Rust - Day 4

Ownership

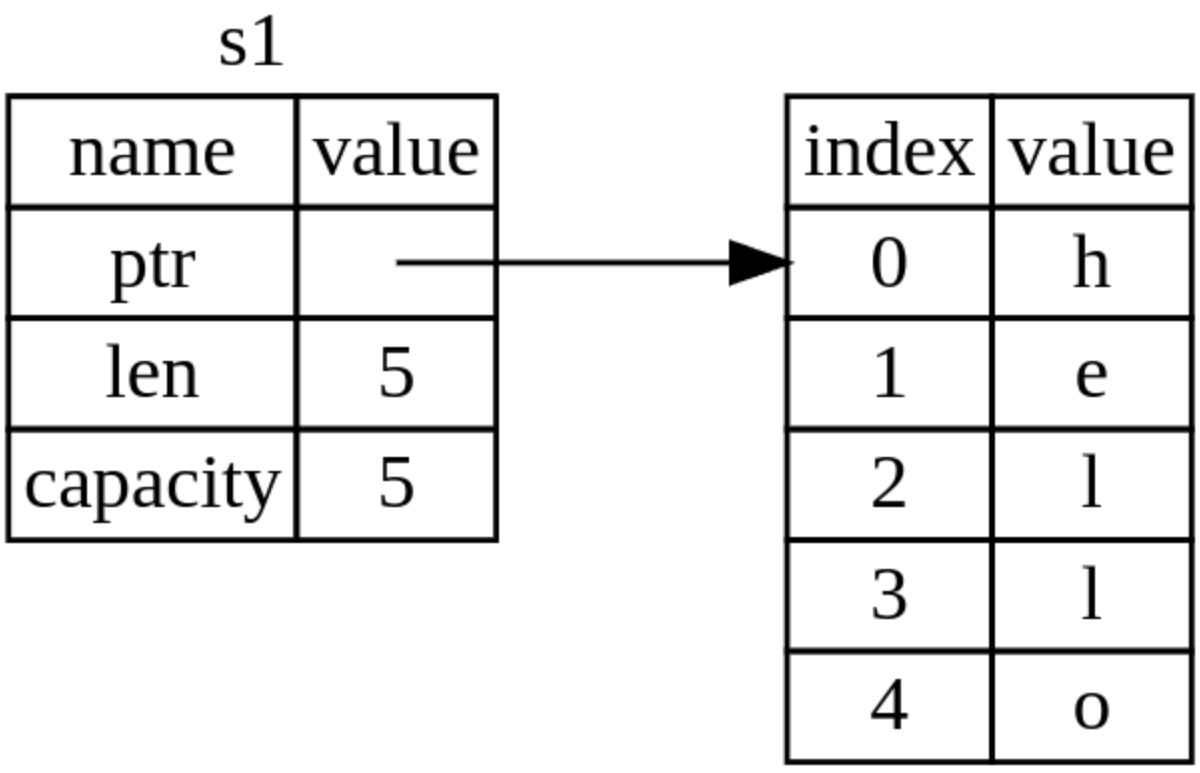
- Ownership is Rust's most unique feature. **It enables Rust to make memory safety guarantees without needing a garbage collector.**
- Topics - Borrowing, Slices and how Rust lays data in the memory.

What is Ownership

- It is a set of rules that govern how a Rust program manages memory.
- All programming languages have to manage the way they use a computer's memory while running programs.
- Some use **Garbage Collectors** that regularly looks for no-longer-used memory and in some languages programmers have to **allocate** and **deallocate** memory.
- **Rust uses a third approach: memory is managed through a system of ownership with a set of rules that the compiler checks.**
- **If any of the rules are violated, the program won't compile. None of the features of ownership will slow down your program while it's running.**

The Stack and the Heap - Read this complete

- Both the stack and the heap are parts of memory available to your code to use at runtime, but they are structured in different ways.
- Stack
  - It stores the data in **LIFO** order with operations like **Pushing onto the stack** and **Popping off the stack**.
  - **All data stored on stack must have a known & fixed size.**
  - **Data with an unknown size at compile time or a size that might change must be stored on the heap instead.**
- Heap
  - It is less organized, when you put data on heap, you request a certain amount of space which is allocated by the memory allocator, after it finds an empty spot in the heap that is big enough, marks it as being in use, and **returns a pointer, which is the address to that location**.
  - This process is called allocating. Pusing values on stack is not considered as allocating. Because the pointer is a know, fixed size, you can store it on the stack, but to get the data, you must follow the pointer.



