Ownership in Rust

Ownership is Rust's **most unique** feature:
it has deep implications for the language.

It enables to make memory safety guarantees **without needing a garbage collector**.

Ownership in Rust

Ownership is Rust's **most unique** feature:
it has deep implications for the language.

It enables to make memory safety guarantees **without needing a garbage collector**.

It's important to understand how it works.

Memory handling

All languages must manage how they use memory while running.

Memory handling

All languages must manage how they use memory while running.

Approach 1: Having a **garbage collector** that regularly looks for no longer used memory as the program runs.

Memory handling

All languages must manage how they use memory while running.

Approach 1: Having a **garbage collector** that regularly looks for no longer used memory as the program runs.

Approach 2: Explicitly **allocate/free memory (C, C++, Assembly, ...)**

Memory handling

All languages must manage how they use memory while running.

Approach 1: Having a **garbage collector** that regularly looks for no longer used memory as the program runs.

Approach 2: Explicitly **allocate/free memory (C, C++, Assembly, ...)**

Rust is different!
Memory is managed through a **set of rules** that the compiler checks. If any rule is violated, the program won't compile.

Memory handling

All languages must manage how they use memory while running.

Approach 1: Having a **garbage collector** that regularly looks for no longer used memory as the program runs.

Approach 2: Explicitly **allocate/free memory (C, C++, Assembly, ...)**

Rust is different!
Memory is managed through a **set of rules** that the compiler checks. If any rule is violated, the program won't compile.

Ownership will **not** slow down your running program.

Ownership: a new concept

Ownership is a new concept for many programmers: it might take some time to get used to it.

It’s a way to develop code that is safe and efficient naturally.

We will focus on examples using a specific data structure: **Strings**.

string literals

main.rs

```rust
fn main() {
    {
        // 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    } // this scope is now over, and s is no longer valid
}
```

Strings

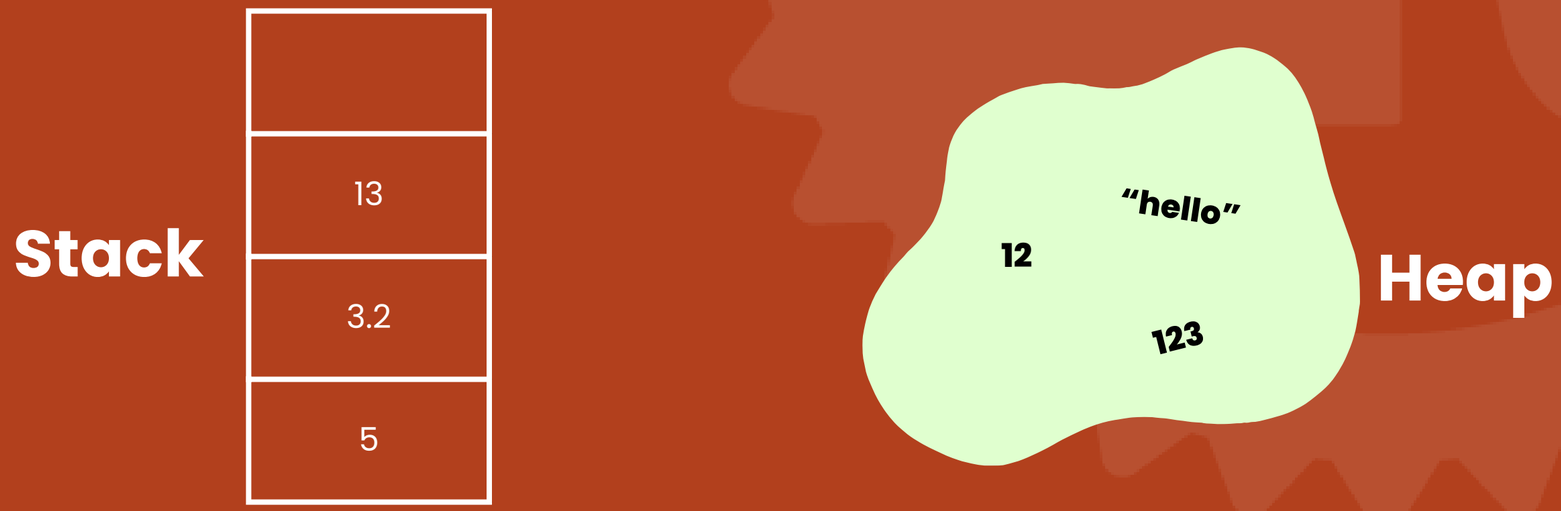
```rust
// 5. Ways Variables and Data Interact: Clone
{
    let s1: String = String::from("hello");
    let s2: String = s1.clone();

    println!("s1 = {}, s2 = {}", s1, s2);
}
```

The Stack and the Heap

Many languages don’t require you to think about the **Stack** and **Heap** often.

