Rusty Ownership and the Lifecycle's Stone

•••

Magic is Cool

but needs a degree from Hogwarts

Rust

Low-level language

High-level appearance

Fast or Safe? Pick one

Fast: C, C++

Safe: Every managed lang out there

Garbage Collector

Pros: You do nothing

Cons: You pay for it

Random Unrelated Logo

Rust is weird

Does not have a garbage collector

Does not let you manage memory

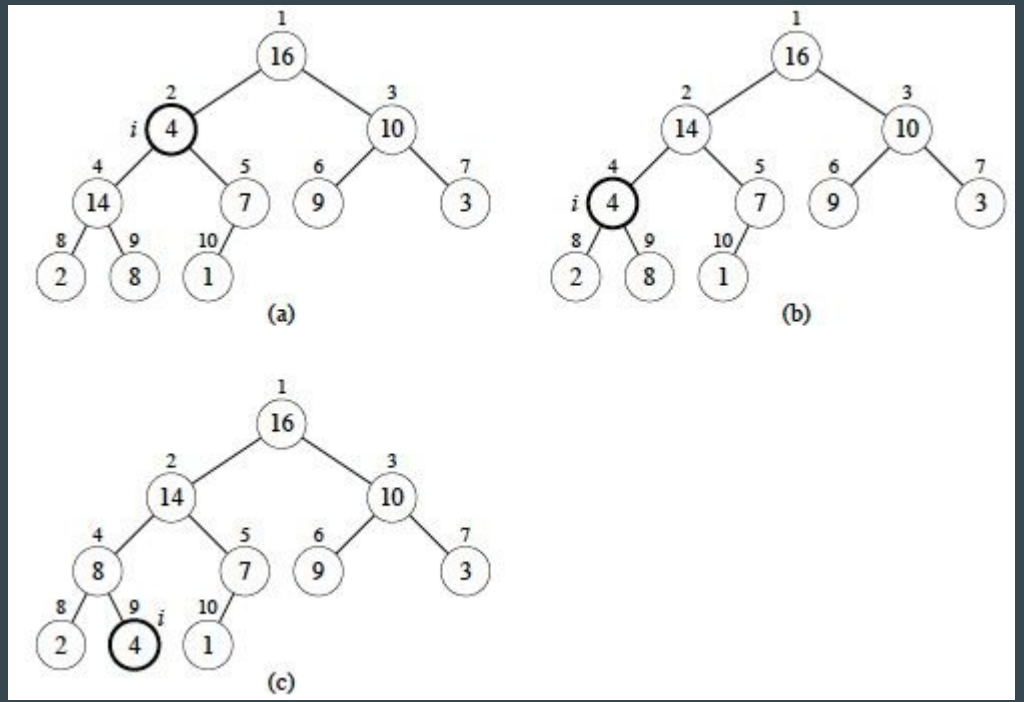
Memory Basics

The Stack

- Last-in, First-out
- Fixed length data

The Heap

What is the heap?



The Heap

*Meaning of **heap** in English*

heap

noun [C]

UK /hi:p/ US /hi:p/

C2

an untidy pile or mass of things:

- *a heap of clothes/rubbish*

Sharon White/Photographer's Choice/GettyImages

The Heap

The Heap

The GOOD part:

- Do whatever you want

The BAD parts:

- Keep a mapping between the parts of code and the data they use of the heap
- Minimize data duplication
- Cleanup unused data

Running a tight ship

Ain’t nobody got time fo ‘dat

We are AGILE!!11111!!!1!!

Ownership

- Easy but far reaching concept
- Innovation introduces weirdness

Ownership of what?
Ownership of values

Who owns values?
Variables own values

Rust's Ownership Commandments

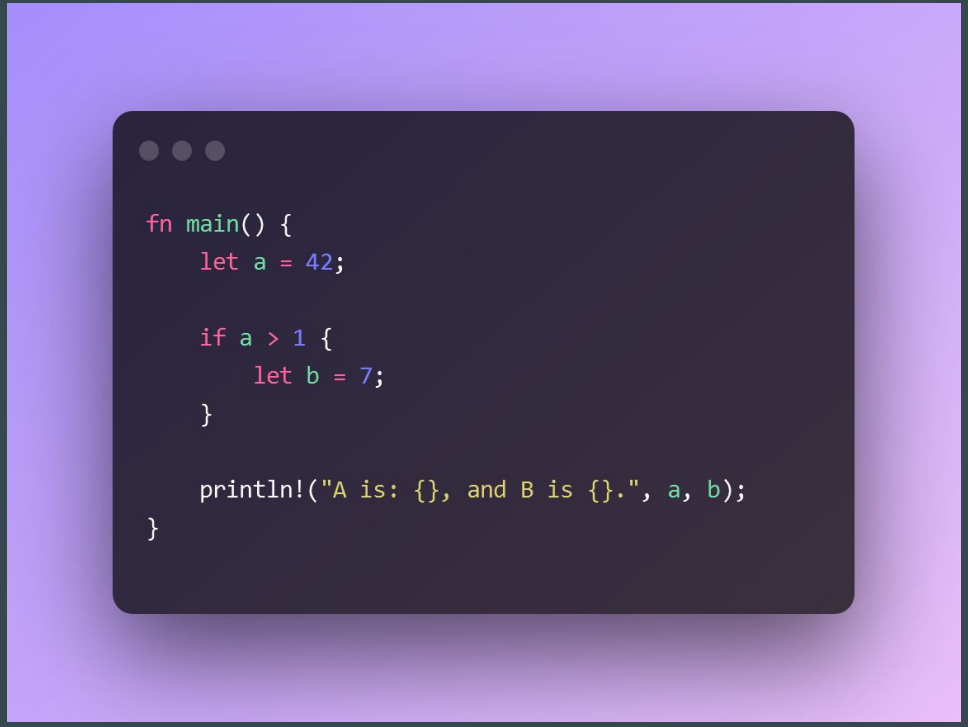
1. Thou shall not have a value without an owner.

