Day-2

Ownership

What is Ownership?

Ownership is Rust's most unique feature, and it enables Rust to manage memory safely without a garbage collector. Once you understand ownership, the rest of Rust becomes much easier.

Rust 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.

Ownership Rules

1. Each value in Rust has a variable that's its owner.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value is dropped.

Plain Text

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // ownership moves here

    // println!("{}", s1); // Error: value borrowed here after move
    println!("{}", s2); // works fine
}


```

Plain Text

```
Before move:
s1 ——> "hello"

After move:
s1      (invalid)
s2 ——> "hello"

s1 is no longer valid after the move.

Rust does not do a shallow copy; it prevents double-free errors by invalidating the original
owner.
```

Ownership and the Stack

Rust uses the stack to store data types whose size is known and doesn't change (i.e., Copy types). When such a variable goes out of scope, it's popped off the stack.

Data stored on the stack:

- Scalar types: `i32`, `f64`, `char`, `bool`
- Tuples with only stack data: `(i32, bool)`
- Arrays with fixed size: `[i32; 3]`

These are known as **Copy types**, meaning they don't move ownership — they are simply duplicated.

Plain Text

```
fn main() {
    let x = 5;
    let y = x; // x is copied, not moved
```

```
    println!("x: {}, y: {}", x, y); // Both are valid
}



Stack Behavior Diagram

Stack:
+-------+     <--- top
|   y   |  ← copy of x (value: 5)
+-------+
|   x   |  (value: 5)
+-------+
```

Ownership and the Heap

In Rust, heap-allocated data is managed by smart pointers (like `String`, `Vec`, `Box<T>`, etc.) stored on the stack. The actual data they point to lives on the heap.

When the owner goes out of scope, Rust automatically deallocates the heap memory (thanks to ownership rules)

Common Types That Use the Heap:

	Data Type	Stored On	Description
1	`String`	Heap	Growable, UTF-8 text
2	`Vec<T>`	Heap	Growable array/vector
3	`Box<T>`	Heap	Smart pointer for heap allocation
4	`HashMap<K, V>`	Heap	Key-value pairs, dynamic size
5	`&str` (slice)	Heap*	Reference to string, often pointing to heap data

Note: The pointer + length + capacity are stored on the stack, while the actual contents live on the heap.

Example with String (Heap)

Plain Text

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // Ownership is moved

    // println!("{}", s1); // Error: s1 moved
    println!("{}", s2);
}

Diagram: Stack vs Heap in Ownership
         Stack                     Heap
+---------------------+     +----------------------+
| s1 (pointer)        | --> |   h  e  l  l  o      |
| length: 5           |     +----------------------+
| capacity: 5         |
+---------------------+


-----------------------------------------------------------
```

```
         Stack
+---------------------+
| s1 (pointer)        |
| length: 5           |
| capacity: 5         |
+---------------------+


         Stack                      Heap
+---------------------+      +----------------------+
| s2 (pointer)        | ---> |   h  e  l  l  o     |
| length: 5           |      +----------------------+
| capacity: 5         |
+---------------------+

```

- Heap is used for data with unknown or variable size at compile time.
- String, Vec, Box, and collections use the heap.
- Ownership ensures automatic cleanup without garbage collection.
- Stack stores pointers to heap data, while the heap stores actual data.

Stack vs Heap in Ownership

	Type	Stored On	Ownership Rules	Example
1	`i32`, `bool`	Stack	Copy (no move)	`let a = 10;`
2	`String`	Heap*	Move ownership by default	`let s = String::from("hi");`
3	`Vec<i32>`	Heap*	Move unless referenced	`let v = vec![1, 2];`

Note: String and Vec store metadata (pointer, length, capacity) on the stack, but their actual data lives on the heap.

Clone Heap Data

Plain Text

```
let s1 = String::from("hello");
let s2 = s1.clone(); // deep copy heap data

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


Stack:                  Heap:
+------+            +------------------+
|  s1  | ---------> | h e l l o        |
+------+            +------------------+
|  s2  | ---------> | h e l l o        |
+------+            +------------------+


```

Reference and borrowing Ownership in Rust?

In Rust, instead of moving ownership of a variable, you can borrow it using references. This allows other parts of your code to use the value without taking ownership.

Types of References:

	Type	Keyword	Mutable?	Ownership?	Copies Data?
1	Immutable Ref	`&T`	❌	No	No
2	Mutable Ref	`&mut T`	✅	No	No

Immutable Reference Example (`&T`)

Plain Text

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1); // Pass by reference

    println!("Length of '{}' is {}", s1, len); // s1 still valid
}

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

Mutable Reference Example (`&mut T`)

Plain Text

```
fn main() {
    let mut s = String::from("hello");
    change(&mut s); // Mutable borrow

    println!("Modified string: {}", s);
}

fn change(s: &mut String) {
    s.push_str(", world!");
}
```

- Only **one mutable reference** is allowed at a time.
- You **can't mix** mutable and immutable references in the same scope.

Plain Text

```
Immutable Reference:
Stack:
+------+        +------------------+
|  s1  | ------> | h e l l o        |  ← Heap
+------+        +------------------+

calculate_length(&s1) uses s1 via reference, ownership not moved.

Mutable Reference:
Stack:
+------+        +--------------------------+
|  s   | ------> | h e l l o ,   w o r l d  |  ← Heap
+------+        +--------------------------+

change(&mut s) modifies data in-place on heap.
```

	Concept	Example	Ownership Taken?	Mutation Allowed?
1	Immutable reference	`&s1`	❌ No	❌ No
2	Mutable reference	`&mut s1`	❌ No	✅ Yes
3	Ownership transfer	`let s2 = s1`	✅ Yes	✅ Yes

Multiple Immutable References (Same Reference)

Plain Text

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

    let r1 = &s;
    let r2 = &s;

    println!("r1 = {}, r2 = {}", r1, r2); // both valid
}
```

Mutable + Immutable Together

Plain Text

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

    let r1 = &s;
    let r2 = &mut s; // ❌ ERROR: cannot borrow as mutable because it's already borrowed as immutable

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