Transferring Ownership

CS128 Honors Ownership Module

Slides by Matt Geimer (FA21)
Presented 9/22/2021

Recap

Ownership

1. Each value in Rust has a variable that’s called its owner.

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

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

Recap

Ownership

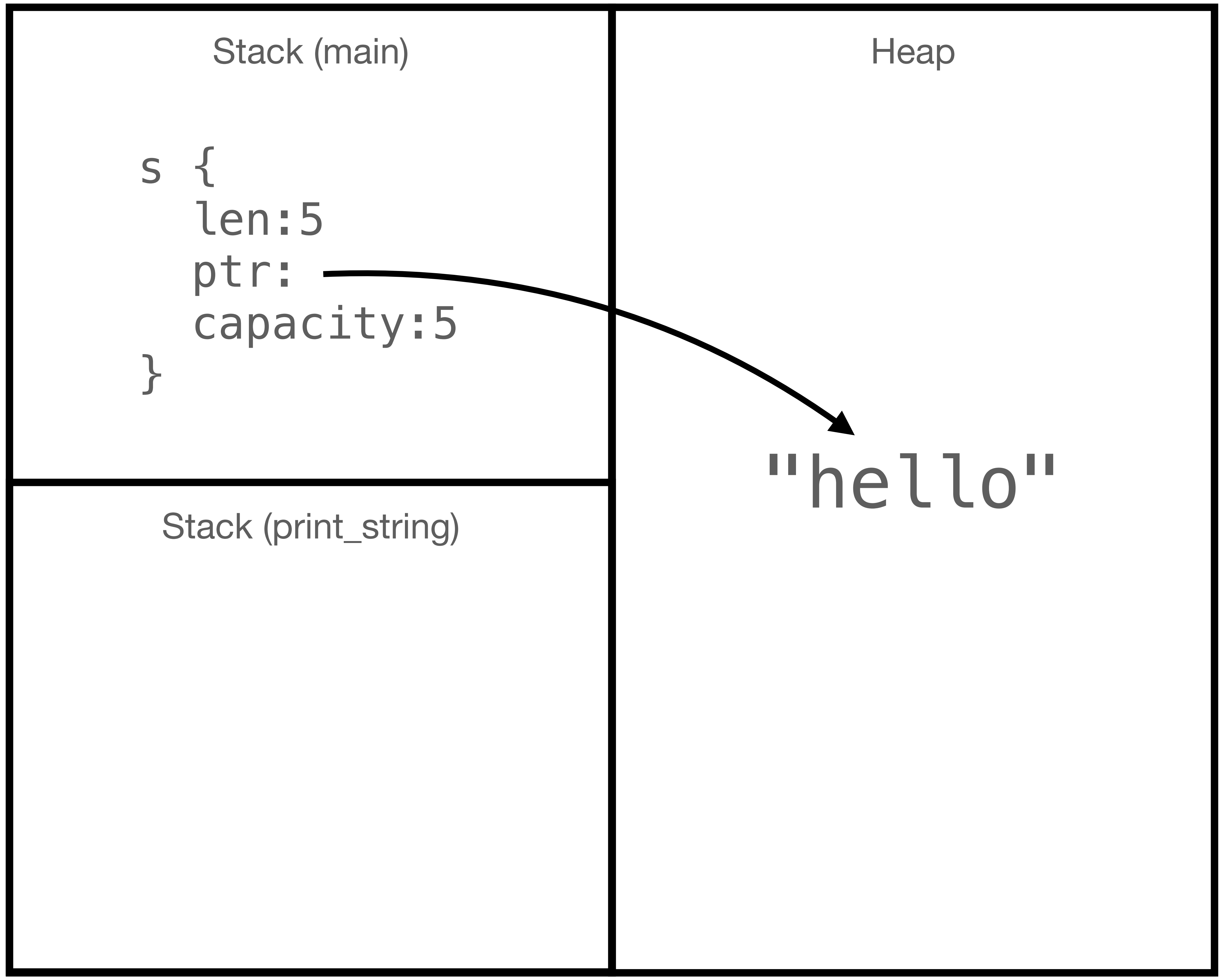
- All scalar types are **copied by value**
  - This means ownership (practically) doesn't apply to scalar types
- Ownership rules apply to values stored on the **heap** (variable size)

Ownership & Functions

- **Functions take ownership** of variables by default
- What does this look like practically?