- Pushing on the stack is faster than allocating on the heap, because allocator never has to search for a place to store new data.
- When your code calls a function, the values passed into the function (including, potentially, pointers to data on the heap) and the function's local variables get pushed onto the stack. When the function is over, those values get popped off the stack.
- **Keeping track of what parts of code are using what data on the heap, minimizing the amount of duplicate data on the heap, and cleaning up unused data on the heap so you don't run out of space are all problems that ownership addresses.**

Ownership rules

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

Variable Scope

- **A scope is the range within a program for which an item is valid**

```
{                      // s is not valid here, it's not yet declared
    let s = "hello";   // s is valid from this point forward

    // do stuff with s
}                      // this scope is now over, and s is no longer valid
```

- Here `s` is a **string literal** with a hardcoded value.
- When `s` comes *into* scope, it is valid.
- It remains valid until it goes *out of* scope.

The String Type

- **string literal and String type are two different types in rust - Read more!!**
- The types covered previously are of a known size, can be stored on the stack and popped off the stack when their scope is over, and can be quickly and trivially copied to make a new, independent instance if another part of code needs to use the same value in a different scope.
- String literals are convenient, but they aren't suitable for every situation. One reason is that **they're immutable**. Another is that **not every string value can be known when we write our code**: for example, **take user input and store it** For these situations, **Rust has a second string type, `String`.**
- This manages data allocated on the heap and is able to store an amount that is unknown at compile time.
- **You can create a `String` from a string literal using the `from` function**

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

Hey everyone, new to Rust and just checking my high level understanding of String vs str.

A `String` type is a container for a `str` that is stored on the heap. String keeps the ownership, `str` is simply a reference and will most commonly be seen as &str.

The heuristic I have in my head is that a String type should only be seen as part of the type that owns it. For example, as part of a struct definition.

When being passed into methods, or returning data from a method, then use an &str type as this will work as a reference to the original ownership.

I imagine it's a little bit more nuanced than that, but is that a good general rule of thumb?

- This kind of string *can* be mutated:

```
let mut s = String::from("hello");
s.push_str(", world!"); // push_str() appends a literal to a String
println!("{s}"); // This will print `hello, world!`
```

Memory and Allocation

- In the case of a string literal, we know the contents at compile time, so the text is hardcoded directly into the final executable. This is why **string literals are fast and efficient.**
- But these properties only come from the **string literal's immutability**.
- We can't put a blob of memory into the binary for each piece of text whose size is unknown at compile time and whose size might change while running the program.
- With the `String` type, in order to support a **mutable**, **growable piece of text**, we need to **allocate an amount of memory on the heap**, **unknown at compile time**, to hold the contents.
  - The memory must be requested from the memory allocator at runtime.
  - We need a way of returning this memory to the allocator when we're done with our `String`.
- **Allocation is done by us, when we call `String::from`. It requets the memory it needs. This behavior is universal in programming languages.**
- **However, the second part is different.**
- **In languages with `Garbage Collector`, it handles the cleaning of unused memory and in languages without `Garbage Collector`, it's our responsibility to identify when memory is no longer being used and explicitely free it.

> Doing this correctly has historically been a difficult programming problem. If we forget, we'll waste memory. If we do it too early, we'll have an invalid variable. If we do it twice, that's a bug too. We need to pair exactly one `allocate` with exactly one `free`.

- 🌟 **Rust takes a different path: the memory is automatically returned once the variable that owns it goes out of scope.**

```
{
    let s = String::from("hello"); // s is valid from this point forward

    // do stuff with s
}                                  // this scope is now over, and s is no
                                   // longer valid
```

- When a variable goes out of scope, Rust calls a special function for us. This function is called **drop**, and it's where the author of `String` can put the code to return the memory. Rust calls **drop** automatically at the closing curly bracket.
- It may seem simple right now, but the **behavior of code can be unexpected in more complicated situations when we want to have multiple variables use the data we've allocated on the heap**. Let's explore some of those situations now.

