Borrowing, Slices, & Program Memory

Lecture 8

Slides: Neil Kaushikkar

Goals For Today

- Answering Your Questions
- Quick Review Ownership & Borrowing
- Program Memory
- Slices of **String**s and **Vec**tors

Reminders

- HW5 releasing tonight due 2/21 at 11:59 pm CT
- HW4 due 2/16 at 11:59 pm CT
- MP0 due 2/15 at 11:59 pm CT
- MP1 released yesterday, due 2/27 at 11:59 pm CT
- We'll be releasing an anonymous feedback survey in the next few days
    - Please let us know what you are thinking, what we can improve, what we should keep doing, etc...
    - If we get 40 responses by Sunday 2/19, we will give everyone 2% extra credit
    - if we have 60 response, we will give everyone 4% extra credit

Answering Your Questions!

- “Unclear specifications in assignments”
- “I wish the explanations were a little more robust; there were times when I was mislead by what I felt was a poorly worded phrase or sentence.”
  - We are constantly tweaking assignment feedback to word things better and make certain requirements clearer
  - If you have a question about some wording, drop a message in Discord, DM a member of course staff, and we’ll clear things up
  - We can only make things clearer when students tell us they are unclear, so please keep asking clarifying questions!

Answering Your Questions!

- “I know the course is fast-paced so I understand the difficulty of the homework but I think it would also be great to give us some simpler problems to get us used to the semantics of Rust”
- “lot to unpack past couple of lectures, maybe more practice?”
  - This one’s on us
  - Going forward, we will try to get out some extra practice problems to reinforce concepts from lecture
  - In the meantime, we set up an in-person office hours to give students a chance for in-person help, interaction with course staff
  - We also have office hours every day of the week! Please come to these!
  - If office hour times or the discussion section times, don’t line up with your schedule, reach out to us and we can move certain times around

Answering Your Questions!

- “Is there an easier way to convert &str to String instead of just .to_string(), or is that something we need to always do when returning Strings?”
  - **.to_string()**
  - **.to_owned()**
  - **String::from(“hello”)**
  - There are more ways but pretty much different flavors of the above…

Answering Your Questions!

- “I was wondering whether solutions are going to be released to see if there are more efficient solutions than the ones I had”
  - We can record some solution walkthroughs for HWs that are past due
  - We want to wait for the 70% credit deadline to expire to be fair to all other students
  - If you come to our discussion section, we would be more than happy to walk you through an efficient solution!

Answering Your Questions!

- "This language makes me furious"
  - Rust has a STEEP learning curve
  - Our goal with lectures, HWs, and MPs is to get you a good understanding so that when you begin working on your final projects, you have a much better understanding
    - (and hopefully it no longer makes you furious)

Ownership Review

- 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

```rust
fn main() {
    let s = String::from("hello");
    // ...
    {
        let w = String::from("world");
        // do something with w...
    }   // w is dropped here
    // ...
}   // s is dropped here
```

```rust
fn main() {
    let x = String::from("hello");

    let y = x; // y now OWNS the String "hello"

    // println!("{}", x); // THIS LINE WON'T COMPILE
    println!("{}", y);
}
```

Reference:

- https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

References Review

- An ampersand (&) represents a *reference*
- Allows you to refer to some value without taking *ownership* of it
- We call the action of creating a reference *borrowing*

Reference:

- https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Borrowing Review

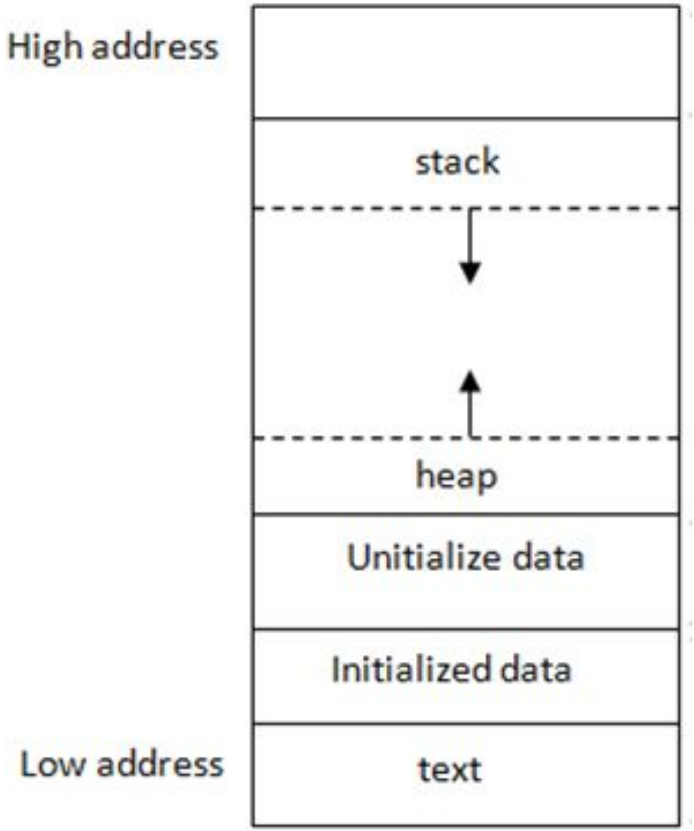
- At any given time, you can have either:
  - one mutable reference using **&mut** or...
  - An infinite number of immutable references using **&**

```rust
fn main() {
    let mut x: String = String::from("hello");

    // creates a MUTABLE reference to x
    let y = &mut x;

    // ERROR: trying to create a SECOND MUTABLE reference to x
    x.push_str(" world!");

    println!("x = {} and y = {}", x, y);
}
```

