# Ownership Continued

CS110L
Jan 12, 2022

# Logistics

- Solutions for week 1 exercises released! Give ‘em a read & put questions in Slack.
- Week 2 exercises will be released today and will be due Tuesday
  - Handout is on the website.
  - You should have gotten an invitation from CS110L to join a repository on GitHub. It will be called `week2-yourSUNetID`.
    - Starter code is here; submit by pushing to this repo.
  - Please check that you got this, and let me know if you didn’t!
    - If you didn’t fill out the intro form, you don’t have a repo. Fill out the form & let me know on Slack.
    - If you’re not officially enrolled, you don’t have a repo. Message me on Slack & I’ll get you one.
- Reminder: no class on Monday; remote on Wednesday.

# Today: more ownership & Rust!

# Previously on 110L...

# Ownership (From The Rust Book!)

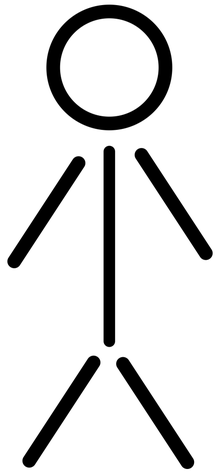
## Ownership Rules

First, let's take a look at the ownership rules. Keep these rules in mind as we work through the examples that illustrate them:

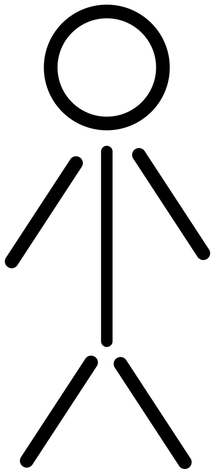
- Each value in Rust has a variable that's called its *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

# Ownership (from Monday)

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

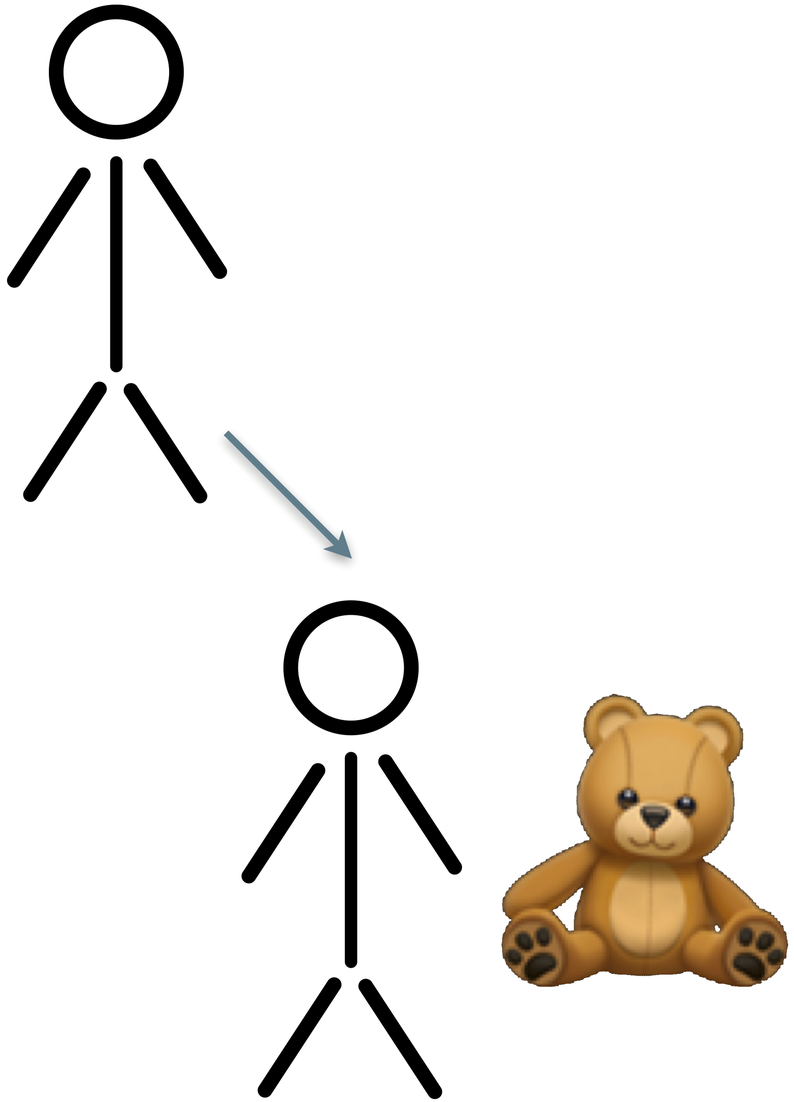


julio;



ryan;

