Lecture 2

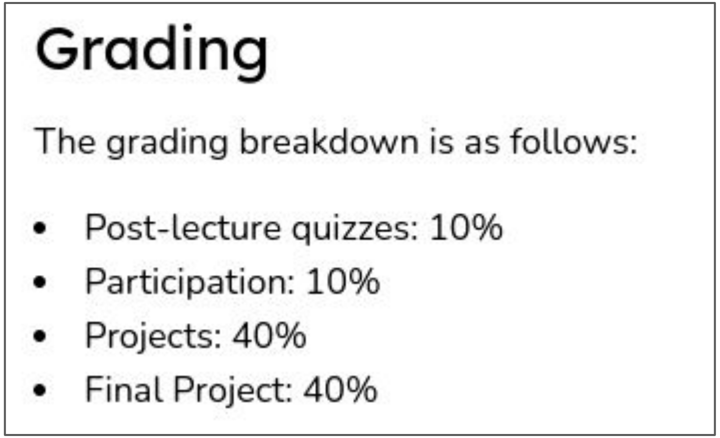
Ownership

“Participation”

“Participation” ≠ “Mandatory Attendance”

Please do come to class if you can! We try to make it valuable.

If not, you can also participate on EdStem by asking questions as you go through the readings/slides



Grading

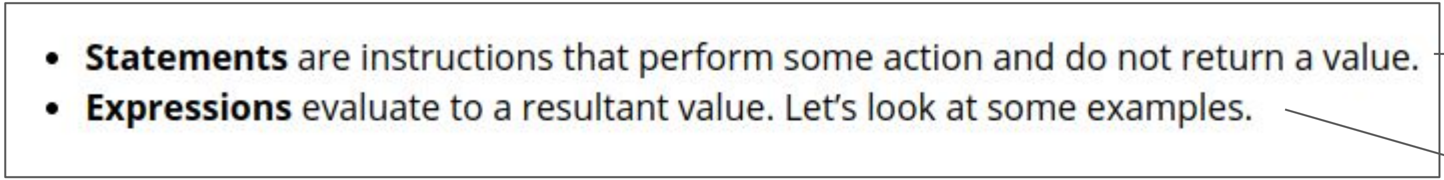
The grading breakdown is as follows:

- Post-lecture quizzes: 10%
- Participation: 10%
- Projects: 40%
- Final Project: 40%

https://www.cis.upenn.edu/~cis1905/2024fall/syllabus/

Clarifications After Lecture 1

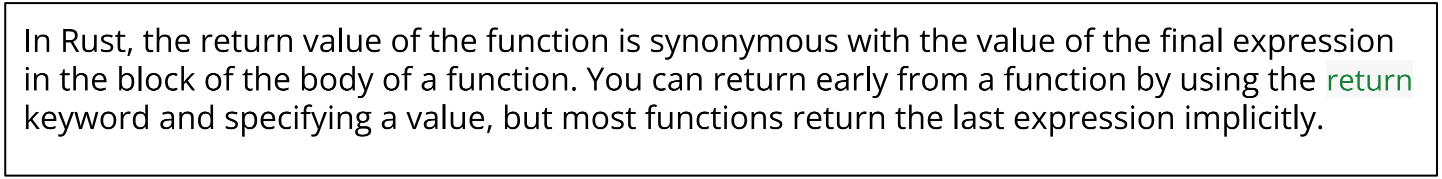
- Statements/expressions/semicolons/`return`
- *Declarative* vs. *imperative*
  - read as *functional* vs. *object-oriented*
- Stack/heap/memory/pointers
  - Covered today!



- **Statements** are instructions that perform some action and do not return a value.
- **Expressions** evaluate to a resultant value. Let's look at some examples.

`let`

everything else



In Rust, the return value of the function is synonymous with the value of the final expression in the block of the body of a function. You can return early from a function by using the `return` keyword and specifying a value, but most functions return the last expression implicitly.

https://doc.rust-lang.org/book/ch03-03-how-functions-work.html

Today: Ownership!

But first: the stack and heap

Where can data be allocated?

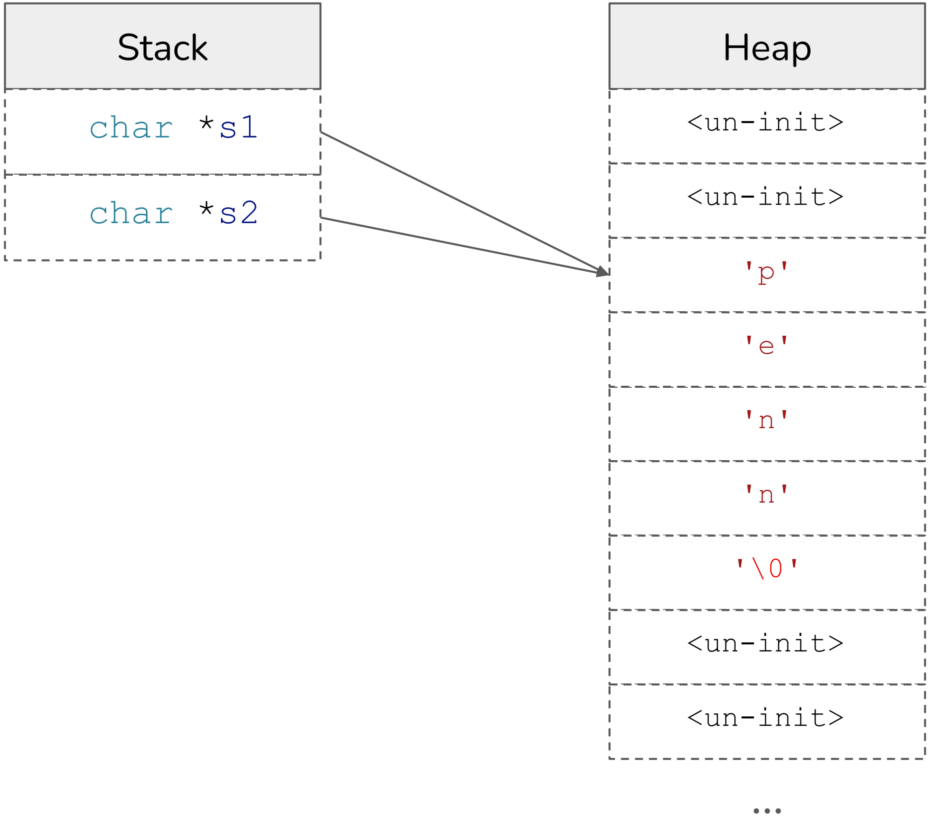
	Static memory	Stack	Heap

Example?	`static float PI = 3.14;`
	OR
	`char *s = "cis1905";`
		`void foo() {`
		`  Point p = {.x=1,.y=2};`
		`}`
			`char *init_username() {`
			` char *s;`
			` malloc(&s, username_length);`
			` ...`
			`}`
			`void drop_username(char *s) {`
			` free(name);`
			`}`

How long does it live?	Entire program lifetime	Until end of function	Until explicitly deallocated with `free`
Pros/Cons?	Zero cost Fixed-size	Low performance cost Can't outlive function	Supports allocations of unknown size Error-prone