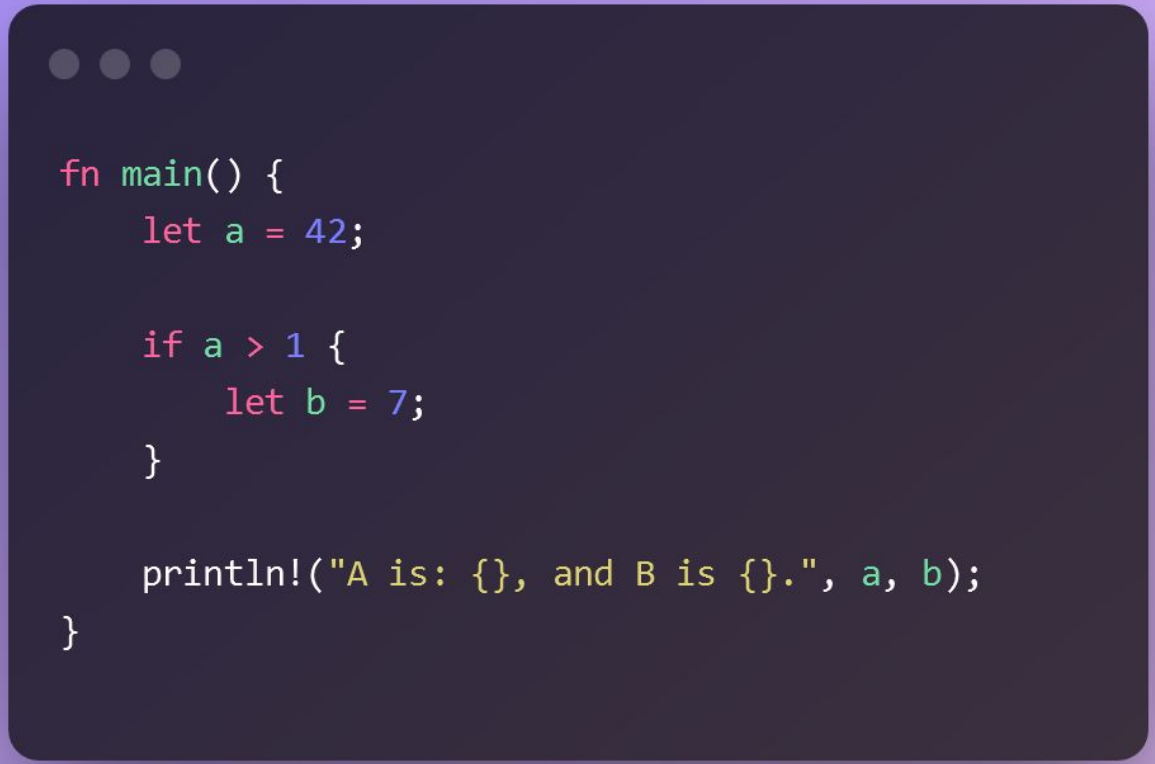
2. Thou shall not have multiple owners for a single value.

3. Thou shall sacrifice the value in a pyre once its owner's life has no scope left.

Out of Scope? Out of Memory

What is the problem with this code?



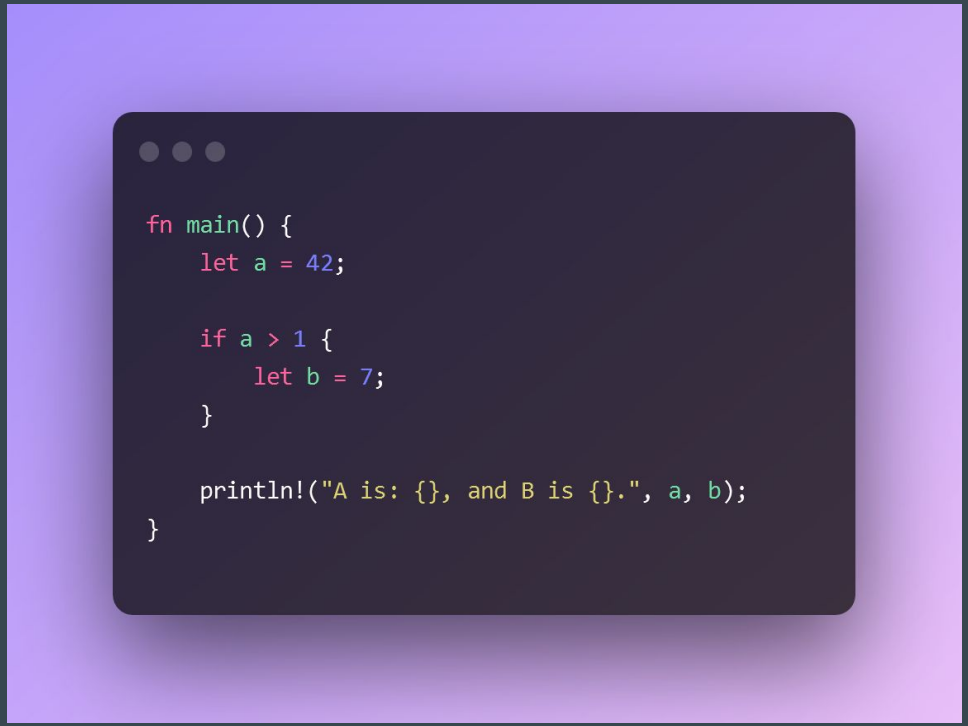


```rust
fn main() {
    let a = 42;

    if a > 1 {
        let b = 7;
    }

    println!("A is: {}, and B is {}.", a, b);
}
```

Out of Scope? Out of Memory

It does not compile!

