When Lifetimes Give You Lemons

Philip Kristoffersen

Philip Kristoffersen

Work

Consultant
Developer/Architect/Lead
C#, TypeScript, Java

Personal

Open Source Developer
Rust enthusiast
Rust!

BOiLR
Play all your games from steam

Show you why **Rust** has **lifetimes**, **what** they do and **when** and **how** you use them

Goal of this talk

OPINION!

Rust Lifetime Annotations are not complex

They are unfamiliar

If you are a Rust beginner

Keep calm and don't panic!()

I will try to bring everyone up to speed

If you are a Rust pro

Stick Around

I will get into some advanced topics by the end

No live coding

In the interest of time

But come talk to me after, and we can try out some things

Agenda

1 **Memory Safety**
Speedrun of memory safety

2 **Lifetimes**
What are lifetimes?

3 **Lifetime annotations**
How do i use lifetimes?

4 **Lemonade**
Tips and tricks for handling lifetimes

How and why is Rust memory safe?

1. Memory Safety

Memory Management

Allocation

Tell the computer that we need X memory

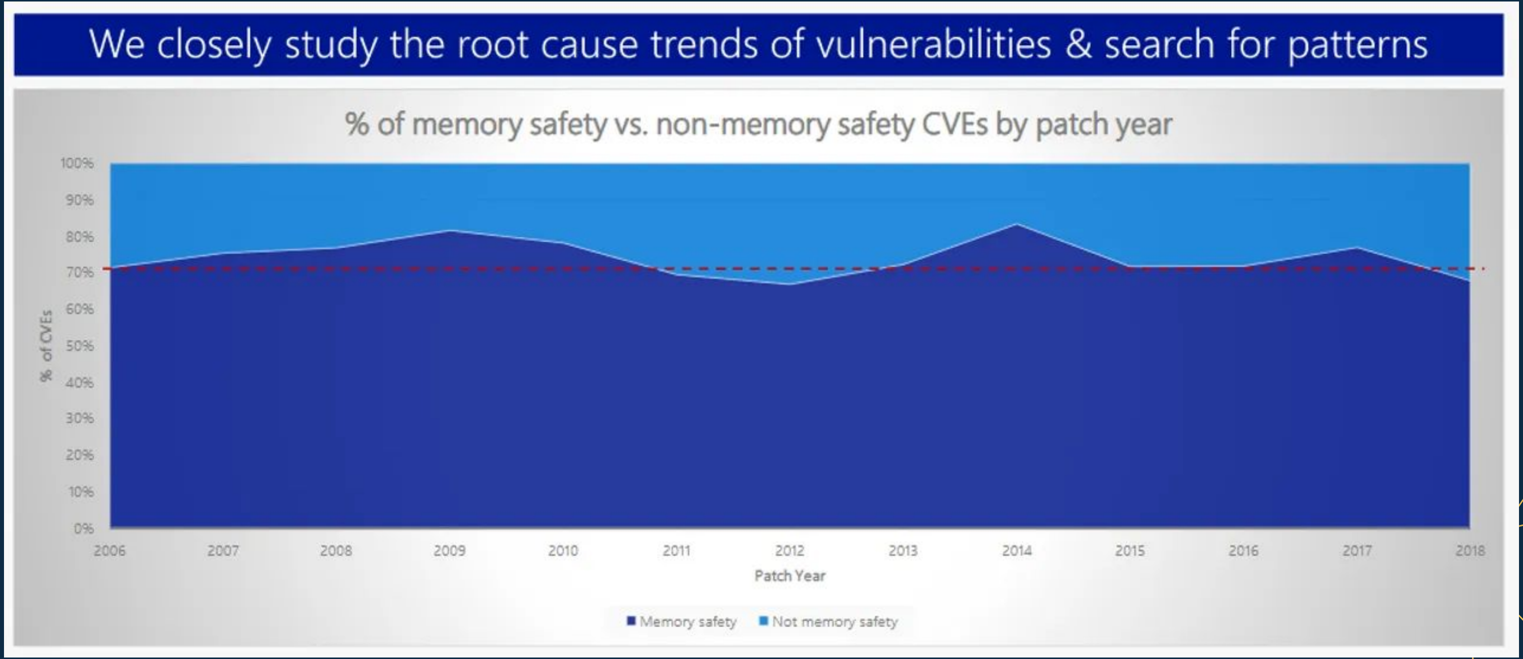
De-Allocation aka. Drop

Tell the computer that we don’t need X memory anymore

Memory Management

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    // Allocate memory
    int* ptr = (int*)malloc(sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed\n");
        return -1;
    }