Ownership & Functions

```
fn main() {
    let s = String::from("hello"); ←
    print_string(s);

    let x = 5;
    print_num(x);
}

fn print_string(some_string: String) {
    println!("{}", some_string);
}

fn print_num(some_integer: i32) {
    println!("{}", some_integer);
}
```



Stack (main)

```
s {
  len:5
  ptr:
  capacity:5
}
```

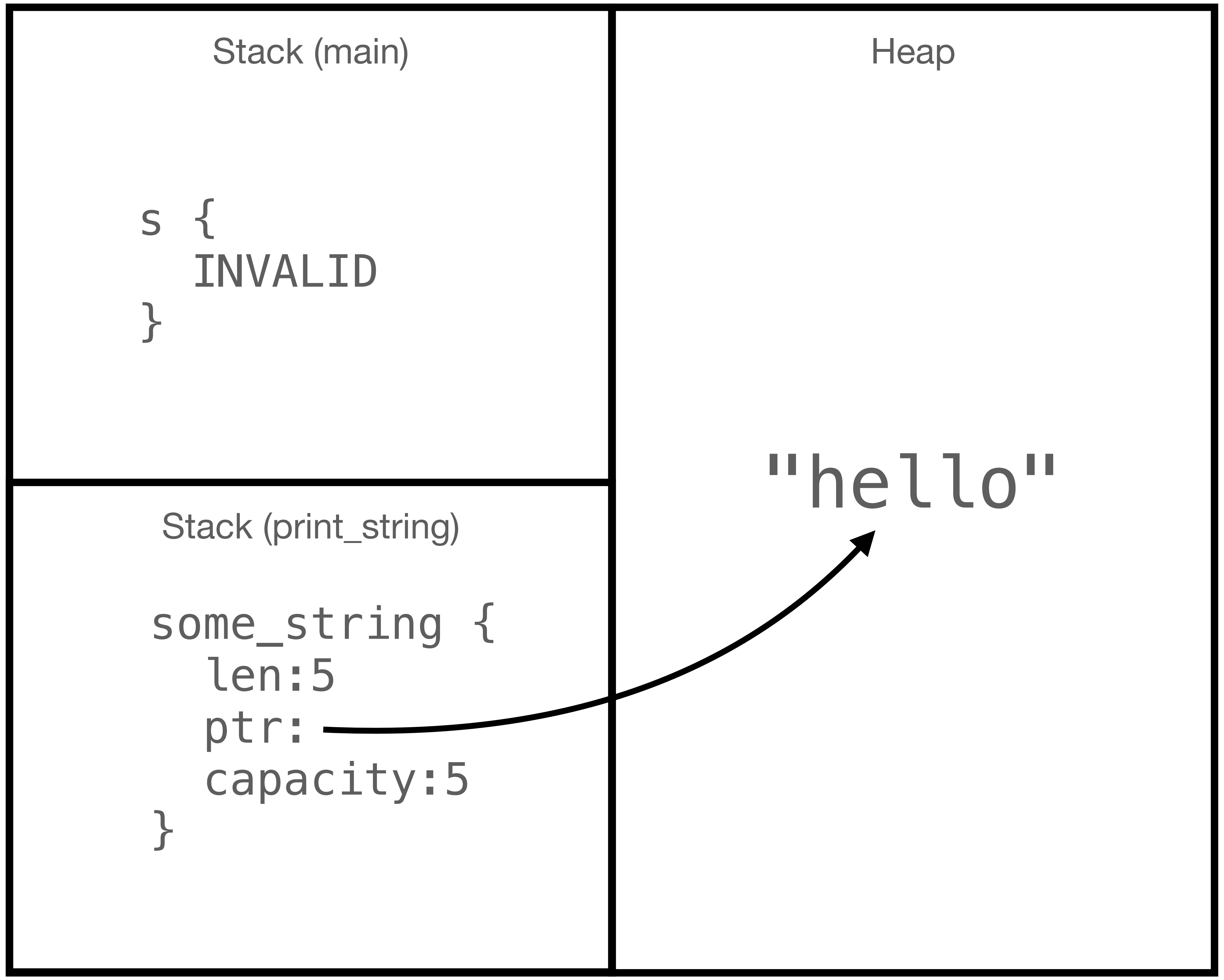
Heap

"hello"

