# Welcome!

Install the Rust toolchain, so you can follow along with the workshop:

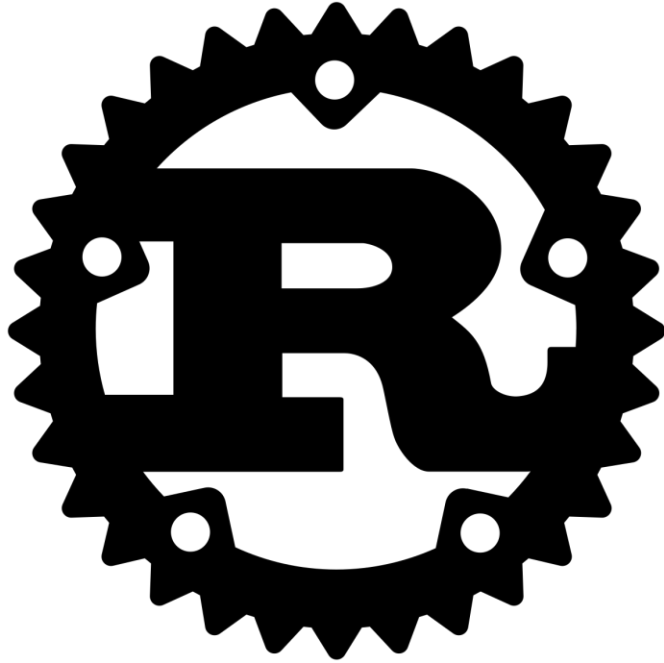https://www.rust-lang.org/learn/get-started

Download some example code, and these slides:

https://github.com/neon64/rust_workshop_materials

Please feel free to type in the chat / unmute your mic to ask questions at any time!

- this workshop is meant to be interactive if you so prefer! ☺

# Introduction to Rust

CHRIS CHAMBERLAIN – CISSA 2021

# Overview

# Why learn Rust?

- Broaden your horizon of what is possible in a language
  - A compiler that helps you write correct and efficient code

- Beginning to be adopted widely across industry:
  - Dropbox (new sync engine),
  - Cloudflare (core edge logic)
  - npm Inc.
  - Microsoft
  - Google (Fuschia, likely other projects)
  - Xero,
  - Atlassian (service analyzing source code)
  - Have seem some AWS job listings mention it

- Not prolific as e.g.: Node.js or React,
  - [elitist view] Rust really shines to write *difficult* software, rather than churning out generic web apps

# History of Rust

- First emerged in 2010, created originally by Graydon Hoare from Mozilla Research
  - Some influences: Cyclone, SML/OCaml, C++, Haskell

- Two big projects driving language evolution: Rust compiler, Servo browser engine
  - Many components (WebRender, Stylo) – now in Firefox

- 1.0 release in 2015

- Previously: many developers employed by Mozilla, with relatively autonomous governance

- In 2021, Rust Foundation was announced, taking ownership of the project.
  - Founding Partners: AWS, Huawei, Google, Microsoft, and Mozilla

# Key Characteristics

```rust
fn parse_value(&mut self) -> Value {
    match self.next_char() {
        '0'...'9' => self.parse_length(),
        '#' => self.parse_color(),
        _ => Value::Keyword(self.parse_identifier())
    }
}

fn parse_length(&mut self) -> Value {
    Value::Length(self.parse_float(), self.parse_unit())
}

fn parse_float(&mut self) -> f32 {
    let s = self.consume_while(|c| match c {
        '0'...'9' | '.' => true,
        _ => false
    });
    s.parse().unwrap()
}
```

Some example Rust code out in the wild.
https://github.com/mbrubeck/robinson/blob/master/src/css.rs



## Defining features

abstractions without overhead

memory safety without garbage collection

concurrency without data races

## Rust compared to other languages

Low-level, can run on bare metal without any runtime, *like C*

Strong static typing, lots of rich type system features, *like Haskell*

Zero-cost abstractions, a relatively complex language, *like C++*

Memory-safe by default, protects against segfaults/null pointers, *like Java/Python/JavaScript/[any other managed language],* but **also** against data races.