Variables and Data Interacting with Move

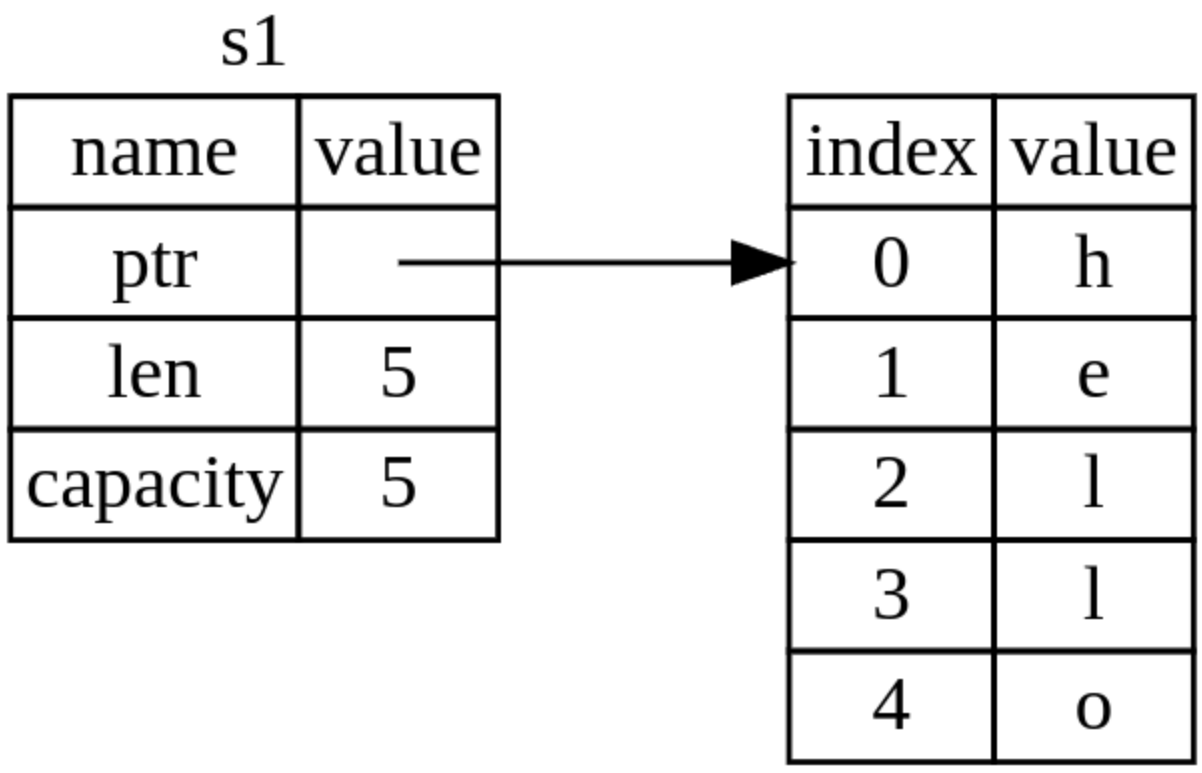
- Multiple variables can interact with the same data in different ways in Rust

```
let x = 5;
let y = x;
```

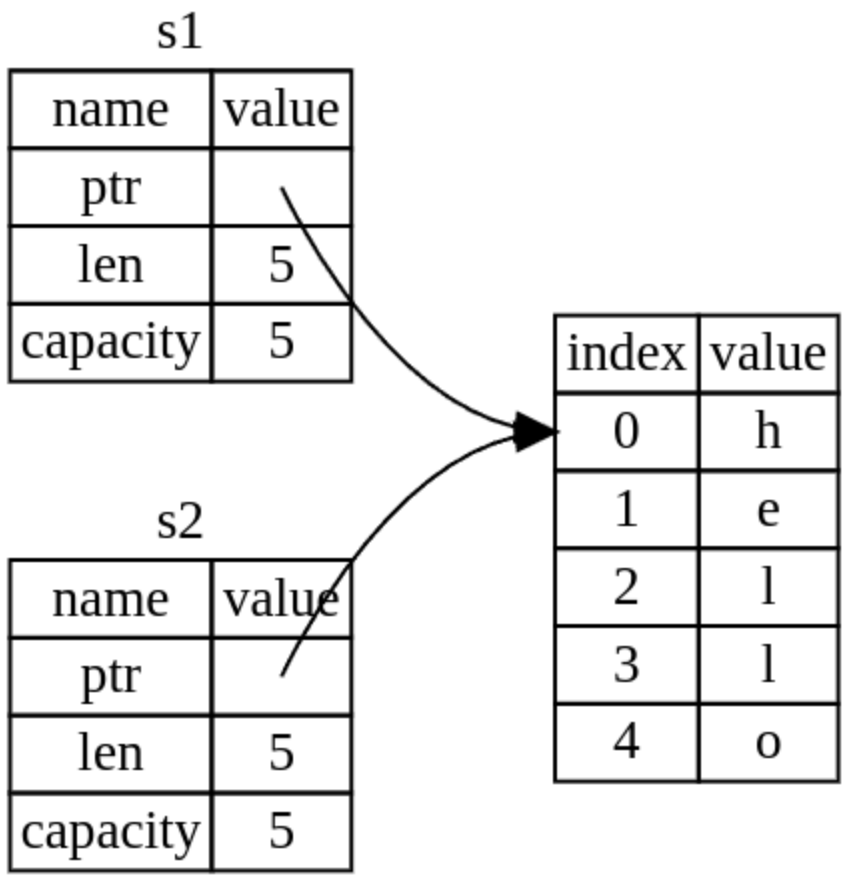
- Here `5` in put into `x` and then a copy of the value of `x` is made and bound to `y`. We now have 2 variables both equal to `5`.
- Integers are simple values with a known, fixed size, and these two `5` values are **pushed onto the stack**.

```
let s1 = String::from("hello");
let s2 = s1;
```

