Ownership 101

- The biggest new feature of Rust
- Compile time check
- Implementation of *affine type system*: Each data/value has exactly **one owner.**
- *Create ownership* of a value with **variable binding** `let var = value;` (immutable binding) or `let mut var = value;` (mutable binding)

Move: Copy vs. non-Copy

```
let x = 1;
println!("x: {}", x);
let y = x; // y has a copy of x value
println!("x: {}", x); // Ok
```

- Primitives are copied by default (*impl Copy trait*)

```
let v = vec![1,2,3]; // Vec doesn't impl Copy
println!("{:?}", v); // Ok
let v1 = v; // v1 is the owner. v is **moved**
println!("{:?}", v); // Compile Error: value used here after move
```

However,

```
let v = vec![1, 2, 3]; // Vec impl Clone instead. Heap thing!
println!("{:?}", v); // Ok
let v1 = v.clone(); // v1 owns a clone of v data
println!("{:?}", v); // Ok
```

Scoping rule

- When binding goes out of scope the value is dropped/destroyed/deallocated (Welcome to Rust memory management)

```
let x = 1;
    {
        let y = 2;
        println!("x + y: {}", x + y); //Ok
```

```
    } // --> y goes out of scope and 2 is destroyed
println!("x + y: {}", x + y); //Error cannot find y
```

Borrowing

- Shared (immutable) reference: **&**
- Mutable reference: **&mut**

```
let v = vec![1, 2, 3]; // Vec impl Clone instead. Heap thing!
println!("{:?}", v); // Ok
let v1 = &v; // v1 owns a reference to v's data
println!("{:?}", v); // Ok
```

Note: Scoping rule applied to references as well
(general lifetime)

```
let v = vec![1, 2, 3];
    {
        let v1 = &v;
        println!("v[0] + v1[0]: {}", v[0] + v1[0]); //Ok
    } // --> v1 goes out of scope so does the reference
println!("v[0] + v1[0]: {}", v[0] + v1[0]) // Error
```

Notorious move errors

- Ownership cannot be transferred while there's a reference to it (prevent dangling pointer)

```
let v = vec![1, 2, 3];
let v_ref = &v;
let v1 = v; // Compile Error: cannot move out of
            // `v` b/c it's borrowed
```

- Cannot move out of the borrowed content

```
let mut v = vec![1, 2, 3];
let v1 = &mut v;
let mut v2 = *v1; // Compile Error:
                  // cannot move out of the borrowed content.
                  // Consider using a reference
v2.push(4);
```

Summary of borrowing rules (memory safety)

- Cannot borrow something that doesn't exist! (obvious)
- Either one can happen not both:

  1) One mutable borrow (no data race)
  2) Multiple immutable borrows

```
let mut v = vec![1, 2, 3];
    for elt in &v {
        v.pop(); // pop needs a mutable borrow
        // ERROR: cannot borrow `v` as mutable because
        // it is also borrowed as immutable
    }
```

```
let mut v = vec![1, 2, 3];
    for elt in &mut v {
        v.pop(); // pop needs a mutable borrow
        // Error: two mutable borrows!
    }
```

Exercises:

Try to see whether the code in each problem compiles or not and where the problem is. Is it possible to change the code slightly to make it compile?!

- problem1
- problem2
- problem3
- problem4
- problem5: solution1, solution2
- problem6: solution
- problem7: solution

References

- Rust book v2
- CIS 198: Rust Programming
- TMLL

- users.rust-lang.org
- Stackoverflow