    *ptr = 100;
    printf("Value: %d\n", *ptr);

    // Deallocate memory
    free(ptr);
    ptr = NULL; // Good practice

    return 0;
}
```

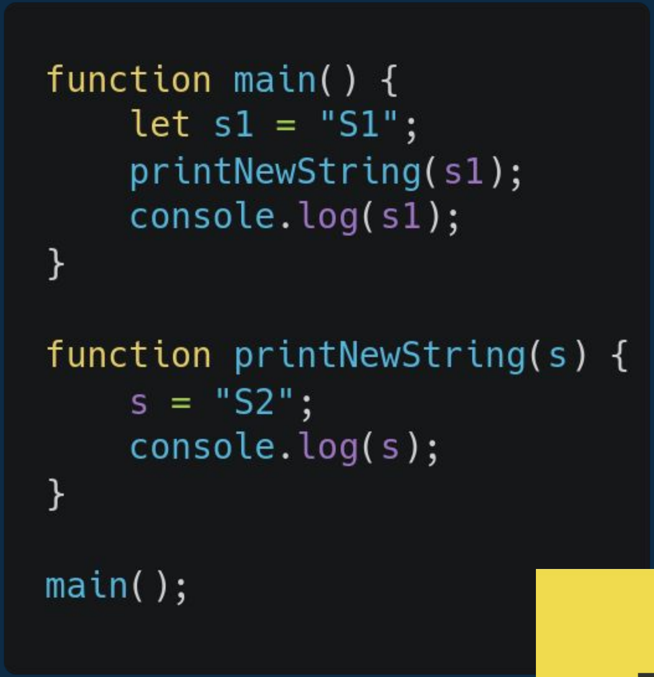
Is it really that hard?



https://www.zdnet.com/article/microsoft-70-percent-of-all-security-bugs-are-memory-safety-issues/

Garbage Collected Languages

GC Languages usually **handle** and **hide** memory management



```
function main() {
    let s1 = "S1";
    printNewString(s1);
    console.log(s1);
}

function printNewString(s) {
    s = "S2";
    console.log(s);
}

main();
```

Rust Memory Management

Rust handles memory
but doesn’t hide it

Rust Borrowing Speedrun

Variables own data

Data is kept as long as it is owned

Variables have lifetimes

From creation to they go out of scope

References to variables

Access a variables data with & and &mut references

What are lifetimes?

2. Lifetimes

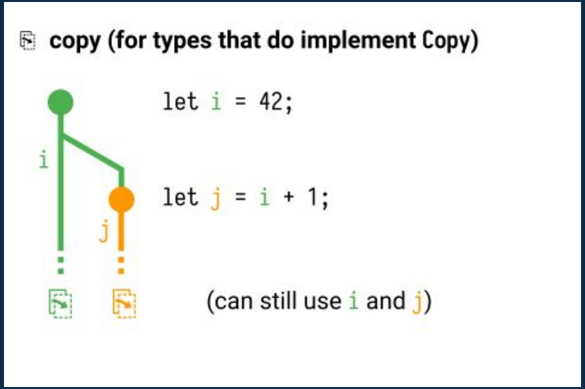
Vocabulary

Copy
Copy the memory and make a new variable own it

Move
Transfer ownership of memory to a new variable

Borrow
Lend memory access to variable

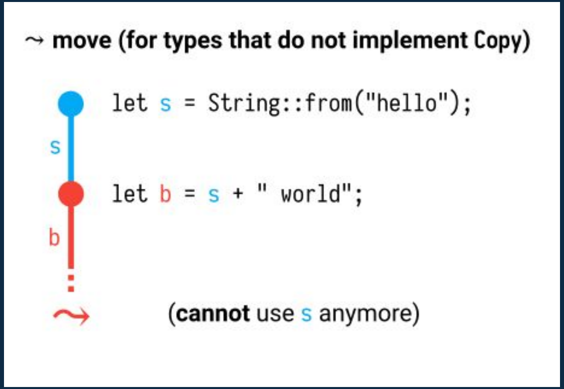
Copy