Reference:

- https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Anatomy of a Program's Memory



Reference:

- https://courses.engr.illinois.edu/cs225/sp2020/resources/stack-heap/

Strings and Substrings

- The **String** type has ownership over its characters
- If we wanted to get a substring, we would like:
  - Some type of reference to a portion of the original **String** (to avoid duplicating out **String** data)
  - The original string to keep ownership of its **char**s

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

s1

name	value
ptr	
len	5
capacity	5

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

Reference:

- https://doc.rust-lang.org/book/ch04-03-slices.html

Enter String Slices

- The **String** type has ownership over its characters
- If we wanted to get a substring, we can take a slice:
  - A *string slice* (**&str**) is a reference to a portion of a **String**
  - This reference can be of substring or the ENTIRE string – it's a reference!
  - The original string still has ownership of the **char**s

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

let hello = &s[0..5];  // same as &s[..5]
let world = &s[6..11]; // same as &s[6..]
let hello_world = &s[..];
```

Reference:

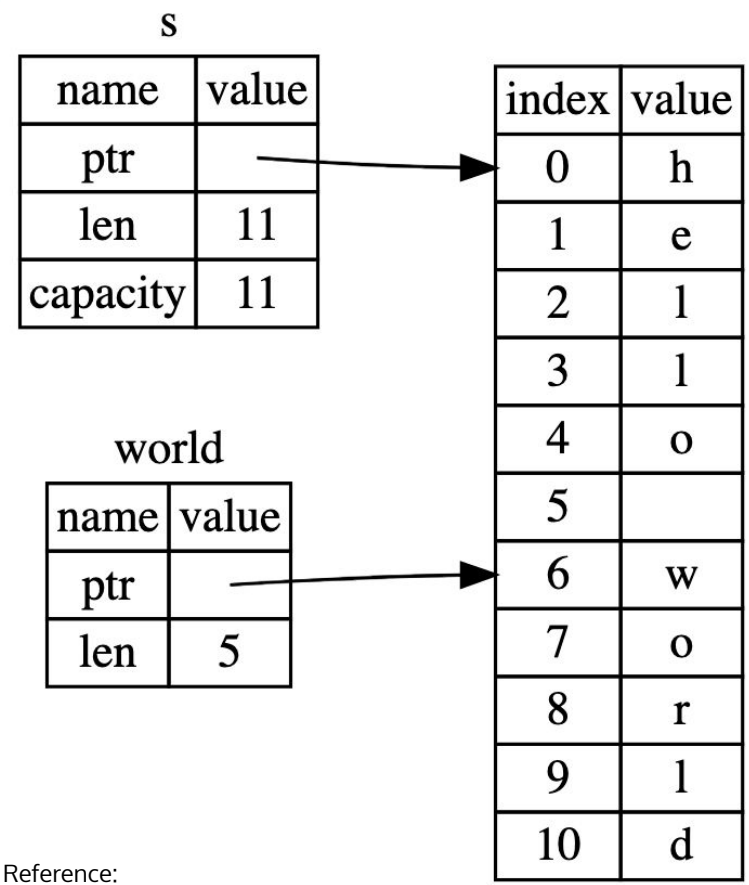
- https://doc.rust-lang.org/book/ch04-03-slices.html

Creating String Slices

- Use & to create a reference and specify a range
  - [*start*..*stop*] - index *start* (inclusive) to *stop* (exclusive)
  - [..*stop*] - index 0 to *stop* (exclusive)
  - [*start*..] - index *start* (inclusive) to the end of the **String**
  - [..] - index 0 to the end of the **String** (equivalent to a normal borrow)
- Slices are **READ-ONLY** (aka immutable)
  - Why do you think that is?

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

let hello = &s[0..5];  // same as &s[..5]
let world = &s[6..11]; // same as &s[6..]
let hello_world = &s[..];
```