Stack (print_string)

Ownership & Functions

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

    let x = 5;
    print_num(x);
}

fn print_string(some_string: String) {
    println!("{}", some_string); ←
}

fn print_num(some_integer: i32) {
    println!("{}", some_integer);
}
```



Stack (main)

```
s {
  INVALID
}
```

Stack (print_string)

```
some_string {
  len:5
  ptr:
  capacity:5
}
```

Heap

"hello"

Ownership & Functions

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

    let x = 5;
    print_num(x);
}

fn print_string(some_string: String) {
    println!("{}", some_string);
}

fn print_num(some_integer: i32) {
    println!("{}", some_integer);
}
```

Ownership & Functions

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

    let x = 5;
    print_num(x);

    print_string(s);
}

fn print_string(some_string: String) {
    println!("{}", some_string);
}

fn print_num(some_integer: i32) {
    println!("{}", some_integer);
}
```

```
error[E0382]: use of moved value: `s`
 --> src/main.rs:8:18
  |
2 |     let s = String::from("hello");
  |         - move occurs because `s` has type `std::string::String`,
  |           which does not implement the `Copy` trait
3 |     print_string(s);
  |                  - value moved here
...
8 |     print_string(s);
  |                  ^ value used here after move

error: aborting due to previous error

For more information about this error, try `rustc --explain E0382`.
```

So all hope is lost?

Ownership & Functions

- Just like functions can take ownership, functions can also **give ownership**

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

    print_string(s);
}

fn print_string(some_string: String) -> String {
    println!("{}", some_string);
    return some_string
}
```

Ownership & Functions

- Just like functions can take ownership, functions can also **give ownership**

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

    print_string(s);
}

fn print_string(some_string: String) -> String {
    println!("{}", some_string);
    return some_string
}
```

- Of course, this **can get messy fast** when passing many variables around

If we had to do this **every** time, nobody would write Rust code

Borrowing

- Borrowing is the temporary use of a variable
- Borrowing is accomplished by **referencing** a variable
- To reference a variable, use `'&'`

Borrowing

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

    let x = 5;
    print_num(x);

    print_string(s);
}

fn print_string(some_string: String) {
    println!("{}", some_string);
}

fn print_num(some_integer: i32) {
    println!("{}", some_integer);
}
```

Borrowing

```
fn main() {
    let s = String::from("hello");
    print_string(&s);

    let x = 5;
    print_num(x);

    print_string(&s);
}

fn print_string(some_string: &String) {
    println!("{}", some_string);
}

fn print_num(some_integer: i32) {
    println!("{}", some_integer);
}
```

```
hello
5
hello
```

Borrowing

- It’s also possible to make borrowed variables mutable using &mut

```
fn main() {
    let mut s = String::from("hello");
    print_string(&mut s);

    println!("{}", s);
}

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

```
hello
hello, world!
```

The Catch!

- Borrowed variables have one rule:
- You can **either** have
  - unlimited immutable borrowed variables

  OR

  - one mutable borrowed variable
- But **NOT** both

The Catch!

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

    let r1 = &s; // no problem
    let r2 = &s; // no problem
    let r3 = &mut s; // BIG PROBLEM

    println!("{} {} {}", r1, r2, r3);
}
```