```
let i = 42;

let j = i + 1;
```

https://rufflewind.com/img/rust-move-copy-borrow.png

Move

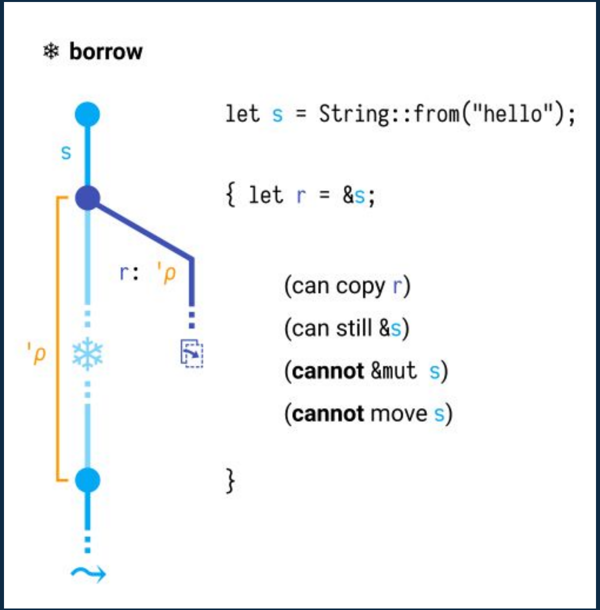

⤳ **move (for types that do not implement Copy)**

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

let b = s + " world";
```

(**cannot** use s anymore)

https://rufflewind.com/img/rust-move-copy-borrow.png

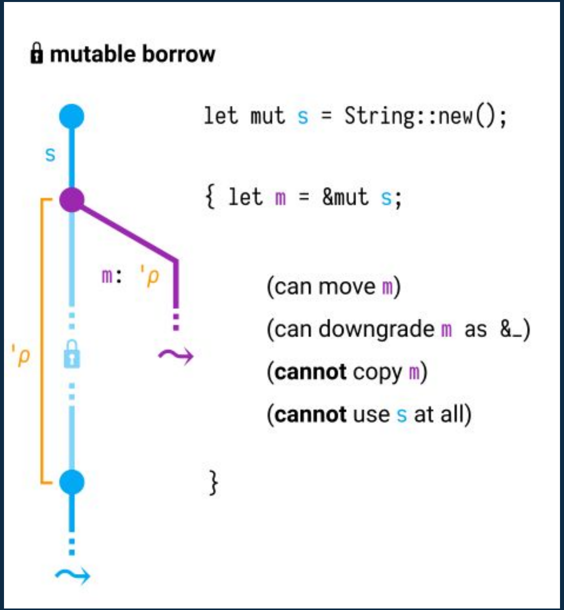
Borrow



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

{ let r = &s;

      (can copy r)
      (can still &s)
      (cannot &mut s)
      (cannot move s)

}
```

https://rufflewind.com/img/rust-move-copy-borrow.png

Mutable Borrow


mutable borrow

```
let mut s = String::new();

{ let m = &mut s;
```

(can move m)

(can downgrade m as &_)

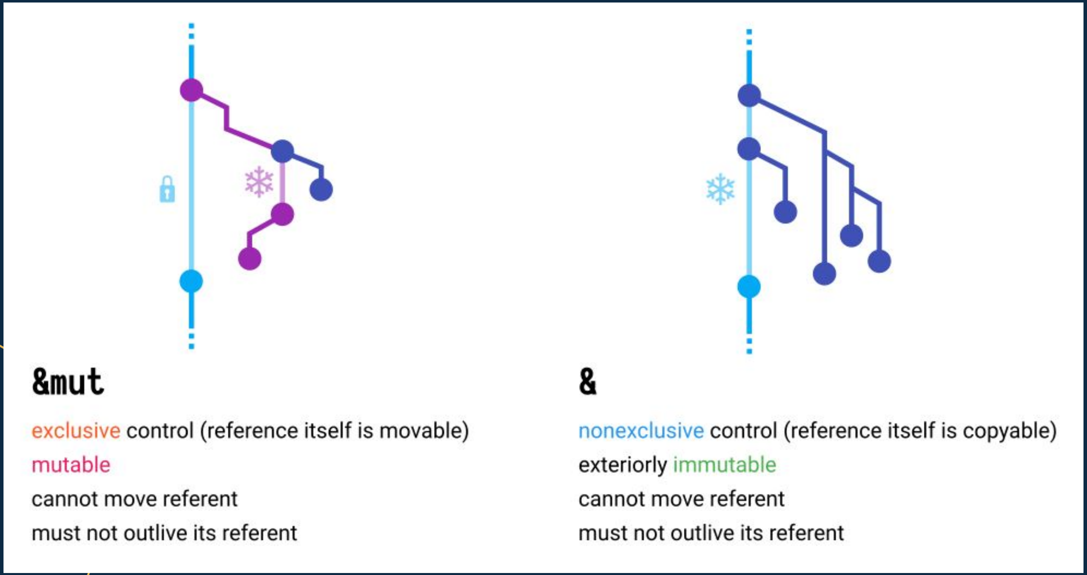
(**cannot** copy m)

(**cannot** use s at all)