# Rust "as a better C"

Rust is a *systems programming language*, in the sense of C, C++

• Control over memory allocations, no garbage collector, minimal runtime

Rust is very fast

• Compiles through the LLVM compiler backend (same as clang),
  • performance more or less equal to C/C++,
  • in some cases, stronger ownership guarantees allow optimizer to beat C/C++
  • in the real world, fearless concurrency can lead to faster code on multithreaded systems

# Demo time!

Goals:

- Download/install Rust, Rust Analyzer

- Hello-world application, variables/functions/structs etc...
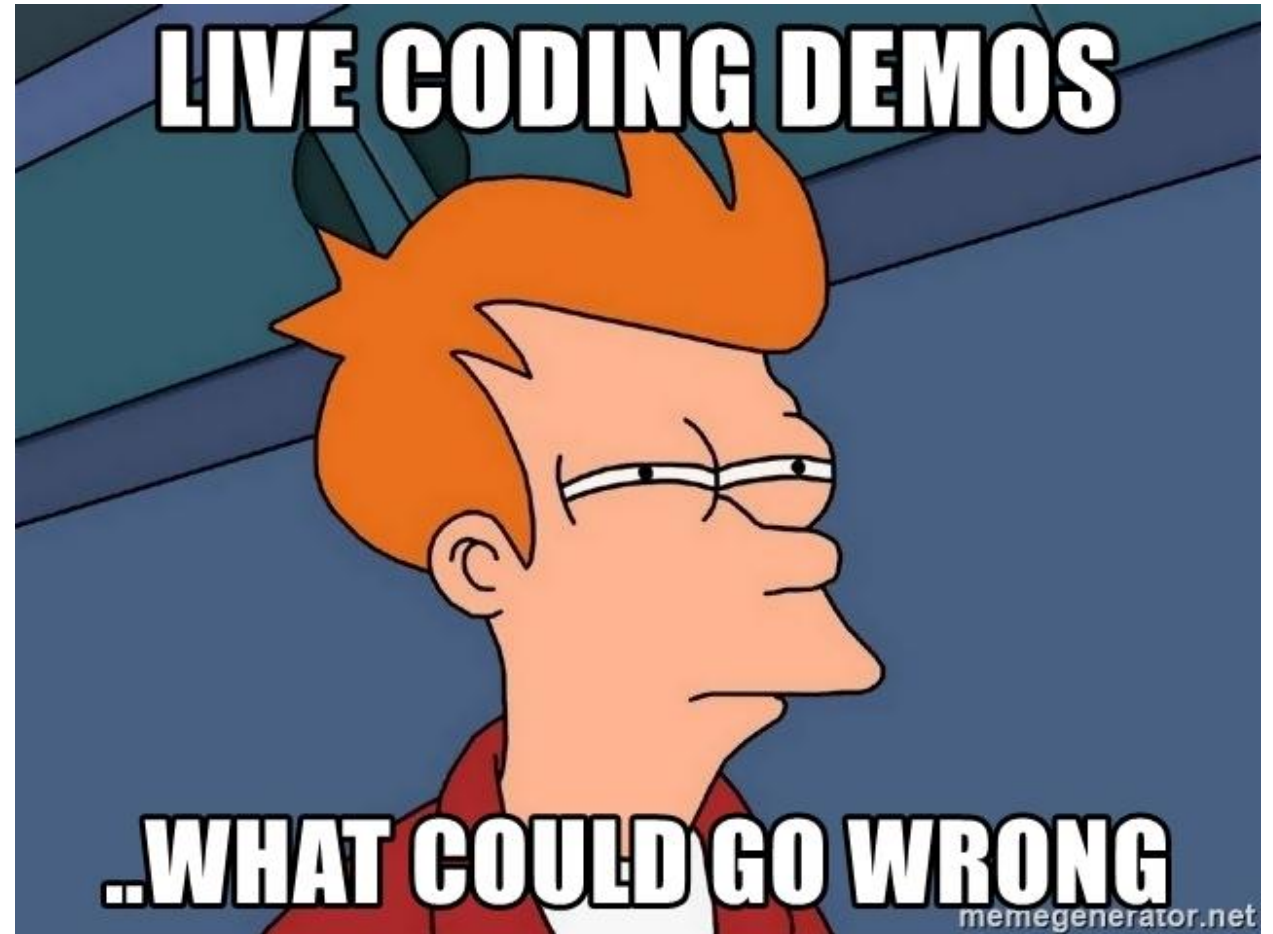
# What is Rust trying to protect against?

The motivation for ownership/borrowing.

