Rust Introduction

Scott Mabin

Background on Rust

- Rust is a systems programming language with the slogan "fast, reliable, productive: pick three."
- 1.0 release back in 2015
- 6 week release cycle
- Previously governed by Mozilla, but is now managed by an independent non-profit organization, the Rust Foundation.

Why Rust?

ESPRESSIF

- It's fast, compiling down to machine code just like C
  - Memory is deallocated as it goes out of scope, no garbage collection required.
- Eliminates a whole class of memory and syncronization bugs at compile time
  - In 2019 Microsoft announced that over 70% of CVEs in the last 12 years related to their system level software (written in C or C++) were memory safety bugs.
- Package management with `cargo`
  - Similar to esp-idf component manager, but supporting the entire language.
- Imperative language, but with strong functional elements

Terminology and basic tooling

What is a crate?

- Synonymous with a library/project
- Two types
  - binary crate - application or project
  - library crate

Cargo

- Manages the
  - Download and compilation of crates in a project
  - Documentation generation for the project
  - Running of tests
- Default repository is crates.io, but allows custom repositories
  - It's also possible to use git or path dependencies which is very useful for development
- Functionality can be extended with plugins (more on this later)

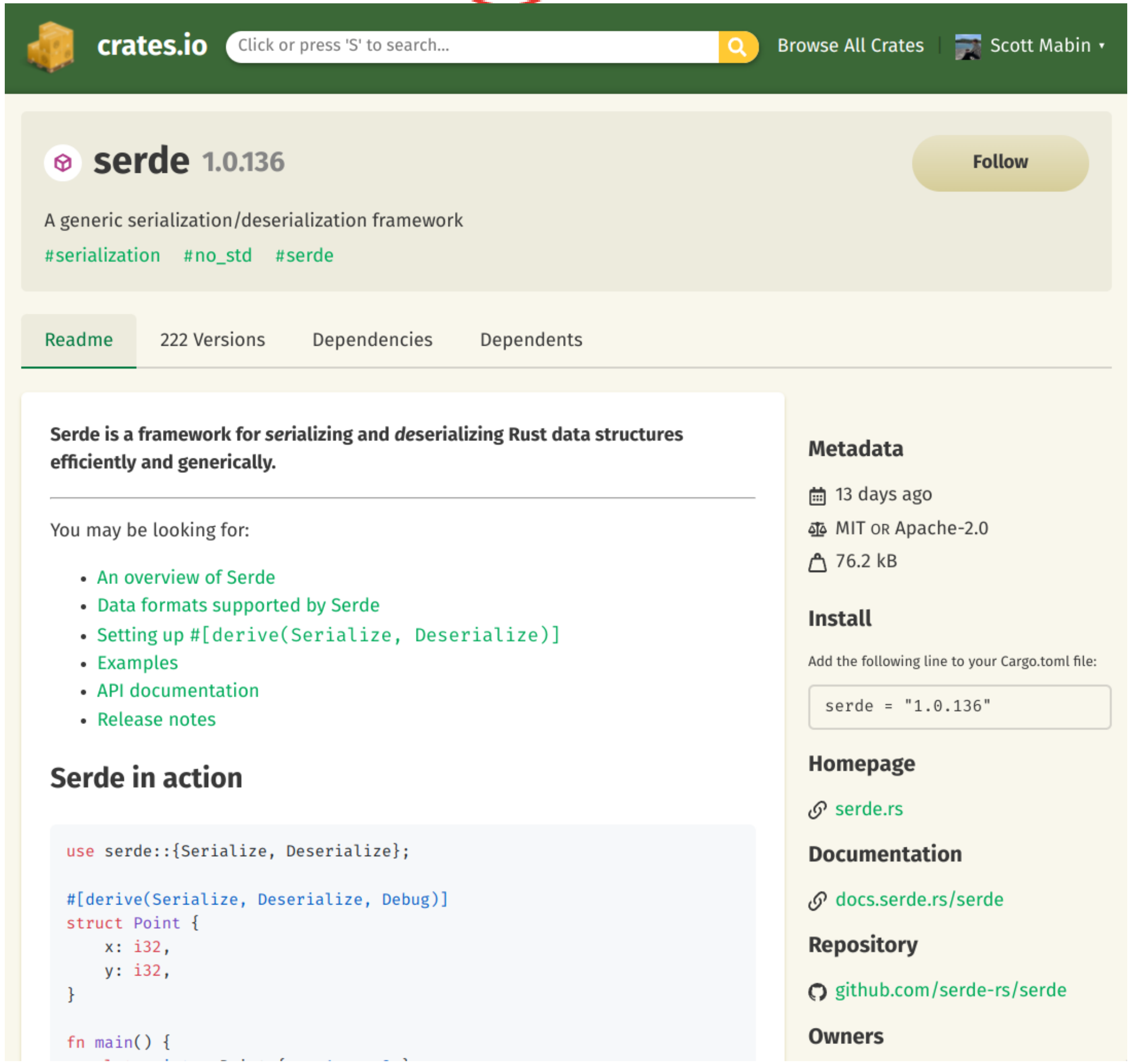
Using a external library

For example, adding `serde` to our application.

```toml
# Cargo.toml
[dependencies]
serde = "1.0.136"
```

