Ownership Continued

Ryan Eberhardt and Julio Ballista

April 8, 2021

Logistics

- Week 1 exercises due at 11:59 pm PST. Let us know if you need more time!
- Week 2 exercises will be released today and will be due next Thursday
- If you're comfortable, post reflections on the #reflections channel on Slack. Great way to synthesize learning and get a sense for the lessons everyone else is picking up!
- Today: More on Ownership and Rust :D

Previously on 110L...

Ownership

```
let julio = Bear::get();
let ryan = julio;
```

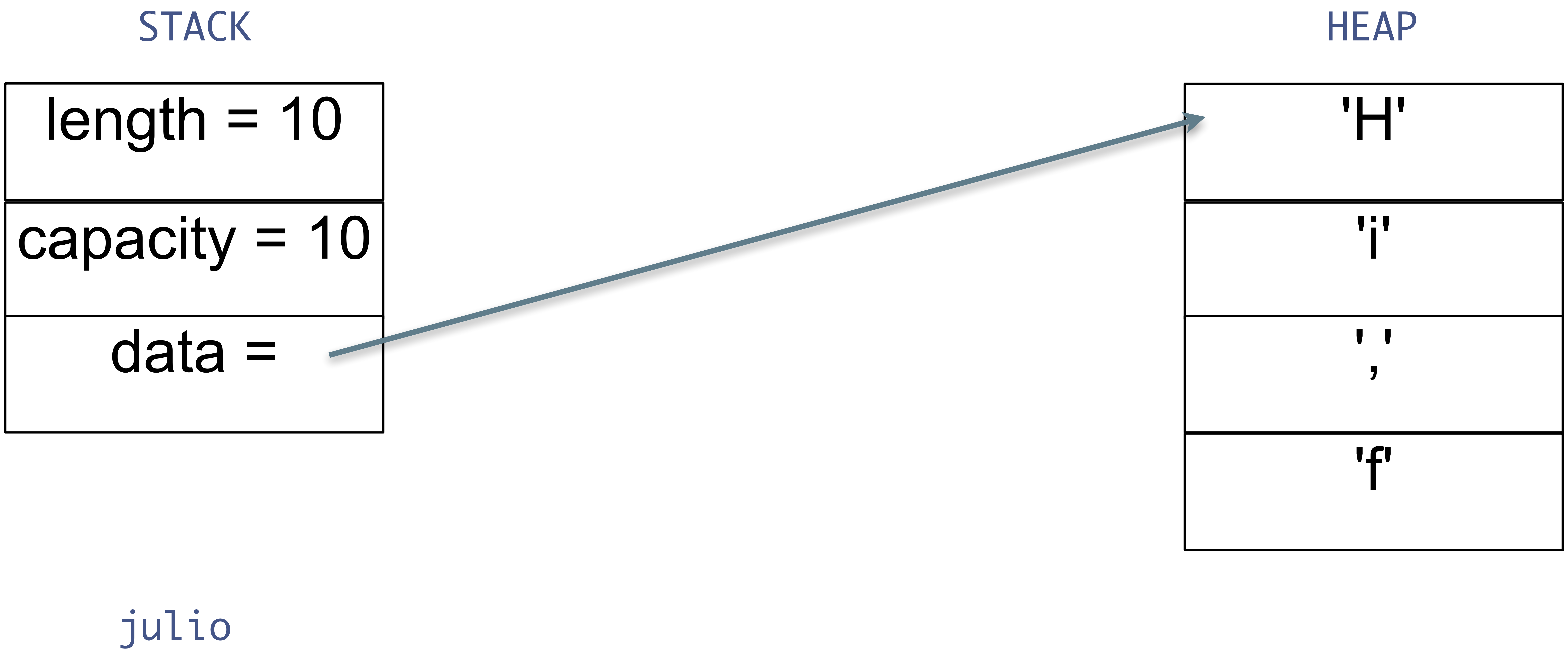
`julio;`

`ryan;`

How does ownership transfer actually look in memory?

