Rust Bootcamp 2021

Day 2

Day 2 Activities

- ✓ Ownership and the Borrow Checker
- ✓ Structs
- ✓ Exercises and Homework

Ownership

Memory ownership is a central principle of Rust, so we'll spend some time going over what this means, especially in practice.

Here are the three rules of ownership in Rust:

- 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.

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html

Scopes

As in C++ and other languages, Rust has scopes for variables.

```
{                       // 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}                       // this scope is now over, and s is no longer valid
```

Braces can be used anywhere, and nested. When a variable goes out of scope it is "dropped" or deleted. This is similar to C++ RAII.

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#variable-scope

Copy vs Move

For primitive types, Rust will copy them when assigned:

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

x and y both are equal to 5, with different “copies” of 5. This is because they have a known, fixed size at compile time.

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ways-variables-and-data-interact-move

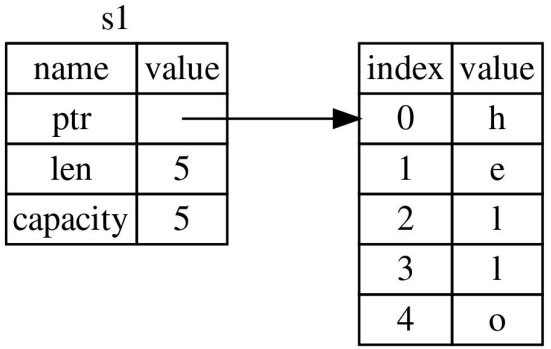
Copy vs Move

For advanced types, Rust cannot automatically copy them

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

The problem is that a String is actually two blocks of memory, one on the stack and one on the heap.

If we copied the stack portion (a shallow copy), we'd have two owners, which violates Rust's ownership rules.

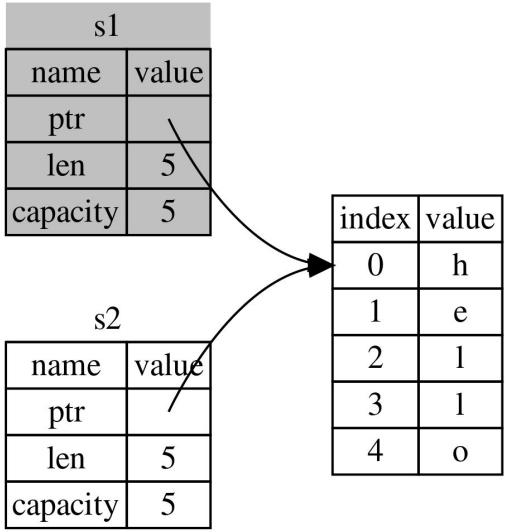


https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ways-variables-and-data-interact-move

Copy vs Move

For advanced types, Rust cannot automatically copy them

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



By default, assignment will do a *move* operation here. This invalidates s1, while s2 has the string.

A full (deep) copy is potentially an option, but is expensive, so must be done explicitly with clone().

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ways-variables-and-data-interact-move

Ownership and Functions

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

    let x = 5;
    makes_copy(x);
}
```

```
fn takes_ownership(some_string: String)
{
    println!("{}", some_string);
}

fn makes_copy(some_integer: i32) {
    println!("{}", some_integer);
}
```

A function acts much like a new variable assignment with ownership, where only primitive types copy.

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ownership-and-functions

Ownership and Functions - return values

```
let s1 = gives_ownership();

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

let s3 = takes_and_gives(s2);
```

```
fn gives_ownership() -> String {
    String::from("hello")
}

fn takes_and_gives(s: String) -> String
{
    s
}
```

One option if you want to use a move with a function parameter is to return the value back.

https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html#ownership-and-functions

Ownership and Functions - borrowing

A better option is *borrowing* - making a reference to a variable.

```
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 {
    s.len()
}
```

https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Ownership and Functions - mutable references

If we want to edit the reference, we need to make everything mutable.

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

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

https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Ownership - mutable references

Mutable references have one big restriction: you can have only one mutable reference to a particular piece of data in a scope.

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

let r1 = &mut s;
let r2 = &mut s;
```

This code is illegal, as it tries to make two references. One of the advantages of this restriction is that data races can be prevented at compile time.

https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

Ownership - Slices

Slices let you reference a contiguous sequence of elements in a collection. This is a reference, not a copy, and follows the rules of Rust references.