Heap Programming Challenges

```
int main() {
   char *s1;
   malloc(&s1, 5);
   *s1 = {'p','e','n','n','\0'};

   char *s2 = s1;
   free(s1);
   printf("%s\n", s2);
```



Stack

char *s1

char *s2

Heap

<un-init>

<un-init>

'p'

'e'

'n'

'n'

'\0'

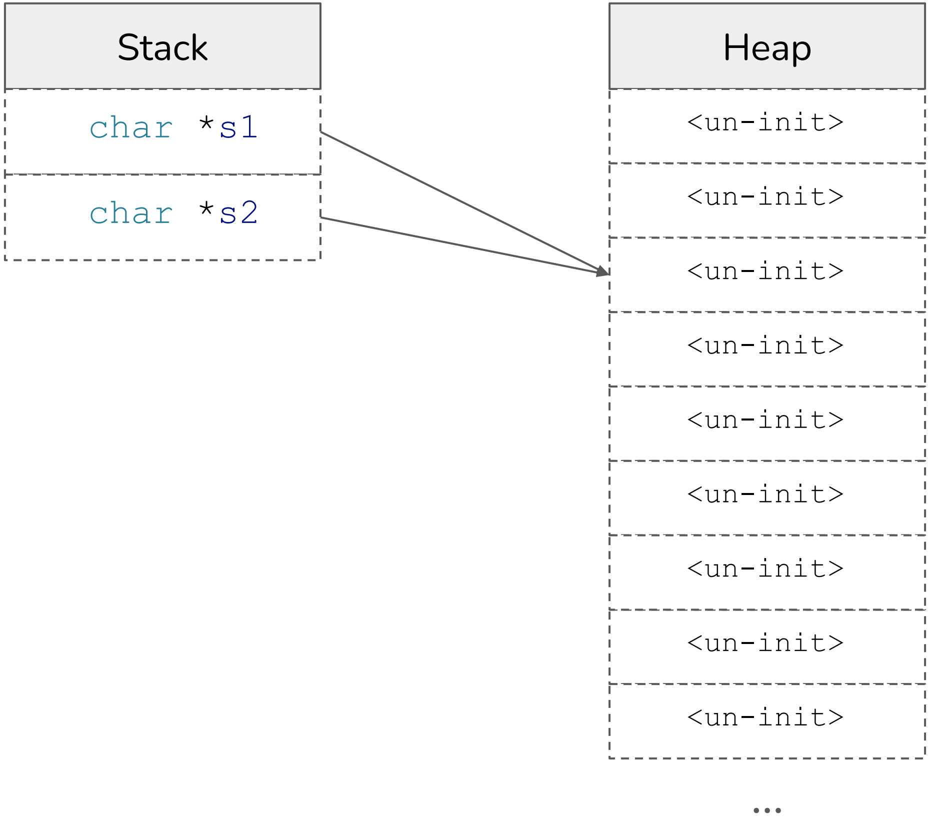
<un-init>

<un-init>

...

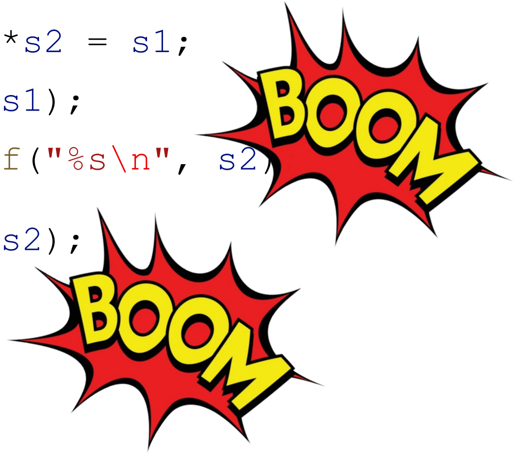
Heap Programming Challenges

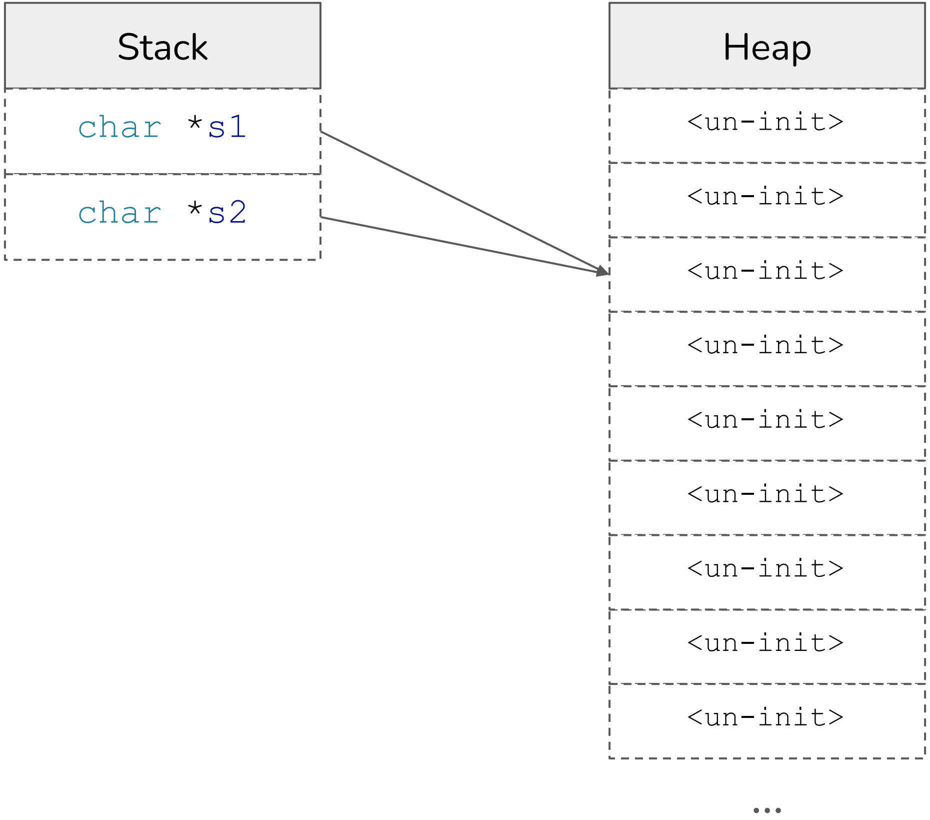
```
int main() {
   char *s1;
   malloc(&s1, 5);
   *s1 = {'p','e','n','n','\0'};

   char *s2 = s1;
   free(s1);
   printf("%s\n", s2)
```



Heap Programming Challenges

```
int main() {
   char *s1;
   malloc(&s1, 5);
   *s1 = {'p','e','n','n','\0'};

   char *s2 = s1;
   free(s1);
   printf("%s\n", s2)

   free(s2);
```





What went wrong here?