```
}
```

https://rufflewind.com/img/rust-move-copy-borrow.png

Reference Lifetimes



&mut

exclusive control (reference itself is movable)
mutable
cannot move referent
must not outlive its referent

&

nonexclusive control (reference itself is copyable)
exteriorly immutable
cannot move referent
must not outlive its referent

https://rufflewind.com/img/rust-move-copy-borrow.png

Sometimes Rust needs a little help

3. Lifetime annotations

Using Lifetime Annotations

Rarely used

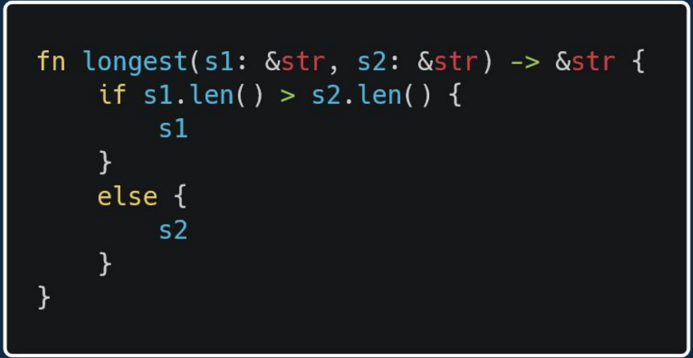
Most of the time you don’t have to use lifetime annotations

References only

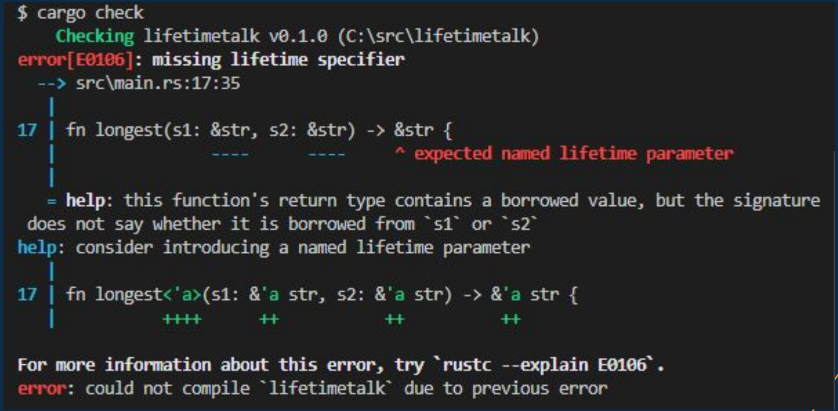
Only when you start working with references will you need annotations

```
fn lower_contains(s1: &str, s2: &str) -> bool {
    let l1 = s1.to_ascii_lowercase();
    let l2 = s2.to_ascii_lowercase();
    l1.contains(&l2)
}
```

Using Lifetime Annotations



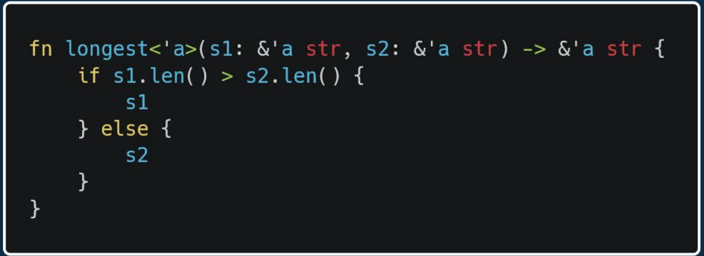
```rust
fn longest(s1: &str, s2: &str) -> &str {
    if s1.len() > s2.len() {
        s1
    }
    else {
        s2
    }
}
```



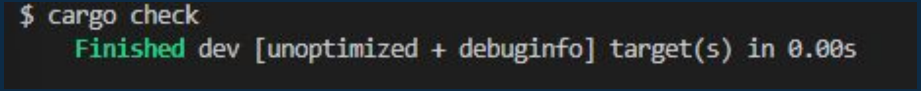
```
$ cargo check
    Checking lifetimetalk v0.1.0 (C:\src\lifetimetalk)
error[E0106]: missing lifetime specifier
  --> src\main.rs:17:35
   |
17 | fn longest(s1: &str, s2: &str) -> &str {
   |                ----      ----     ^ expected named lifetime parameter
   |
   = help: this function's return type contains a borrowed value, but the signature
 does not say whether it is borrowed from `s1` or `s2`
help: consider introducing a named lifetime parameter
   |
17 | fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
   |           ++++      ++           ++          ++