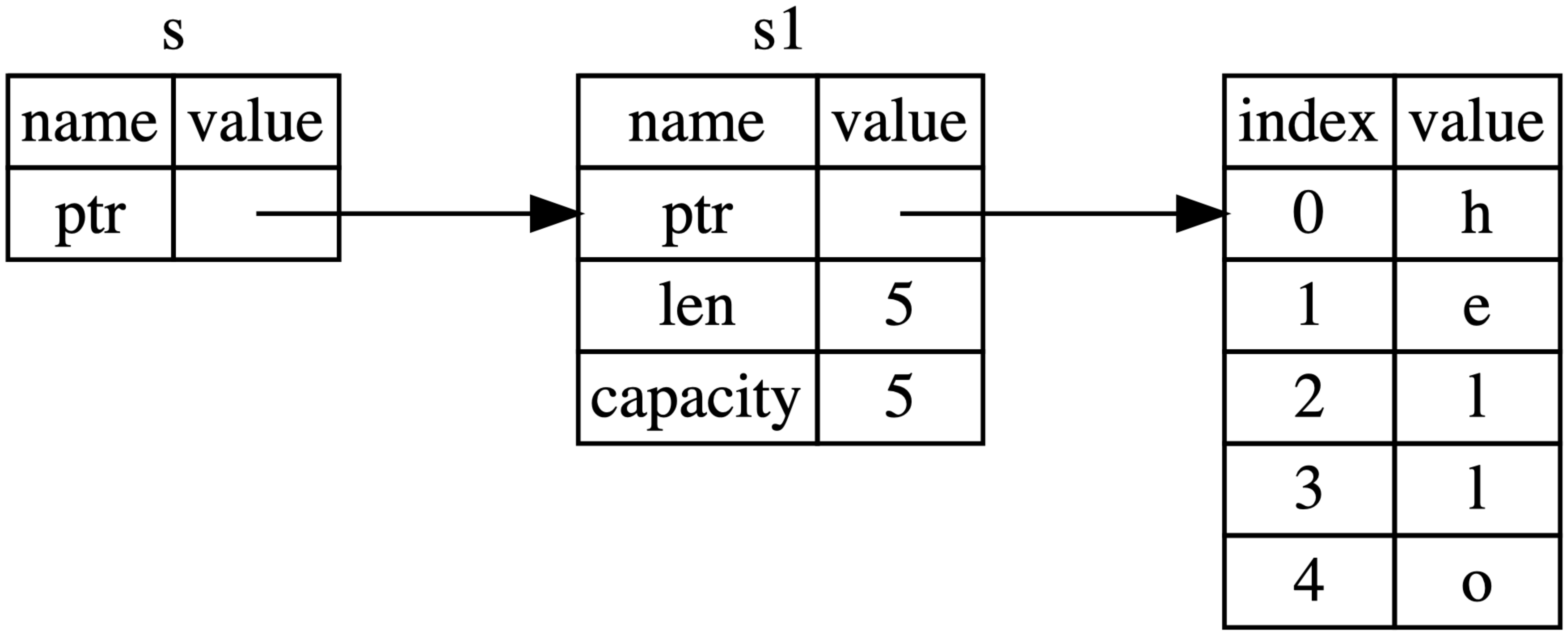
The Catch!

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

    let r1 = &s; // no problem
    let r2 = &s; // no problem

    println!("{} {}", r1, r2);

    let r3 = &mut s; // no problem
    println!("{}", r3);
}
```

← r1 & r2 go out of scope here

Dereferencing

- How are these variables being stored in memory?
- s = &s1



s

name	value
ptr	

s1

name	value
ptr	
len	5
capacity	5

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

Dereferencing

```
fn main() {
    let mut num_to_increment = 5;
    increment_by_five(&mut num_to_increment);
    println!("{}", num_to_increment);
}