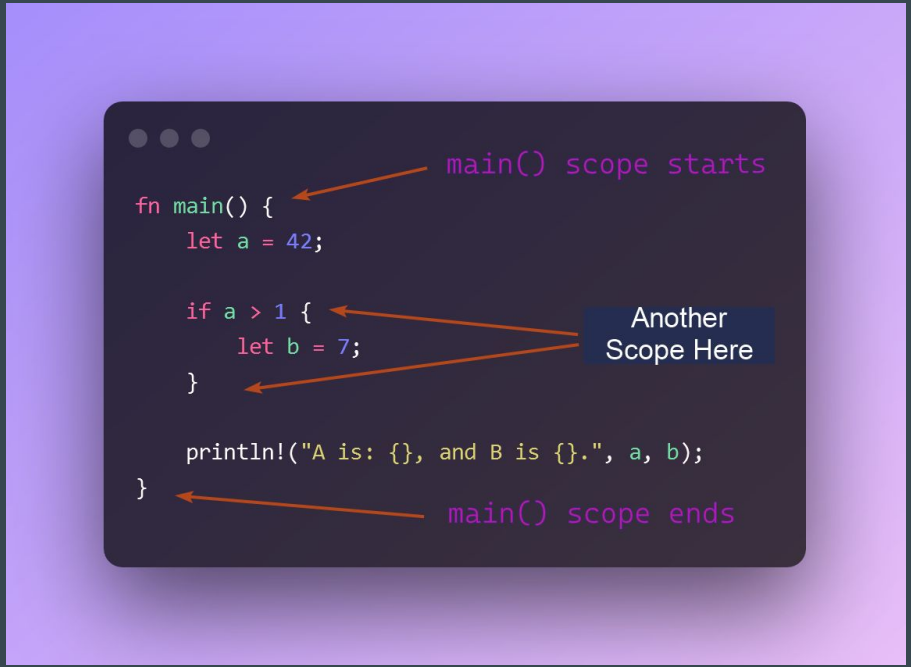
cannot find value `b` in this scope

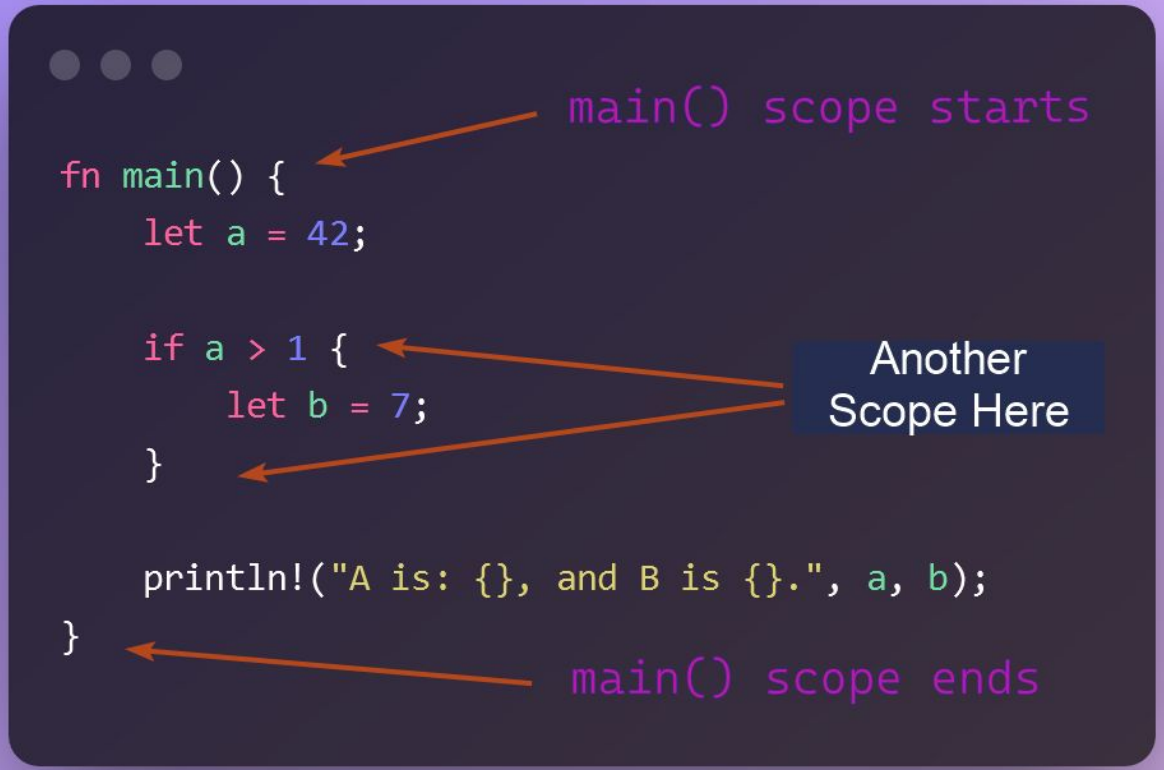


```rust
fn main() {
    let a = 42;

    if a > 1 {
        let b = 7;
    }

    println!("A is: {}, and B is {}.", a, b);
}
```