For more information about this error, try `rustc --explain E0106`.
error: could not compile `lifetimetalk` due to previous error
```

Why does this fail?

Using Lifetime Annotations

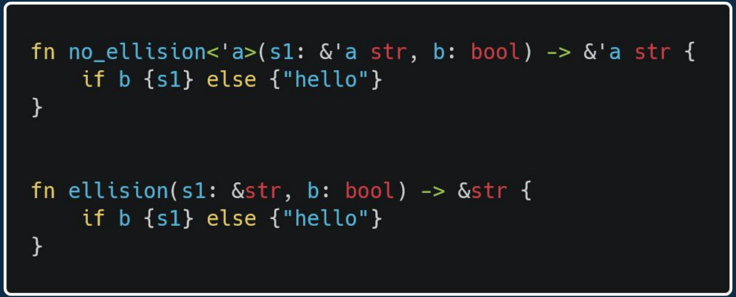


```rust
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```



```
$ cargo check
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
```

Lifetime ellison



```
fn no_ellision<'a>(s1: &'a str, b: bool) -> &'a str {
    if b {s1} else {"hello"}
}


fn ellision(s1: &str, b: bool) -> &str {
    if b {s1} else {"hello"}
}
```

Equivalent functions

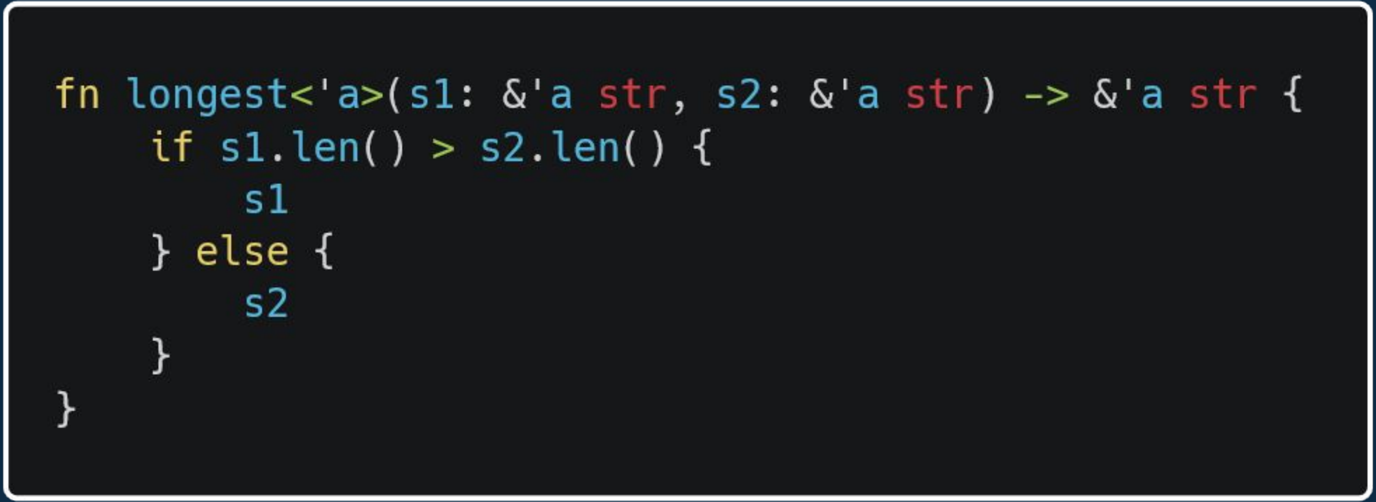
Only need lifetime annotations when Rust can't figure it out itself

Lifetime annotations don't change how long any of the references live.

They only describe relationships to the Rust compiler, which it uses to enforce borrowing rules.

Lifetime annotations are compiler helpers!