String Slices Under the Hood




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

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

Reference:

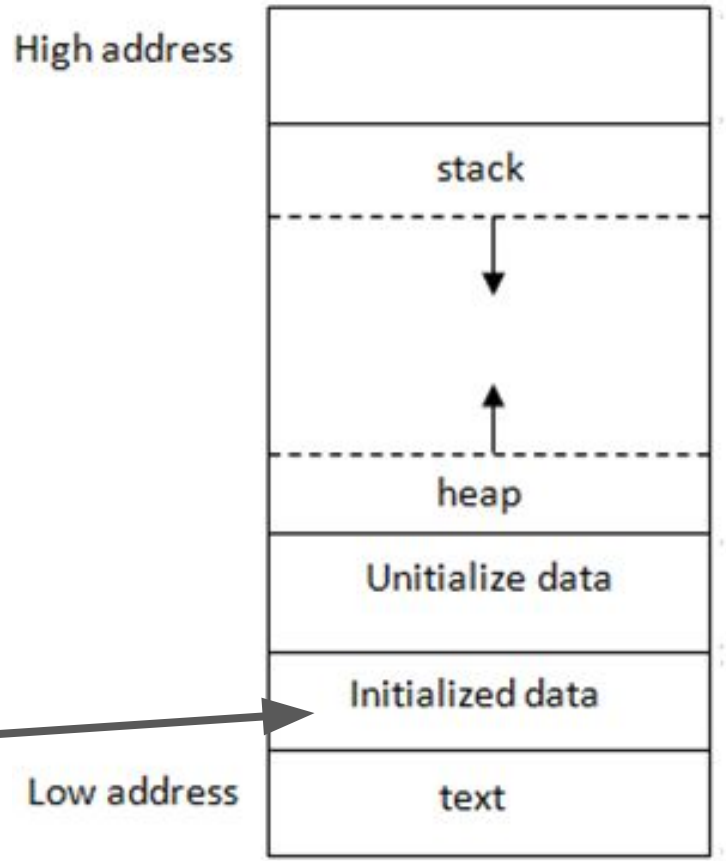
- https://doc.rust-lang.org/book/ch04-03-slices.html