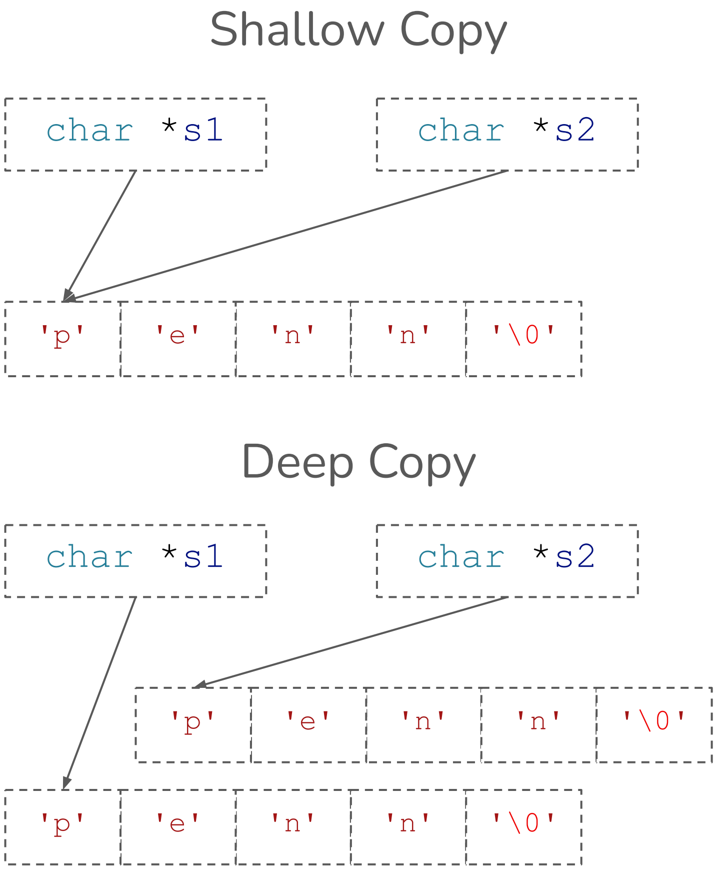
1. Shallow copies vs. deep copies
2. Who is in charge of freeing data?

Recall from lecture 1:

How can we prevent memory safety issues…

- buffer overflow
- use-after-free
- double free

…while still giving the programmer control of heap allocations?



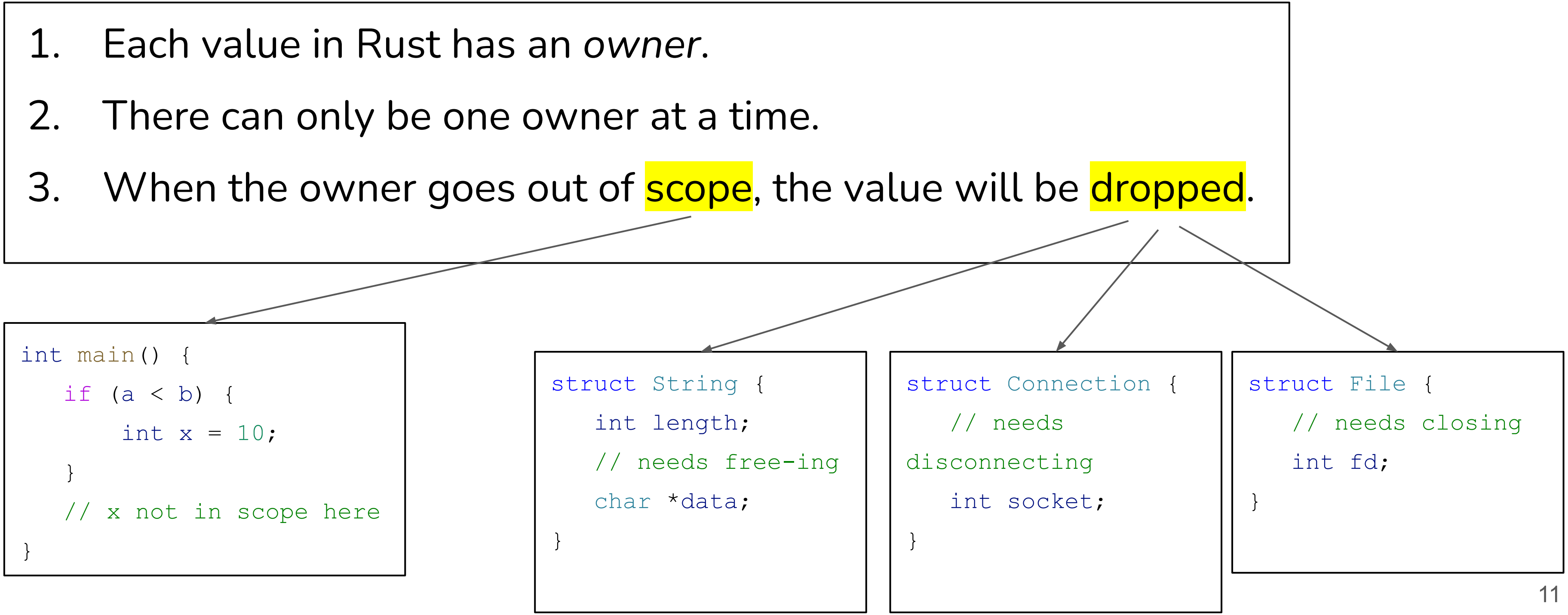
Ownership!

Three golden rules:

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

Ownership!

Three golden rules:



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

```
int main() {
   if (a < b) {
       int x = 10;
   }
   // x not in scope here
}
```

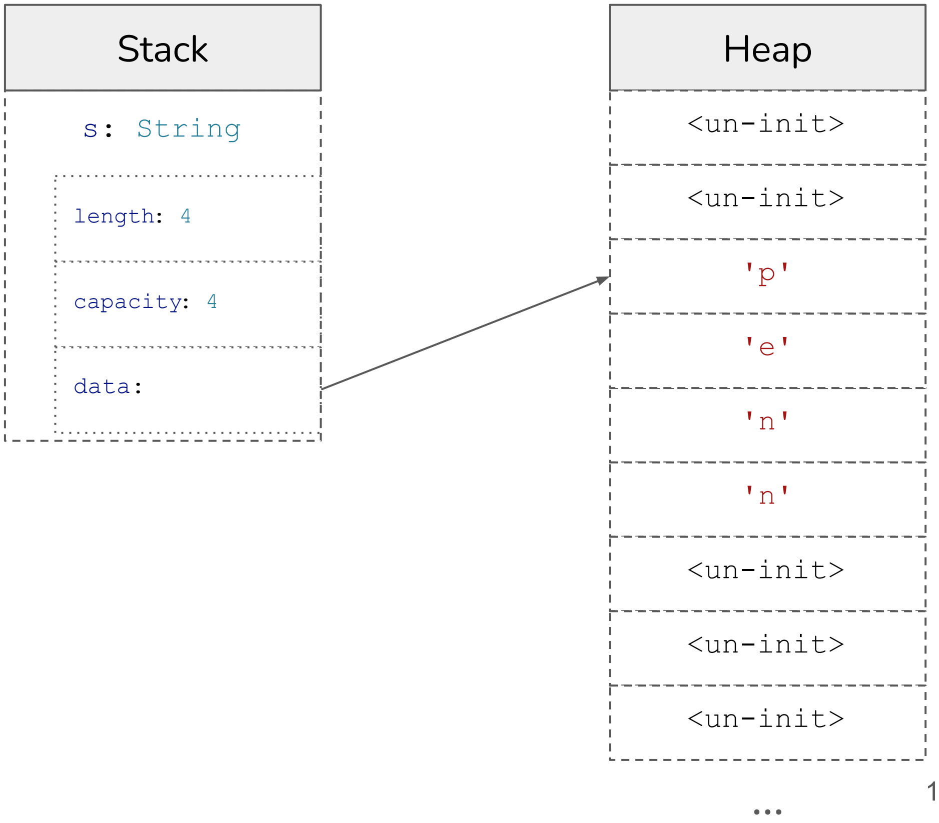
```
struct String {
   int length;
   // needs free-ing
   char *data;
}
```

```
struct Connection {
   // needs
disconnecting
   int socket;
}
```

```
struct File {
   // needs closing
   int fd;
}
```