Rust the Butler drops the plate

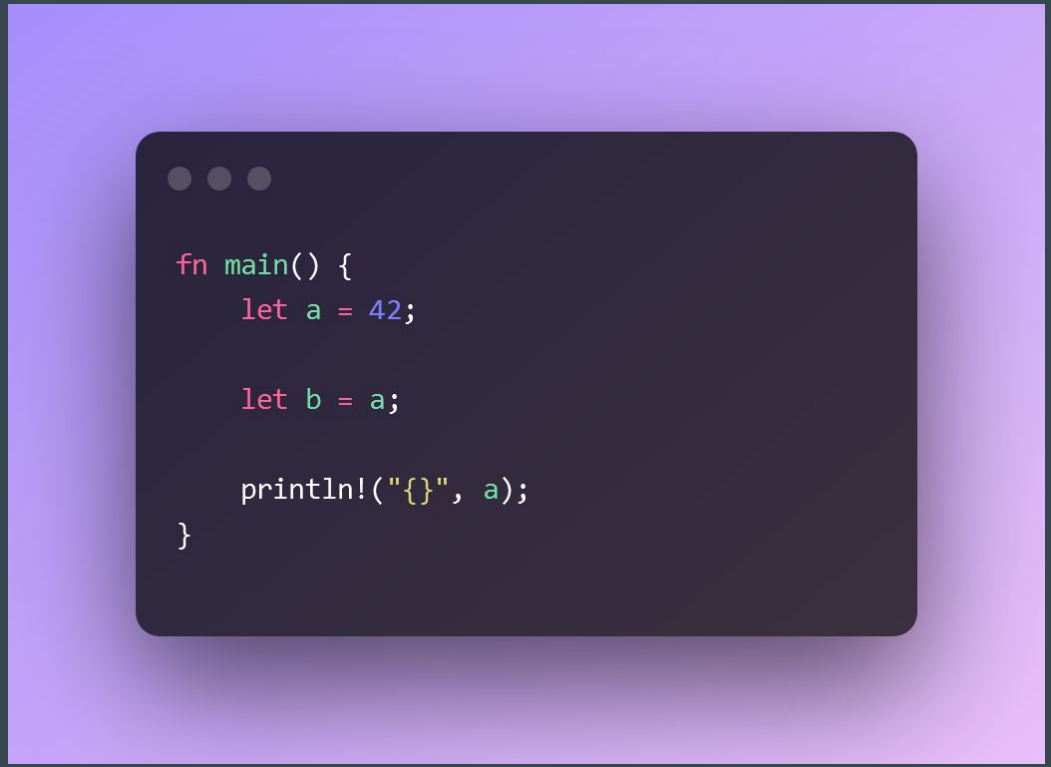
Trivial stuff deferred to the compiler



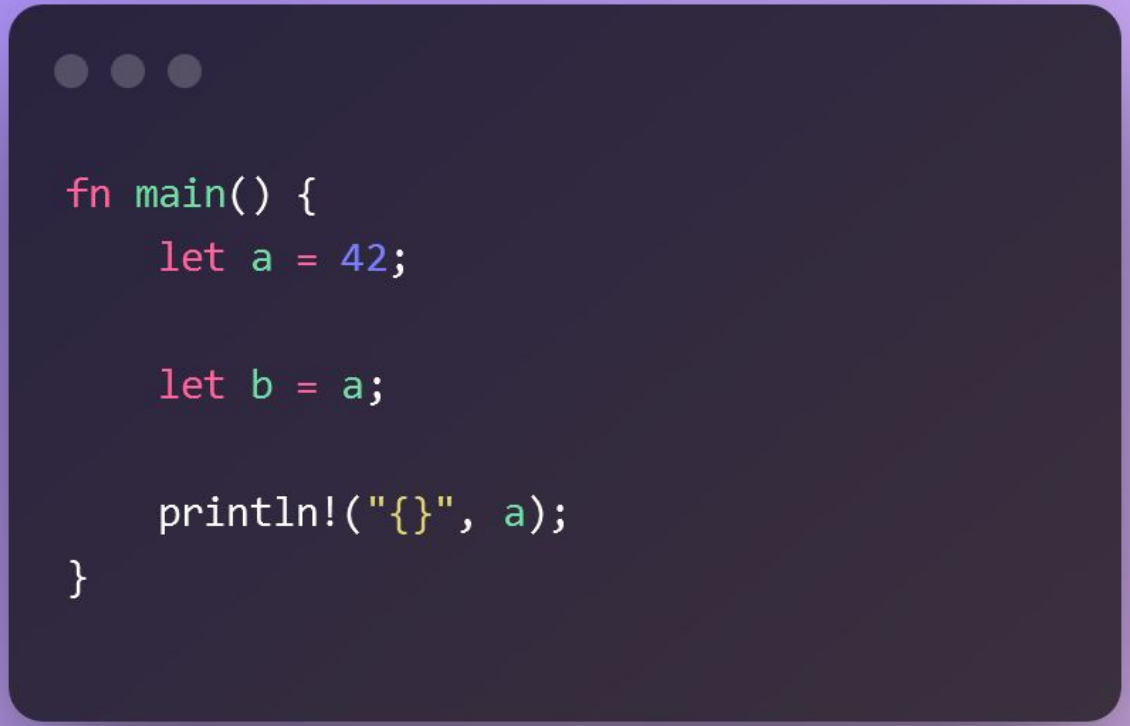

main() scope starts
fn main() {
let a = 42;
if a > 1 {
let b = 7;
}
Another
Scope Here
println!("A is: {}, and B is {}.", a, b);
}
main() scope ends

The burdens of Ownership

What's the problem with this code?



```
fn main() {
    let a = 42;

    let b = a;

    println!("{}", a);
}
```

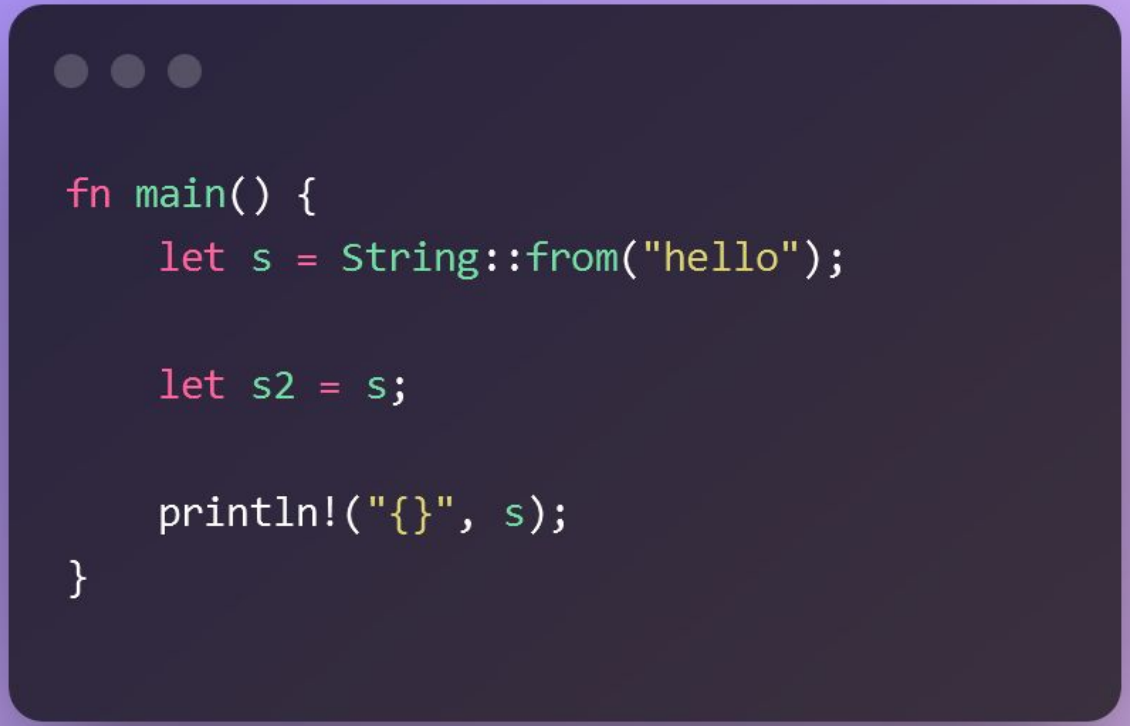

fn main() {
let a = 42;
let b = a;
println!("{}", a);
}

It was a trick question

42 is fixed length, Rust used the Stack, nothing to see here

The burdens of Ownership

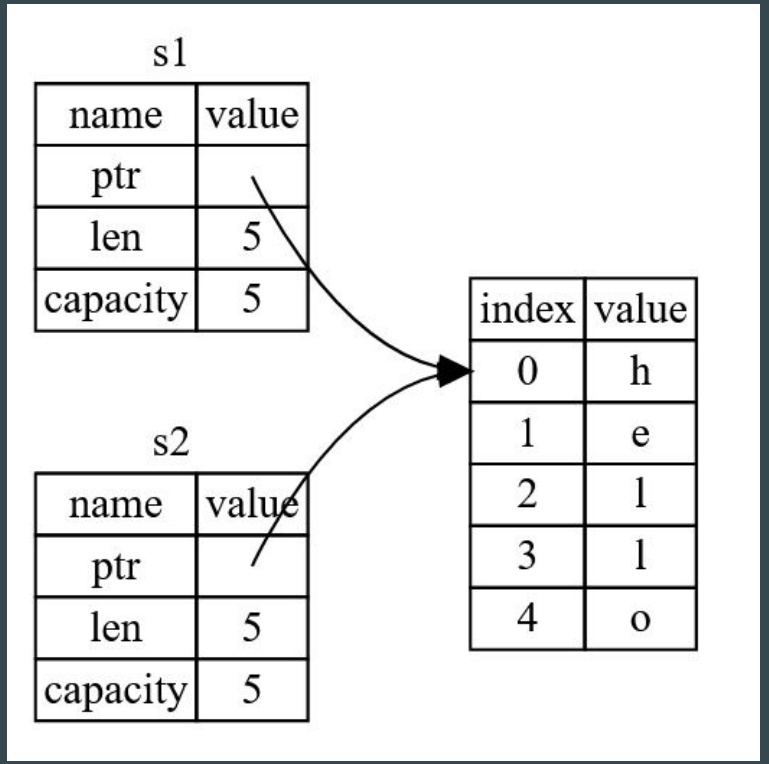
Strings require dynamic allocation
Strings go to the Heap



