Rust 101 - Lecture 2

Jordan Hall

2022

DoCSoc

Roadmap


Rust 101
Concurrency
Mutex/Arc
Send/Sync Traits
Rayon
Types
Enums
Structs
Compound Types
Primitives
Build Tools
Formatter
Clippy
Automated Tests
Cargo
Functions
Pure Functions
Closures
Higher-Order Functions
Smart Pointers
Rc
Box
Traits
Generic Types
Traits
Lifetimes
Control Flow
If/While/For/Loop Statements
If/While Let
Control Flow as Expressions
Question Mark Operator
The Borrow Checker
Borrowing
Ownership
References

Before We Start

The slides are now over on my GitHub page (so you can make notes):

https://github.com/JordanLloydHall/RustCourse

Quick Recap

Last time we took a look at variables, basic types, control flow, and functions. Most of these ideas are not particularly new. They look similar to structures of other languages. But then we came across something weird:

A Problem with Functions (continued)

```
fn length_of_username(user: User) -> usize {
    user.username.len()
}
fn main() {
    let tony = User {
        active: false,
        username: String::from("Tony"),
        email: String::from("OohBaby42@haskell.org"),
        sign_in_count: 12345
    };
    let l = length_of_username(tony);
    assert_eq!(4, l);
    assert!(!tony.active)
}
```

A Problem with Functions (continued)

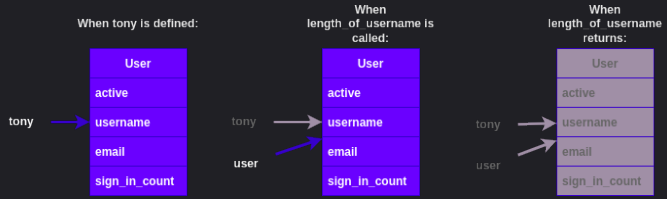
```
error[E0382]: use of moved value: `tony`
  --> src/main.rs:20:14
   |
12 |     let tony = User {
   |         ---- move occurs because `tony` has type `User`, which does not implement
   |                the `Copy` trait
...
18 |     let l = length_of_username(tony);
   |                                ---- value moved here
19 |     assert_eq!(4, l);
20 |     assert!(!tony.active)
   |             ^^^^^^^^^^^ value used here after move

For more information about this error, try `rustc --explain E0382`.
```

Borrows and Ownership

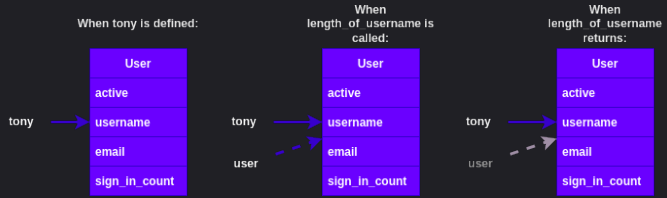
```
fn length_of_username(user: &User) -> usize {
    user.username.len()
}
fn main() {
    let tony = User {
        active: false,
        username: String::from("Tony"),
        email: String::from("OohBaby42@haskell.org"),
        sign_in_count: 12345
    };
    let l = length_of_username(&tony);
    assert_eq!(4, l);
    assert!(!tony.active)
}
```

Borrows and Ownership (continued)

Without the reference, we had the following flow diagram:



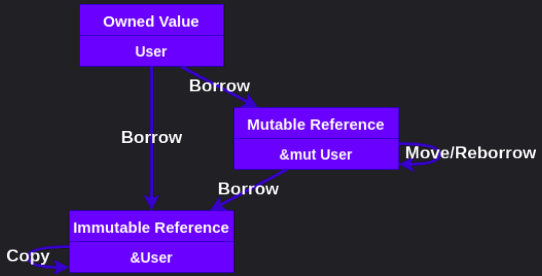
With the reference, we had the following flow diagram:

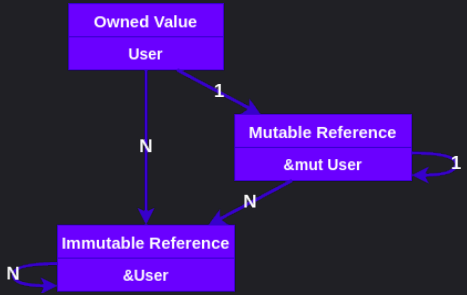


Borrows and Ownership

```
fn sign_in_user(user: &mut User) {
    user.sign_in_count += 1;
    user.active = true;
}
fn main() {
    let mut tony = User {
        active: false,
        username: String::from("Tony"),
        email: String::from("OohBaby42@haskell.org"),
        sign_in_count: 12345
    };
    sign_in_user(&mut tony);
    assert_eq!(12346, tony.sign_in_count);
    assert!(tony.active);
}
```

Borrow Hierarchy





Questions:

1. Why can't you have more than one mutable reference at the same time?
2. Why can't an immutable reference create a mutable reference?
3. Why can a mutable reference create immutable references, but a owned value can't create both a mutable and immutable reference at the same time?

Borrow Hierarchy (continued)

```
fn main() {
    let s = String::from("It's Over, Anakin. I Have The High Ground");

    let ref1: &String = &s;
    let ref2: &String = ref1;

    println!("{}\n{}", ref1, ref2);
}
```

Questions:

1. Does this code compile?
2. If not, why?

Borrow Hierarchy (continued)

```
fn main() {
    let s = String::from("It's Over, Anakin. I Have The High Ground");

    let ref1: &String = &s;

    println!("{}\n{}\n{}", ref1, s);
}
```

Questions:

1. Does this code compile?
2. If not, why?

Borrow Hierarchy (continued)

```
fn main() {
    let mut s = String::from("It's Over, Anakin. I Have The High Ground");

    let mut_ref: &mut String = &mut s;
    let immut_ref: &String = &s;

    println!("{}\n{}", mut_ref, immut_ref);
}
```

Questions:

1. Does this code compile?
2. If not, why?

Borrow Hierarchy (continued)

```
error[E0502]: cannot borrow `s` as immutable because it is also borrowed as mutable
  --> src/main.rs:16:30
   |
15 |     let mut_ref: &mut String = &mut s;
   |                                ------ mutable borrow occurs here
16 |     let immut_ref: &String = &s;
   |                              ^^ immutable borrow occurs here
17 |
18 |     println!("{}\n{}", mut_ref, immut_ref);
   |                        ------- mutable borrow later used here

For more information about this error, try `rustc --explain E0502`.
```

Strings!

```
fn main() {
    let s1: &str = "Hello World!";
    let s2: &str = &s1[..5];
    println!("{}, {}", s1, s2);
}
```

Traits

What Traits are

Traits are a collection of methods that can be called on things that implement them.

They describe what a type with that trait is able to do.

First Look at Traits

```
struct User {
    active: bool,
    username: String,
    email: String,
    sign_in_count: u64,
}
```

First Look at Traits (continued)

```
impl User {
    fn login(&mut self) -> bool {
        if self.active {
            false
        } else {
            self.sign_in_count += 1;
            self.active = true;
            true
        }
    }
...
```

First Look at Traits (continued)

```
...
    fn logout(&mut self) -> bool {
        if self.active {
            self.active = false;
            true
        } else {
            false
        }
    }
}
...
```

First Look at Traits (continued)

```
...
fn main() {
    let mut tony = User {
        active: true,
        username: String::from("Tony"),
        email: String::from("OohBaby42@haskell.org"),
        sign_in_count: 12345
    };
    tony.logout();
    tony.login();
    assert!(tony.active);
    assert_eq!(tony.sign_in_count, 12346);
}
```

First Look at Traits (continued)

```
impl User {
    fn login(&mut self) -> bool {
        if self.active {
            false
        } else {
            self.sign_in_count += 1;
            self.active = true;
            true
        }
    }
    fn logout(&mut self) -> bool {
        if self.active {
            self.active = false;
            true
        } else {
            false
        }
    }
}
```

A Second Look at Traits

```
trait Quote {
    fn quote(&self) -> String;
}

struct MaceWindu;
struct Tony;

impl Quote for MaceWindu {
    fn quote(&self) -> String {
        String::from("You are on this council, but we do not grant you the rank of Master.")
    }
}
impl Quote for Tony {
    fn quote(&self) -> String {
        String::from("Ohhhhhhh baby!")
    }
}
```