In Rust (systems language), whether a value is on the stack or the heap affects how the language behaves and why you must make certain decisions.

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.



The Stack

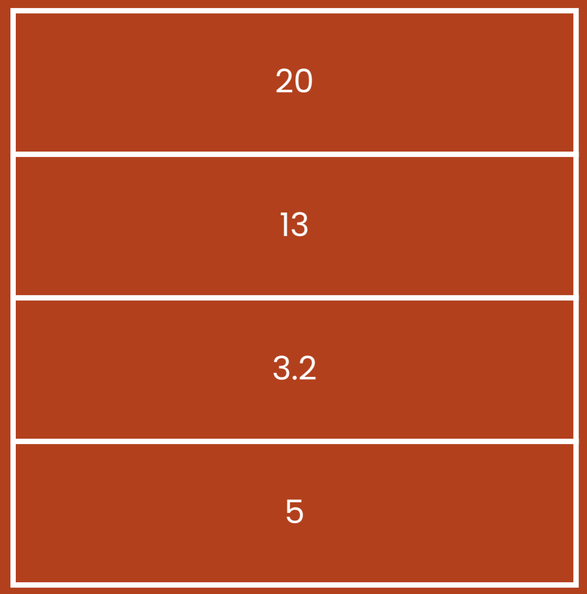
The stack stores values in the order it gets them and removes the values in the opposite order (LIFO).

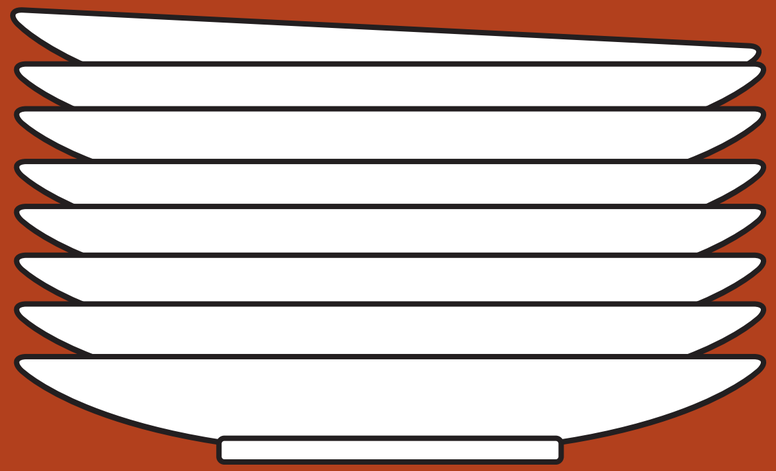
- **pushing:** add onto the stack.
- **popping:** removing data off the stack.

All data stored on the stack must have:

- known size.
- fixed size.

Data with an unknown size at compile time or size that might change **can't be stored on the Stack**. They use the **Heap** instead.





The stack is like a pile of plates.

The Heap

Heap: less organized: when you put data on the heap, you request some space.

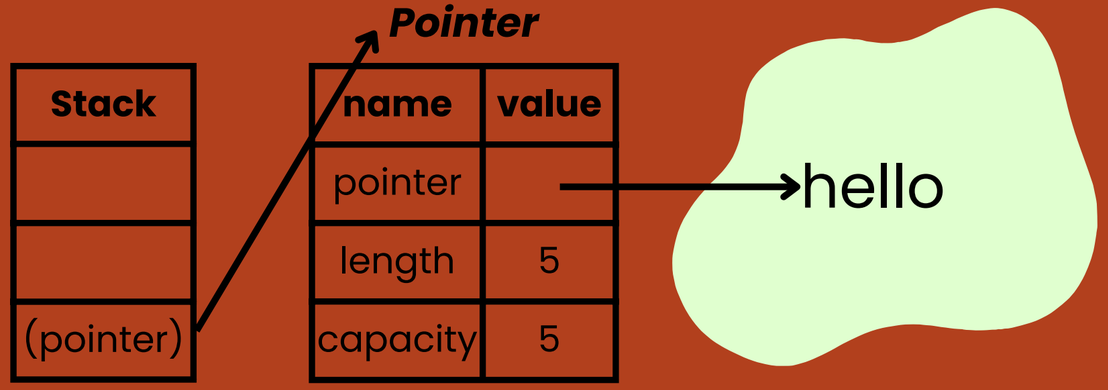
The memory allocator:

1. finds a big enough empty spot in the heap.
2. marks it as being in use.
3. returns a pointer: the location's address.