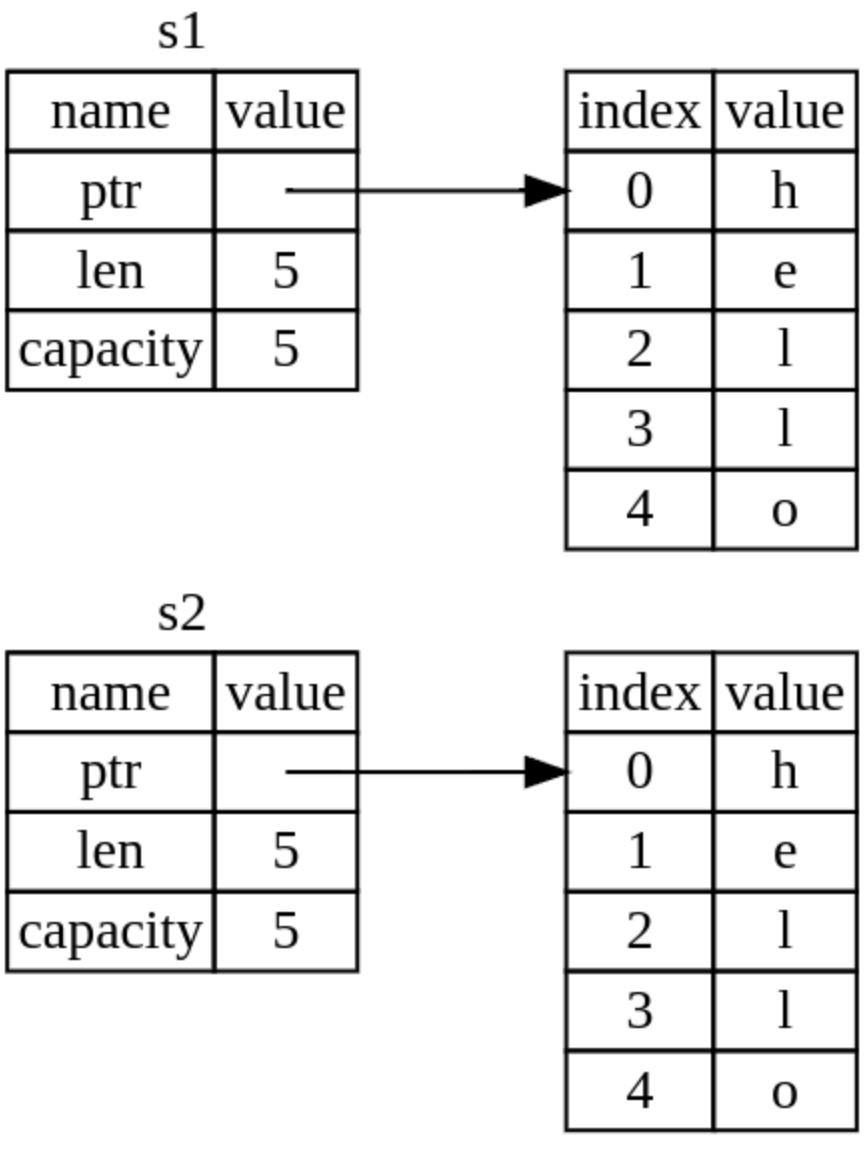
- Now, this seems the same but isn't.
- Under the hood, A `String` is made up of 3 parts:
  - **A pointer to the memory that holds the content of the string**
  - **Length - How much memory, in bytes, the contents of the `String` are currently using.**
  - **Capacity - It is the total amount of memory, in bytes, that the `String` has received from the allocator.**