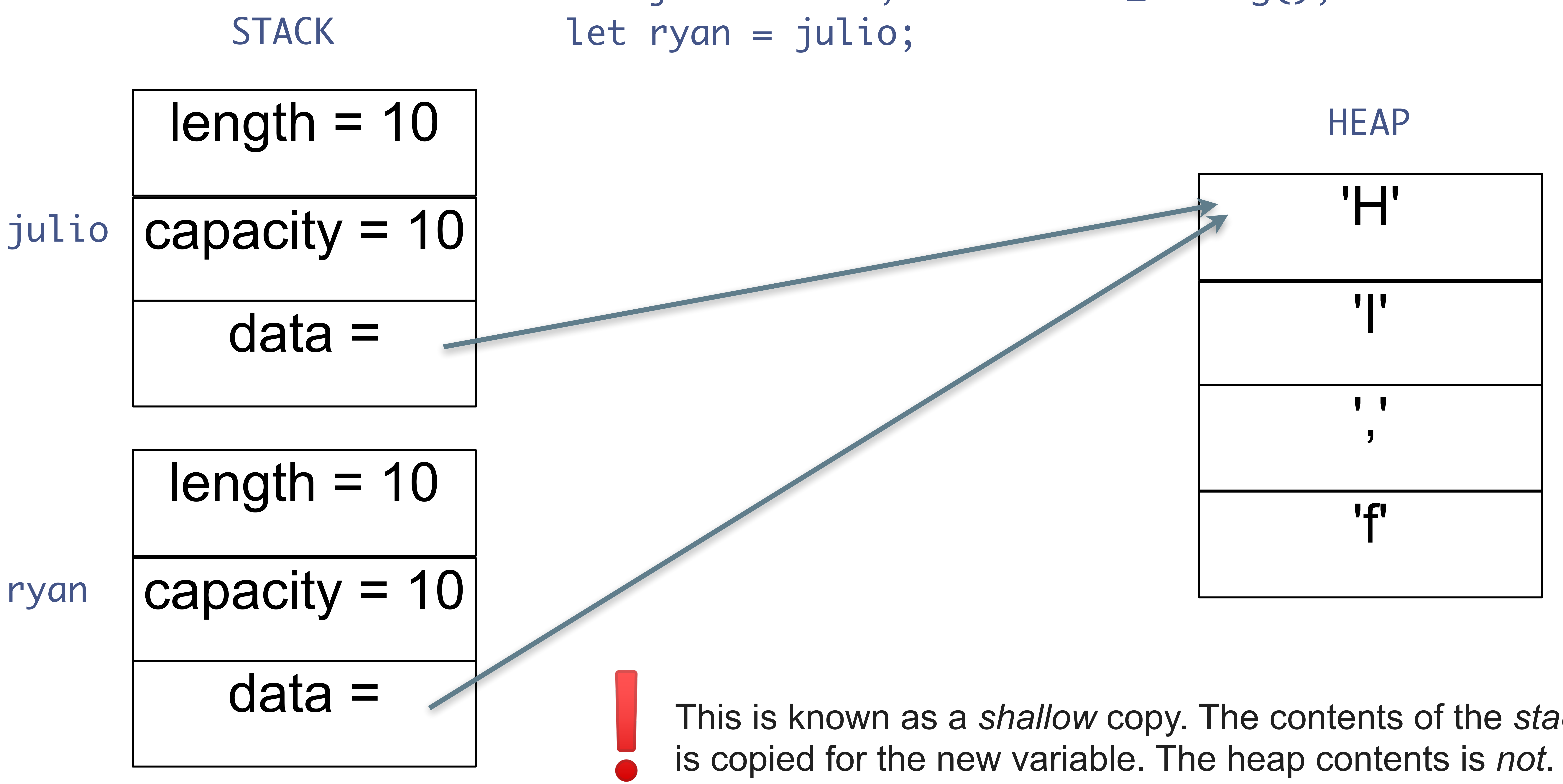
Ownership in Memory

```
let julio = "Hi,friends".to_string();
```