# Pitfalls of C/C++

- Null pointers / uninitialized values

- Dangling pointers / use-after-frees / double-frees

- Buffer overflows

- Data races

# Consequences of undefined behaviour

- Program crashes (SIGSEGV)

- Bad data, corruption

- The compiler is allowed to assume anything about your program now…

# Consequences of undefined behaviour

*The compiler is allowed to assume anything about your program now...*

Here are some references of unexpected results after optimisations are applied, if you are curious:

- https://en.cppreference.com/w/cpp/language/ub

- http://blog.llvm.org/2011/05/what-every-c-programmer-should-know_14.html

# Dangling pointer access in C

```c
typedef struct {
    int x;
    int y;
} useful_data_t;

int main() {
    useful_data_t* data = malloc(sizeof(useful_data_t));
    data->x = 3;
    data->y = 5;

    int *dangling = &data->x;

    // In the real world, this call to `free` could be
    //  hidden away in some other function.
    free(data);

    printf("should print zero: %d\n", *dangling);

    return 0;
}
```

# A quote from the Rust Book

*"In languages with a garbage collector (GC), the GC keeps track and cleans up memory that isn't being used anymore, and we don't need to think about it. Without a GC, it's our responsibility to identify when memory is no longer being used and call code to explicitly return it, just as we did to request it.*

*Doing this correctly has historically been a difficult programming problem. If we forget, we'll waste memory. If we do it too early, we'll have an invalid variable. If we do it twice, that's a bug too. We need to pair exactly one allocate with exactly one free."*

# Ownership

The claim is that ownership solves all three of these desirable properties.
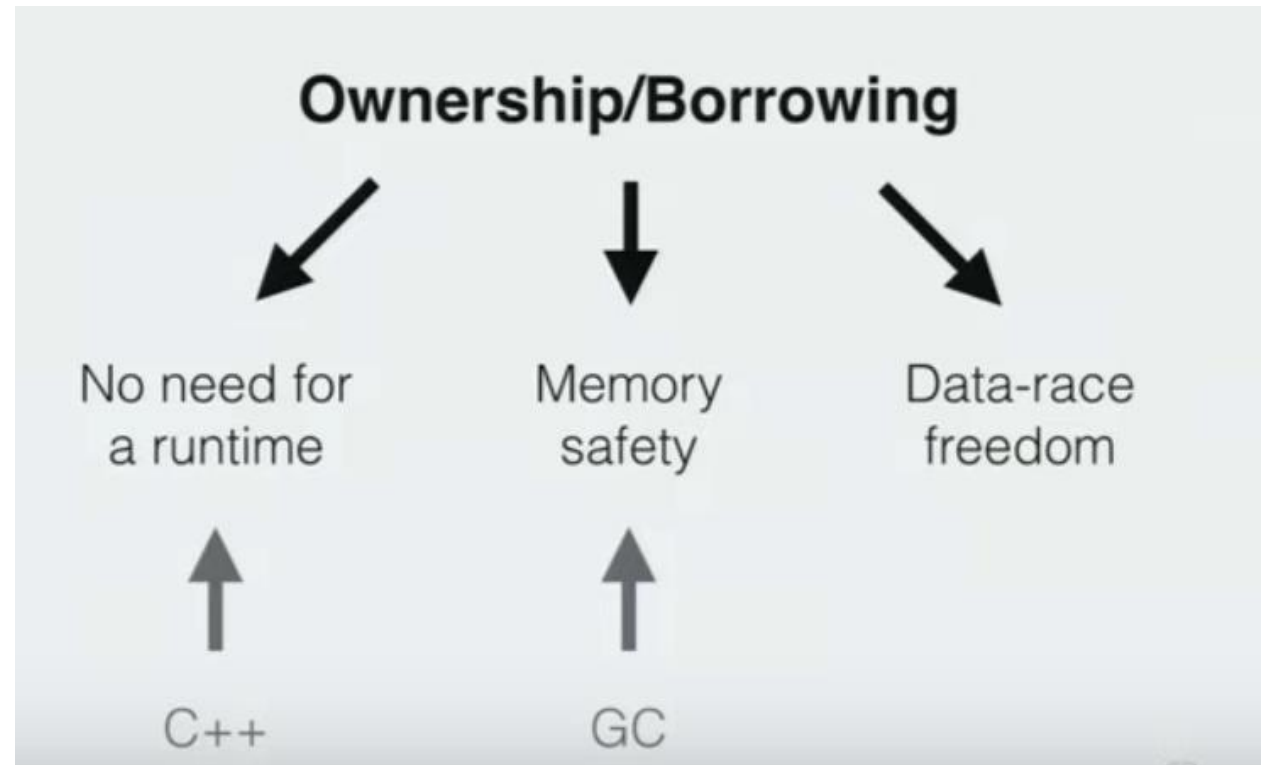
# Three rules of **ownership**

Three rules:

- **Every** value in Rust has a variable that's called its *owner*.