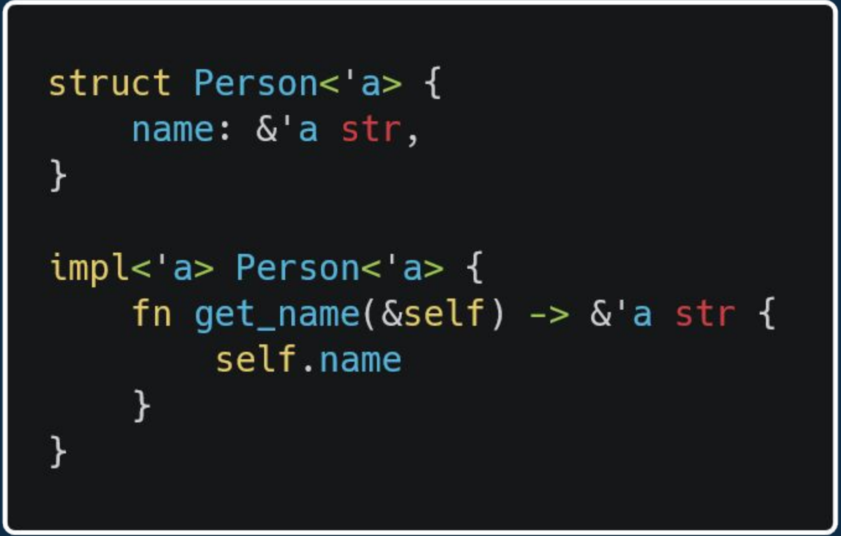
Function Lifetimes



```rust
fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}
```

The returned reference lives (at least) as long as the input references

Struct & Impl Lifetimes

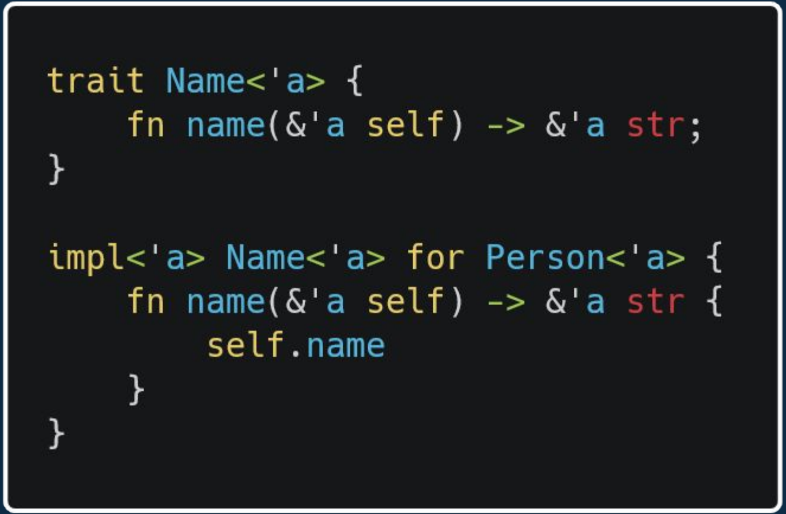


```rust
struct Person<'a> {
    name: &'a str,
}

impl<'a> Person<'a> {
    fn get_name(&self) -> &'a str {
        self.name
    }
}
```

Every instance of Person cannot outlive the reference to the string slice it contains.

Returns a reference that lives as long as the Person struct it is called on

Trait Lifetimes

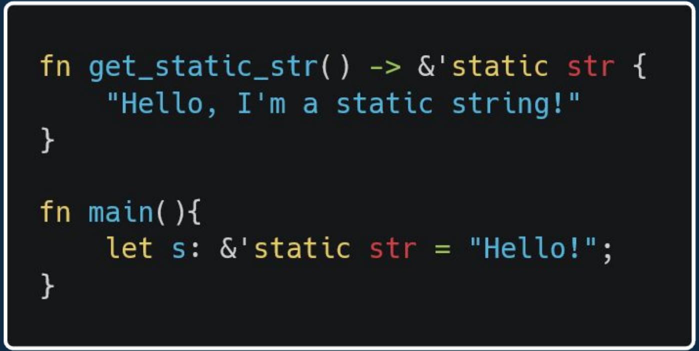


```
trait Name<'a> {
    fn name(&'a self) -> &'a str;
}

impl<'a> Name<'a> for Person<'a> {
    fn name(&'a self) -> &'a str {
        self.name
    }
}
```

Borrows self and returns a string slice with the same lifetime as self.

