L17: Ownership, Borrowing and Sharing

Rachata Ausavarungnirun

(rachata.a@tggs.kmutnb.ac.th)

March 10th, 2020

Architecture Research Group

Software System Engineering

Thai-German Graduate School, KMUTNB

Ownership

- What should be considered for a memory to be safe?

- Who else share the memory?
  - How to handle parallel data updates - Correctness

- When do I need the memory?
  - When to free?

- Rust introduces an ownership system that provide memory safety

Lifetime of a Resource

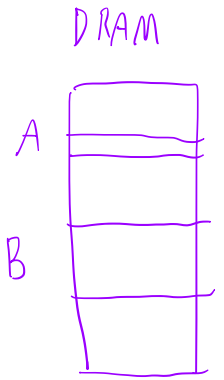
- Rust will find the lifetime of each resource - When the variable is n't used anymore
  - This can be done through checking the scope
  - This can also be done through liveness analysis
    - We will talk about this during the compiler section

- Resources are freed when their lifetime end

Scope Example - Range of coverage

- { Scope of S => Everyone know "S" until the end bracket
  let s = “hello”
  // Do something here
  }
  ← Can free "S" from this (No one use S)

- From the above example, s is not value until the variable is declared, and no longer valid at the end



Garbage collector will check for the use of A and B
if A doesn't be used, free A

Ownership Rules

(Think of it as money)

- Every value has a single owner at any given time
  - Why is this good?

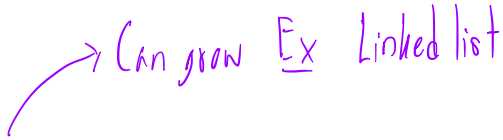
(Has to return to the owner)

- Anyone can borrow a reference to a value
  - This borrowed reference cannot outlive the value

- To change a value, you need an exclusive access
  - The only one who can change a value at the instance of time

Single Ownership Examples

- Variables own their values
- Structs own their fields
- Allocated value (on the heap) has a single pointer that own the value
- If the owner drops the value, value is dropped

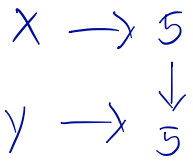


String Types

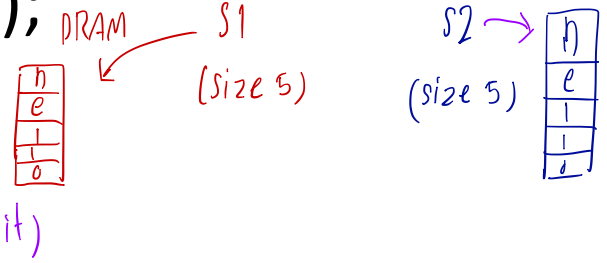
- There are two way to represent strings
  - Rust: &str and String
- let mut s = String::from("hello");
  - What is the type of “hello”?
  - What is the type of s?
- s.push_str(", world!");
  - push_str() appends a literal to a String
- println!("{}", s);
  - This will print `hello, world!`

Copying Data

- fn main() { let x = 5; let y = x; }
  - Bind value 5 to x, then copy x to y



- What if I do this?
- fn main() {
  let s1 = String::from("hello");
  let s2 = s1;
  }



- Use of moved value s1
  - But only one thing can use s1 at the same time
  - And copy is not implemented for String
    - Why is a deep copy expensive vs. move?

Clone - Deep Copy

- If you still want to perform a deep copy, use clone

- let s1 = String::from("hello");
- let s2 = s1.clone();

- Types that has the copy trait - Don't need Deep copy (because it is simple)
  - Integer types
  - Boolean types
  - Floating point types
  - Character types
  - Tuples where everything inside is a copy type

Ownership w/ Function Calls

- Permision to do something which that variable

- Passing value to a function is similar to assignment

- Consider copy vs. move

- See lecture-example1.rs

Ownership w/ Function Calls

- **fn** main() {
  let name = format!("…");
  helper(name); // ownership of "name" is with helper
  helper(name); // This violate the ownership rule
  }

(name gives ownership with helper)

Take name and modify it at the same time (Error)