# Function calls transfer ownership (from Monday)

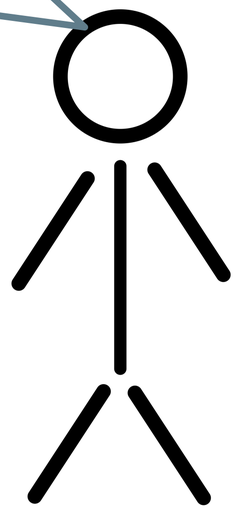


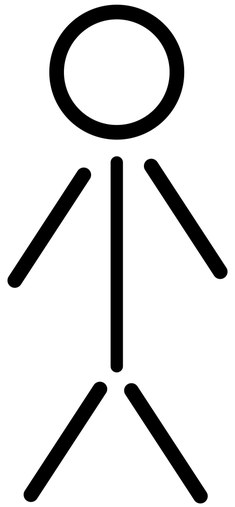
```
fn my_cool_bear_function(/* parameter */ ) {
  // Do stuff
  // Value (bear) will go out of scope – freed!
}


fn main() {
  let julio = Bear::get();
  my_cool_bear_function(julio);
  /* ^This transfers ownership to parameter in function */

  /* julio no longer owns the toy D: Compiler wont let you
  use it! */
}
```

# Borrowing (from Monday)

Hey, my_cool_bear_function, you could BORROW this toy. Just give it back when you're done!

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





Thank you, this means you'll have to put the toy back when you're done though!

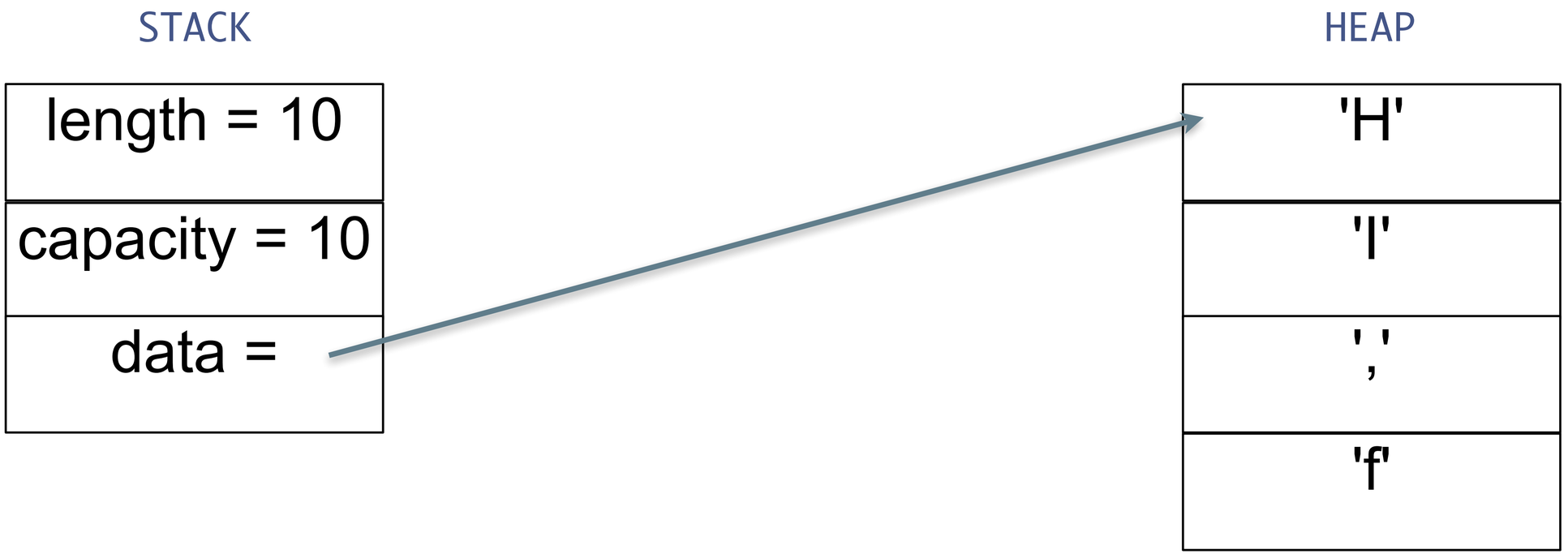
```
let julio = ...
```

```
my_cool_bear_function(Bear: &Bear)
```