- **This group of data is stored on the stack.**
- **🌟 When we assign `s1` to `s2`, the `String` data is copied, meaning we copy the pointer, the length, and the capacity that are on the stack. We do not copy the data on the heap that the pointer refers to.**



- If **rust copied the heap data instead**, then the operation `s2=s1` could be **very expensive in terms of `runtime performance`** if the data on the heap is large.



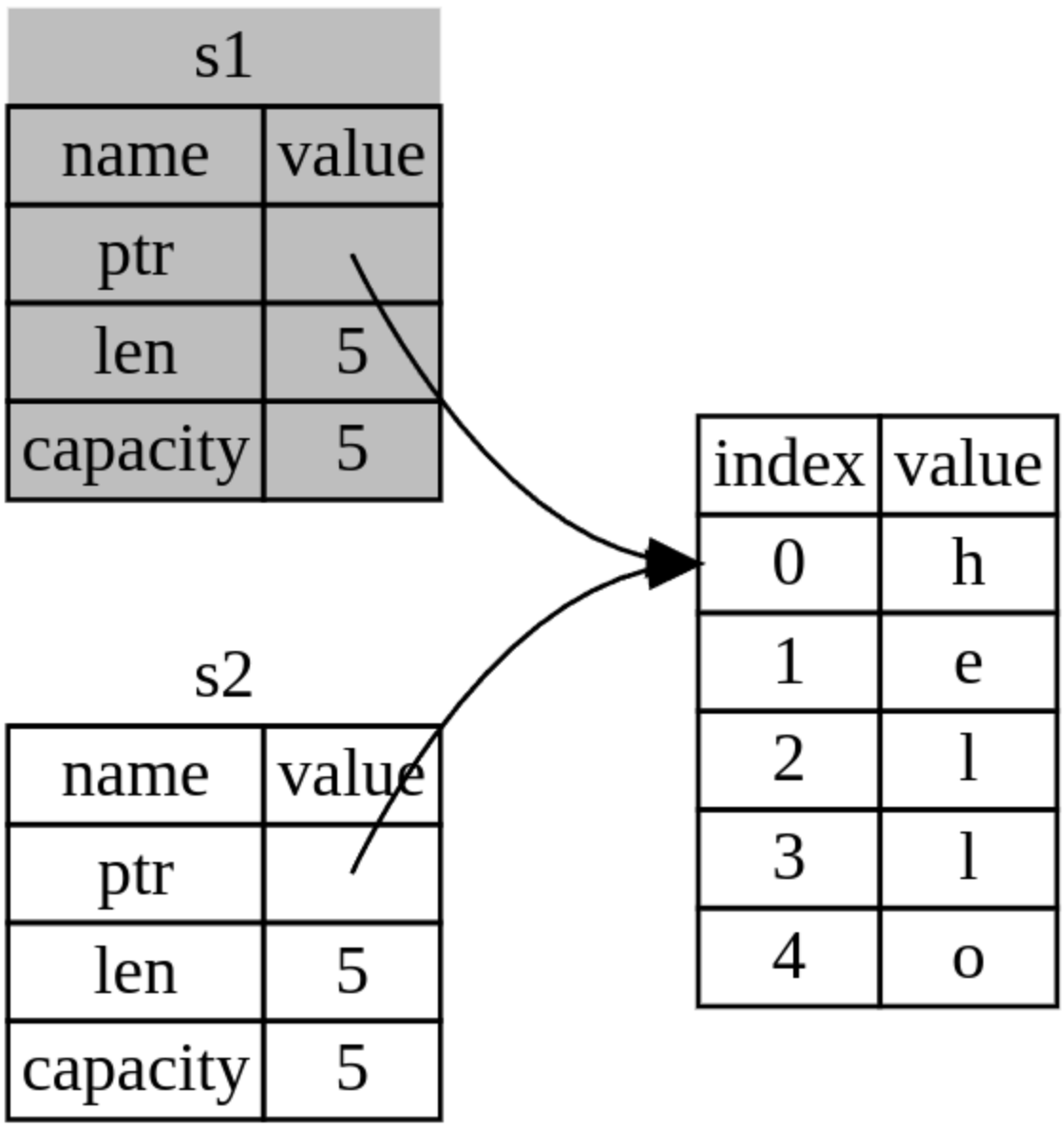
- **We understood that if a variable goes out of scope, Rust automatically calls the `drop` function to clean up the heap memory of that variable.** But if two variables are pointing to the same location, then when `s2` and `s1` will try to free the **same memory**.
- **This is known as a *double free* error and is one of the memory safety bugs we mentioned previously.**
- **🌟 To ensure memory safety, after the line `let s2 = s1;`, Rust considers `s1` as no longer valid. Therefore, Rust doesn't need to free anything when `s1` goes out of scope**

```
  let s1 = String::from("hello");
  let s2 = s1;
  println!("{s1}, world!");
```