String Literals in Memory

```
let world: &str = "world";
```

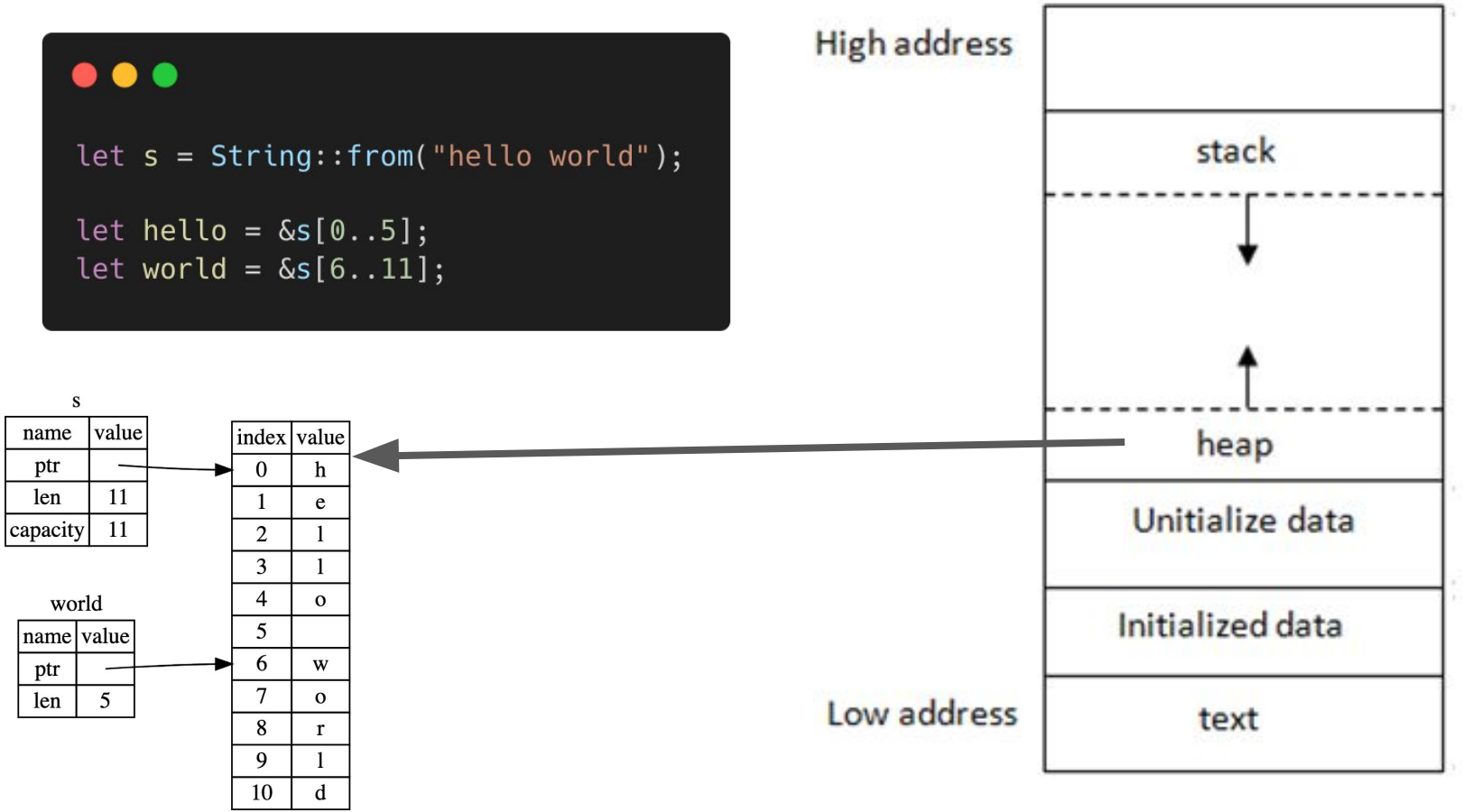
world

name	value
ptr	
len	5

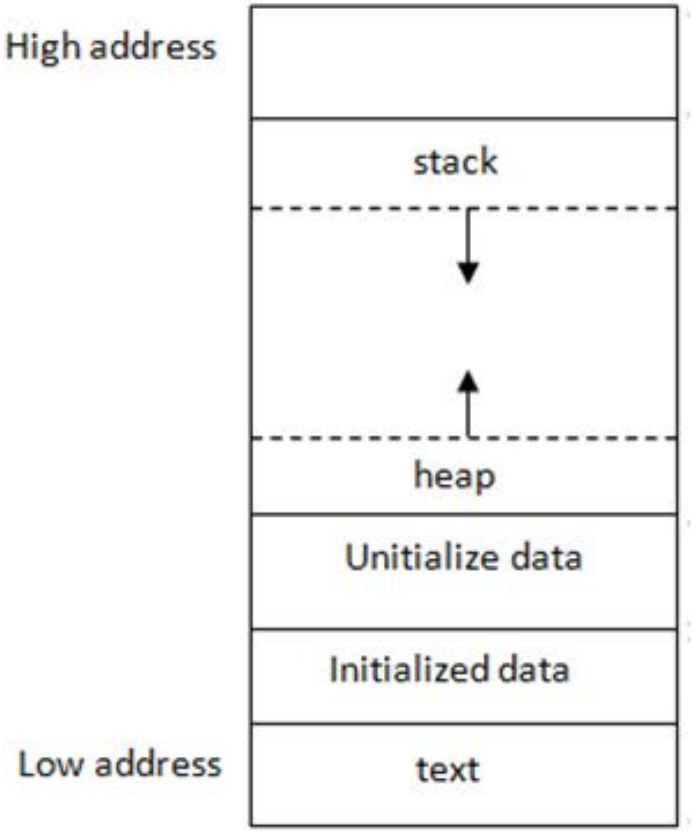


String Slices in Memory

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

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



Let's Find a String in Program Memory!



Reference:

- https://courses.engr.illinois.edu/cs225/sp2020/resources/stack-heap/

Slices Example

Vector Slices

- Constructed the same way as a **String** slice
  - Borrow the original vector
  - Specify a range with the [*start*..*stop*] notation
- Again, slices are **READ-ONLY** (aka immutable)
- Vector slices have type **&[T]**
  - The vector has elements of type **T** (any type)
  - A borrow to an array (vectors just have arrays under the hood!)

Vector Slices