Examining Ownership with Strings

Numeric types are too simple. Next week we'll talk about defining custom data types, but for now we'll use `std::String`, Rust's built-in String type



Stack

s: String

length: 4

capacity: 4

data:

Heap

<un-init>

<un-init>

'p'

'e'

'n'

'n'

<un-init>

<un-init>

<un-init>

...

```
fn main() {
    let s = String::from("penn");
}
```

Examining Ownership with Strings

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

generic function to trigger destructor

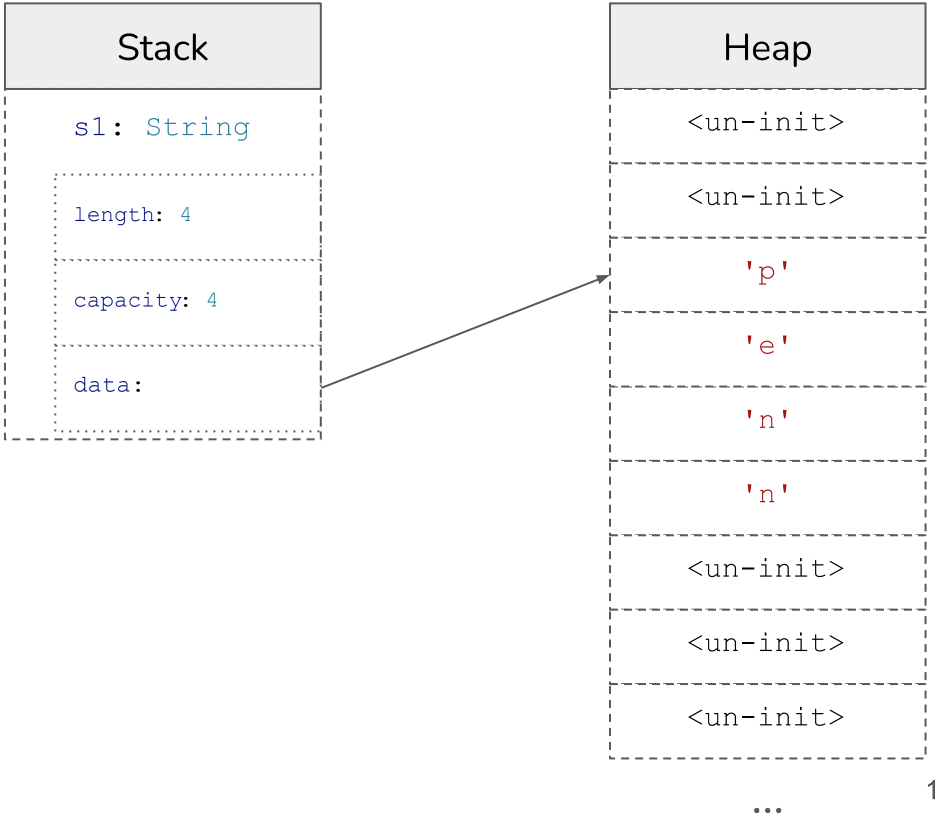
1. Each value in Rust has an *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

No double free !

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
```

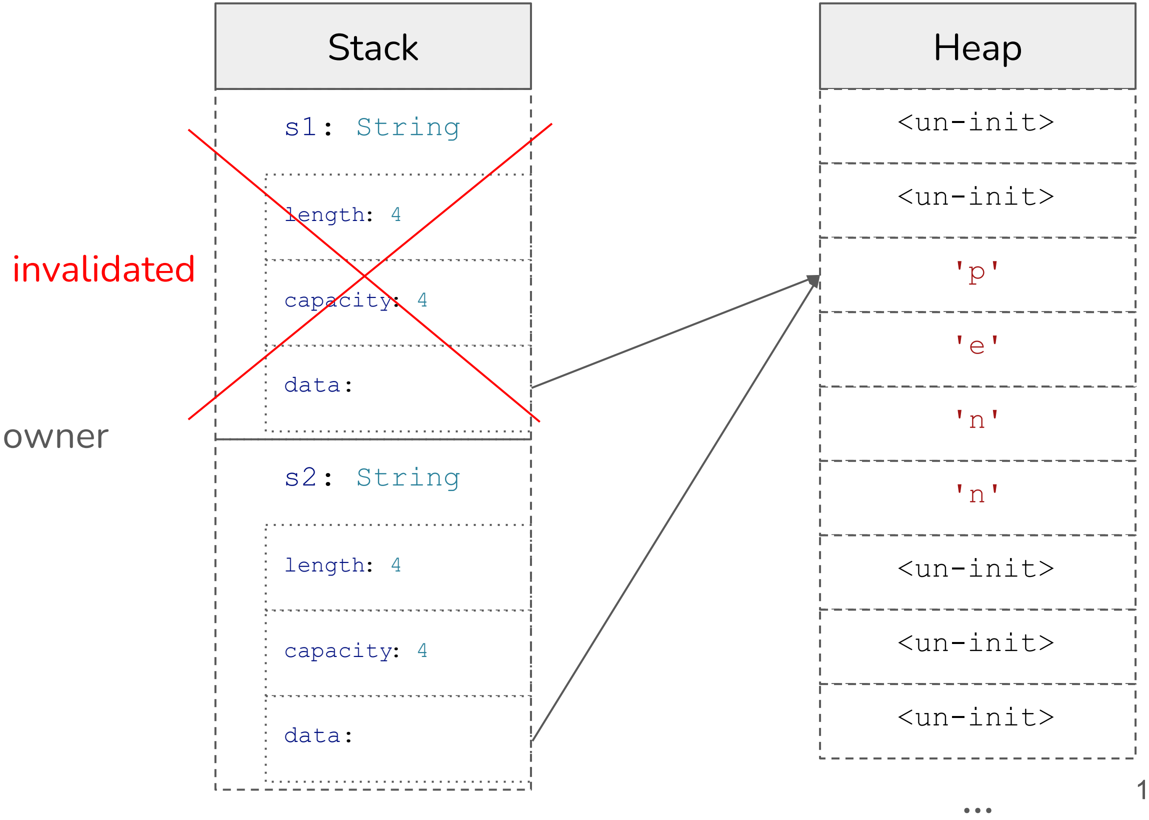
What’s in a move?

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



What’s in a move?

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



Move ≈ shallow copy + invalidate old owner

Moves are fast! *(O(1))*

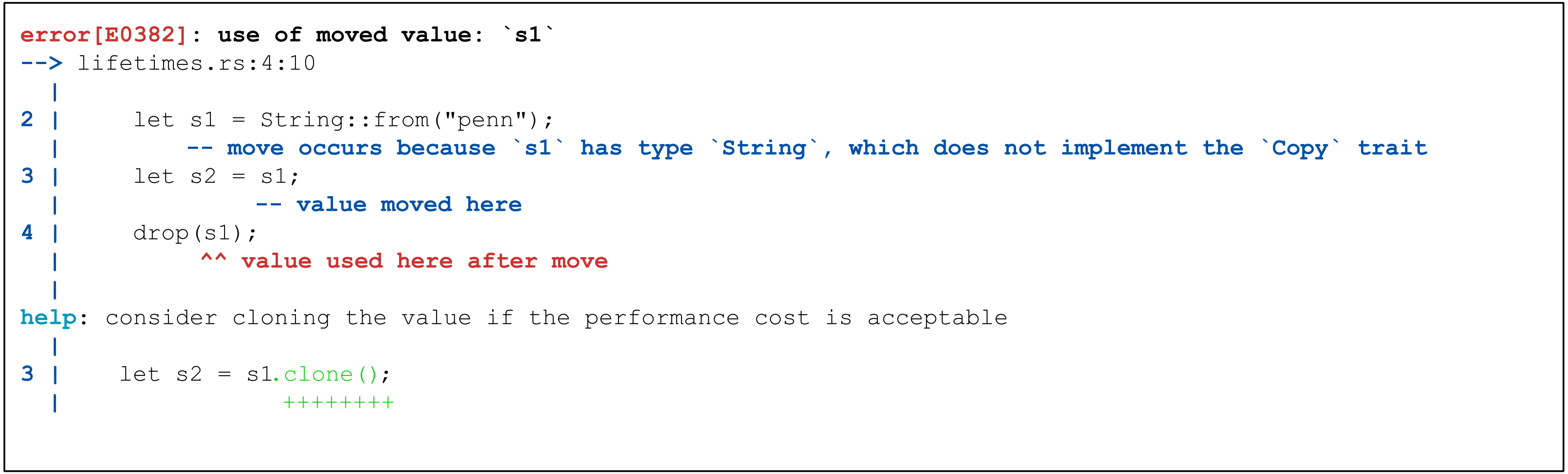
What’s in a move?

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

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
```