This process is called **allocating**.

You can **store** the pointer on the stack.

To get the data, **follow the pointer**.



The heap is like a Restaurant

The Stack

Fixed size, known at compile time

What goes on the **STACK:**

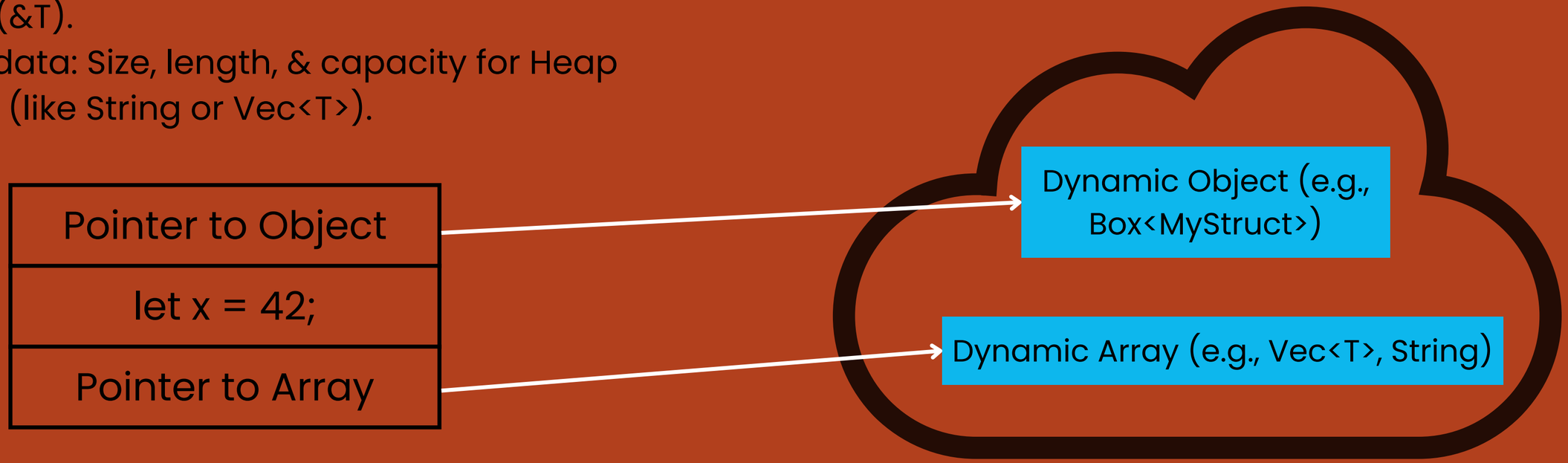
- **Primitive Values:**
  - Integers, Floats, Booleans, Characters.
- **Fixed Collections:**
  - Arrays
  - Tuples (if their contents are Stack-allocated).
- **Memory Management:**
  - References & Pointers: The memory address itself (&T).
  - Metadata: Size, length, & capacity for Heap types (like String or Vec<T>).

The Heap

Variable size, known at runtime. Managed by Ownership.

What goes on the **HEAP:**

- **Dynamic Collections:**
  - Strings: The actual text data (the bytes of "hello world").
  - Vectors (Vec<T>): The list of elements can grow or shrink.
  - HashMaps, BTreeMaps, LinkedLists, etc.
- **Boxed values** (using the smart pointer <Box>)



Pushing/Allocating

Pushing to the stack is fast: the allocator never has to search for a place to store new data, as it's always at the top of the stack.

Allocating space on the heap is harder: the allocator must first find a big enough space to hold the data, then perform bookkeeping to prepare for the next allocation.