Ownership in Memory

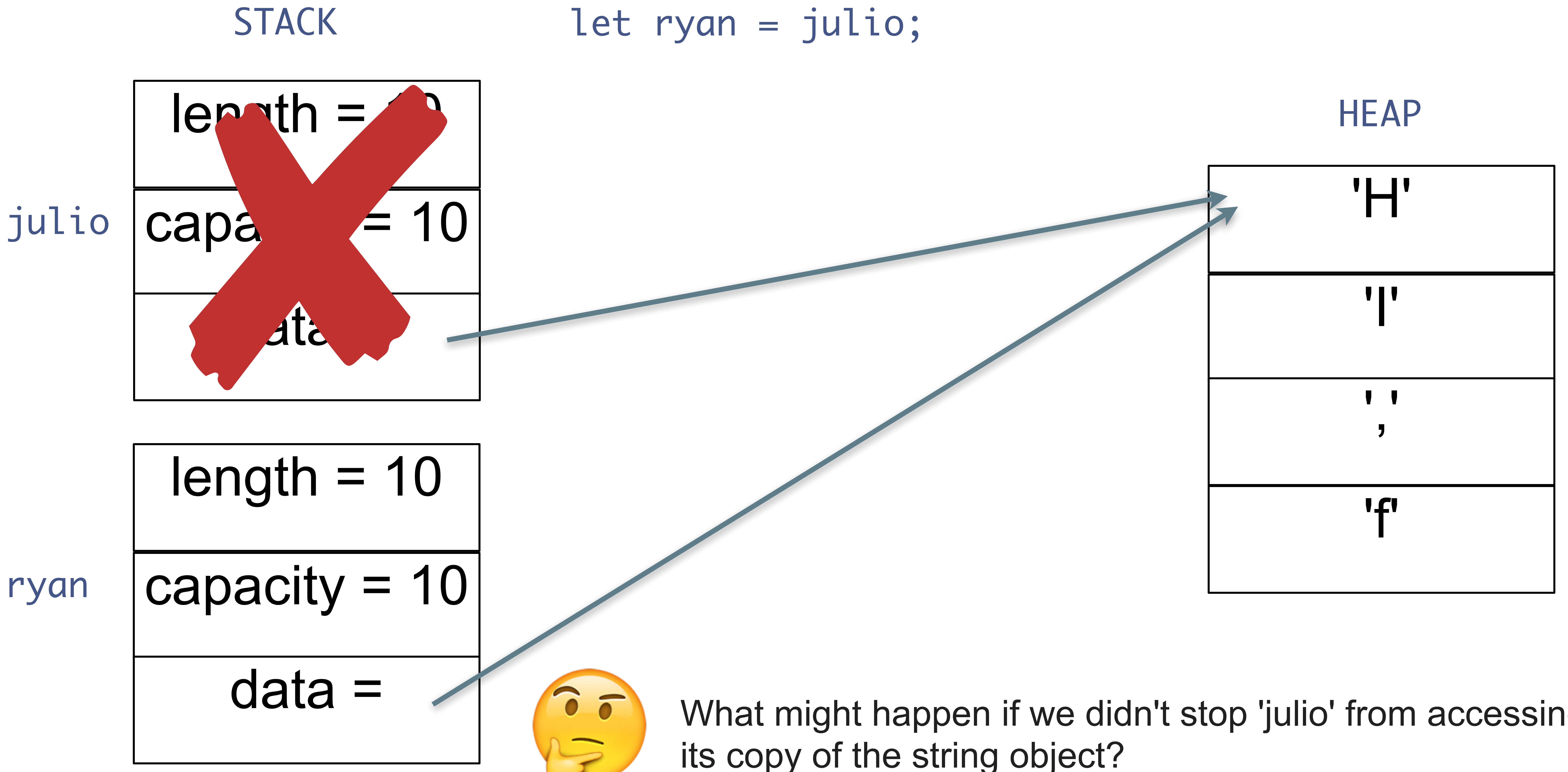
```
let julio = "Hi,friends".to_string();
let ryan = julio;
```



❗ This is known as a *shallow* copy. The contents of the *stack* is copied for the new variable. The heap contents is *not*.

Ownership in Memory

```
let julio = "Hi,friends".to_string();
let ryan = julio;
```



What might happen if we didn't stop 'julio' from accessing the values in its copy of the string object?

Ownership in Memory

- When we reach the end of a scope (designated by curly-braces), the ***Drop*** function is called.
- You can think of this being a special function to properly free() the entire object (maybe multiple pointers to free, so the function will have that implementation)
- Similar to the destructor in C++
- Types with the Rust ***Drop*** trait have a ***Drop*** function to call (more on traits soon!)