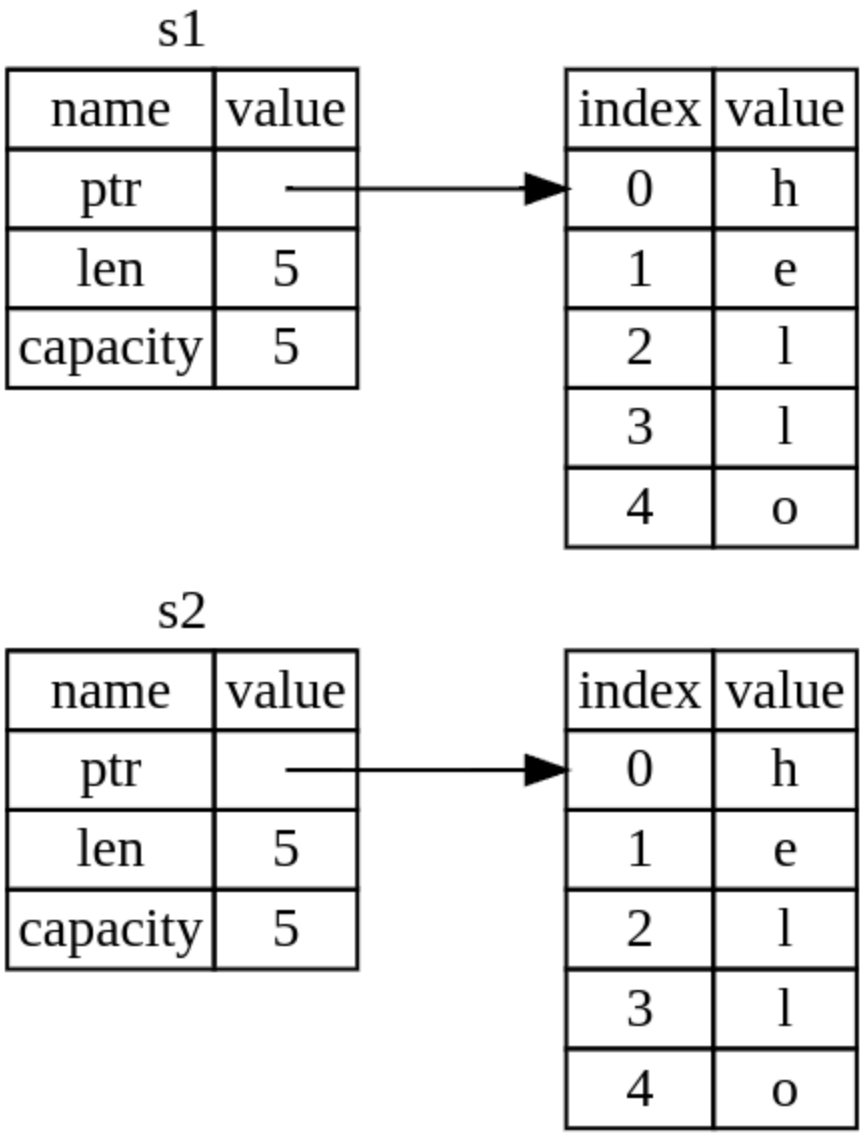
```
$ cargo run
  Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:15
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{s1}, world!");
  |              ^^^^ value borrowed here after move
  |
  = note: this error originates in the macro `$crate::format_args_nl` which comes from the expansion of the macro
`println` (in Nightly builds, run with -Z macro-backtrace for more info)
help: consider cloning the value if the performance cost is acceptable
  |
3 |     let s2 = s1.clone();
  |                ++++++++

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` (bin "ownership") due to 1 previous error
```