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

    let s2 = s;

    println!("{}", s);
}
```

The burdens of Ownership

Copy is by reference



The burdens of Ownership

Double burger has undefined taste

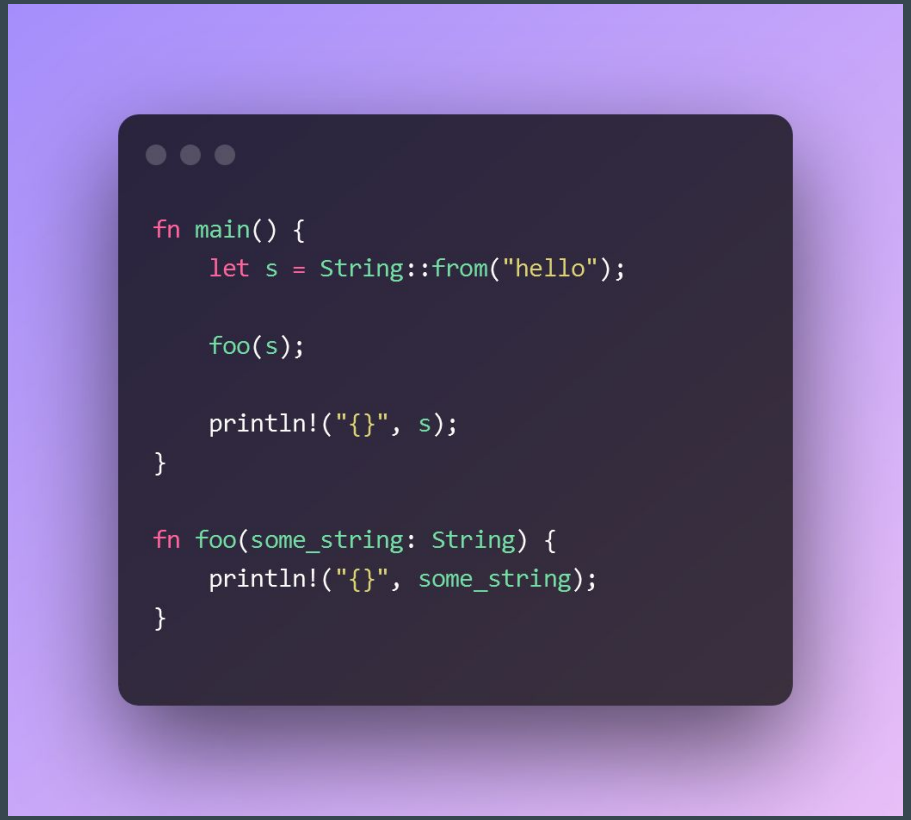
The burdens of Ownership

This doesn’t compile!

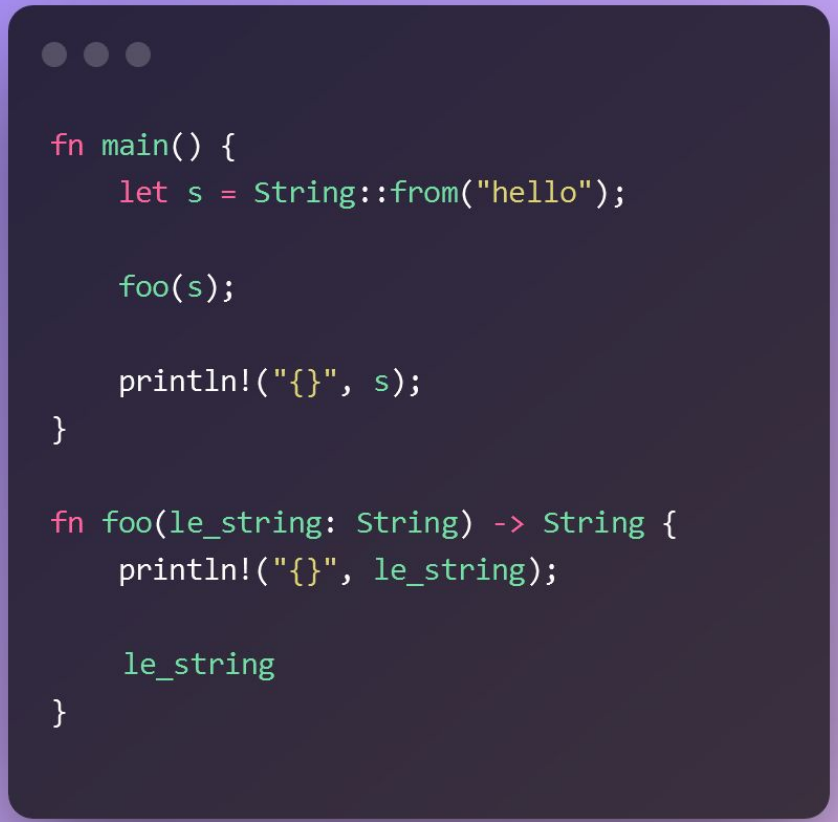
Remember that we can only have a single owner.

Passing a variable to a function means moving ownership

And it doesn’t come back on its own



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

    foo(s);

    println!("{}", s);
}

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

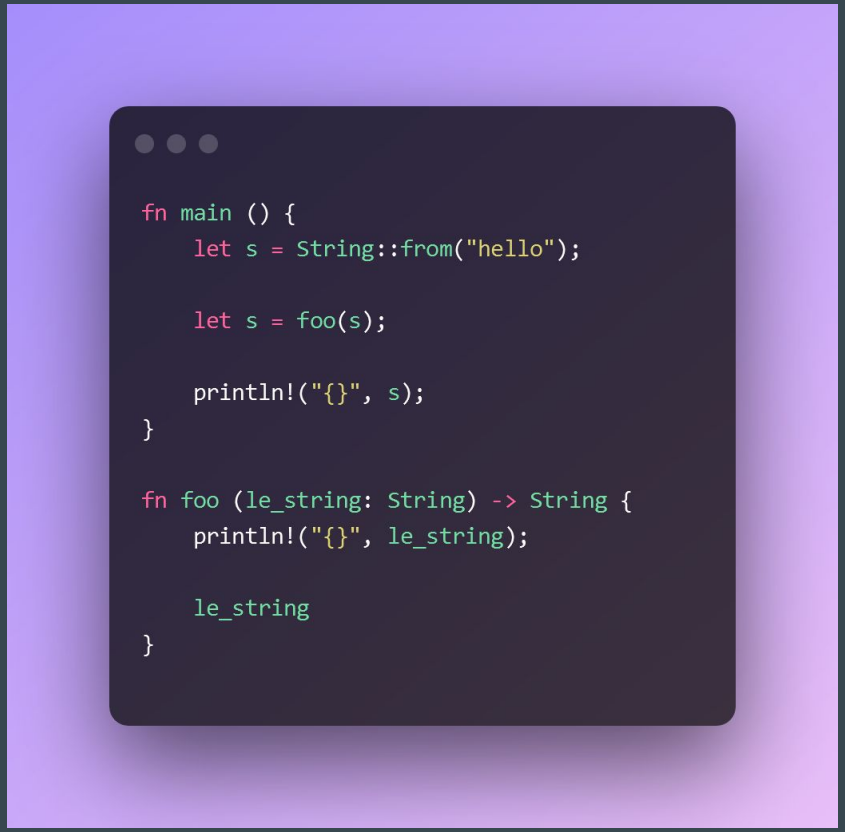