What’s in a move?

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



```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
  |
help: consider cloning the value if the performance cost is acceptable
  |
3 |    let s2 = s1.clone();
  |               ++++++++
```

What’s in a move?

```
fn main() {
    let s1 = String::from("penn");
    let s2 = s1.clone();
    drop(s1);
    drop(s2);
}
```

`clone`:

- Deep copy
- Available on most built-in types
- Automatically derive for your own types
- When is a type not cloneable?

What’s in a move?

```
fn main() {
   let s1 = String::from("penn");
   let s2 = s1.clone();
   drop(s1);
   drop(s2);
}
```

```
fn main() {
   let s1: u32 = 1337;
   let s2 = s1;
   drop(s1);
   drop(s2);
}
```

Why no error??

`clone`:

- Deep copy
- Available on most built-in types
- Automatically derive for your own types
- When is a type not cloneable?

`Copy`:

- Types with trivial `clone` functions can be marked copy
- In that case move==clone and you don’t have to worry about ownership
- These types often also have trivial destructor functions

What types are `Copy`?

- All numeric types (integers and floats)
- `bool`
- `char`
- Tuples if their members are `Copy` (e.g. `(i32, f64)`)

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

Ways to transfer ownership

1. Assignment (see previous example)
2. Function calls

```
fn main() {
    let s1 = String::from("penn");
    print_str(s1);
    drop(s1);
}
fn print_str(s: String) {
    println!("{}", s);
}
```

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     print str(s1);
  |               -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
```

Ways to transfer ownership

1. Assignment (see previous example)
2. Function calls

Hang on... why do you need ownership to print?

```
fn main() {
    let s1 = String::from("penn");
    print_str(s1);
    drop(s1);
}
fn print_str(s: String) {
    println!("{}", s);
}
```

```
error[E0382]: use of moved value: `s1`
--> lifetimes.rs:4:10
  |
2 |     let s1 = String::from("penn");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     print str(s1);
  |               -- value moved here
4 |     drop(s1);
  |          ^^ value used here after move
```

Borrowing

Need access to a value without owning it?

- Try borrowing
- Defaults to immutable, can also borrow mutably

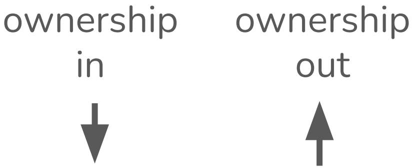
```
fn main() {
   let s1 = String::from("penn");
   print_str(&s1);
   drop(s1);
}
fn print_str(s: &String) {
   println!("{}", s);
}

fn clear_str(s: &mut String) {
   s.clear();
}
```

✅

What about return values?

Return values can transfer ownership too

```
fn main(){
    let s = String::from("I love Rust");
    let with_ferris = add_ferris(s);
}
```



ownership in

ownership out

```
fn add_ferris(s: String) -> String {
    s + "🦀"
}
```

Other ways to write this function

- Pros and Cons?

```
fn add_ferris1(s: &String) -> String {
    s.clone() + "🦀"
}
fn add_ferris2(s: &mut String) {
    s.push_str("🦀")
}
```

View Types

&str and &[T]

A Motivating Example

```
/// Returns last 4 chars of course name
/// e.g. cis1905 -> 1905
fn course_code(course: &String) -> &str {
    ...
}
```

How would you implement this function based on the signature?

- Talk with your neighbor

One solution: create a new string and copy bytes from the `course` string to it

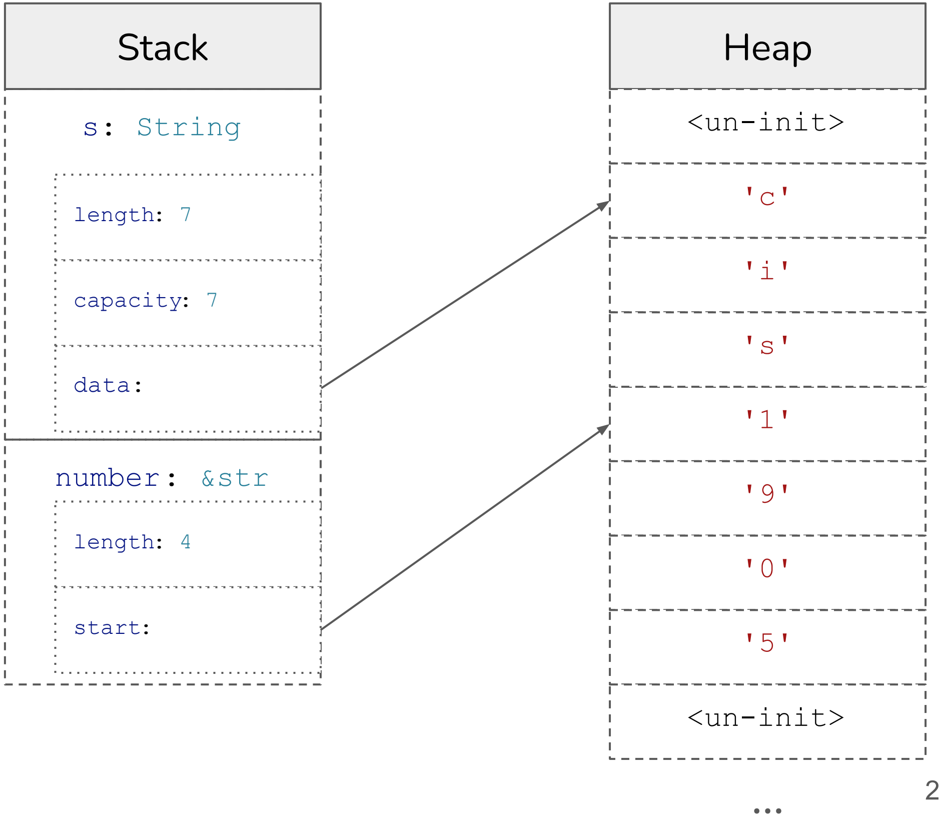
- Inefficient—the bytes already exist in memory so why copy?

A Motivating Example

