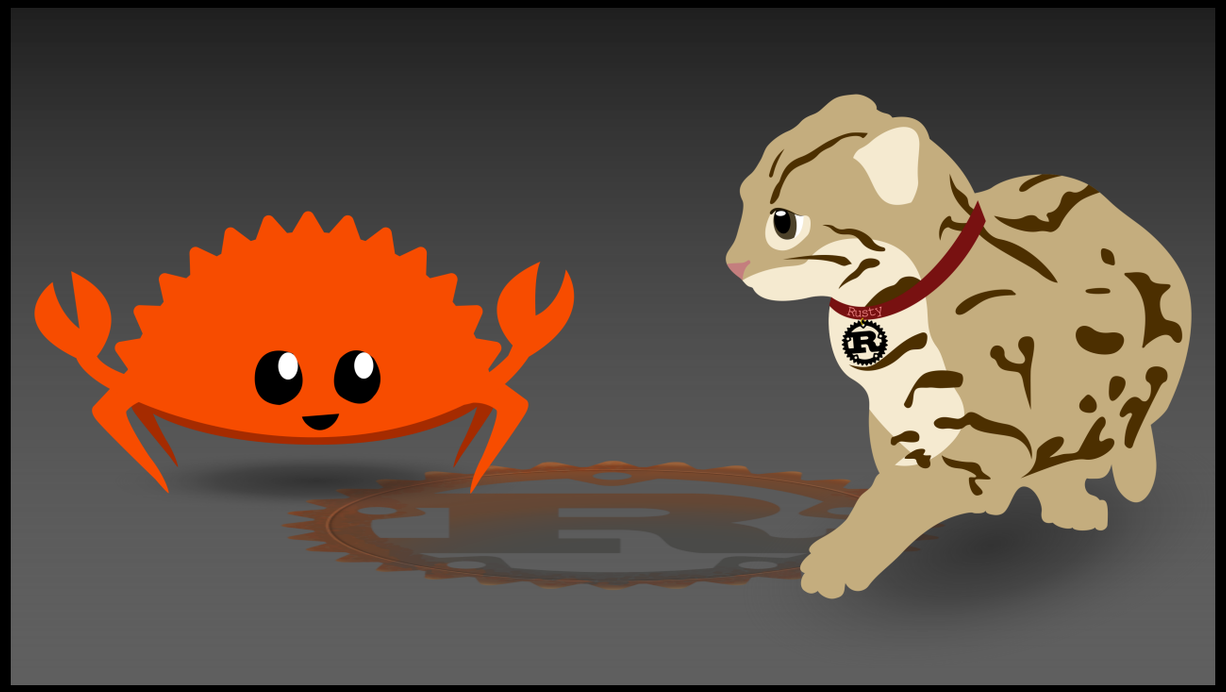

The Beauty of Rust

Lifetimes

Peter Zdankin

Today's topics

The bad and the good

- We need to talk about lifetimes
  - Awkward concept to explain, but sometimes relevant

- Cargo
  - The best build system. Period.

Look at this cute cat
before we proceed

What are lifetimes?

Lifetimes

- Every Reference has a lifetime
- The lifetime indicates how long a reference is valid
- Most of the time you don’t have to think about lifetimes
- Lifetimes have always been in Rust
- Have you noticed them so far?

```rust
fn main(){
    let r;
    {
        let x = 5;
        r = &x;
    }
    println!("r: {}", r);
}
```

```
joy on □ master [?] via R v1.34.1 → cargo run
   Compiling joy v0.1.0 (/Users/peterzdankin/Projects/rust/joy)
error[E0597]: `x` does not live long enough
  --> src/main.rs:6:14
   |
6  |         r = &x;
   |             ^ borrowed value does not live long enough
7  |     }
   |     - `x` dropped here while still borrowed
...
10 | }
   | - borrowed value needs to live until here

error: aborting due to previous error

For more information about this error, try `rustc --explain E0597`.
error: Could not compile `joy`.
```

The Borrow Checker

- Checks whether all borrows obey da rulez
- Remember the borrowing rules?

```
{
    let r;                          // ----------+-- 'a
                                    //           |
    {                               //           |
        let x = 5;                  // -+-- 'b   |
        r = &x;                     //  |        |
    }                               // -+        |
                                    //           |
    println!("r: {}", r);           //           |
}                                   // ----------+
```

Lifetimes

- We are able to annotate lifetimes with any name
- Commonly used: ‘a, ’b, ’c
- Lifetimes can be ordered

- If lifetime c is smaller than lifetime b, then a value with lifetime c can’t be assigned to a variable that is in b

```
{
    let x = 5;                  // ----------+-- 'b
                                //           |
    let r = &x;                 // --+-- 'a  |
                                //   |       |
    println!("r: {}", r);       //   |       |
                                // --+       |
}                               // ----------+
```

You've earned another cat


<'A>
<'B>

Why lifetimes?