- There can only be **one** owner at a time.

- When the owner goes out of scope, the value will be dropped.

If you're familiar with modern C++, this is strikingly similar to C++'s
notion of *Resource Acquisition Is Initialization (RAII),*
except more strictly enforced.

# Demo time!

Goals:

- Demonstrate ownership.

- Motivate the need for something more flexible (borrowing)

# Ownership: an example
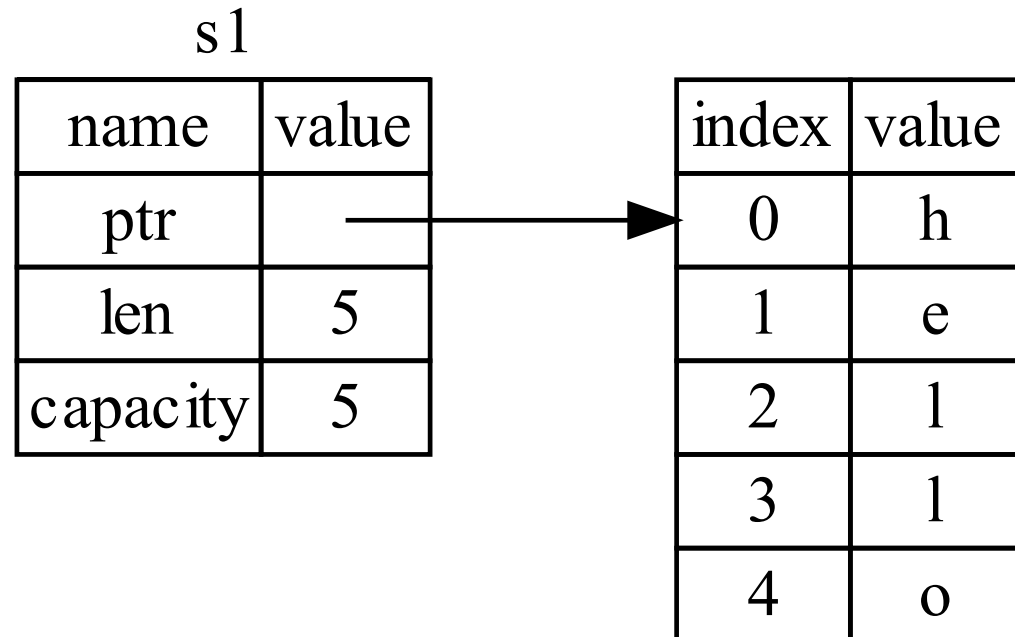
```rust
let s1 = String::from("hello");
let s2 = s1;
```

s1

| name | value |
|------|-------|
| ptr | |
| len | 5 |
| capacity | 5 |

| index | value |
|-------|-------|
| 0 | h |
| 1 | e |
| 2 | l |
| 3 | l |
| 4 | o |

# Ownership

```
let s1 = String::from("hello");
let s2 = s1;
```

s1

| name | value |
|------|-------|
| ptr | |
| len | 5 |
| capacity | 5 |

s2

| name | value |
|------|-------|
| ptr | |
| len | 5 |
| capacity | 5 |

| index | value |
|-------|-------|
| 0 | h |
| 1 | e |
| 2 | l |
| 3 | l |
| 4 | o |

# Ownership

```
let s1 = String::from("hello");
let s2 = s1;
```

**s1**

| name | value |
|------|-------|
| ptr | |
| len | 5 |
| capacity | 5 |

| index | value |
|-------|-------|
| 0 | h |
| 1 | e |
| 2 | l |
| 3 | l |
| 4 | o |

**s2**

| name | value |
|------|-------|
| ptr | |
| len | 5 |
| capacity | 5 |

| index | value |
|-------|-------|
| 0 | h |
| 1 | e |
| 2 | l |
| 3 | l |
| 4 | o |