```
/// Returns last 4 chars of course name
/// e.g. cis1905 -> 1905
fn course_code(course: &String) -> &str {
    course[3..7]
}

fn main() {
    let s = String::new("cis1905");
    let number = course_code(&s);
}
```

“Fat pointer”:

- A pointer along with some data
- Never see just `str`, always `&str` or `&mut str`
- Function arg should always be `&str`, never `&String`. Why?



Stack

s: String

length: 7

capacity: 7

data:

number: &str

length: 4

start:

Heap

<un-init>

'c'

'i'

's'

'1'

'9'

'0'

'5'

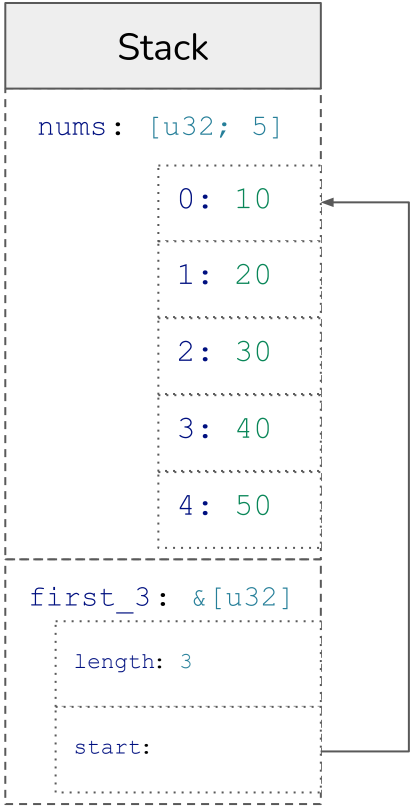
<un-init>

...

Another Fat Pointer: &[T] ("Slice")

Like &str, but for collections of any type

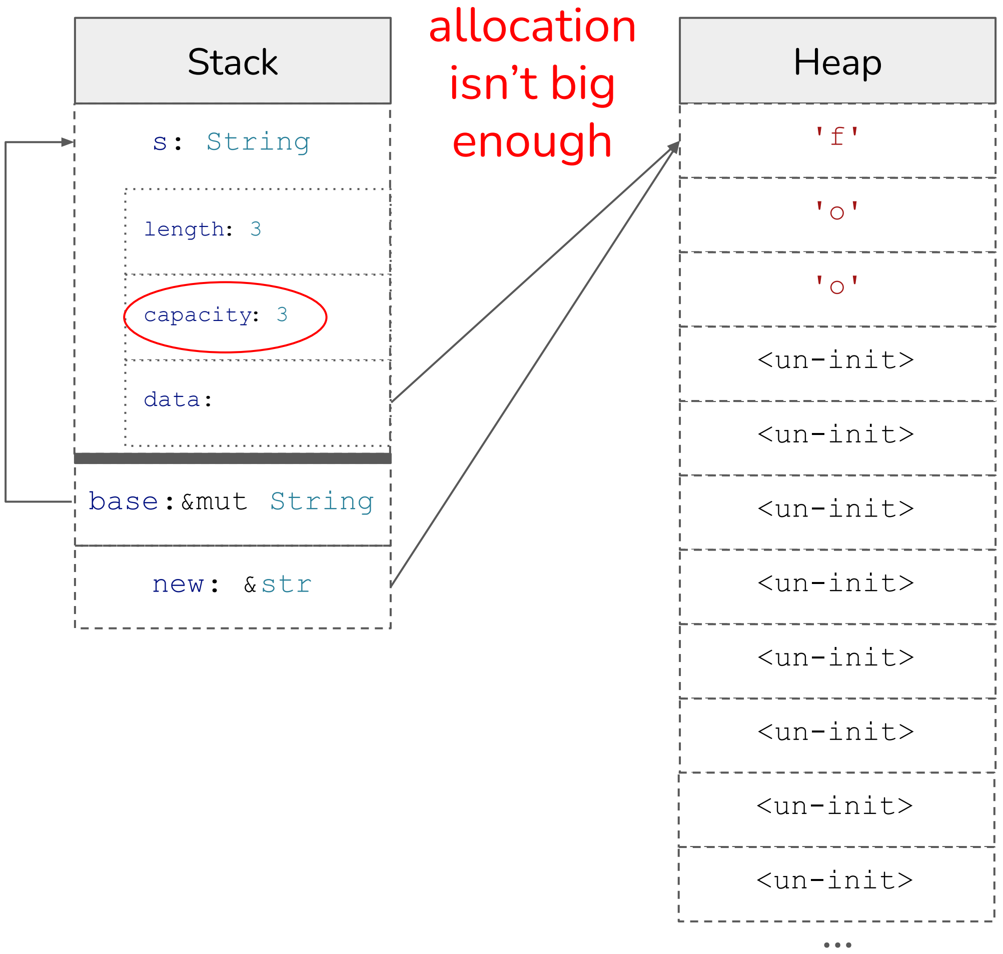
```
fn main() {
    let nums = [10, 20, 30, 40, 50];
    let first_3 = first_3(&nums);
}
fn first_3(arr: &[u32; 5]) -> &[u32] {
    &arr[0..3]
}
```



Controlling mutability

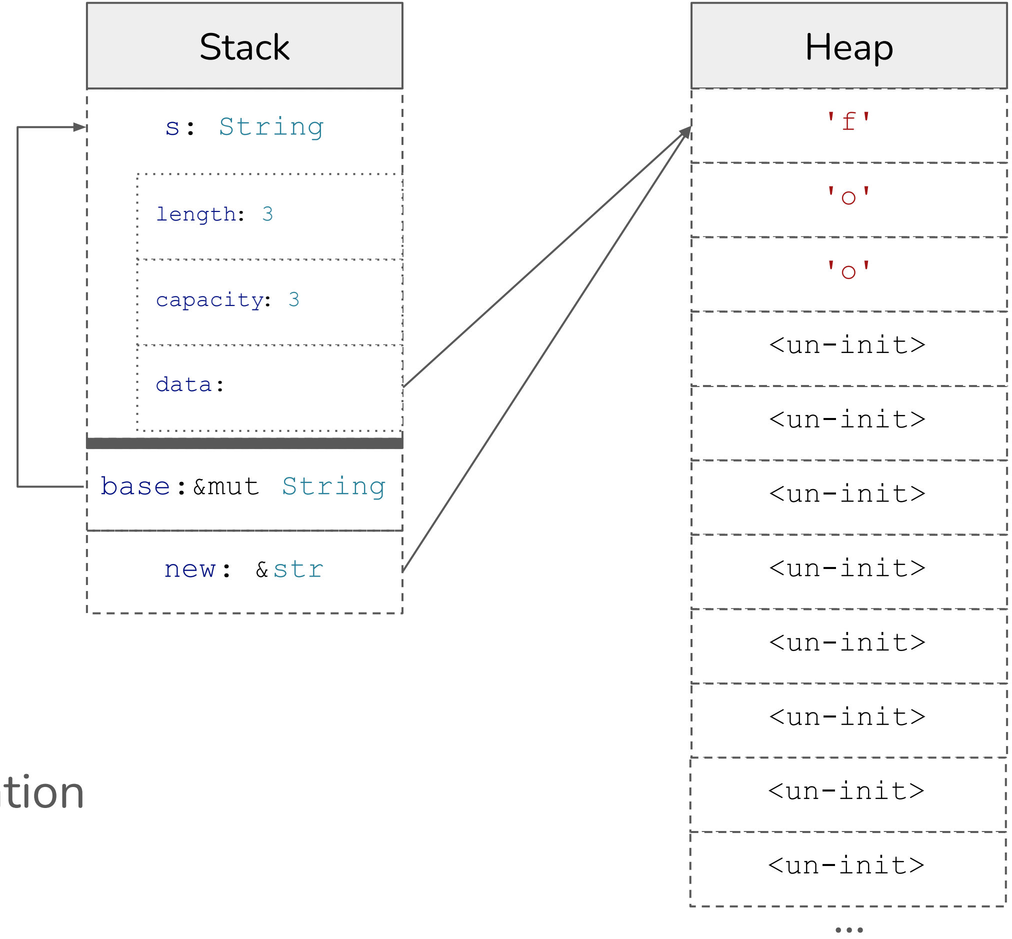
An example