Accessing data

Accessing data in the heap is slow: you have to follow a pointer

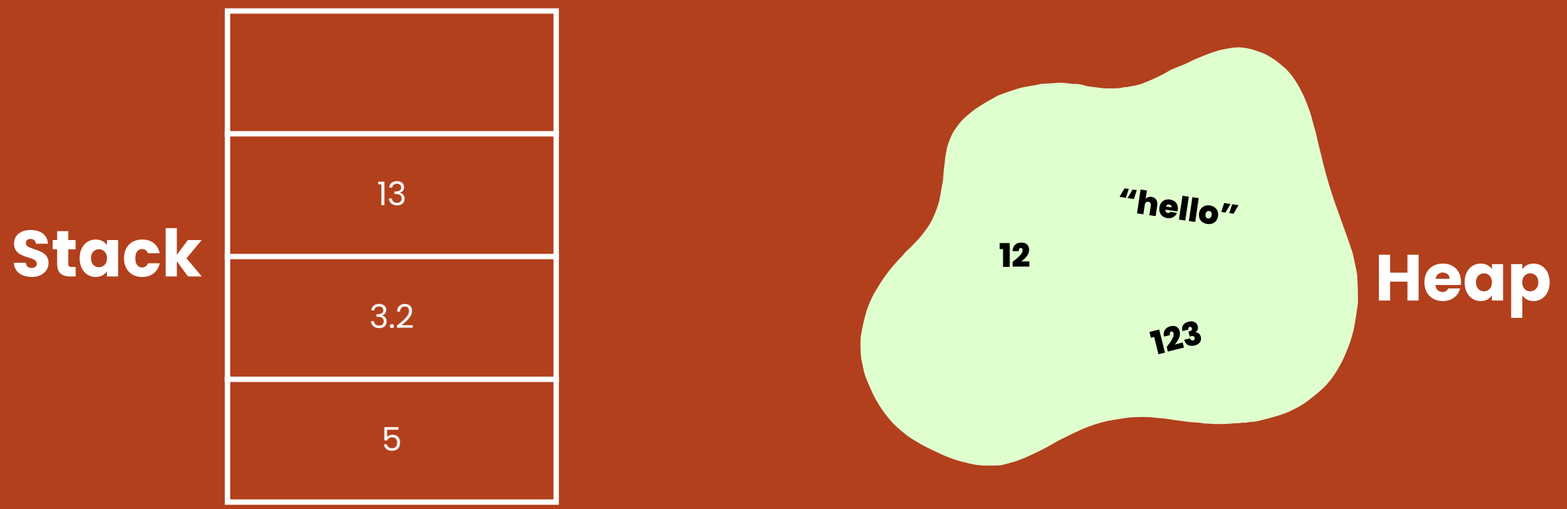
In a restaurant, getting all the orders at one table before moving on to the next is most efficient.

A processor can do its job **better** if it works on **data close to other data** (stack) **rather than far away** (heap).

Functions

When you call a function, the **values are passed into the function** (including pointers to data on the heap), and **the function's local variables get pushed onto the stack**.

When the function is over, values get **popped off** the stack.



Ownership's purpose

The **primary purpose** of ownership is to **manage heap data**.

- Keeping track of **what code** is using **what data** on the **heap**.
- **Minimizing** the amount of **duplicate data** on the **heap**.
- **Cleaning up** unused data on the **heap.**

Heap

Ownership Rules

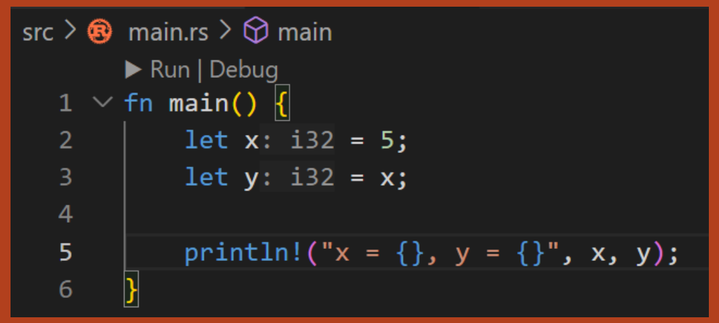
1. **Each value** has an **owner**.

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

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

DEMO

COPY Stack-Only Data



it's valid without clone!

Types such as integers (known size at compile time) are **stored entirely on the stack**: copies of the values are quick to make. No reason to prevent x from being valid after we create y. No need to call clone in this case.

Rust has a special annotation (**Copy** trait) for types stored on the stack: When we use the "=", it doesn't move, but it's simply copied, and the old one is still valid.

If a type has the **Drop** trait (e.g **String**), you can't add the **Copy** trait.

COPY Who can implement it

General rule: any group of simple scalar values can implement Copy.

Nothing that requires allocation or is some form of resource can implement Copy.

Some types that 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.