# Ownership

```rust
let s1 = String::from("hello");
let s2 = s1;
```
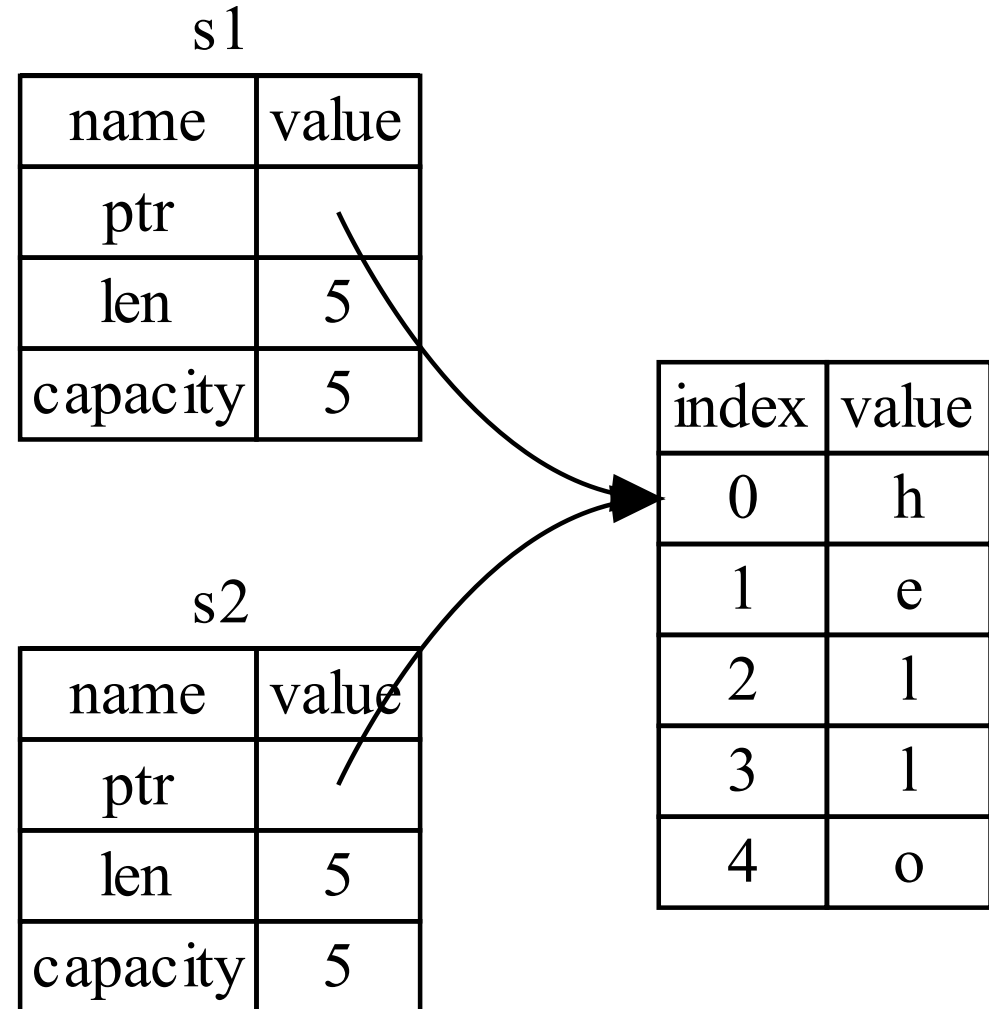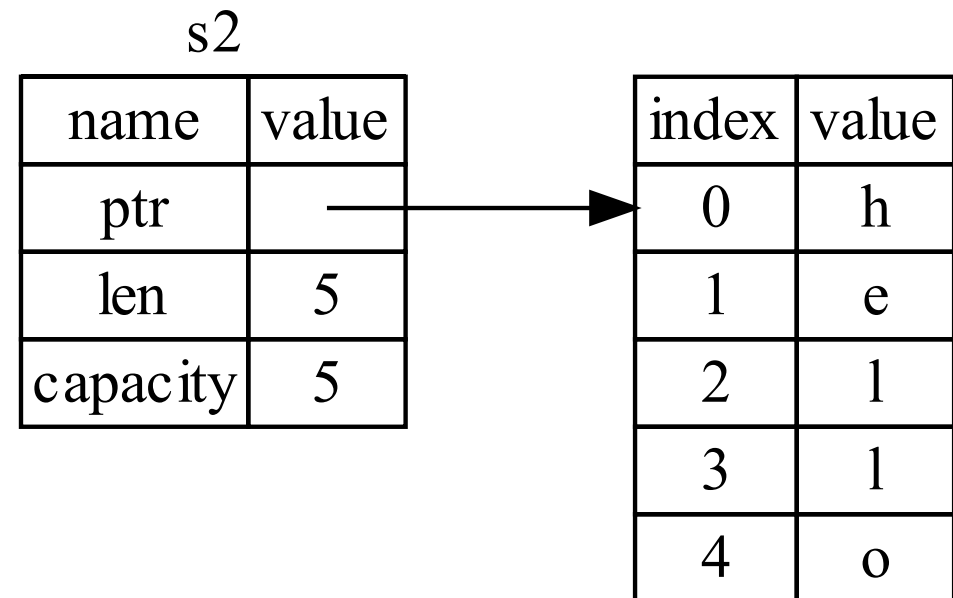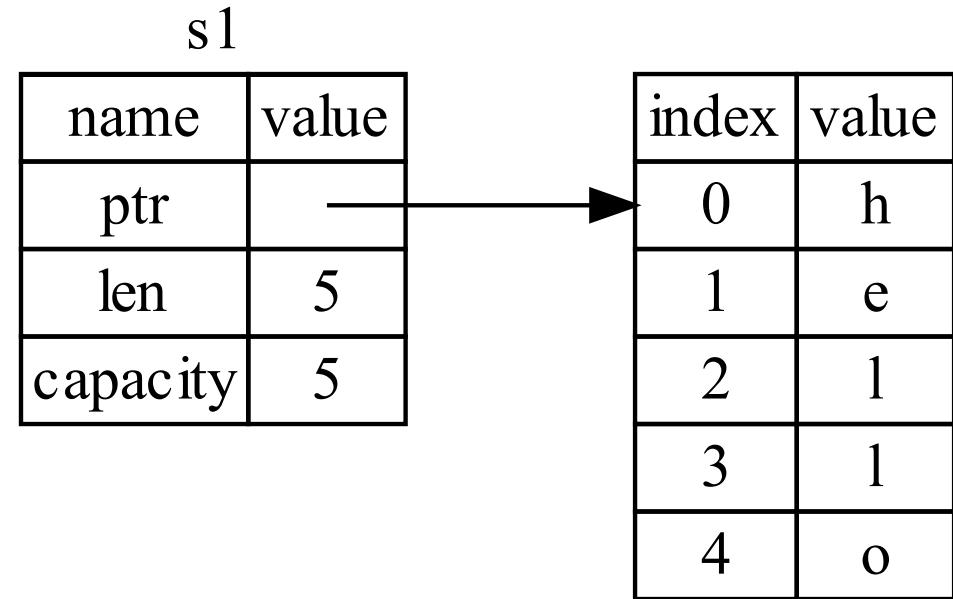
| s1 | |
|---|---|
| name | value |
| ptr | |
| len | 5 |
| capacity | 5 |