# What does ownership look like in memory?

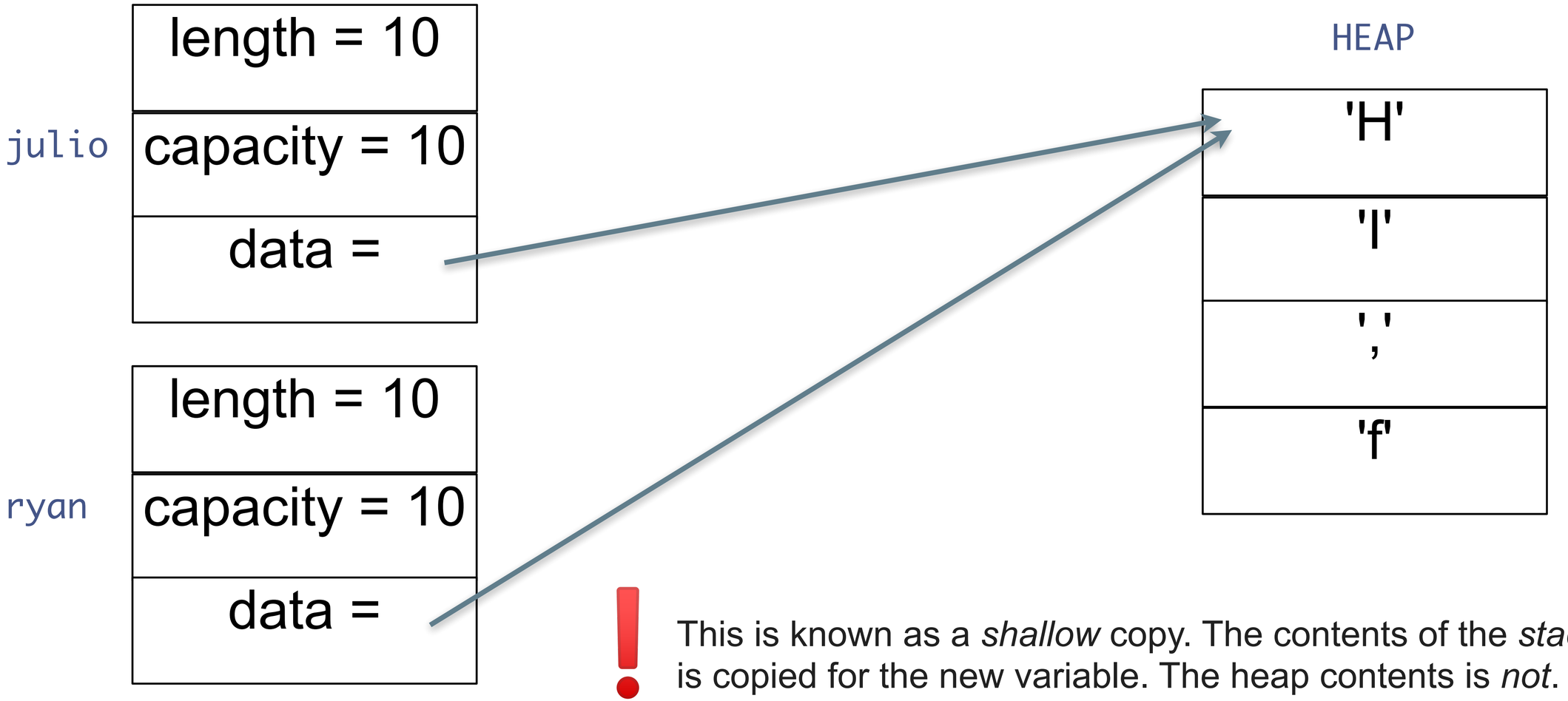
# Ownership in Memory

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



