices, Threads, Traits

Winter 2023

Huanyi Chen
huanyi.chen@uwaterloo.ca

UNIVERSITY OF WATERLOO

Rules

- Recall the concept: Ownership

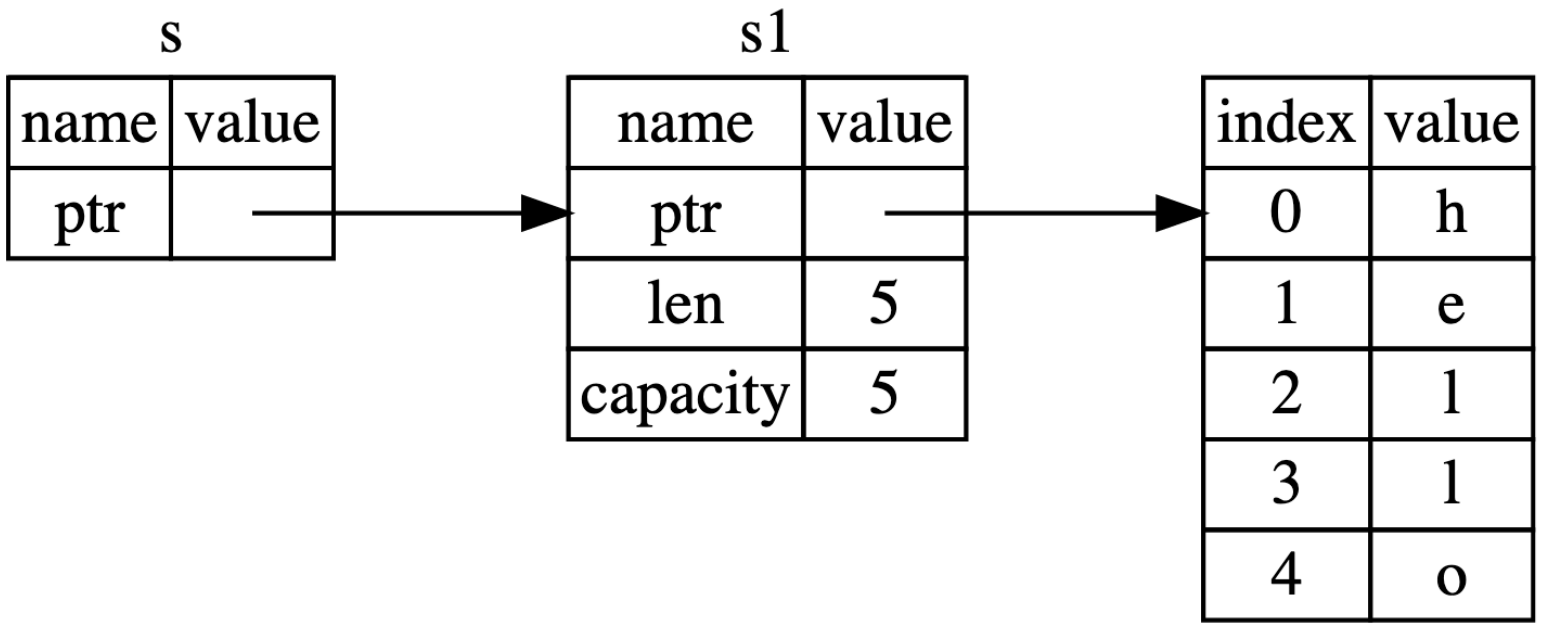
1. Every value has a variable that is its owner
2. Only one owner at a time
3. When the owner goes out of scope, the value is dropped

Borrowing and References

- Temporarily use the data
- Compiler will complain if you do not give back

Borrowing and References

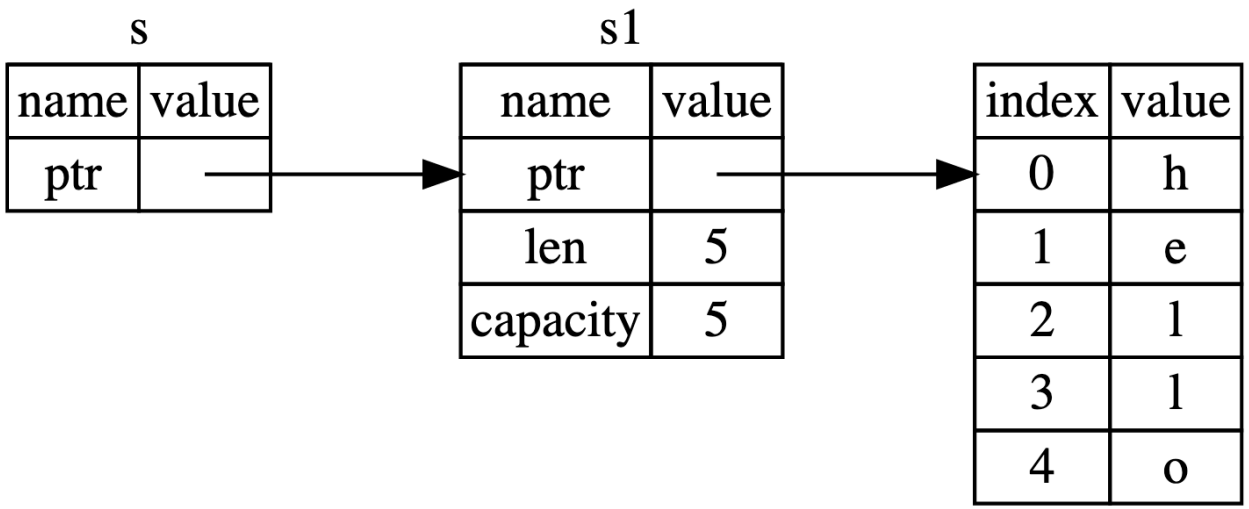
- To claim a reference
- use &
- `let s = &s1;`