The returned string slice lifetime will match that of the string slice in person.

Static Lifetimes



```
fn get_static_str() -> &'static str {
    "Hello, I'm a static string!"
}

fn main(){
    let s: &'static str = "Hello!";
}
```

‘static is a keyword

Lives as long as the whole program

Usually static string slices

Tips and tricks for handling lifetimes

3. Lemonade

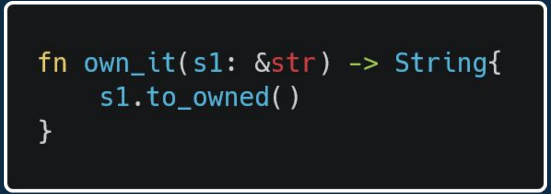
Clone / To Owned

Deep copies the data into a new variable

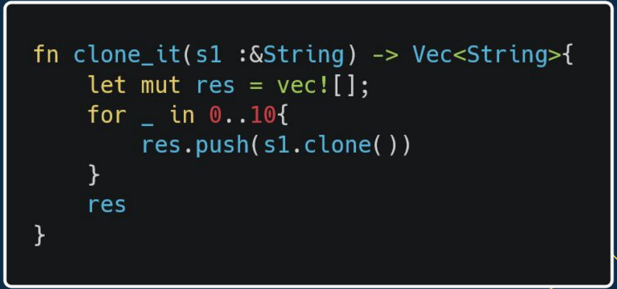
Expensive, but sometimes it is fine

Tips for working with lifetimes

1. Make it work with clone
2. Benchmark
3. Maybe remove the clone



```
fn own_it(s1: &str) -> String{
    s1.to_owned()
}
```



```
fn clone_it(s1 :&String) -> Vec<String>{
    let mut res = vec![];
    for _ in 0..10{
        res.push(s1.clone())
    }
    res
}
```

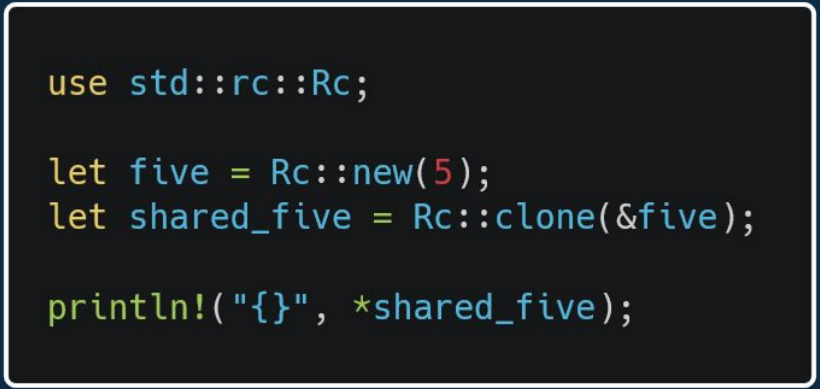
Reference Counting (RC)

Smart Pointer

Tracks number of references, drops memory when there are 0 references.

Single Threaded

Clones the reference, not the memory

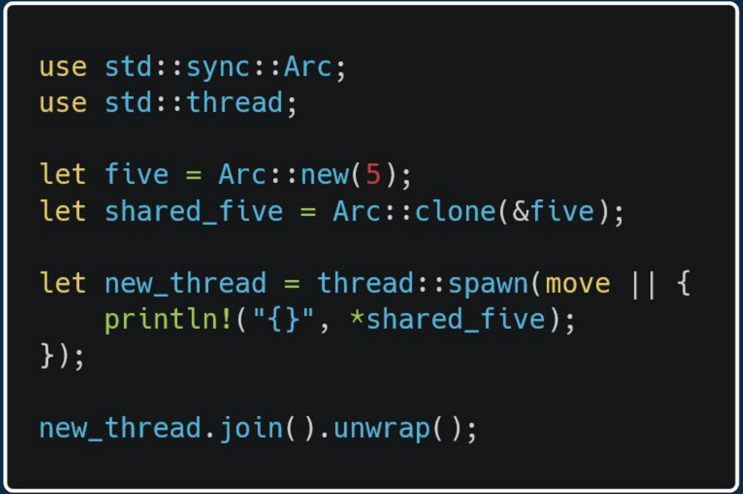


```rust
use std::rc::Rc;

let five = Rc::new(5);
let shared_five = Rc::clone(&five);

println!("{}", *shared_five);
```