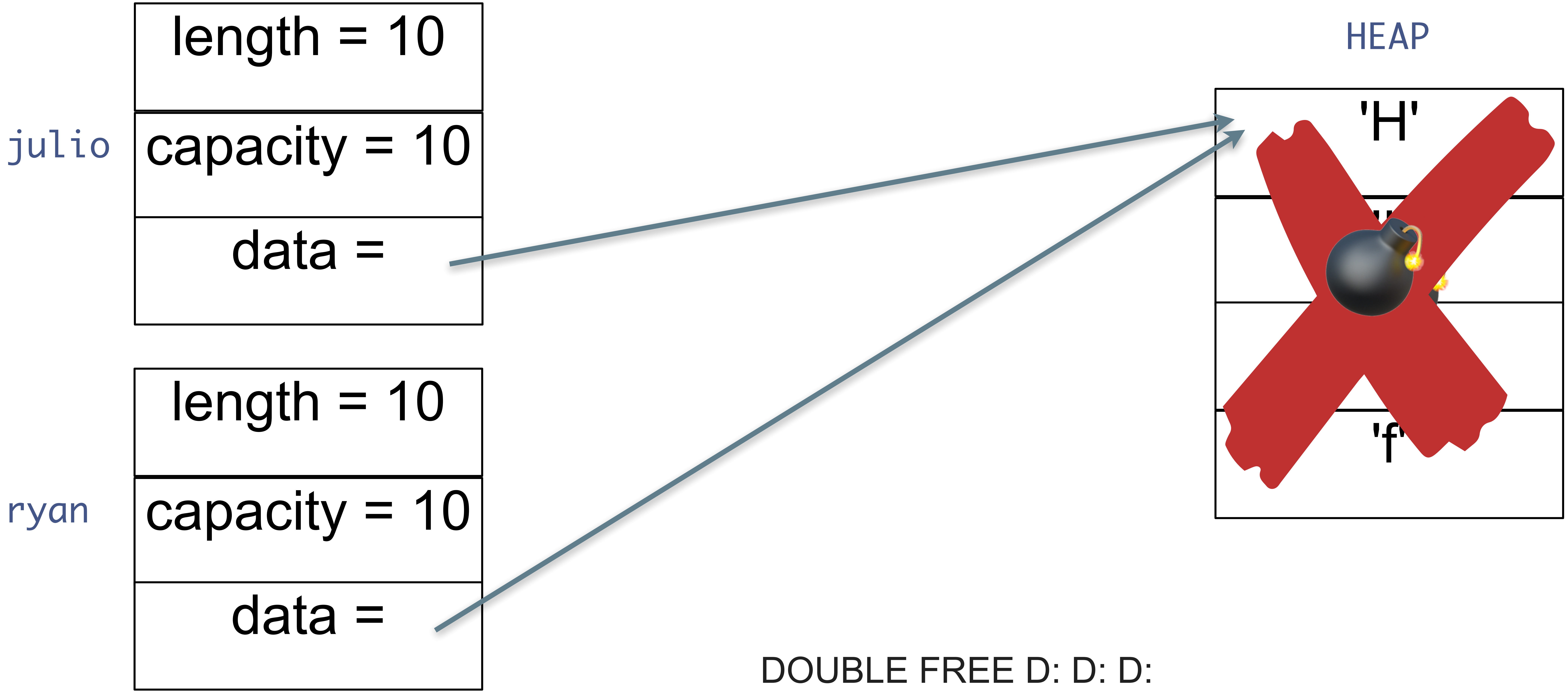
```
fn main() {
  let julio = "Hi,friends".to_string();
  let ryan = julio;
}
```

End of variable scope!
Drop function called for variables *owning* values

Ownership in Memory

```
let julio = "Hi,friends".to_string();
let ryan = julio;
```

STACK



DOUBLE FREE D: D: D:

Ownership in Memory: Recap

- We make *shallow* copies of variables when passing ownership, and we *invalidate* previous variables that no longer own the variable.
- The invalidation is to prevent double-frees - much safer when we know exactly who should call the **Drop** function.
- If you wanted to make a deep copy (copy the data on the *heap)*, Rust has the *clone* function.

Clone function

```
let julio = "Hi,friends".to_string();
let ryan = julio.clone();
```

Now, julio and ryan have their own heap data!

Questions?

Ownership in Memory

```
let julio = 10;
```

Ownership in Memory

```
let julio = 10;
let ryan = julio
```

STACK

HEAP

julio | value = 10

ryan | value = 10

HEAP
?
?
?
?

What might happen if we didn't stop 'julio' from accessing the values in its copy of the number object?