- Imagine you want to get the longer of two strings
- You pass two references
- How should the function be implemented?

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}
```

Is this safe?

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

And here?

```
fn main() {
    let string1 = String::from("long string is long");

    {
        let string2 = String::from("xyz");
        let result = longest(string1.as_str(),
string2.as_str());
        println!("The longest string is {}", result);
    }
}
```

And here?

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

If it’s not clear, we have to make it clear

Add lifetime hints

- First, do a list of the lifetimes that occur
- Then, assign the lifetimes to your input/output parameters
- You can leave some annotations out

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

Lifetimes do not do something

```rust
fn longest<'a>(x: &str, y: &str) -> &'a str {
    let result = String::from("really long string");
    result.as_str()
}
```

```
error[E0597]: `result` does not live long enough
  --> src/main.rs:15:5
   |
15 |     result.as_str()
   |     ^^^^^^ borrowed value does not live long enough
16 | }
   | - borrowed value only lives until here
   |
```

Let’s go back

```
fn main() {
    let string1 = String::from("long string is long");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }
    println!("The longest string is {}", result);
}
```

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

Lol just add lifetimes idiot

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

```
joy on □ master [?] via R v1.34.1 → cargo run
   Compiling joy v0.1.0 (/Users/peterzdankin/Projects/rust/joy)
error[E0597]: `string2` does not live long enough
 --> src/main.rs:6:44
  |
6 |         result = longest(string1.as_str(), string2.as_str());
  |                                            ^^^^^^^ borrowed value does not live long enough
7 |     }
  |     - `string2` dropped here while still borrowed
8 |     println!("The longest string is {}", result);
9 | }
  | - borrowed value needs to live until here
```

MORE KITTEN

Next level: Lifetimes in structs

- Work similarly
- A struct that contains a reference, can not live longer than the reference
- Borrow checker will be angry if you misbehave

```
struct ImportantExcerpt<'a> {
    part: &'a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split('.')
        .next()
        .expect("Could not find a '.'");
    let i = ImportantExcerpt { part: first_sentence };
}
```

Rust history

- Rust has come a long way
- It changed a lot over the years
- http://venge.net/graydon/talks/intro-talk-2.pdf
- Look at these slides, if you want to see the slides Graydon Hoare (Creator of Rust) presented to Mozilla
- We’ve come such a long way

Introducing: Rust

- **Rust** is a language that mostly cribs from past languages. **Nothing new**.
- Unapologetic interest in the static, structured, concurrent, large-systems language niche.
  - Not for scripting, prototyping or casual hacking.
  - Not for research or exploring a new type system.
- Concentrate on **known** ways of achieving
  - more safety,
  - more concurrency,
  - less mess.

Earliest archeological Rust code

3500 BC Egyptian Wall

Lifetime elusion rules

- In Pre 1.0 Rust, all lifetimes needed to be annotated
- All code with references looked like this!

```
fn first_word<'a>(s: &'a str) -> &'a str {
```

- Rust introduced rules that would simplify the code

Rule 1

- Each reference parameter gets its own lifetime

```
fn first_word(s: &str) -> &str {
```

```
fn first_word<'a>(s: &'a str) -> &str {
```

Rule 2

- If we have one input reference, that lifetime is set to the output, if it’s a reference

```
fn first_word(s: &str) -> &str {
```

```
fn first_word<'a>(s: &'a str) -> &’a str {
```

- Now a programmer doesn’t need to write this

Rule 3

- If we have multiple input references and one of them is &self or &mut self, the output lifetime is the lifetime of self

What if 2 references?

- Let’s apply rule 1

```
fn longest(x: &str, y: &str) -> &str {
```

```
fn longest<'a, 'b>(x: &'a str, y: &'b str) -> &str {
```

- We can’t apply Rule 2
- We have no lifetime for the output and need to do it ourselves

What happens here?

```rust
use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(x: &'a str, y: &'a str, ann: T) -> &'a str
    where T: Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Exercises

- Write a function, that receives two references and returns the first
- Replace the ? In the following function signature:
  fn func<?>(a: &? i32, b: &? i32) -> &? i32

- Such that both {a} and {b} are valid bodies for the function
- Write a struct, that stores a &str and implement a method that compares if this &str is longer or shorter than a new parameter &str