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..];
let hello_world = &s[..];
```

Note that the ranges allow dropping the first or second index as a shortcode for “all the rest.” You can even drop both indexes to return a reference to the whole string.

https://doc.rust-lang.org/book/ch04-03-slices.html

Ownership - Slices

```
fn first_word(s: &String) -> &str {
    let bytes = s.as_bytes();
    for (i, &item) in bytes.iter().enumerate() {
        if item == b' ' {
            return &s[0..i];
        }
    }
    &s[..]
}
fn main() {
    let mut s = String::from("hello world");
    let word = first_word(&s);
    s.clear();                      // error! reference exists!
    println!("the first word is: {}", word);
}
```

Ownership - Slices

Slices also work on other data structures besides strings.

```
let a = [1, 2, 3, 4, 5];

let slice = &a[1..3];
```

Containers that support indexing are generally sliceable.

https://doc.rust-lang.org/book/ch04-03-slices.html#other-slices

Structs

A struct looks basically the same as in C or C++.

```
struct User {
    username: String,
    sign_in_count: u64,
    active: bool,
}
```

```
let user1 = User {
    active: true,
    username: String::from("user"),
    sign_in_count: 1,
};
```

To use a struct, all values must be defined (in any order).

https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Structs

To get a value from a struct, dot notation is used.

```
let user1 = User {
    active: true,
    username: String::from("user"),
    sign_in_count: 1,
};
println!(“username is {}”, user1.username);
```

You may also assign to the value using dot notation.

https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Structs

Field init shorthand is available:

```
fn build_user(username: String) -> User {
    User {
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Structs - update syntax

When creating a new instance from an existing instance, a shorthand is available using the .. operator

```
let user1 = User {
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};
let user2 = User {
    username: String::from("anotherusername567"),
    ..user1
};
```

https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Structs - tuple structs

Tuple structs do not have named fields, just types.

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let black = Color(0, 0, 0);
let origin = Point(0, 0, 0);

let black_val0 = black.0;
let origin_val2 = origin.2;
```

https://doc.rust-lang.org/book/ch05-01-defining-structs.html

Structs - storing references

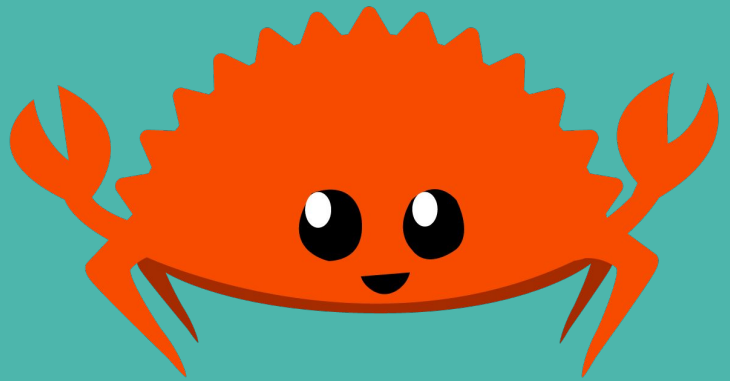
Storing references in structs is difficult because of a feature called lifetimes.

```
struct User {
    username: &str,
    sign_in_count: u64,
    active: bool,
}
```

The problem is that the compiler doesn't know how long the reference should live - both when it gets created and when it should be dropped. While this can be solved, it is far easier to store only owned types.

https://doc.rust-lang.org/book/ch05-01-defining-structs.html#ownership-of-struct-data

End of Lecture



How to read cmdline arguments

In today's exercises and homework, you might need to read arguments from the cmdline.

```
use std::env;

fn main() {
    let args: Vec<String> = env::args().collect();

    let query = &args[1];
    let filename = &args[2];

    println!("Searching for {}", query);
    println!("In file {}", filename);
}
```

How to read from a file

In today's exercises and homework, you might need to read from a file.

```
use std::fs;

fn main() {
    let filename = "foo.txt";
    let contents = fs::read_to_string(filename)
        .expect("Something went wrong reading the file");
}
```

Some useful resources

https://doc.rust-lang.org/std/primitive.str.html

In particular, functions like split and startswith.

Exercises

Find and fix the errors in exercises/day2/check_errors

Write a program that can take a log file as input and query for various things.

Sample queries:

- List lines containing the user dglo at the WARNING level
- List all lines between two times
- For all lines that have “memory error”, print the total average memory size

Starter code and log file available at exercises/day2/logfile