Then to use it.

```rust
//! main.rs
use serde;
```



How a Rust project is structured

```
├── Cargo.lock
├── Cargo.toml
├── src/
│   ├── lib.rs
│   ├── main.rs
│   └── bin/
│       ├── named-executable.rs
│       ├── another-executable.rs
│       └── multi-file-executable/
│           ├── main.rs
│           └── some_module.rs
├── examples/
│   ├── simple.rs
│   └── multi-file-example/
│       ├── main.rs
│       └── ex_module.rs
└── tests/
    ├── some-integration-tests.rs
    └── multi-file-test/
        ├── main.rs
        └── test_module.rs
```

Hello world in Rust

ESPRESSIF

```rust
//! main.rs
fn main() {
  println!("Hello world!")
}
```

Writing Rust code

Enumerations - C like

Rust's enums are most similar to algebraic data types in functional languages, such as F#, OCaml, and Haskell but can also be C like too.

```
enum Chip { // C like enum
  Esp32,
  Esp32c3,
  Esp8266
}
```

```
enum Chip { // C like enum with values
  Esp32 = 123,
  Esp32c3 = 555,
  Esp8266 = 999
}
```

```
let c3 = Chip::Esp32c3; // Example usage
```

Enumerations - C like matching

```rust
let chip = Chip::Esp32c3;
match chip {
  Chip::Esp32c3 => println!("It's a C3 yay!"),
  other => println!("It's a {:?}!", other),
}
```

Enumerations - Algebraic

```rust
enum Chip {
  Esp32 { revision: u8 }, // named field
  Esp32c3,
  Esp8266
}
```

```rust
enum Chip {
  Esp32(u8), // anonymous field
  Esp32c3,
  Esp8266
}
```

```rust
// Example usage
let esp32r0 = Chip::Esp32 { revision: 0 };
```

Enumerations - Algebraic matching

```rust
let esp32 = Chip::Esp32 { revision: 0 };
match esp32 {
  // matching with fixed constants
  Chip::Esp32 { revision: 0 } => println!("It's a revision esp32r0! You're old school."),
  // matching with variable bindings
  Chip::Esp32 { revision } => println!("It's a esp32r{}!", revision),
  // wildcard catch all
  _ => panic!("Not an esp32!"),
}
```

playground link

Error handling - Unrecoverable

```rust
panic!("Oh no!");
```

```rust
let reason = "Cosmic Ray";
panic!("Panic reason: {}", reason);
```

Error handling - Result type

```rust
enum Result<T, E> {
    Ok(T),
    Err(E),
}
```

```rust
let possible_err: Result<&str, &str> = Ok("All good!");
let assume_okay_or_panic = possible_err.unwrap();
```

```rust
let possible_err: Result<&str, &str> = Err("Oops, there was an error");
let assume_okay_or_panic = possible_err.unwrap(); // panic here
```

Error handling

```rust
enum Error {
  Checksum
}

pub fn read_u32_from_device() -> Result<u32, Error>
```

```rust
let mut num_samples = 0;
for _ in 0..100 {
  match read_u32_from_device() {
   Ok(data) => {
     num_samples += 1;
     println!("Received: 0x{:04X}", data);
   },
   Err(e) => println!("{:?} error occured, but it's okay! Lets keep going!", e),
  }
}
```

playground link

Mutability

Unlike most programming languages, every variable and reference is immutable by default.

```rust
let x = 5;
x = 6; // compile error, x is not mutable
```

It is also possible to redefine or shadow variable names.

```rust
let x = 5; // x is 5
let x = 6; // x is 6
```

To declare something that should be mutable, use the `mut` keyword.

```rust
let mut x = 5;
x = 6; // x now 6
```