- **fn** helper(name: String) {
  println!(..);
  }

Move, Clone and Copy

- Move: pass variables that cannot be copy around

- Clone: custom code to make a copy

- Copy: implicitly done

Borrowing - Take something and promise to return it back

- This temporarily transfer the rights to use to someone
  - Use & to refer to the item you want to borrow
- **fn** main() {
  let name = format!("…");
  let r = &name; - r borrow name
  helper(r);
  helper(r); } Read only, and don't modify (Cannot modify)
  }
- **fn** helper(name: &String) {
  println!(..);
  }

Borrowing → Immutable

- You cannot write to something you borrow
- This is ok

```
fn helper(name: &String) {
  println!("{}", name);
}
```

- **This is not ok** - Modify the content

```
fn helper(name: &String) {
  name.push('x');
}
```

- This is called a shared reference - Can be many readers

Lending - Same as borrowing

- Lending is for the owner to give the permission to someone else
- **fn** main() {
  let name = format!("…");
  helper(&name[1..]); // Lend part of the string
  helper(&name); // Lend an entire string
  }
- **fn** helper(name: &str) {
  println!(..);
  }

Mutable Borrowing/Lending - Allow to change

- You can use the keyword **mut** to make the item mutable
- ```
  fn main() {
    let mut name = …;
    update(&mut name);
    println!(“{}”, name);
  }
  ```
- Then you can modify the string you borrow