```
fn str_append(
      base: &mut String,
      new: &str) {
    base.push_str(new);
}

fn main() {
    let mut s = String::from("foo");
    str_append(&mut s, &s);
}
```



An example

```
fn str_append(
      base: &mut String,
      new: &str) {
    base.push_str(new);
}

fn main() {
    let mut s = String::from("foo");
    str_append(&mut s, &s);
}
```

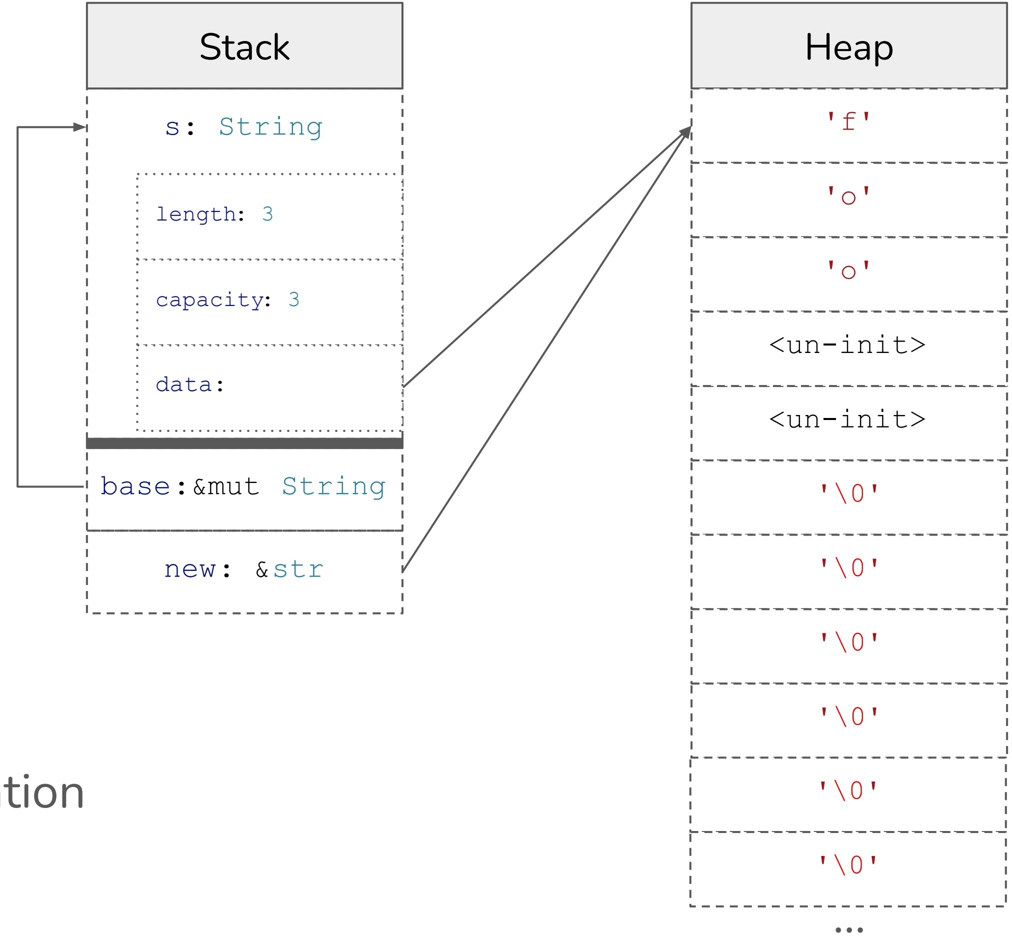


Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

An example

```
fn str_append(
      base: &mut String,
      new: &str) {
   base.push_str(new);
}