```rust
fn main() {
    let s = String::from("hello");

    foo(s);

    println!("{}", s);
}

fn foo(le_string: String) -> String {
    println!("{}", le_string);

    le_string
}
```

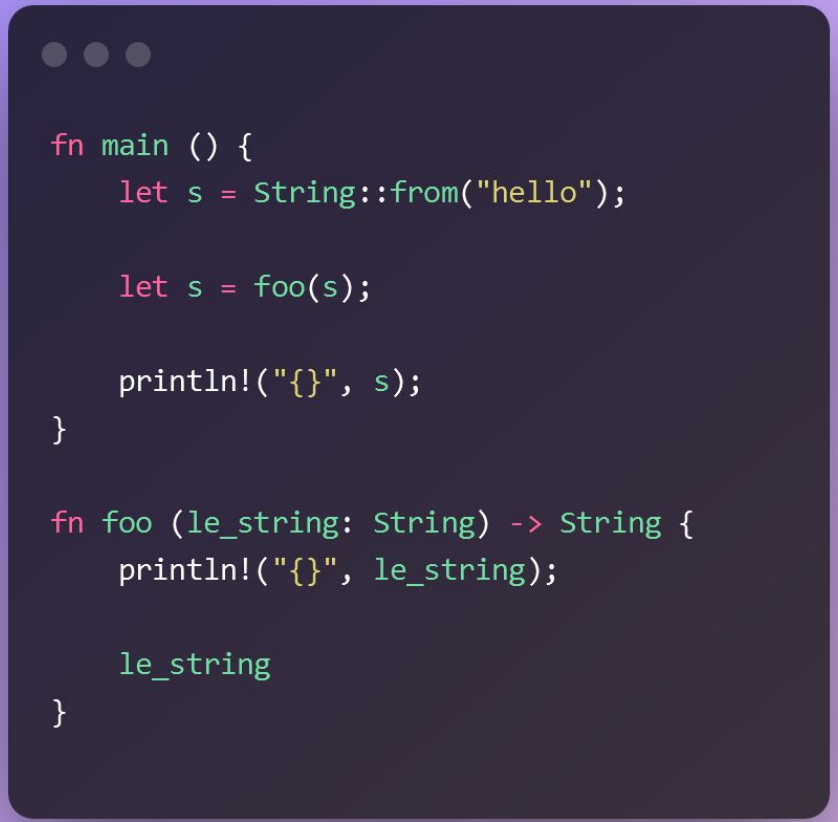
The naive solution

Why bother RTFM?

Just solve it in a completely unscalable way!



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

    let s = foo(s);

    println!("{}", s);
}

fn foo (le_string: String) -> String {
    println!("{}", le_string);

    le_string
}
```

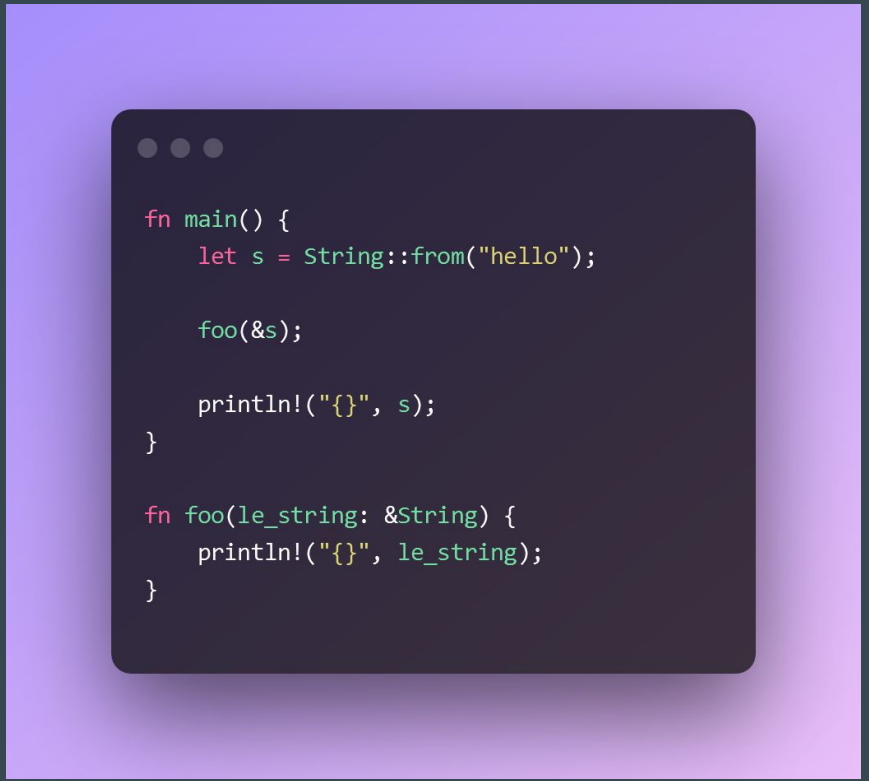


```rust
fn main () {
    let s = String::from("hello");

    let s = foo(s);

    println!("{}", s);
}

fn foo (le_string: String) -> String {
    println!("{}", le_string);

    le_string
}
```