MOVE

a String holding the value “hello” bound to s1

s1

name	value
pointer	→
length	5
capacity	5

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

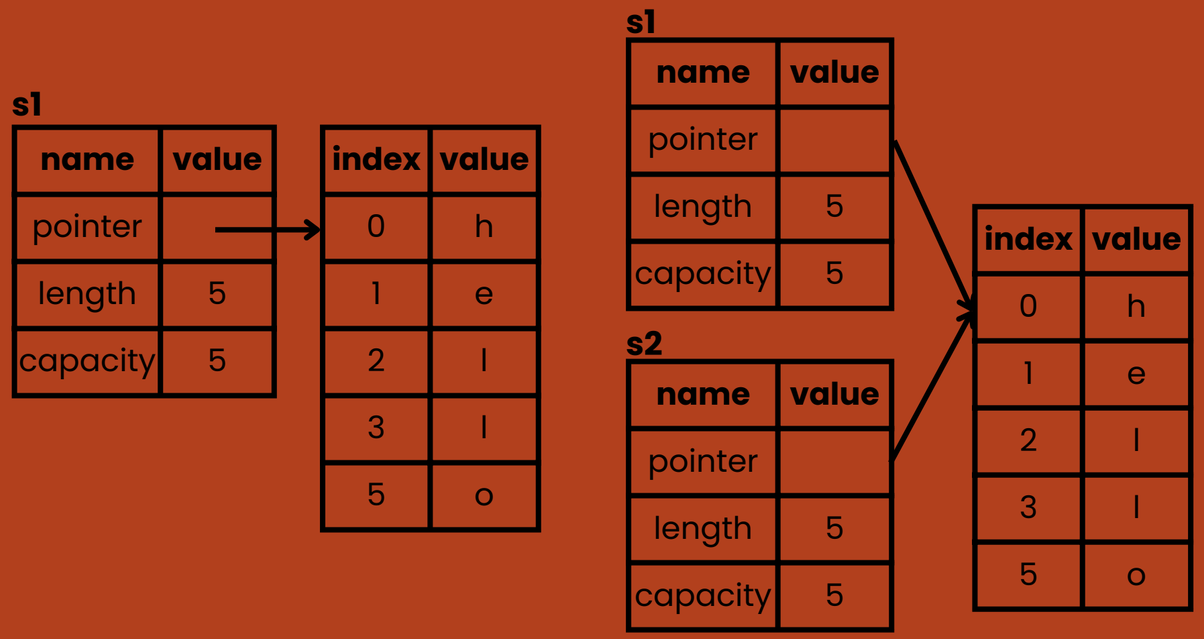
src > main.rs

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

MOVE

a String holding the value “hello” bound to s1

s2 has a copy to s1's pointer, length and capacity



src > main.rs

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

MOVE

a String holding the value “hello” bound to s1

s1

name	value
pointer	
length	5
capacity	5

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

s2 has a copy to s1's pointer, length and capacity

s1

name	value
pointer	
length	5
capacity	5

s2

name	value
pointer	
length	5
capacity	5

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

If Rust copied the heap data when using = ...