`return` in Rust

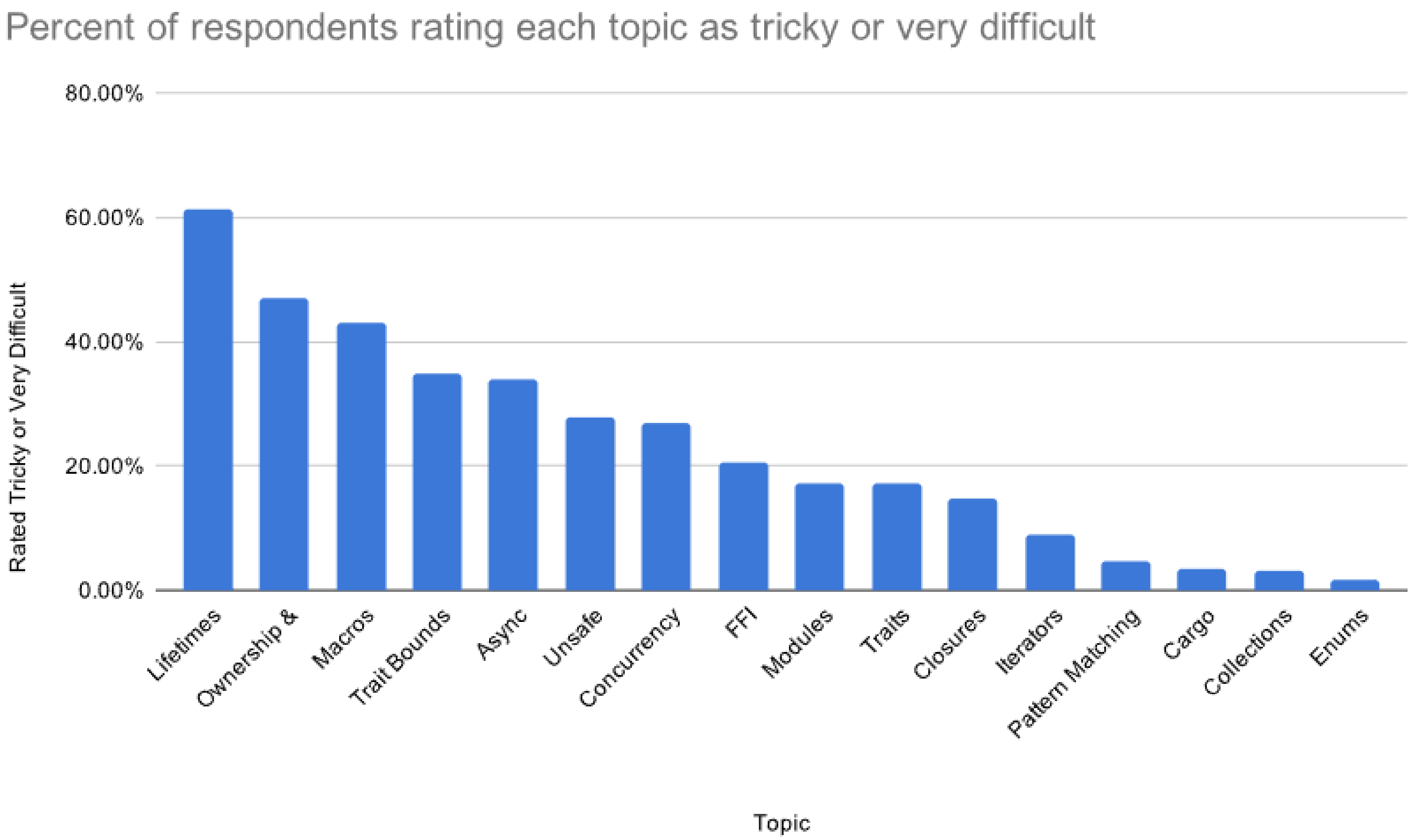
Everything in Rust is an expression, its typical to not use the `return` keyword, unless returning early.

```
fn new(seed: u32) -> Result<u32, &'static str> {
  if seed == 0 {
    return Err("Seed can't be zero"); // early return here
  }

  // important to note there is a missing semicolon, adding a semicolon
  // would make that line of code a statement. (which makes this example fail to compile)
  // Equivilent to `return Ok(seed);`
  Ok(seed)
}
```

Playground link with more examples

Ownership & Lifetimes



Ownership

- Fundamental set of rules that governs how a Rust program manages memory.
- Applies to both stack *and* heap memory.

Ownership

The three ownership rules:

- Each 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.