| s2 | |
|---|---|
| name | value |
| ptr | |
| len | 5 |
| capacity | 5 |

| index | value |
|---|---|
| 0 | h |
| 1 | e |
| 2 | l |
| 3 | l |
| 4 | o |

# But isn't that overly restrictive?

**YES!**

# References and borrowing

```rust
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

1. Use the `&` symbol before a type to denote an (immutable) reference.

2. Use the `&` operator to create a reference

# References

implemented as pointers

| s | |
|------|-------|
| name | value |
| ptr | |

| s1 | |
|----------|-------|
| name | value |
| ptr | |
| len | 5 |
| capacity | 5 |

| index | value |
|-------|-------|
| 0 | h |
| 1 | e |
| 2 | l |
| 3 | l |
| 4 | o |
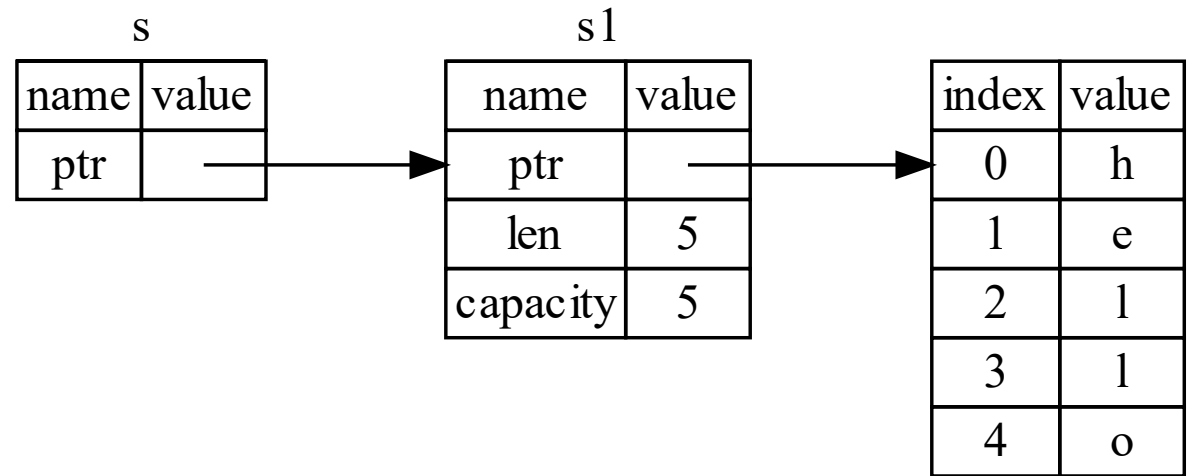
# References are immutable

(by default)

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

    change(&s);
}

fn change(some_string: &String) {
    some_string.push_str(", world");
}
```

This will not compile ^, since we cannot change the String through an **immutable** reference.