Borrowing and References

- You can think `s` is a pointer to `s1`
- When you use `s`, the compiler will be smart enough to know that you want to access `s1`
- No `*s` needed

- *Deref Coercion:* if you want to read more about it



Mutable `refs`

- `ref` are immutable by default
- But you can explicitly create mutable ones
- `let s = &mut s1;`

- But `s1` needs to be mutable (`let mut s1 = ...`)

Multiple `muts`?

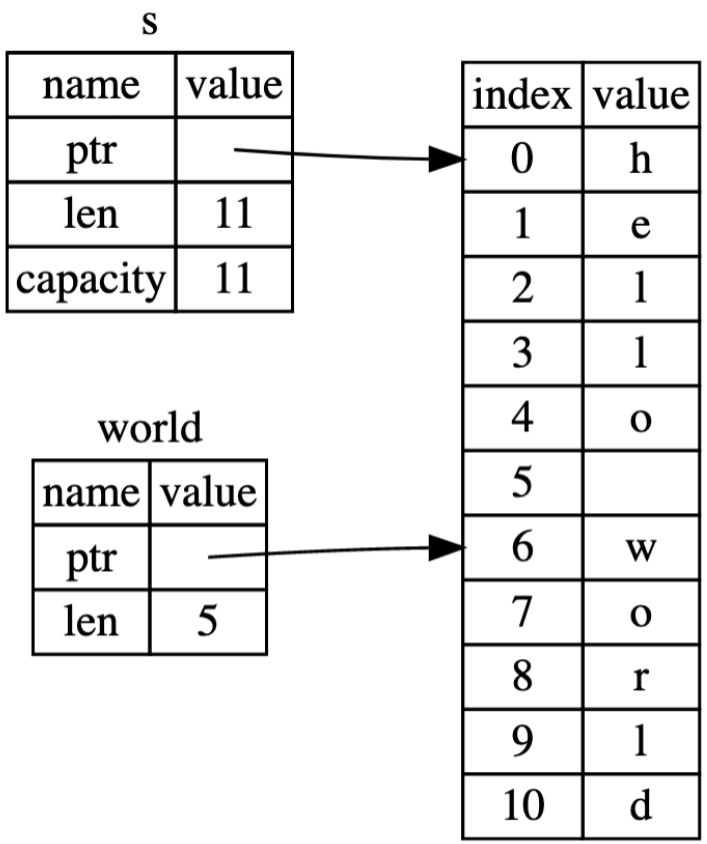
- To prevent race conditions

1. While a mutable reference exists, the owner can't change the data
2. There can be only one mutable reference at a time, and while there is, there can be no immutable references.

- Make them simple

1. Only one write
2. No read when a write exists

Slices




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

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

- More than Strings

- Vectors
- Collections

Unwrap the Panic

- Both `Ok` and `Err` are variants of ***enum*** `Result`
- Since `Result` implements the function
  - `unwrap`
  - `expect`
- Thus, `Ok` and `Err` have the functions, too
- But they show different behaviors
  - e.g., `unwrap` on `Ok` returns the value, on `Err` will call `unwrap_failed`, which calls `panic!`

e.g., for `expect()`

```
pub fn expect(self, msg: &str) -> T
where
    E: fmt::Debug,
{
    match self { // pattern matching
        Ok(t) => t,
        Err(e) => unwrap_failed(msg, &e),
    }
}
```

Fearless Concurrency

- Races? Tend to be solved in compile time!
- Less runtime debugging
- Introducing new code does not introduce new bugs (likely)

Closure

- *Closure* --- an anonymous function that can capture some bits of its environment

```
fn  add_one_v1   (x: u32) -> u32 { x + 1 }  // original function
```

- Anonymous function

```
let add_one_v2 = |x: u32| -> u32 { x + 1 }; // closure
let add_one_v3 = |x|             { x + 1 }; // closure
let add_one_v4 = |x|               x + 1  ; // closure
```

- With capturing

```
let add_one_v5 = ||              { x + 1 }; // x is in the environment
add_one_v5();
```

Threads

- Data communication
  - Capturing
  - Message passing
  - Shared state

Traits: Defining Shared Behavior

- Similar to interfaces

```
pub trait FinalGrade {
    fun final_grade(&self) -> f32;
}

impl FinalGrade for Enrolled_Student {
    fn final_grade(&self) -> f32 {
        // Calculation of average according to syllabus rules goes here
    }
}
```

Traits

- You can only define traits on your own types.
- You can create a default implementation.
- Traits can be used as a return type or method parameter.
- Use the + to combine multiple traits in a parameter.

```
e.g., pub fn notify<T: Summary + Display>(item: &T) {
```

Traits

- Send: allows transferring ownership between threads
- Sync: means a particular type is thread-safe

In-class exercises

- See lectures/flipped/L03.md
- You can create a repo called “ece459-practice” and push your code there
- You can add me as a member if you want