ECE 459
Programming for Performance

Lecture 4
Rust: Breaking the Rules for Fun and Performance

Winter 2023

Huanyi Chen
huanyi.chen@uwaterloo.ca

UNIVERSITY OF WATERLOO

Multiple Ownership

- Reference Counting

- `Rc<T> // Rc stands for reference counted`

Reference Counted

```
use std::rc::Rc;

fn main() {
    let s = String::from("hello");
    let rc = Rc::new(s);
    println!("{}", rc);
}
```

Multiple Ownership

```
fn main() {
    let s = ExampleStruct {
        description: String::from(
            "this is a struct"
        )
    };
    let rc = Rc::new(s);
    let rc_clone = rc.clone();
    println!("rc       : {:?}", rc);
    println!("rc_clone: {:?}", rc_clone);
}
```

Multiple Ownership

```
fn main() {
    let s = ExampleStruct {
        description: String::from(
            "this is a struct"
        )
    };
    let rc = Arc::new(s);
    let rc_clone = rc.clone();
    // spawn a thread and use rc_clone
    // ..
    println!("rc       : {:?}", rc);
    println!("rc_clone: {:?}", rc_clone);
}
```

- Rc<T> is not thread-safe
- Use Arc<T> (atomically reference counted)

Lifetimes

```
fn longest(x: &str , y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes

- to prevent *dangling references*

- Annotations
  - `'a`
  - `'b`
  - `etc.`

  - `'static` (Not recommended ❌)

Lifetimes

- Annotations are meant to tell Rust how generic lifetime parameters of multiple **references** relate to each other
- Annotations don’t change how long references live, really.
- They just describe the relationships between the lifetimes of references.
- Analogy: expiration dates on food.

Lifetimes

```
fn longest<'a> (x: &'a str , y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes

```
foo<'a, 'b>

// `foo` has lifetime parameters `'a` and `'b`
// and the lifetime of foo cannot exceed that of either
'a or 'b.
```

Lifetimes Elision

1. Compiler assigns a lifetime parameter to each parameter that's a reference (focus on input)
   - `fn foo<'a>(x: &'a i32);`
   - `fn foo<'a, 'b>(x: &'a i32, y: &'b i32);`
   - etc.

2. If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters

3. If there are multiple input lifetime parameters, but one of them is `&self` or `&mut self` because this is a method (of a struct), the lifetime of `self` is assigned to all output lifetime parameters.

More examples

- lifetime-elision
- https://doc.rust-lang.org/reference/lifetime-elision.html?highlight=lifetime#lifetime-elision-in-functions

- lifetime in general
- https://doc.rust-lang.org/stable/rust-by-example/scope/lifetime.html

Unsafe

```
unsafe {
```

- Call an unsafe function/method
- Access or modify a mutable static variable
- Implement an unsafe trait
- Access the fields of a union
- Dereference a raw pointer

```
}
```

In-class exercises

- See lectures/flipped/L04.md
- You can create a repo called “ece459-practice” and push your code there
- You can add me as a member if you want