fn main() {
   let mut s = String::from("foo");
   str_append(&mut s, &s);
}
```

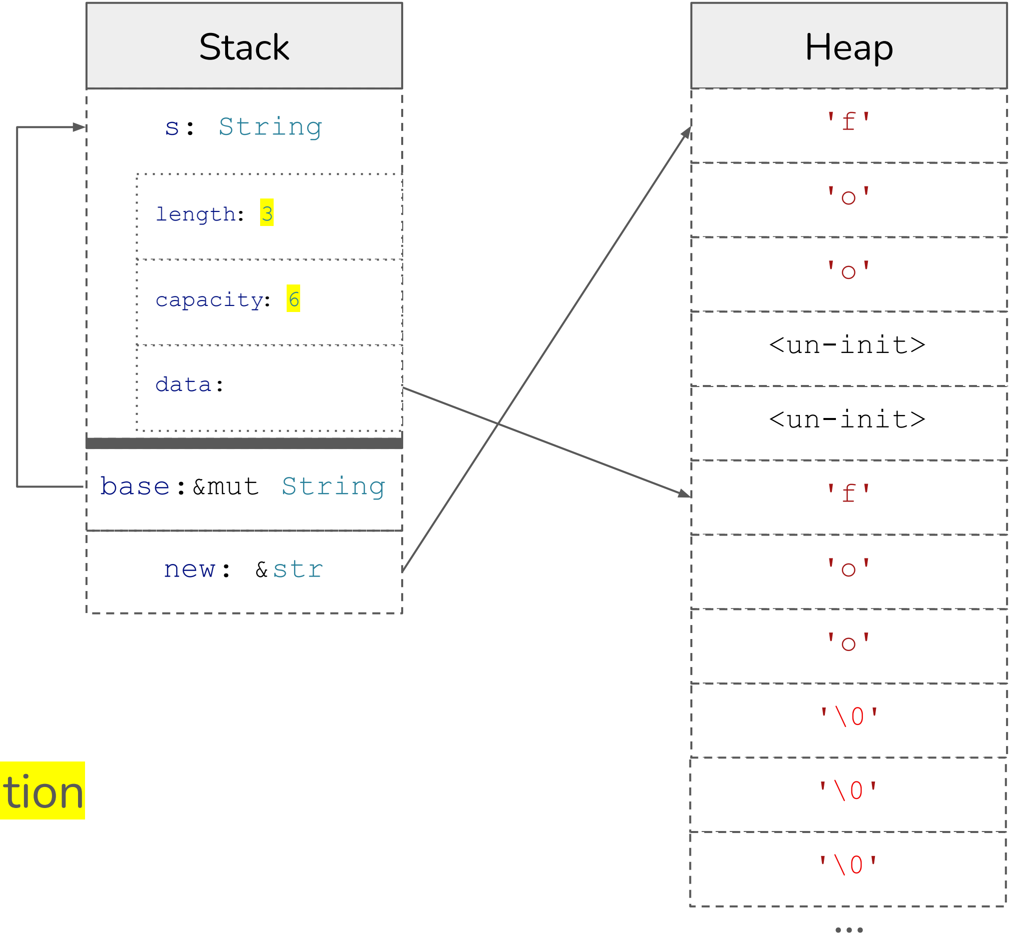


Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

An example

```
fn str_append(
      base: &mut String,
      new: &str) {
   base.push_str(new);
}

fn main() {
    let mut s = String::from("foo");
    str_append(&mut s, &s);
}
```



Stack

s: String

length: 3

capacity: 6

data:

base:&mut String

new: &str

Heap

'f'

'o'

'o'

<un-init>

<un-init>

'f'

'o'

'o'

'\0'

'\0'

'\0'

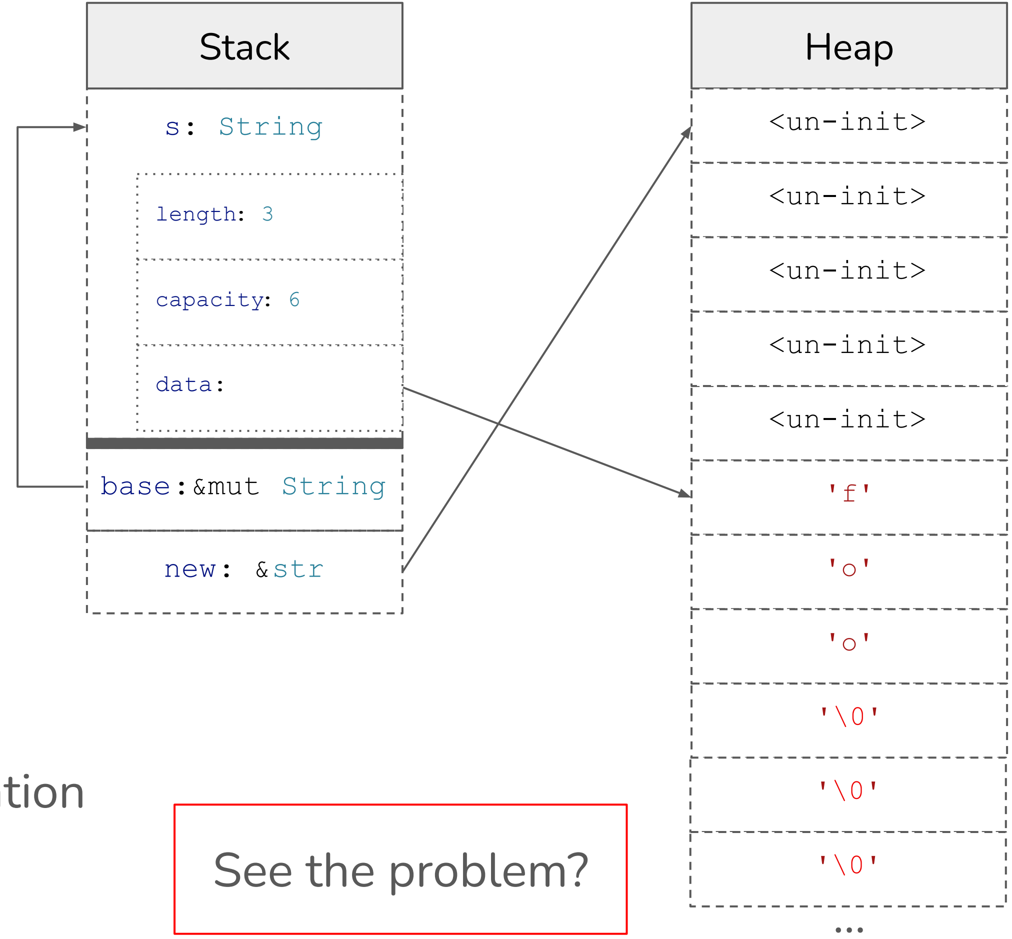
...

Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

An example

```
fn str_append(
     base: &mut String,
     new: &str) {
    base.push_str(new);
}

fn main() {
    let mut s = String::from("foo");
    str_append(&mut s, &s);
}
```



Growing a string:

1. Allocate new memory
2. Copy old data to new allocation
3. Free old allocation

See the problem?

The Rule of References:

- At any given time, you can have either one mutable reference or any number of immutable references.
- References must always be valid.

An example

```
fn str_append(
     base: &mut String,
     new: &str) {
   base.push_str(new);
}

fn main() {
   let mut s = String::from("foo");
   str_append(&mut s, &s);
}
```

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed as mutable
--> lifetimes.rs:9:24
  |
9 |     str_append(&mut s, &s);
  |     ---------- ------  ^^ immutable borrow occurs here
  |     |          |
  |     |          mutable borrow occurs here
  |     mutable borrow later used by call
```

Quiz

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

let r1 = &s;
let r2 = &s;
println!("{} and {}", r1, r2);

let r3 = &mut s;
println!("{}", r3);
```

Does it compile? Talk to your neighbor

Quiz

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

let r1 = &s;
let r2 = &s;
println!("{} and {}", r1, r2);

let r3 = &mut s;
println!("{}", r3);
```

Does it compile? Talk to your neighbor

Yes! Compiler is smart enough to know when you’re done using a reference

Recap

Ownership:

1. Each value in Rust has an *owner*.

2. There can only be one owner at a time.

3. When the owner goes out of scope, the value will be dropped.

Transfer ownership with *move* (like a shallow copy)

- When assigning
- When calling/returning from functions

Opt out of moving by `clone`ing (performance hit)

References:

To avoid transferring ownership, borrow an owned value to get a reference

- Nothing happens when reference goes out of scope

References can be immutable or mutable

- At any given time, you can have either one mutable reference or any number of immutable references.

References must always be valid.