fn increment_by_five(num: &mut i32) {
    *num += 5;
}
```

Why didn’t we have to do this before?

In some cases, the Rust compiler does it for you

😢

Why are we doing all this?

What do Java & C++ do?

Java

- Uses the “Garbage collector”
- Periodically goes around checking if memory is still being used
- Automatic memory management
- **SLOW!!!**

C++

- Programmer manually allocates/frees memory
- **Prone to human errors**
- Means system can focus on actually running code
- Fast

What do Java & C++ do?

Rust

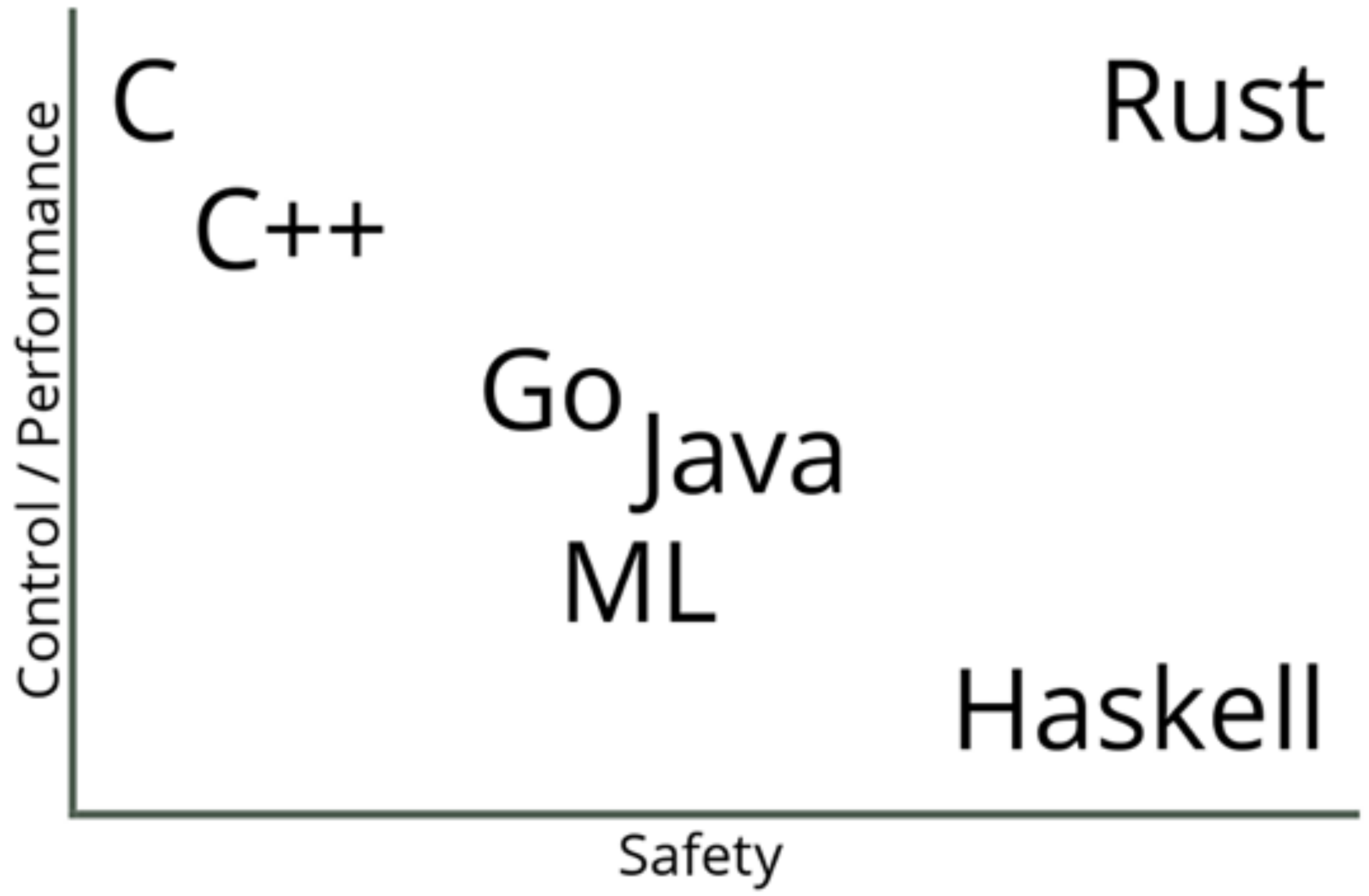
- **Compiler automatically inserts memory allocations/frees**
- “Automatic” memory management
- Means the system can focus on actually funning code
- Fast

C++

- Programmer manually allocates/frees memory
- **Prone to human errors**
- Means system can focus on actually running code
- Fast

Comparisons

NOT TO SCALE



Summary

Ownership in Functions

- Functions giving/receiving ownership
- Introduced Borrowing
- Dereferencing
- Why do manual memory management?

Transferring Ownership

CS128 Honors Ownership Module

Slides by Matt Geimer (FA21)
Presented 9/22/2021