# Ownership in Memory

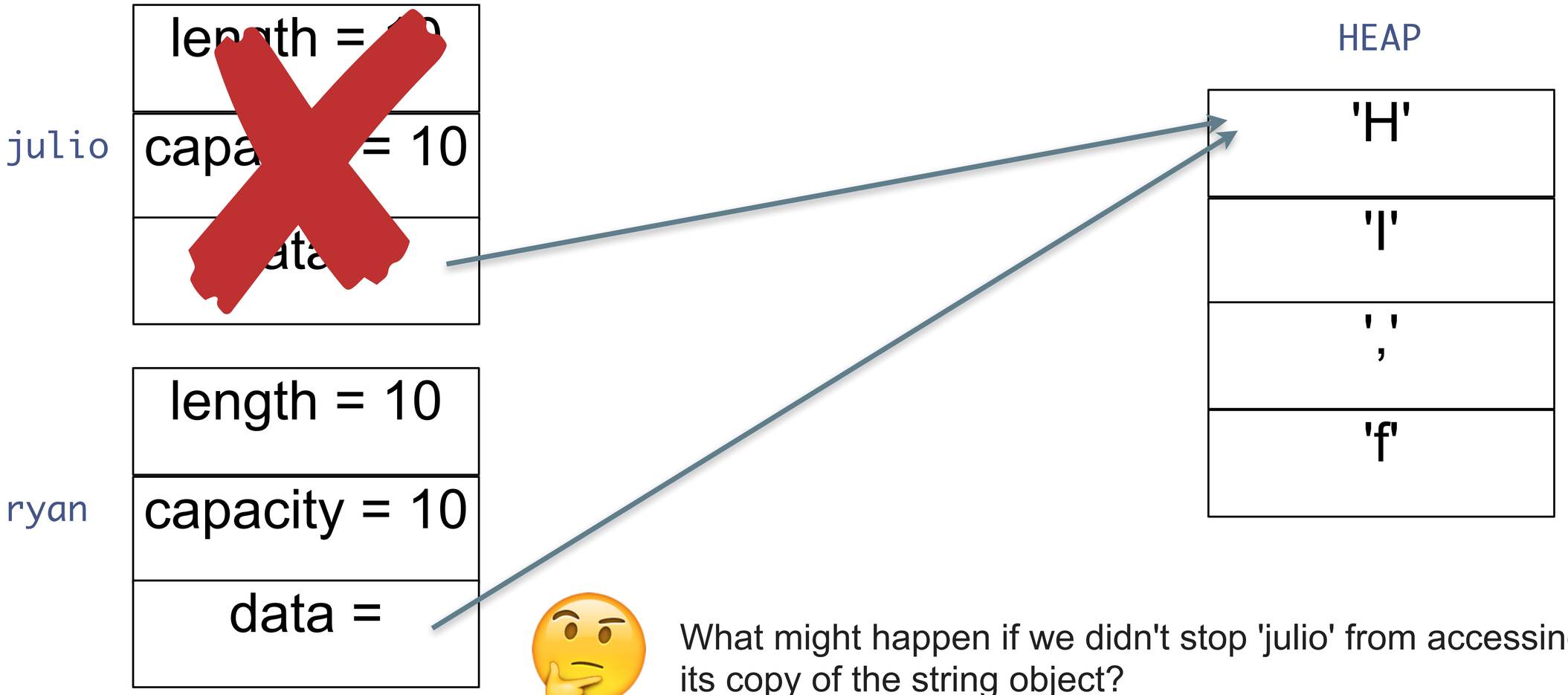
```
let julio = "Hi,friends".to_string();
let ryan = julio;
```