A Second Look at Traits (continued)

```
fn main() {
    let windu = MaceWindu;
    let tony = Tony;

    assert_eq!(
        windu.quote(),
        String::from("You are on this council, but we do not grant you the rank of Master.")
    );
    assert_eq!(tony.quote(), String::from("Ohhhhhhh baby!"));
}
```

Using Traits in Functions

```
fn print_quote_long<T: Quote>(quoter: &T) {
    println!("{}", quoter.quote());
}
fn print_quote_short(quoter: &impl Quote) {
    println!("{}", quoter.quote());
}
fn main() {
    let windu = MaceWindu;
    print_quote_long(&windu);
    print_quote_short(&windu);
}
```

More Than One Trait

```
trait Name {
    fn name(&self) -> String;
}
impl Name for MaceWindu {
    fn name(&self) -> String {
        String::from("Mace Windu (Jedi Master)")
    }
}
fn print_quote<T: Name + Quote>(quoter: &T) {
    println!("{} - {}", quoter.quote(), quoter.name())
}
fn main() {
    let windu = MaceWindu;
    let tony = Tony;
    print_quote(&windu);
}
```

More Than One Trait (continued)

```
trait Name {
    fn name(&self) -> String;
}
fn print_quote<T: Name + Quote>(quoter: &T) {
    println!("{} - {}", quoter.quote(), quoter.name())
}
fn main() {
    let tony = Tony;
    print_quote(&tony);
}
```

More Than One Trait (continued)

```
error[E0277]: the trait bound `Tony: Name` is not satisfied
  --> src/main.rs:32:17
   |
32 |     print_quote(&tony);
   |     ----------- ^^^^^ the trait `Name` is not implemented for `Tony`
   |     |
   |     required by a bound introduced by this call
   |
note: required by a bound in `print_quote`
  --> src/main.rs:26:19
   |
26 | fn print_quote<T: Name + Quote>(quoter: &T) {
   |                   ^^^^ required by this bound in `print_quote`

For more information about this error, try `rustc --explain E0277`.
```

More Than One Trait (continued)

```
fn print_quote<T: Name + Quote>(quoter: &T) {
    println!("{} - {}", quoter.quote(), quoter.name())
}

fn print_quote<T>(quoter: &T)
where
    T: Name + Quote,
{
    println!("{} - {}", quoter.quote(), quoter.name())
}
```

Supertraits

```
trait Name {
    fn name(&self) -> String;
}
trait Quote: Name {
    fn quote(&self) -> String;
    fn speak(&self) {
        println!("{} says {}", self.name(), self.quote())
    }
}
```

Supertraits (continued)

```
struct Dog;
impl Name for Dog {
    fn name(&self) -> String {
        String::from("Dog")
    }
}
impl Quote for Dog {
    fn quote(&self) -> String {
        String::from("Wooof!")
    }
}
```

Supertraits (continued)

```
fn main() {
    let dog = Dog;
    dog.speak();
}
```

Supertraits (continued)

```
struct Cat;
impl Quote for Cat {
    fn quote(&self) -> String {
        String::from("Meow")
    }
}
```

Supertraits (continued)

```
error[E0277]: the trait bound `Cat: Name` is not satisfied
  --> src/main.rs:11:6
   |
11 | impl Quote for Cat {
   |      ^^^^^ the trait `Name` is not implemented for `Cat`
   |
note: required by a bound in `Quote`
  --> src/main.rs:4:14
   |
4  | trait Quote: Name {
   |              ^^^^ required by this bound in `Quote`

For more information about this error, try `rustc --explain E0277`.
```

Constant Generics

We have already seen the most basic use of constant generics. They allow an integer to be a part of a type, known at compile-time.

```
let xs: [i32; 5] = [1, 2, 3, 4, 5];
```

Constant Generics (continued)

```
fn cumulate<const N: usize>(mut arr: [i32; N]) -> [i32; N] {
    let mut sum = 0;
    for i in 0..N {
        sum += arr[i];
        arr[i] = sum;
    }
    arr
}
fn main() {
    let arr = cumulate([1; 10]);
    assert_eq!(arr, [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]);
}
```

A Problem With References

```
fn return_slice(s: &str) -> &str {
    &s[..5]
}
fn main() {
    assert_eq!(return_slice("Hello World!"), "Hello")
}
```