# Mutable references

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

    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

# Aliasing XOR mutability

Any reference in Rust can either
- Be **shared** with others.
- Be **mutable** (writable)

Not both at the same time!

# Safety guarantees

# Protects against iterator invalidation



```rust
fn iterator_invalidation() {
    let mut xs = Vec::new();
    xs.push(1);
    xs.push(2);
    xs.push(3);

    for x in xs.iter() {
        println!("{}", x);
        xs.remove(index: 0);
    }
}
```



```
Problems  Declaration  Console 
<terminated> Example [Java Application] C:\Program Files\Java\jre-10.0.2\bin\javaw.exe (12-Jul-2019, 5:00:57 PM)
List Value:1
List Value:2
Exception in thread "main" List Value:3
java.util.ConcurrentModificationException
        at java.base/java.util.ArrayList$Itr.checkForComodification(Unknown Source)
        at java.base/java.util.ArrayList$Itr.next(Unknown Source)
        at crux19aug.Crux19Aug2018.src.assignments.Example.main(Example.java:24)
```



```
user@host:/tmp/segfault$ cat segfault.c
void main() {
  char *str = "Hello, world!";
  *str = 'A';
}
user@host:/tmp/segfault$ gcc segfault.c -o segfault
user@host:/tmp/segfault$ ./segfault
Segmentation fault
neil@snap:/tmp/segfault$ 
```

ComputerHope.com

# "Fearless" concurrency

- Tools in the "concurrency" toolbox:
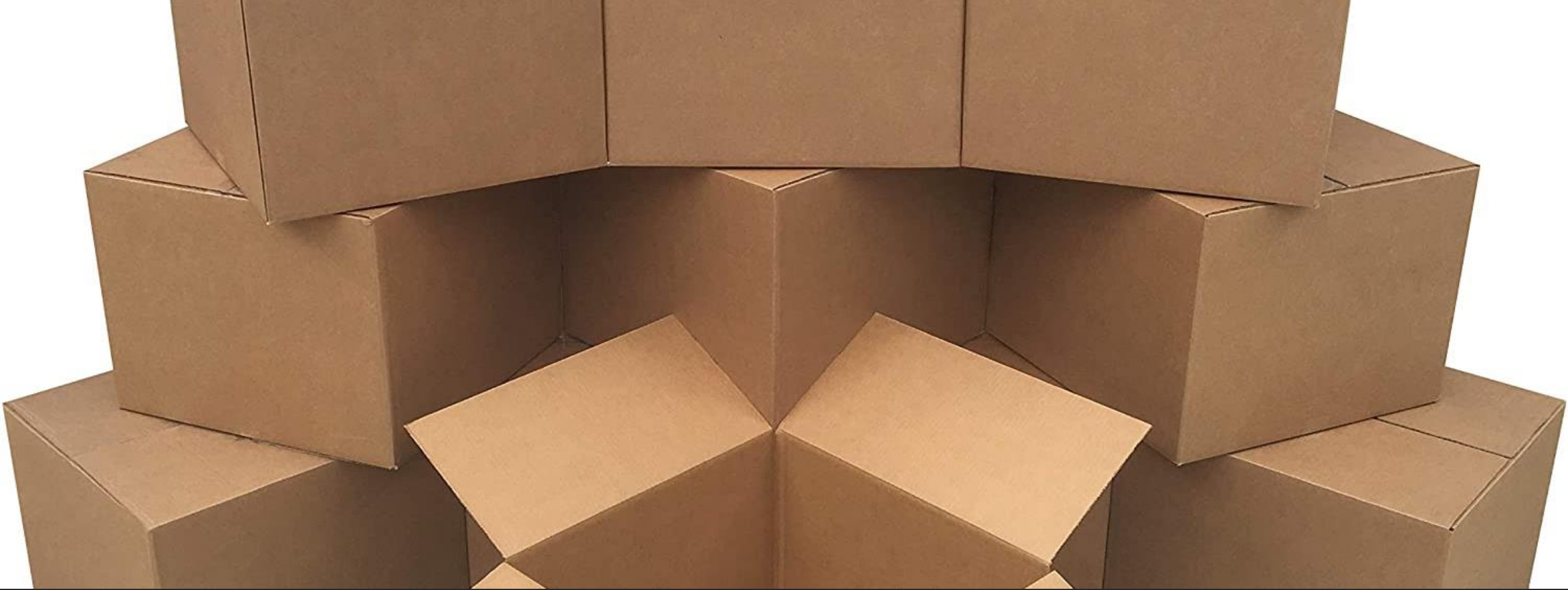  ◦ Shared state: *Atomics*, *Arc, Mutex*
  ◦ Message passing: *Channels*, *Actors*