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

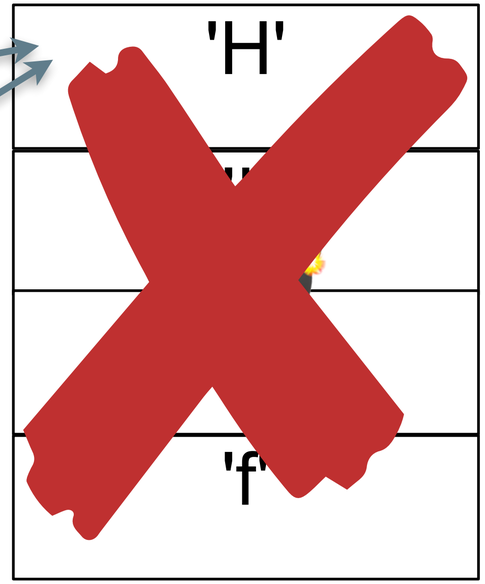
julio

| length = 10 |
| --- |
| capacity = 10 |
| data = |

ryan

| length = 10 |
| --- |
| capacity = 10 |
| data = |

HEAP




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

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 data.
- 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 (create a new object with a copy of the data on the *heap)*, Rust has the *clone* function.

# Clone function

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

julio;

ryan;

Now, julio and ryan have their own heap data!

# Questions?

# Ownership in Memory

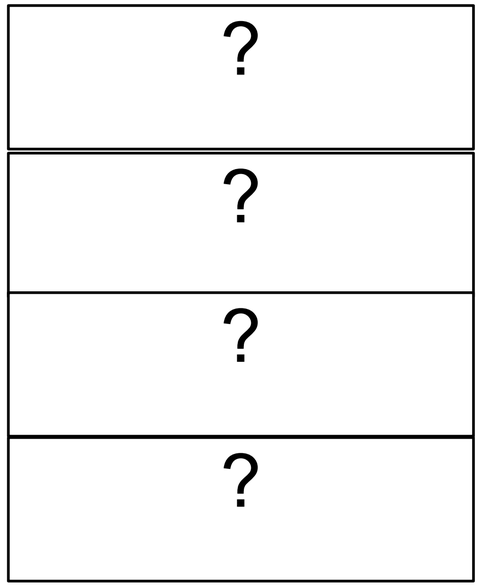
```
let julio = 10;
```

STACK

julio

value = 10

HEAP



# Ownership in Memory

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

STACK

HEAP

julio

value = 10

ryan

value = 10

?

?

?

?

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

# Ownership in Memory

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

STACK

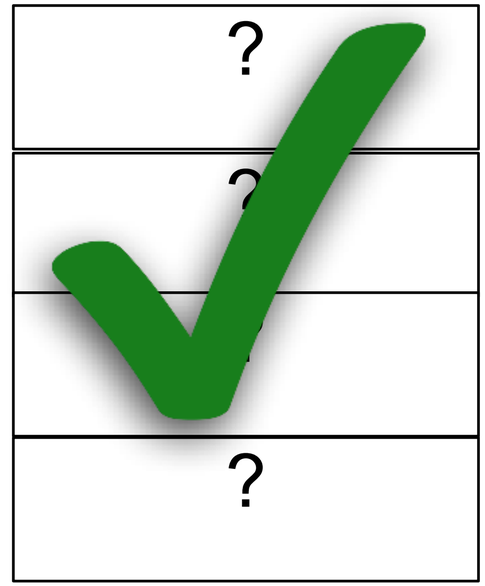
julio

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…)
  - For these types, a “shallow copy” = a full copy
- Objects that only require stack space are typically **copied by default** when assigning variables
  - Types with this property have the **Copy** trait.
    - Instead of transferring ownership, ‘=‘ operator for assignment (e.g., `let ryan = julio`) will create a copy
- If you have the **Copy** trait, Rust won’t let you implement a **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
  |
```

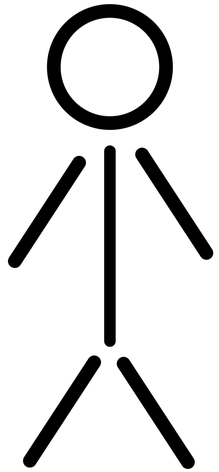
Without the **Copy** trait, Rust assumes ownership is moving!

# Questions?

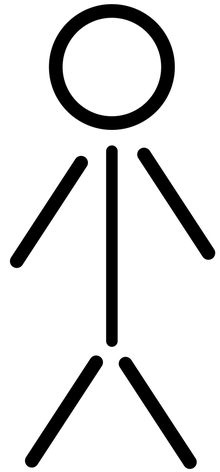
# Borrowing++

# 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 **mut** keyword specifies the data a variable owns to be **mutable.** It's like the opposite *const.*
- The Rust Compiler will *not* compile your code if you change the data owned by any variable that is not declared as mutable.

# 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 these are new variables, 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;

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