s1

name	value
pointer	
length	5
capacity	5

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

s2

name	value
pointer	
length	5
capacity	5

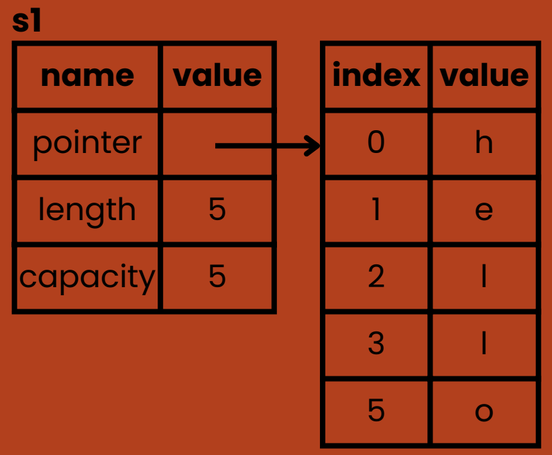
index	value
0	h
1	e
2	l
3	l
5	o

src > main.rs

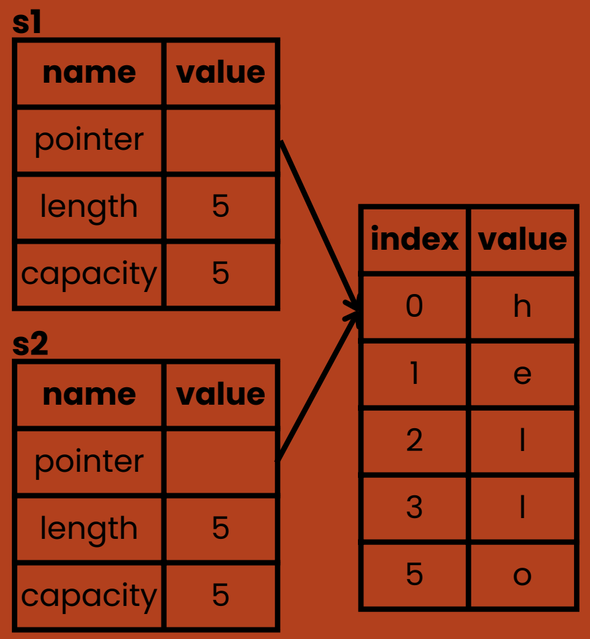
```rust
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
    println!("{}, world!", s1);
}
```

MOVE

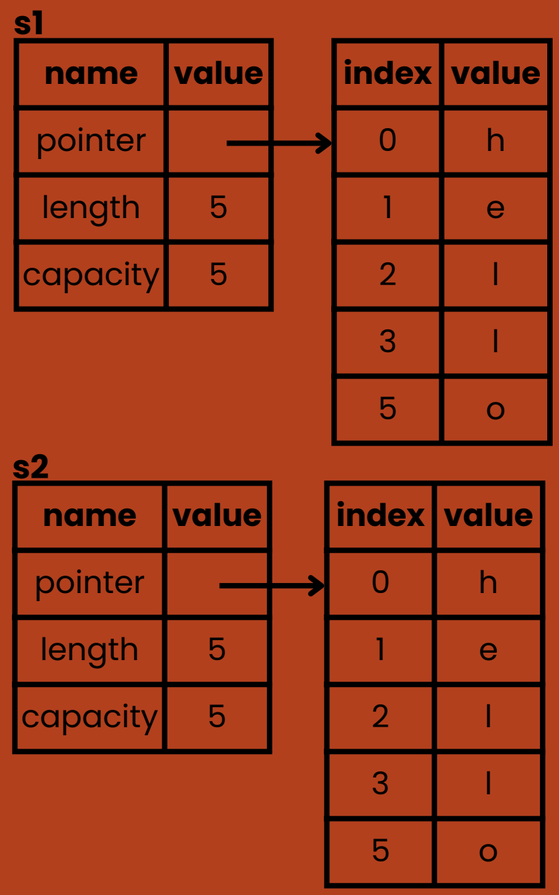
a String holding the value “hello” bound to s1



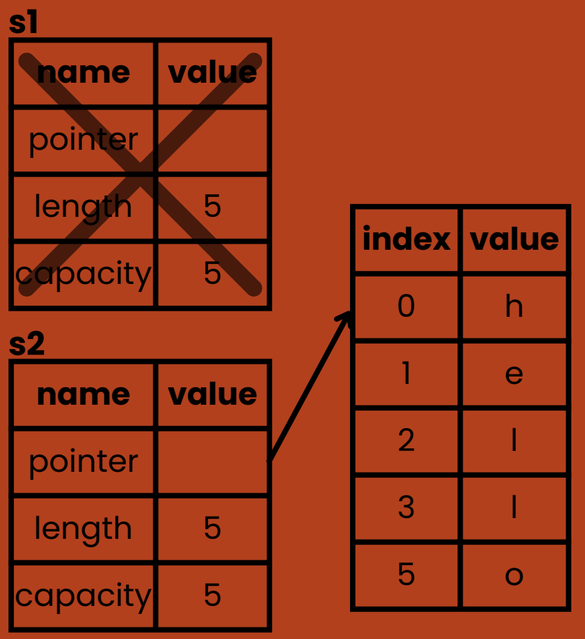
s2 has a copy to s1's pointer, length and capacity



If Rust copied the heap data when using = ...