Ownership - Example

```
let s1 = String::from("hello"); // s1 owns the string
let s2 = s1; // s1 transfers ownership (moves) to s2, leaving s1 empty
```

```
println!("{}, world!", s1); // try to use s1 and we'll get a compile error
```

```
error[E0382]: borrow of moved value: `s1`
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move

For more information about this error, try `rustc --explain E0382`.
error: could not compile `ownership` due to previous error
```

Ownership - `Copy` types

The error on the previous slide talks about a type not implementing the `Copy` trait (more on traits later!). Simply put, if an `struct` or `enum` implements the `Copy` trait, it means it's safe to do a bitwise memcopy to duplicate the value.

Example of a `Copy` type are integers.

```rust
let x = 5; // x owns the integer with value of 5
let y = x; // integer is `Copy`, so x is copied bit for bit into y
```

```rust
println!("(x,y) = ({},{})", x,y); // compiles fine
// prints (x,y) = (5,5)
```

Ownership - `Copy` types

Why is `String` not copy? Well a `String` is just a `Vec` but with the guarantee that all the bytes inside are valid UTF8. Let's look at the memory layout of `Vec`.

```rust
struct Vec<T> {
    ptr: NonNull<T>,
    cap: usize,
    _marker: PhantomData<T>,
}
```

We can see we have a pointer to some memory (on the heap), and a capacity. If we were to do a bitwise copy of our `Vec` structure we'd have two objects with mutable access to the same heap memory! Not good!

Ownership - `Clone`

To duplicate a `String` we'd need some special behavior. This is where `Clone` comes in. `Clone` is a trait like `Copy` that allows us to define what to do when we want to duplicate a `struct` or `enum` that is not `Copy`.

The `Clone` implementation for a `String` allocates *new* memory on the heap with the same capacity, copies the bytes from the current allocation into the new memory and finally returns the new `String`.

Ownership - Borrowing

Moving (transfering ownership) everytime doesn't make sense. Sometimes we just want to *borrow* a value, without any unnecessary `Clone`'s or `Copy`'s.

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

    let len = calculate_length(&s1);

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

/// Takes a _reference_ to a `String`
fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Ownership - Mutable borrowing

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

    change(&mut s);

    println!("{}", s)
}

/// Takes a **mutable** _reference_ to a `String`
/// We can treat this `String` like we _own_ it for the duration of the borrow
fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Ownership - Mutable borrowing

The most important rule about mutable borrowing is that to borrow mutably, there must be no other references (mutable or immutable) to the thing you are trying to borrow.

```
let mut owned: String = "hi".to_owned();
let mut_borrow = &mut owned; // mutable borrow starts here

let borrow = &owned; // can't borrow again whilst a mutable borrow exists!

mut_borrow.push_str("!!!"); // mutable borrow lives till here
```

Ownership - Mutable borrowing

Fortunately we don't have to track this, Rust is kind enough to tell us if we try and borrow again with a friendly compile error.

```
error[E0502]: cannot borrow `owned` as immutable because it is also borrowed as mutable
  --> src/main.rs:9:19
   |
7  |     let mut_borrow = &mut owned;
   |                      ---------- mutable borrow occurs here
8  |
9  |     let _borrow = &owned;
   |                   ^^^^^^ immutable borrow occurs here
10 |
11 |     mut_borrow.push_str("!!!");
   |     -------------------------- mutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
error: could not compile `playground` due to previous error
```

Feel free to play around with the ordering of statements and scope in the playground.

Ownership - Lifetime of a borrow

In most cases, and with the examples on the previous slides the lifetime of the borrows were inferred by the compiler, but this is not always the case.

```
struct SliceContainer {
  bytes: &[u8] // reference to a slice of bytes
}

impl SliceContainer {
  fn print(&self) {
    println!("{:?}", self.bytes);
  }
}
```

Ownership - Lifetime of a borrow

In this case, what happens at run time if we call `print` when the underlying storage for the bytes has be deallocated?

```
fn create_container() -> SliceContainer {
  let data = [0xFF; 12]; // small array of bytes the stack

  SliceContainer {
    bytes: &data[..] // reference to slice of `data`
  }
}
```

When `create_container()` returns, the reference to the slice inside `SliceContainer` is invalidated (`data` is deallocated from the stack). Let's see how Rust solves this at compile time.

Ownership - Lifetime of a borrow

```rust
struct SliceContainer<'a> { // lifetime of struct denoted as 'a
  // reference to a slice of bytes,
  // which **must** live _atleast_ as long as the lifetime 'a
  bytes: &'a [u8]
}

impl<'a> SliceContainer<'a> {
  fn print(&self) {
    println!("{:?}", self.bytes);
  }
}
```

Rust will track the lifetime of any variables used in `SliceContainer` and ensure they live long enough (not dropped before `SliceContainer`'s lifetime `'a`).