```
let mut julio = Bear::get();
let mutable_julio_reference = &mut julio;

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

# Code: Immutable + Mutable References

***does not compile***

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

***compiles!***

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!

# Borrowing + References: The Catch

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

We want both painters to trust that the bear they see won't change while they're trying to paint it!

```
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

- Can have many **immutable** references for a variable in scope at a time.
  - *Think that many painters can paint a picture of the bear, so long as they know no one will change that bear while they're painting.*
- But can only have **one** mutable reference in scope at a time.
  - *Otherwise, the immutable references might see different data than what they initially expected, or two mutable reference's changes might conflict.*

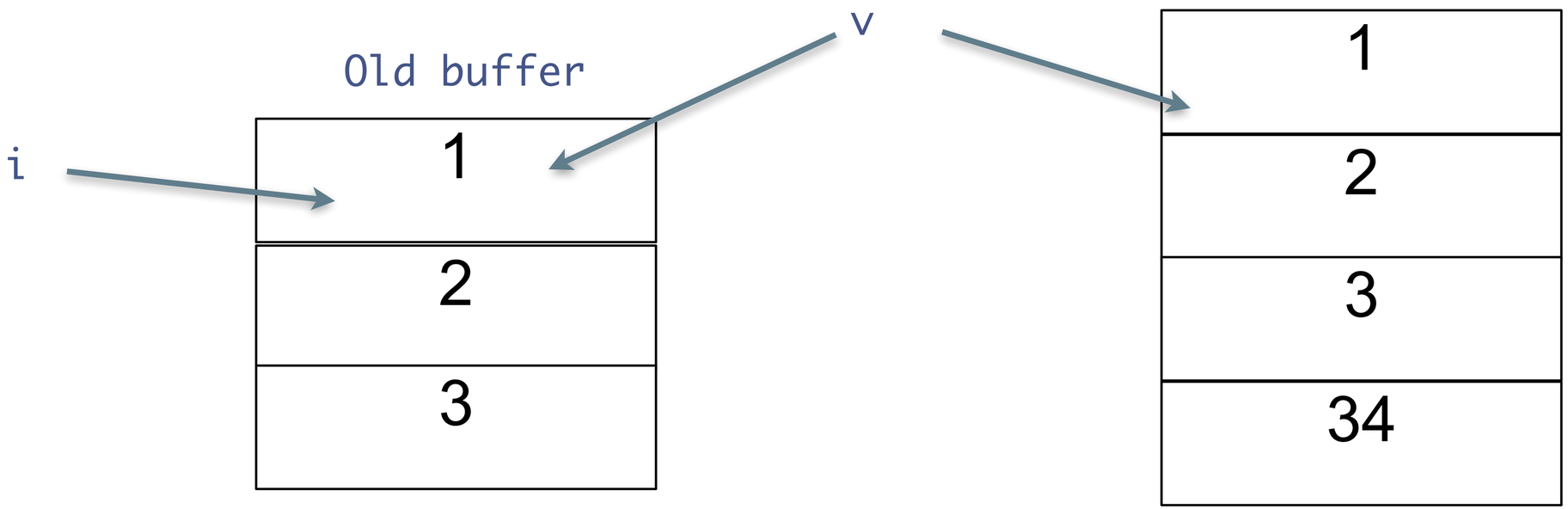
# References Rules

- Can have many **immutable** references for a variable in scope at a time.
- But can only have **one** mutable reference in scope at a time.
- Note: if you create a reference, the original variable is:
  - If reference is mutable: temporarily unusable
  - If reference is immutable: (temporarily) immutable

# 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); /* can cause resize -> moving in memory! */
    }
}
```



# 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 #questions channel!