- **🌟 The concept of copying the pointer, length, and capacity without copying the data probably sounds like making a shallow copy. But, because Rust also invalidates the first variable, instead of being called a shallow copy, it's known as a *move*.**

- Here, we'd say `s1` moved → into `s2`.



-
- **That solves our problem!** With only `s2` valid, when it goes out of scope it alone will free the memory, and we're done.
- **In addition, there's a design choice that's implied by this: Rust will never automatically create "deep" copies of your data. Therefore, any *automatic* copying can be assumed to be inexpensive in terms of runtime performance.**
- To create deep copy we can use `clone()` method which would work like this:

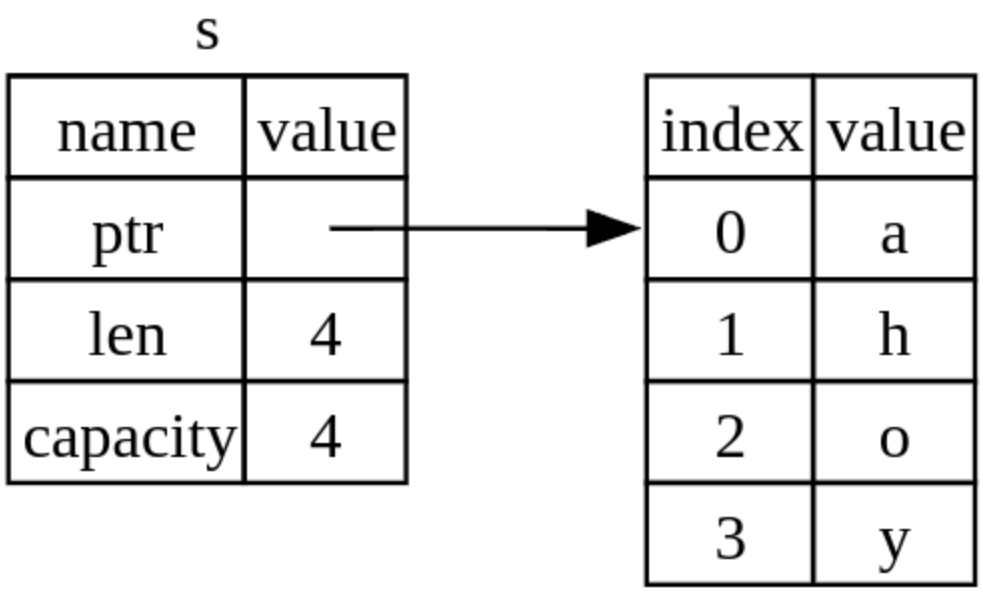


Scope and Assignment

- When you assign a completely new value to an existing variable, Rust will call `drop` and free the original value's memory immediately.

```
let mut s = String::from("hello");
s = String::from("ahoy");
println!("{s}, world!");
```

- We initially declare a variable `s` and bind it to a `String` with the value `"hello"`. Then we immediately create a new `String` with the value `"ahoy"` and assign it to `s`. At this point, nothing is referring to the original value on the heap at all.