Atomic Reference Counting (ARC)

Smart Pointer

Like Rc
Thread safe
More expensive



```rust
use std::sync::Arc;
use std::thread;

let five = Arc::new(5);
let shared_five = Arc::clone(&five);

let new_thread = thread::spawn(move || {
    println!("{}", *shared_five);
});

new_thread.join().unwrap();
```

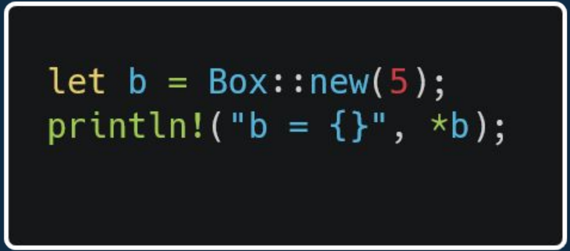
Box

Smart pointer for heap allocation

Deallocates heap memory when box goes out of scope

Use When

- You don't know the exact size at compile tile
- Transfer ownership of large amounts of data without copy
- You only care about a trait, not the specific type

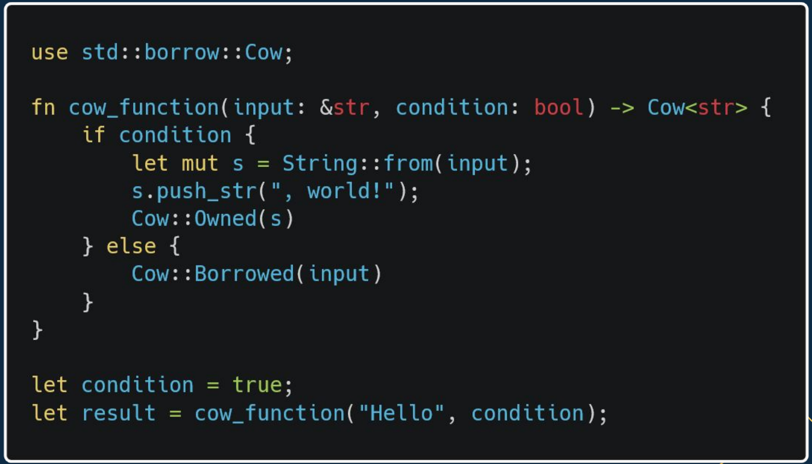


```rust
let b = Box::new(5);
println!("b = {}", *b);
```

Clone On Write (Cow)

Enum, Borrowed/Owned

Use when you don’t know if you need to own or reference at compile time.



```rust
use std::borrow::Cow;

fn cow_function(input: &str, condition: bool) -> Cow<str> {
    if condition {
        let mut s = String::from(input);
        s.push_str(", world!");
        Cow::Owned(s)
    } else {
        Cow::Borrowed(input)
    }
}

let condition = true;
let result = cow_function("Hello", condition);
```

Lifetime Constraints

You can constraint lifetime annotations (just like types)

Read as:
‘a: ‘c = ‘a outlives ‘c

Here both ‘a and ‘b outlives ‘c, so it’s fine to return either.

```rust
fn takes_two_lifetimes<'a, 'b, 'c>(a: &'a str, b: &'b str) -> &'c str
where
    'a: 'c,
    'b: 'c,
{
    if a.len() > b.len() {
        a
    } else {
        b
    }
}

fn main() {
    let c = takes_two_lifetimes("A", "B");
    println!("Hello, {c}!");
}
```

Rust Lifetimes Summarized

Memory Safety
Rust is memory safe, but doesn’t hide memory

Lifetime Annotations
Helps the compiler when it is in doubt

Lifetimes
Not complex, but unfamiliar

Lemonade
There are lots of helper types to manage lifetimes

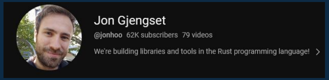
Next Steps

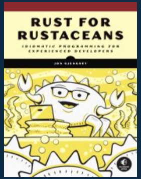
Practise

Write some simple code that uses references

Crust of Rust

Great youtube series by Jon Gjengset


Jon Gjengset
@jonhoo 62K subscribers 79 videos
We're building libraries and tools in the Rust programming language!



Rust for Rustaceans

Great book by Jon Gjengset

Cargo Clippy

Helps you more than any other linter

Thanks & Questions

Contact

PhilipKristoffersen@gmail.com

github.com/PhilipK

Credits

Slides theme adapted from Slidesgo template