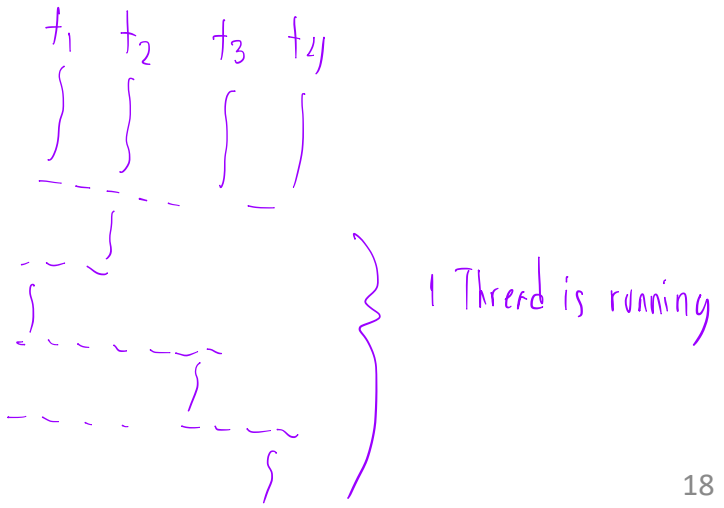
  ```
  fn update(name: &mut String) {
    name.push_str(“…”);
  }
  ```
  ```

Types of Ownership

- Owner: full control, free when done

- Shared reference: many readers and no writer
  - &String
  - This has performance benefit in hardware

- Mutable reference: No other reader, one writer
  - &mut String
  - This create serialization



Rust Solution to Dangling Pointers

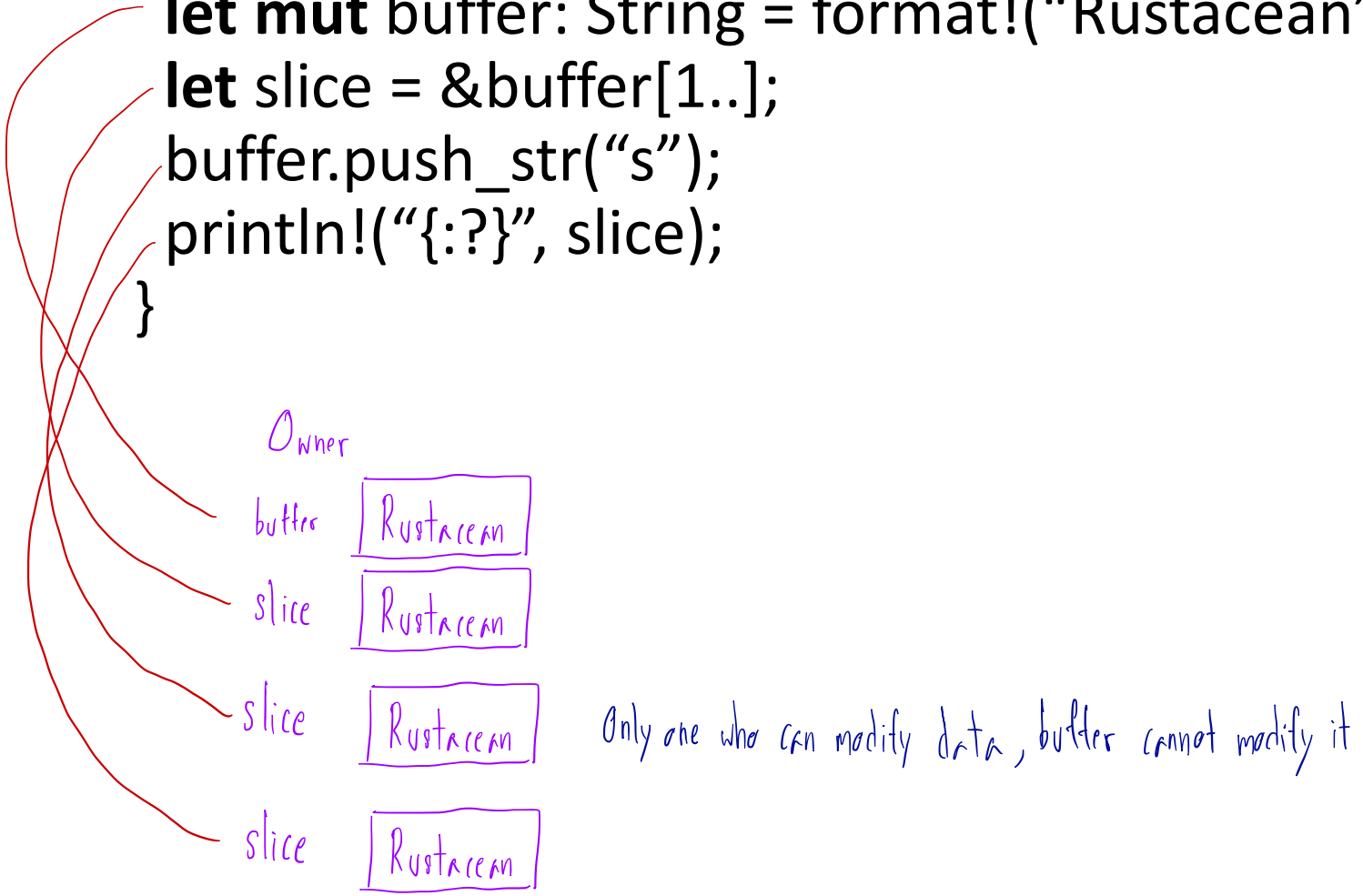
- Compile-time read and write locks → When to lock

- Share reference to X: read locks X
  - Other readers can read
  - No writer
  - Lock last until this reference is out of the scope

- Mutable reference to X: write locks X
  - No other reader and writer - Only one at a time
  - Lock last until this reference is out of the scope

Fix This Code

- **fn** main() {
  let mut buffer: String = format!(“Rustacean”);
  let slice = &buffer[1..];
  buffer.push_str(“s”);
  println!(“{:?}”, slice);
  }



What's Wrong Here?

- **fn** main() {
  let mut buffer: String = format!("Rustacean");
  for i **in** 0 .. buffer.len() {
  let slice = &buffer[i..];
  buffer.push_str("s"); (Wrong)
  println!("{:?}", slice);
  }
  buffer.push_str("s"); (Correct)
  }

Owner
buffer
buffer
slice
slice
slice
buffer

Try! Macro - Try, Catch

- From this code: (Wrong)
- **fn** do_calc() -> Result<i32, String>{
  let a = match do_subcalc1(){
  Ok(val) => val,
  Err(msg) => return Err(msg)
  }
  let a = match do_subcalc2(){
  Ok(val) => val,
  Err(msg) => return Err(msg)
  }
  Ok(a+b)
  }
- This code is tedious

Try! Macro

- Instead, you can do this
- **fn** do_calc -> Result<i32, String>{
  let a = try!(do_subcalc1());
  let b = try!(do_subcalc2());
  Ok(a+b)
  }

- Try will behave similar to “try” in other languages

In-class Exercise 16

- Fix the two codes on canvas so that they compile