Rust stdlib (and crates) build on an "unsafe" foundation to give you efficient building blocks for **safe, data-race-free, highly-multithreaded** code.



no data race

(potentially) data race
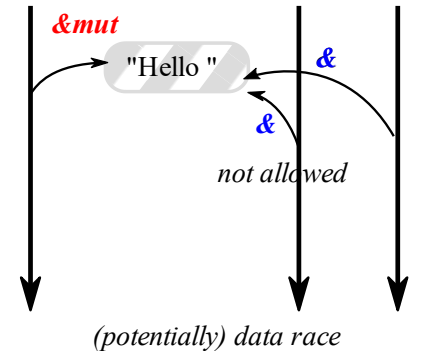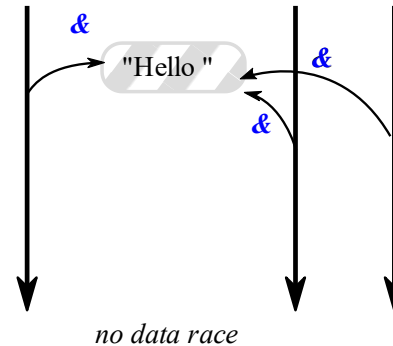
*"Rust teaches you the rules that good C/C++ programmers have in their head anyway"*

- SOMEONE ON THE RUST FORUM

# Appendix: OOP in Rust

# Is Rust object-oriented?

Sort of.

Rust does **not** have classes or data inheritance (only **structs**)

- Inheritance has recently fallen out of favor (see: composition over inheritance)

- Rust's enums can fill some of the void.

Rust **does** have data-hiding through *marking* structs/functions/fields **pub** or not

# Traits and polymorphism

Traits ⇔ interfaces

Rust supports *polymorphism* by implementing **traits** for specific types.

# Traits

We can build functions which work with any particular implementation of a trait (static dispatch)

```rust
pub fn notify<T: Summary>(item: &T) {
    println!("Breaking news! {}", item.summarize());
}
```

# Traits

We can also build functions which use *virtual dispatch* to use support multiple implementations at runtime.

```rust
pub trait Draw {
    fn draw(&self);
}

pub struct Screen {
    pub components: Vec<Box<dyn Draw>>,
}

impl Screen {
    pub fn run(&self) {
        for component in self.components.iter() {
            component.draw();
        }
    }
}
```

This is very similar to $virtual$ in C++

Or using $void*$ for polymorphism in C.

# Conclusion

# Why not Rust?

Some very good points from "matklad" – a major contributor to Rust (main dev of Rust Analyzer)

https://matklad.github.io/2020/09/20/why-not-rust.html

I repeat some of them here:

1. Not all programming is systems programming

2. Complexity

3. Compile times

4. Maturity

Rust's price for improved control is the curse of choice:

```
1   struct Foo       { bar: Bar        }
2   struct Foo<'a> { bar: &'a Bar       }
3   struct Foo<'a> { bar: &'a mut Bar }
4   struct Foo       { bar: Box<Bar>    }
5   struct Foo       { bar: Rc<Bar>     }
6   struct Foo       { bar: Arc<Bar>    }
```

# How to learn more Rust

- I can highly recommend the official Rust book as a learning resource:
  - https://doc.rust-lang.org/stable/book/title-page.html
  - Many examples from my slides are unceremoniously copied from here!


- For a more advanced learner, I can recommend Jon Gjengset's "Crust of Rust" series
  - https://www.youtube.com/watch?v=rAl-9HwD858&list=PLqbS7AVVErFiWDOAVrPt7aYmnuuOLYvOa
  - I started watching some of these just a few months ago and found them very insightful, but he comes from quite an academic standpoint and assumes great familiarity with the language.


- Try and **build** something!
  - A console application, a game, a webapp, an operating system, you name it!

# Thank you!

If you have further questions?