Enter Borrowing

Temporary transfer of ownership

With benefits



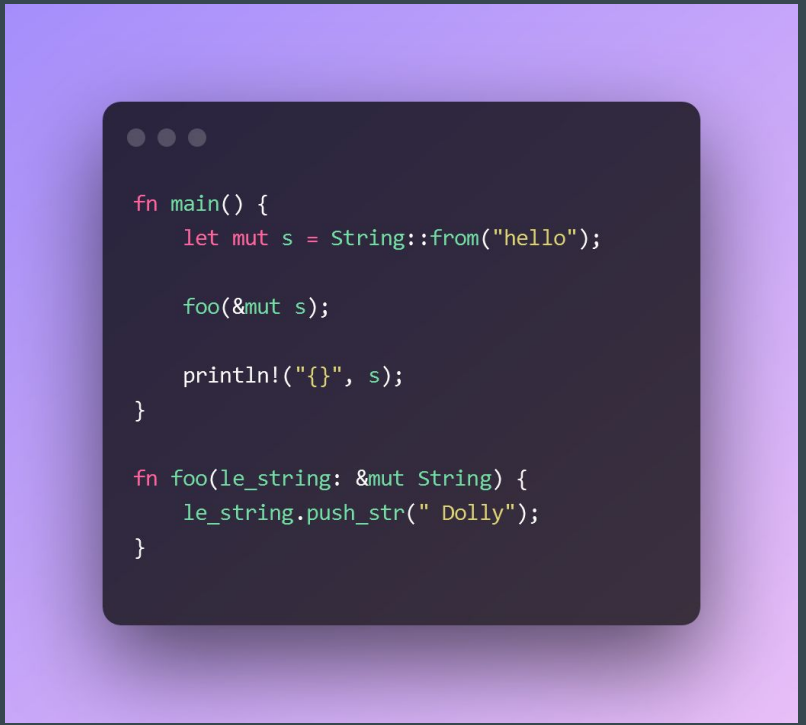
```rust
fn main() {
    let s = String::from("hello");

    foo(&s);

    println!("{}", s);
}

fn foo(le_string: &String) {
    println!("{}", le_string);
}
```

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

    foo(&s);

    println!("{}", s);
}

fn foo(le_string: &String) {
    println!("{}", le_string);
}
```

References

Immutable by default

Just add mut



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

    foo(&mut s);

    println!("{}", s);
}

fn foo(le_string: &mut String) {
    le_string.push_str(" Dolly");
}
```

The rules of Borrowing

1. One mutable borrow at a time - or any number of immutable ones.
2. No Invalid References

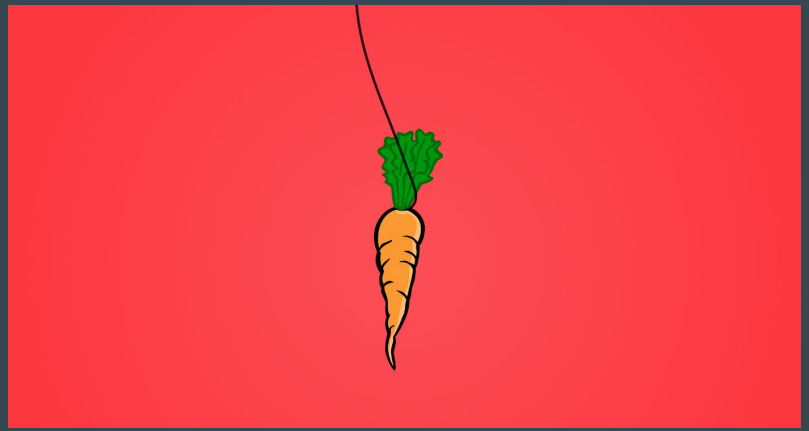

fn main() {
let mut s = String::from("hello");
let s1 = &mut s;
let s2 = &mut s;
println!("{}", s1)
}


fn main() {
let mut s = String::from("hello");
let s1 = &s;
let s2 = &s;
println!("{}", s1)
}


fn main() {
let mut s = String::from("hello");
let s1 = &s;
let s2 = &mut s;
println!("{}", s1)
}

Dangling pointers

A step towards nothingness



```rust
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");

    &s
}
```

Lifetimes

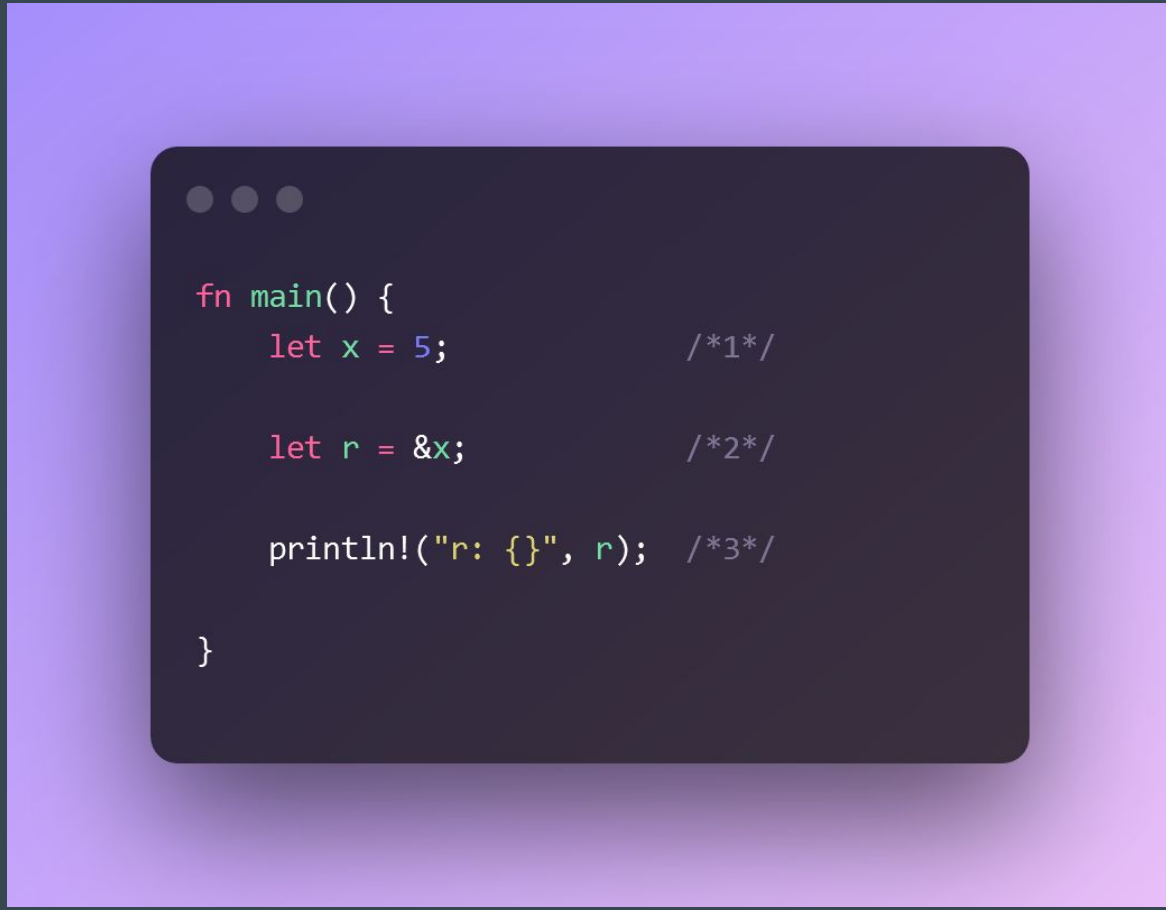
Ownership and Borrowing are fine, they don't cover all cases.