Ownership in Memory

```
let julio = 10;
let ryan = julio
```

STACK

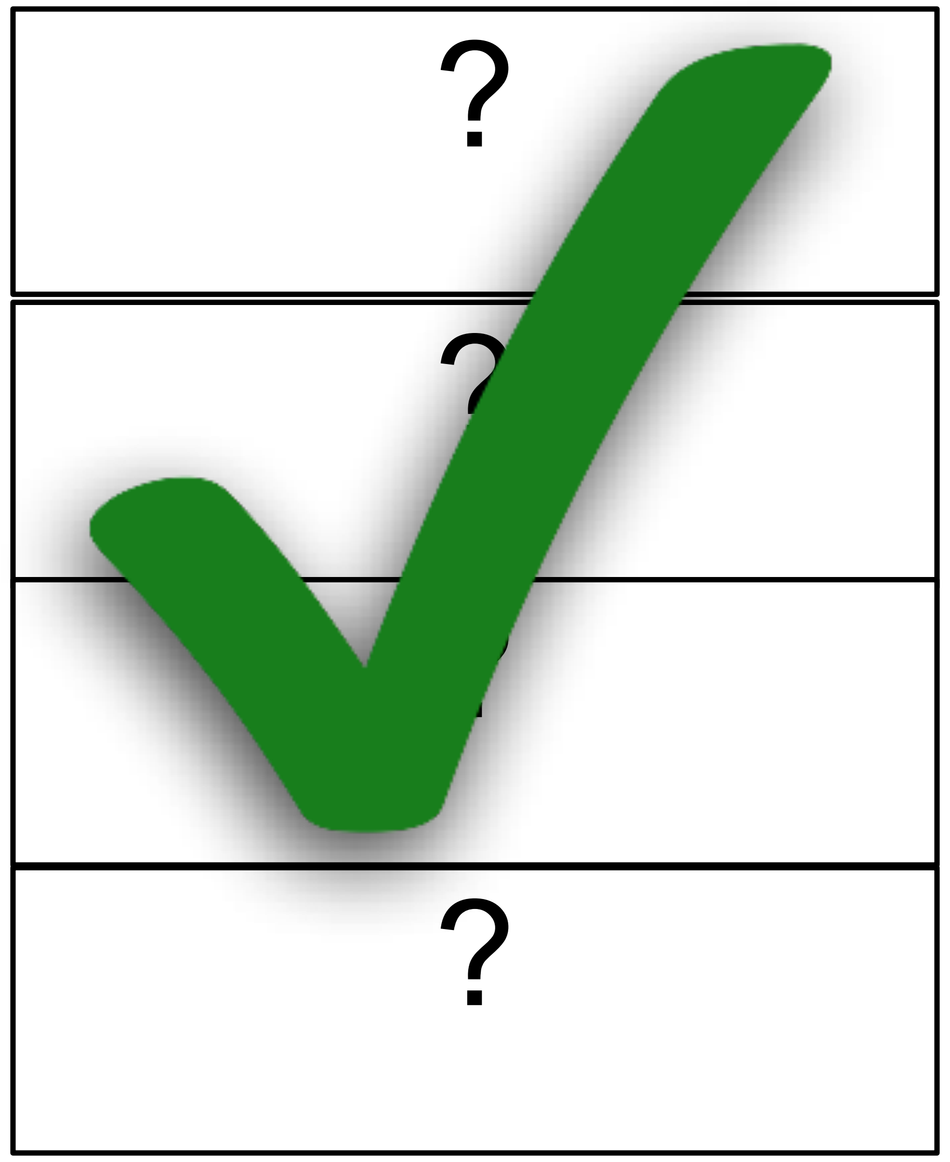
julio

value = 10

ryan

value = 10

HEAP



Absolutely nothing - the heap is safe!

What's going on here?

- Some values in Rust do not make use of the heap, and are stored directly on the stack. (integer types (u32), booleans, etc)..
- These variables are typically copied **by default** when assigning variables, as you don't need to worry about any **Drop** function being called (and hence, no memory issues!!)
- Types with this property have the **Copy** trait.
- If you have the **Copy** trait, you cannot also have the **Drop** trait (why?)