memory representation after s1 has been invalidated



src > main.rs

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

Ownership Rules

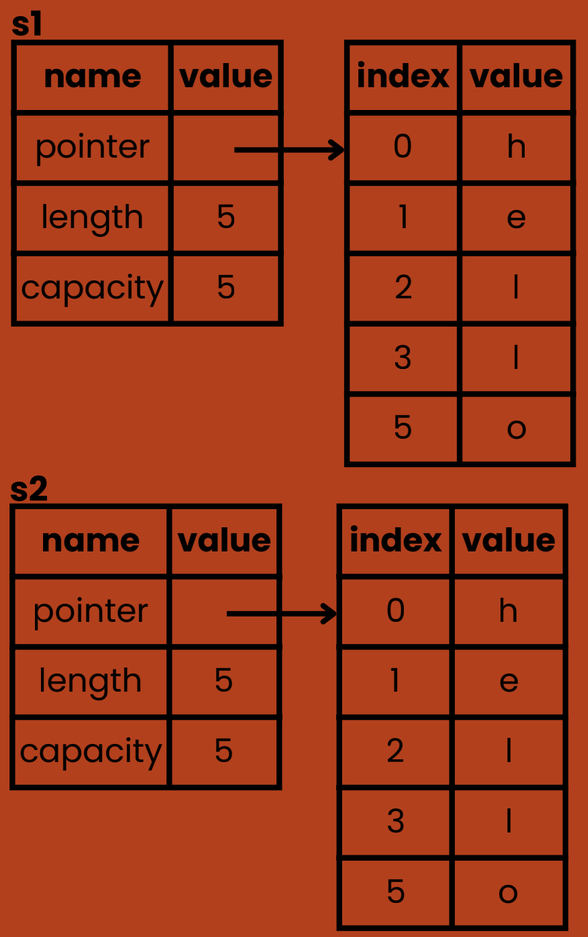
1. **Each value** has an **owner**.

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

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

CLONE

```
src > main.rs > ...
    ▶ Run | Debug
1   fn main() {
2       let s1: String = String::from("hello");
3       let s2: String = s1.clone();
4
5       println!("s1 = {}, s2 = {}", s1, s2);
6   }
7
```



Clone: To deep copy the heap data of the String, not just the stack data

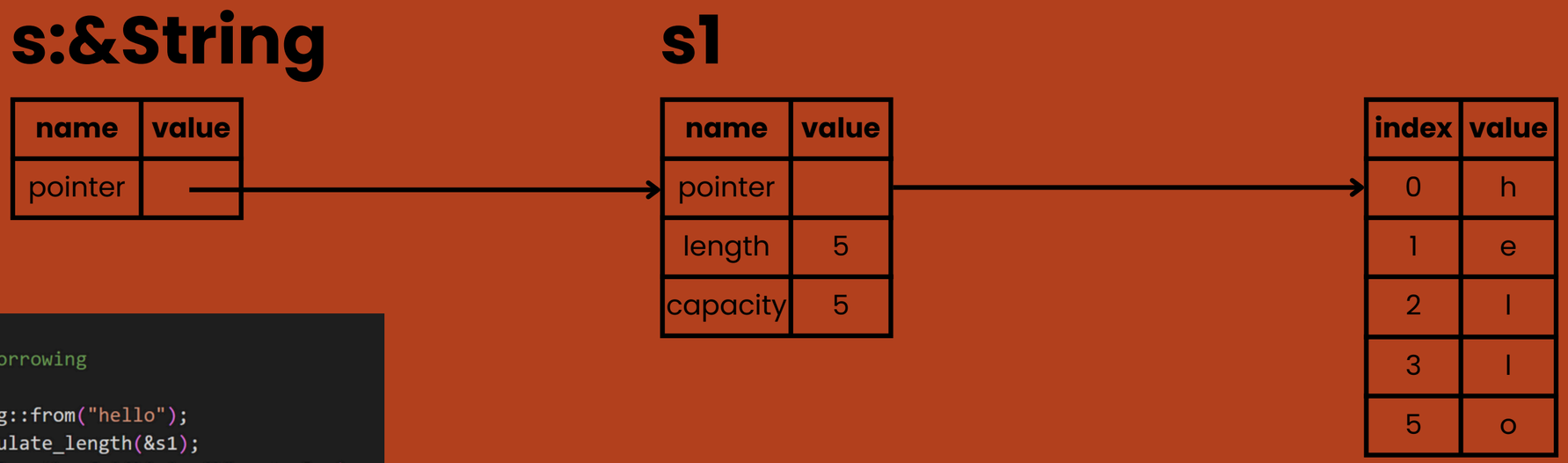
This works fine and explicitly produces the behavior we see on the right: **the heap data gets copied**.

When you see a call to clone, you know that some arbitrary code is being executed: **that code may be expensive**.