Variables and "loans" are LIVE or ... dead, depending on the code block.

Good living

x : { 1, 2 , 3}

r : { 2, 3}



Lifetimes

x : { 3, 4 }

r : { 1, ..., 6}

```
fn main() {
    let r;                  /*1*/

    {                       /*2*/
        let x = 5;          /*3*/
        r = &x;             /*4*/
    }                       /*5*/

    println!("r: {}", r);   /*6*/
}
```

Lifetimes

Sometimes, the compiler has no way to calculate the lifetimes in a complete way.

```rust
fn main() {
    let x = String::from("hello");
    let y = String::from("ciao");

    let z = max(&x, &y);

    println!("{}", z);
}

fn max(s1: &String, s2: &String) -> &String {
    if s1.len() > s2.len() {
        return s1
    }

    s2
}
```

Lifetime annotation

We need a way to enhance the function signature

It’s an ugly looking way - thankfully, don’t have to use it much

```rust
fn main() {
    let x = String::from("hello");
    let y = String::from("ciao");

    let z = max(&x, &y);

    println!("{}", z);
}

fn max<'a>(s1: &'a String, s2: &'a String) -> &'a String {
    if s1.len() > s2.len() {
        return s1
    }

    s2
}
```

Lifetimes

Now the compiler can be sure about borrow checking.

Lifetime annotation does NOT modify lifetime. It’s just a contract.

```rust
fn main() {
    let x = String::from("hello");
    let z;

    {
        let y = String::from("ciao");
        z = max(&x, &y);
    }

    println!("{}", z);
}

fn max<'a>(s1: &'a String, s2: &'a String) -> &'a String {
    if s1.len() > s2.len() {
        return s1
    }

    s2
}
```

```
error[E0597]: `y` does not live long enough
  --> src/main.rs:7:21
   |
7  |         z = max(&x, &y);
   |                     ^^ borrowed value does not live long enough
8  |     }
   |     - `y` dropped here while still borrowed
9  |
10 |     println!("{}", z);
   |                    - borrow later used here
```

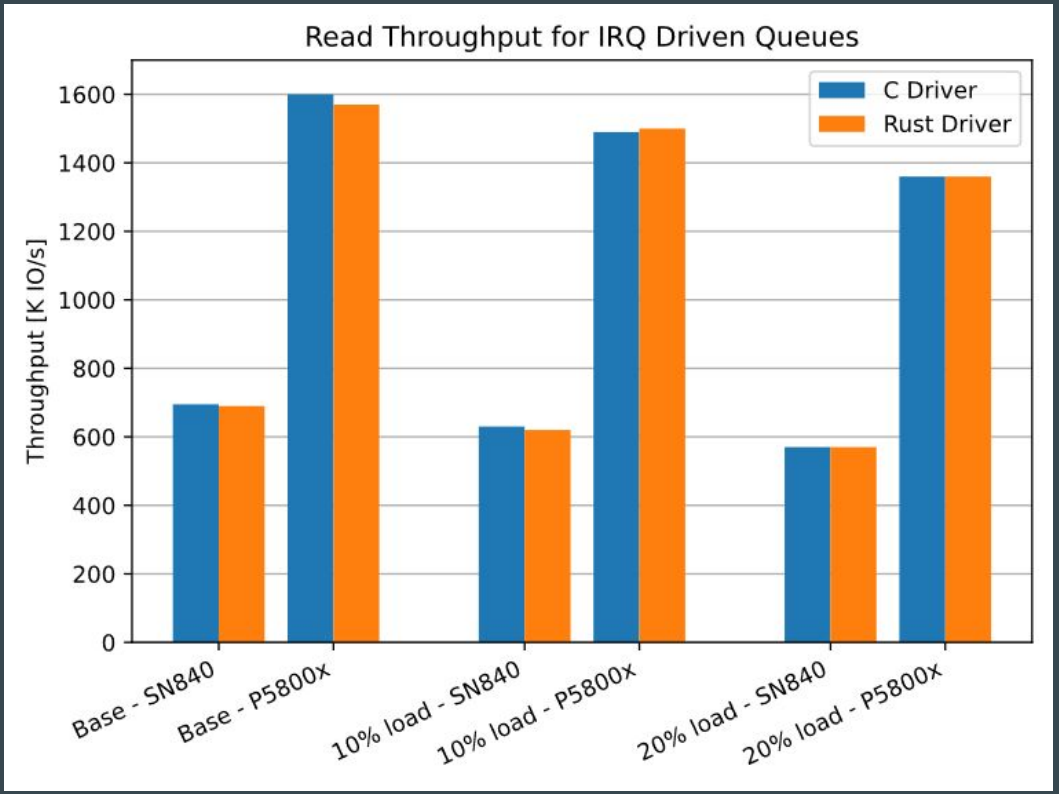
Outro

Rust adds an innovative solution to the dilemma

The BAD: It’s like learning to ride the bicycle

The GOOD: The compiler is actually helpful

Outro - Rust is the future, today



Rust lands in Linux Kernel 6.1

Outro - Rust is the future, today

Web3

The Rust Programming Language

Outro - Your friendly neighborhood coder

Follow/Connect on LinkedIn - Chat on Slack

FSF members may use a GNU/Linux laptop for QR Scanning

LinkedIn