The lifetime name is not important, it can be almost anything for example `'bytes`, but typically it is a single letter.

Ownership - Lifetime of a borrow

Compiling the `create_container()` function again yields the following error message.

```
error[E0515]: cannot return value referencing local variable `data`
  --> src/main.rs:20:3
   |
20 | /   SliceContainer {
21 | |     bytes: &data[..] // reference to slice of `data`
   | |             ---- `data` is borrowed here
22 | |   }
   | |___^ returns a value referencing data owned by the current function

For more information about this error, try `rustc --explain E0515`.
```

playground link

Generics

We've already seen some generics throughout these slides, a generic enum `Result<T, E>` and `Vec<T>` a homogeneous collection. They are a core part of Rust's tools for reducing code duplication.

Generic type parameters (usually denoted by a single letter) are placeholders for a *concrete* type.

Generics

Lets look at the definition of another important enum, `Option`. `Option` is a replacement for null, and represents a situation where a value may or may not exists during runtime.

```
enum Option<T> {
    Some(T),
    None,
}
```

`Option`'s generic type `T` can be filled with any concrete type we like for example

```
let x: Option<u32> = Some(0); // optional u32
```

```
struct CustomItem;

let x: Option<CustomItem> = None; // optional CustomItem
```

Generics - Traits

Traits are Rust's way of defining shared behaviour. They are similiar to what other languages might call interfaces. Lets define our first trait.

```
pub trait Animal {
  fn eat(&self, t: Treat);
}
```

and some `struct`'s that we want to implement the trait for.

```
struct Cat;
struct Dog;
```

and we can't forget a treat for them!

```
struct Treat;
```

Generics - Traits

Lets implement `Animal` for our structs.

```rust
impl Animal for Cat {
  fn eat(&self, _t: Treat) {
    println!("Meow!") // cat for tasty
  }
}
```

```rust
impl Animal for Dog {
  fn eat(&self, _t: Treat) {
    println!("Woof!") // dog for yummy
  }
}
```

Generics - Traits

Now lets make a cage to hold animals. We'll add a treat dispensing method too.

```
struct Cage<T>{
  animal: T,
}

impl<T> Cage<T> {
  pub fn new(animal: T) -> Self {
    Self { animal: animal }
  }
  pub fn dispense_treat(&self) {
    let t = Treat;
    self.animal.eat(t)
  }
}
```

Generics - Traits

```rust
fn main() {
  let mia = Dog;
  let cage = Cage::new(mia);

  cage.dispense_treat();
}
// _should_ print "Woof!"
```

Easy, right? Not quite! Rust will fail to compile this because is doesn't know that whatever the concrete type `T` is filled in with implements `Animal`.

Generics - Traits

In that same line of thinking, what stops us from putting something that does not have `Animal` traits into the cage? Right now we could put whatever we like in here, like a wild `u32`.

```
let nibble = 0u32; // ah, our beloved pet u32, nibble!
let cage = Cage::new(nibble);

cage.dispense_treat();
// what should this print???
```

We need some *trait bounds* on our `Cage`.

Generics - Traits

```
struct Cage<T>{
  animal: T,
}

impl<T> Cage<T>
  where T: Animal // <--- this is the important bit!
{
  pub fn new(animal: T) -> Self {
    Self { animal: animal }
  }
  pub fn dispense_treat(&self) {
    let t = Treat;
    self.animal.eat(t)
  }
}
```

Feel free to play with this example in the playground.

Miscellaneous - `std` vs `no_std`

The Rust Core Library is the dependency-free foundation of The Rust Standard Library. The core library is minimal: it isn’t even aware of heap allocation, nor does it provide concurrency or I/O. These things require platform integration, and this library is platform-agnostic.

Even though the Core library is not aware of heap allocation, it's still possible to do heap allocation in a `no_std` environment. Much like the Core library and Standard library, the Alloc library is also seperate and can be pulled into a `no_std` project.

The Standard library builds on top of Core and Alloc, often re-exporting large parts of each crate.

Miscellaneous - Cargo plugins

By naming a package binary `cargo-X` you can extend the capabilities of cargo. For example, `cargo-espflash` can be invoked with `cargo espflash OPTIONS`. Other examples.

`cargo-edit` is one such tool, and allows the addition, removal and upgrade of dependencies in the projects `Cargo.toml` via the command line.

Thanks for listening!

Questions?

Links & Resources

- "the book" - Official Rust learning book
- rust-by-example
- rustlings
- The esp book - Rust on Espressif chips
- esp-rs organisation
- esp-rs roadmap
- rust embedded book
- gitpod.io/rust - Quickstart rust environment in the browser