Copy Trait Error

```
   Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `julio`
 --> src/main.rs:5:17
  |
2 |   let julio = "Hi friends".to_string();
  |       ----- move occurs because `julio` has type `String`, which does not implement the `Copy` trait
3 |   let ryan = julio;
  |              ----- value moved here
4 |
5 |   println!("{}", julio);
  |                  ^^^^^ value borrowed here after move
```

Without the **Copy** trait, Rust assumes ownership is moving!

Questions?

Borrowing++

Borrowing: Recap

```
let julio = Bear::get();
my_cool_bear_function(&julio)
/* The julio variable can still be used here! */
```

julio; my_cool_bear_function;

What are the rules behind the &?

Variables Rules in Rust

- All pieces of data, by default, are **immutable** in Rust.
- You can imagine that *const* is secretly behind every variable you instantiate.
- The Rust Compiler will *not* compile your code if you change any variable that is not mutable.
- The **mut** keyword specifies a variable to be **mutable.** It's like the opposite *const.*

Mutable Variables

```
let lst = vec![1,2,3];
vec.push(4);
```

```
let mut lst = vec![1,2,3];
vec.push(4);
```

'Borrowing' creates a type!

```
let julio = Bear::get();
my_cool_bear_function(&julio)
/* The julio variable can still be used here! */
```

```
let julio = Bear::get();
let julio_reference = &julio;
my_cool_bear_function(julio_reference);
/* The julio variable can still be used here! */
```

"Borrowing Type" == References!

- & creates a new variable type, known as a **reference** to that type.
- Because this is another type, they too are **immutable** by default, and can be made **mutable** with the **mut** keyword.
- Mutable references can only be made if the actual variable is also mutable!

```
let julio = Bear::get();
let julio_reference = &julio;
// let julio_mutable_reference = &mut julio;

my_cool_bear_function(julio_reference);
/* The julio variable can still be used here! */
```

Code: Immutable + Mutable References

Function takes in a **reference** to a vector!

```
fn append_to_vector(lst: &Vec<u32>) {
    lst.push(3);
}

fn main() {
 let mut lst = vec![1,2,3];
 append_to_vector(&lst);
}
```

Main passes a **reference** to append_to_vector...

Code: Immutable + Mutable References

But it must be a **mutable reference** since the vector is changed!

```
fn append_to_vector(lst: &mut Vec<u32>) {
    lst.push(3);
}

fn main() {
 let mut lst = vec![1,2,3];
 append_to_vector(&mut lst);
}
```

Main must also pass a **mutable reference** through!

Questions?

Borrowing + References: The Catch

```
let mut bear = Bear::get();
```

Because both references are **immutable**, both painters can trust the bear they see!

```
let pink_shirt = &bear;
```

```
let blue_shirt = &bear;
```

Borrowing + References: The Catch

```
let mut bear = Bear::get();
```

```
let pink_shirt = &bear;
let blue_shirt = &bear;
let evil_patrick = &mut bear;
```

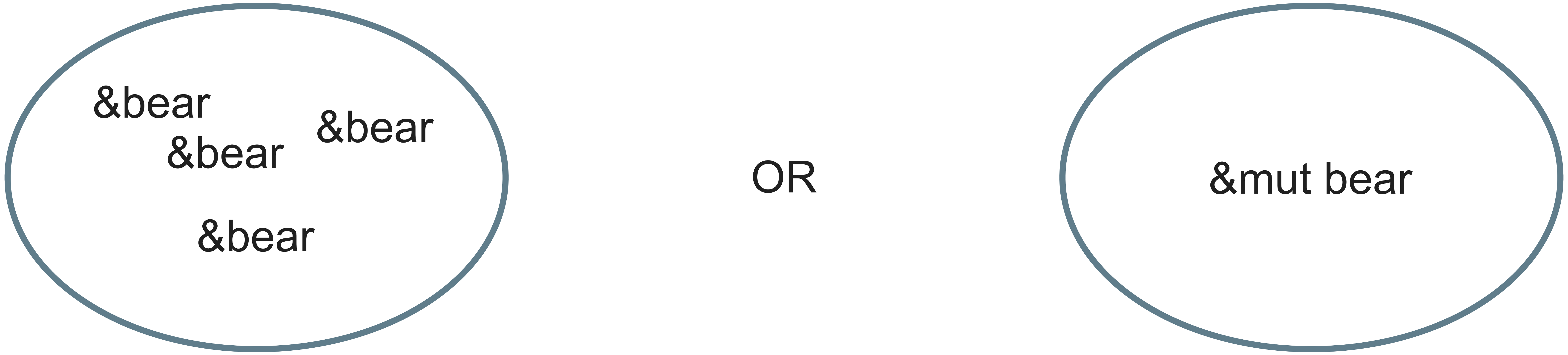
Borrowing + References: The Catch

```
let mut bear = Bear::get();
```

```
let pink_shirt = &bear;        let blue_shirt = &bear;     let evil_patrick = &mut bear;
```