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

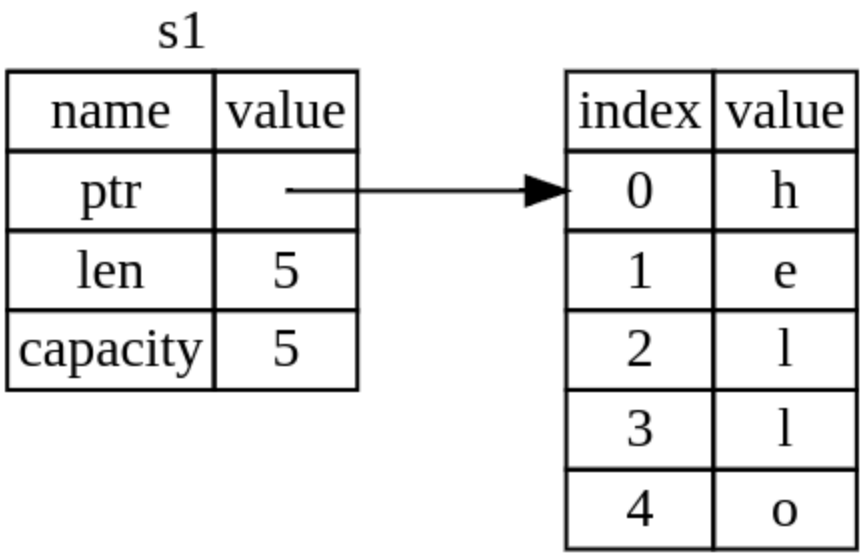
- The original string thus immediately goes out of scope. Rust will run the `drop` function on it and its memory will be freed right away. When we print the value at the end, it will be `"ahoy, world!"`.

Variables and Data Interacting with Clone

- If we *do* want to deeply copy the heap data of the `String`, not just the stack data, we can use a common method called `clone`.

```
let s1 = String::from("hello");
let s2 = s1.clone();
println!("s1 = {s1}, s2 = {s2}");
```

- This works just fine and explicitly produces the behavior shown below, where the heap data *does* get copied.



- 🌟 When you see a call to `clone`, you know that some arbitrary code is being executed and that code may be expensive. It's a visual indicator that something different is going on.

Stack-Only Data: Copy

```
let x = 5;
let y = x;
println!("x = {x}, y = {y}");
```

- This code copies without having to call `clone` with `x` still being valid and not being moved into `y`
- **This is because, types such as integers, have a known size at compile time and hence are stored entirely on the stack, so copies of the actual values are quick to make.**

Copy Trait

- Rust has a special annotation called the `Copy` trait that we can place on types that are stored on the stack, as integers are.
- If a type implements the `Copy` trait, variables that use it do not move, but rather are trivially copied, making them still valid after assignment to another variable.
- Rust won't let us annotate a type with `Copy` if the type, or any of its parts, has implemented the `Drop` trait.
- **Any group of simple scalar values can implement `Copy`, and nothing that requires allocation or is some form of resource can implement `Copy`.**
  - All the integer types, such as `u32`.
  - The Boolean type, `bool`, with values `true` and `false`.
  - All the floating-point types, such as `f64`.
  - The character type, `char`.
  - Tuples, if they only contain types that also implement `Copy`. For example, `(i32, i32)` implements `Copy`, but `(i32, String)` does not.

Ownership and Functions

- The mechanics of passing a value to a function are similar to those when assigning a value to a variable.

- 🌟 **Passing a variable to a function will move or copy, just as assignment does.** Unlike in other languages where the reference is directly passed.

```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward

} // Here, x goes out of scope, then s. But because s's value was moved, nothing
  // special happens.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{some_string}");
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{some_integer}");
} // Here, some_integer goes out of scope. Nothing special happens.
```

- **String `s` ownershp is transfered to the `takes_ownership` function and must be returned to reuse in the main function**. If not returned then it will be dropped when the scope of `takes_ownership` ends.

Returning Values and Scope

- Returning values can also transfer ownership.

```
fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                            // return value into the function
                                            // that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                              // some_string is returned and
                                            // moves out to the calling
                                            // function
}

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
                                                      // scope

    a_string  // a_string is returned and moves out to the calling function
}
```

- 🌟 **When a variable that includes data on the heap goes out of scope, the value will be cleaned up by `drop` unless ownership of the data has been moved to another variable.**
- While this works, taking ownership and then returning ownership with every function is a bit tedious.
- What if we want to let a function use a value but not take ownership? It's quite annoying that anything we pass in also needs to be passed back if we want to use it again, in addition to any data resulting from the body of the function that we might want to return as well.
- **Rust does let us return multiple values using a tuple**

```
fn main() {
    let s1 = String::from("hello");
    let (s2, len) = calculate_length(s1);
    println!("The length of '{s2}' is {len}.");
}

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String
    (s, length)
}
```

> **Return Statement in RUST**
>
> Is it bad practice to omit return statements in Rust? **No, it's encouraged**. Rust is intended to be an expression-based language. In particular the last line of your function can be an expression rather than a statement, which is understood to be the value that the function returns.

- Rust has a feature for using a value without transferring ownership, called *references*.