Borrowing & References

REFERENCES

&String = reference: It allows to refer to some value without taking Ownership



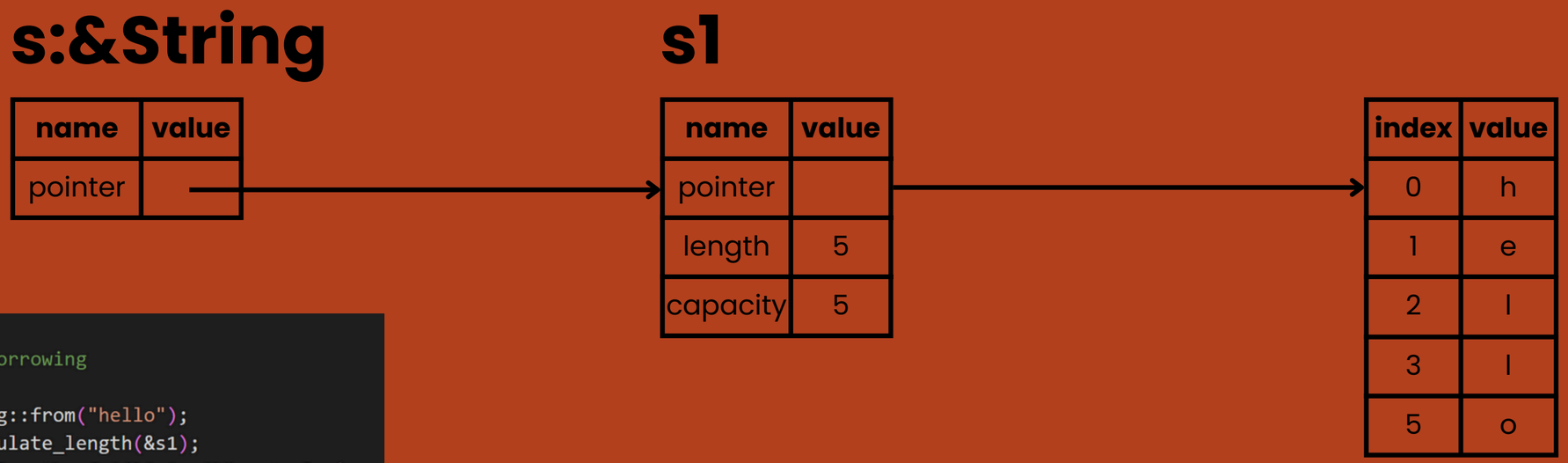
src > main.rs

```rust
//References and Borrowing
fn main(){
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}", s1, len);
}

fn calculate_length(s: &String) -> usize{
    let length = s.len();
    length
}
```

&s1 = creates a reference to s1, but it doesn't own it
s: &String = the parameter s is a reference

REFERENCES

&String = reference: It allows to refer to some value without taking Ownership



src > main.rs

```rust
//References and Borrowing
fn main(){
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}", s1, len);
}

fn calculate_length(s: &String) -> usize{
    let length = s.len();
    length
}
```

&s1 = creates a reference to s1, but it doesn't own it
s: &String = the parameter s is a reference

The Action of creating a reference is called BORROWING

REFERENCES RULES

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

```rust
fn main() {
    let mut greeting = String::from("Hello");

    // Example with immutable references
    let greet_ref1 = &greeting; // Works: Immutable reference
    let greet_ref2 = &greeting; // Works: Another immutable reference
    println!("Immutable references: {} and {}", greet_ref1, greet_ref2);

    // Attempt to create a mutable reference after immutable ones
    // Does NOT work: Cannot borrow as mutable because it is also borrowed as immutable
    // let greet_mut_ref = &mut greeting;

    // Immutable references go out of use here
    // Now we can safely create a mutable reference
    // For demo: mutable reference is created in a new scope to avoid errors
    {
        // Since greet_ref1 and greet_ref2 are no longer used, this works
        // Works: Mutable reference, after immutable ones are no longer in use
        let greet_mut_ref = &mut greeting;
        greet_mut_ref.push_str(", world!");
        println!("Mutable reference: {}", greet_mut_ref);
    }

    // Note: If you uncomment the line that attempts to create a mutable reference
    // while immutable ones are active, the compiler will enforce the borrowing rules
    // and prevent the code from compiling.
}
```

2. References must always be valid.

src > main.rs

```rust
fn main() {
    // Example of a potentially invalid reference
    let reference_to_nothing;

    {
        let some_number = 42;
        reference_to_nothing = &some_number;
        // This will not compile because `some_number` goes out of scope here,
        // and `reference_to_nothing` would be a dangling reference.
        // println!("Reference: {}", reference_to_nothing);
    }

    // Correct usage ensuring references are valid
    let some_number = 42; // `some_number` is declared in the outer scope
    let reference_to_something = &some_number; // Valid ref: `some_number` in scope

    println!("Valid reference: {}", reference_to_something);
    // This works: `reference_to_something` remains valid throughout its use.
}
```