References Rules

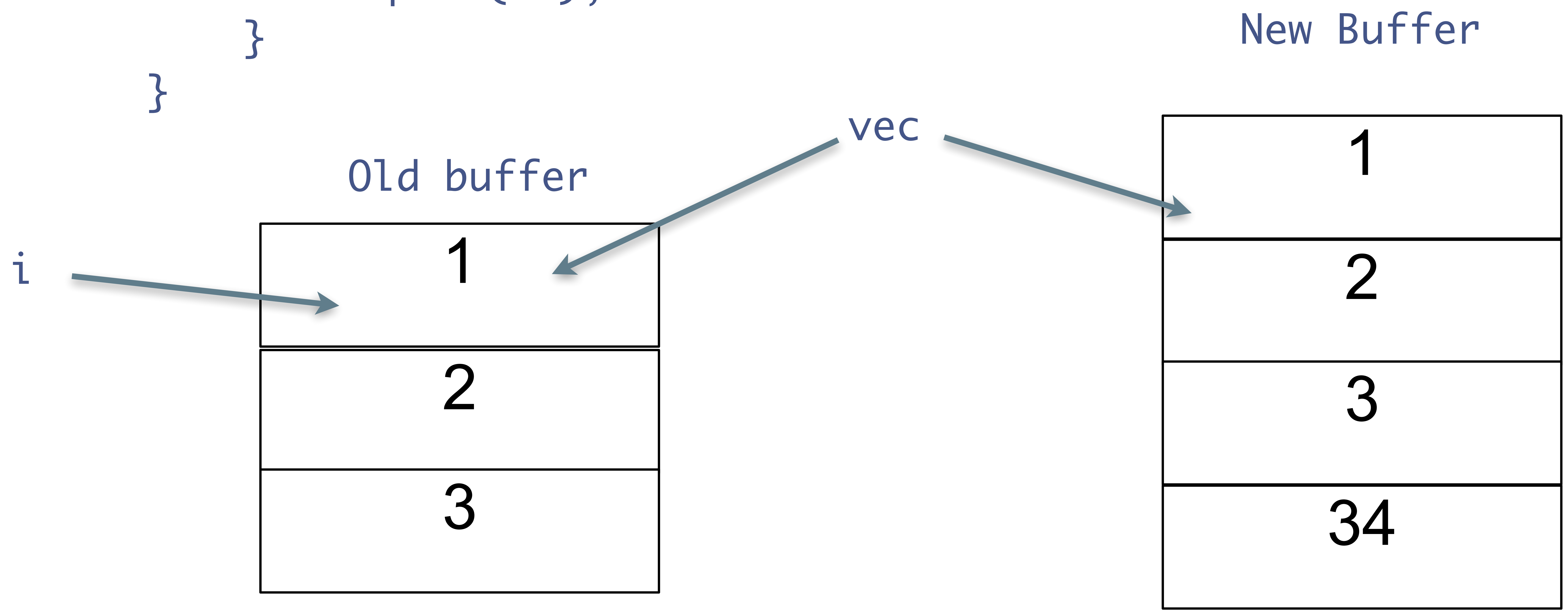
- We can have many kinds of **immutable** references for a variable (Think that many painters can paint on their canvas, so long as they know no one will change that painting)
- But we can only have **one** mutable reference at a time. Otherwise, the immutable references might see different data than what they initially expected.



Code Example

Iterator Invalidation Avoided!

```
fn main() {
    let mut v = vec![1, 2, 3];
    /* This for loop borrows the vector above to do its work. */
    for i in &mut v {
        println!("{}", i);
        v.push(34);
    }
}
```



References Recap [End]

- With the ownership and borrowing rules, many different kinds of memory errors are avoided :D
- But they do lead to trickier code to write - the Rust compiler will fight with you as you write these programs
- Take it slow, ask questions in the #rust-questions channel!