A Problem With References (continued)

```
fn return_largest_str(s1: &str, s2: &str) -> &str {
    if s1.len() >= s2.len() {
        s1
    } else {
        s2
    }
}
fn main() {
    let s1 = String::from("Hello World!");
    let s2 = String::from("Wow!");
    assert_eq!(return_largest_str(&s1, &s2), "Hello World!")
}
```

A Problem With References (continued)

```
 --> src/main.rs:1:46
  |
1 | fn return_largest_str(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
  |
1 | fn return_largest_str<'a>(s1: &'a str, s2: &'a str) -> &'a str {
  |                       ++++      ++           ++          ++

For more information about this error, try `rustc --explain E0106`.
```

Lifetimes

Lifetimes are an extremely important feature of Rust.

They serve the purpose of informing the borrow checker how long references are meant to be in scope for.

Some very common lifetime patterns are elided by the Rust compiler:

```
fn return_slice(s: &str) -> &str {
    &s[..5]
}
```

Is actually expanded by the compiler to be:

```
fn return_slice<'a>(s: &'a str) -> &'a str {
    &s[..5]
}
```

Lifetimes (continued)

```
fn main() {
    let slice: &str;
    {
        let s = String::from("Hello world!");
        slice = return_slice(&s);
    }
    assert_eq!(slice, "Hello")
}
```

Lifetimes (continued)

```
error[E0597]: `s` does not live long enough
  --> src/main.rs:18:30
   |
18 |         slice = return_slice(&s);
   |                              ^^ borrowed value does not live long enough
19 |     }
   |     - `s` dropped here while still borrowed
20 |
21 |     assert_eq!(slice, "Hello")
   |     -------------------------- borrow later used here

For more information about this error, try `rustc --explain E0597`.
```

Lifetimes (continued)

```
fn return_largest_str<'a>(s1: &'a str, s2: &'a str) -> &'a str {
    if s1.len() >= s2.len() {
        s1
    } else {
        s2
    }
}
fn main() {
    let s1 = String::from("Hello World!");
    let s2 = String::from("Wow!");
    assert_eq!(return_largest_str(&s1, &s2), "Hello World!")
}
```

Static Lifetimes

There's a special lifetime, called static lifetime. It looks like `'static`.

Its purpose is to inform the compiler when something will live as long as the whole program. A good example of this are string literals:

```
fn main() {
    let s: &str;
    {
        s = "Hello There!";
    }
    assert_eq!(s, "Hello There!");
}
```

Like in most languages, string literals are put onto the .data section of the compiled assembly file. `&str` are references to this data, and so should be accessible from any scope.

Static Lifetimes (continued)

```
fn return_str() -> &'static str {
    "Ohhh Baby!"
}
```

Traits From the Standard Library

Clone, Copy, Debug, Display, Default, From/Into and Iterator/IntoIterator

The Clone Trait

The `Clone` trait allows you to perform a deep copy of an object. It's an explicit way to duplicate data.

```
pub trait Clone {
    fn clone(&self) -> Self;

    fn clone_from(&mut self, source: &Self) { ... }
}
```

The Clone Trait (continued)

```
impl Clone for User {
    fn clone(&self) -> User {
        User {
            active: self.active,
            username: self.username.clone(),
            email: self.email.clone(),
            sign_in_count: self.sign_in_count
        }
    }
}
```

The Clone Trait (continued)

We can often use Rust's powerful macros to automate implementing simple traits. Here's how we do it:

```
#[derive(Clone)]
struct User {
    ...
}
fn main() {
    let s1 = String::from(
        "According to all known laws of aviation, \
        there is no way a bee should be able to fly.",
    );
    let s2 = s1.clone();
    println!("{}\n{}", s1, s2);
}
```

The Copy Trait

The `Copy` trait allows you to implicitly clone an object instead of moving or explicitly cloning it.

```
pub trait Copy: Clone { }
```

The Copy Trait (continued)

```
fn main() {
    let x = 42;
    let y = x;
    println!("x: {}, y: {}", x, y);
}
```

The Debug Trait

The `Debug` trait allows you to print all of the information about an object. It is meant to be programmer-facing as it can be quickly implemented using the macro if all associated types also implement `Default`.

```
pub trait Debug {
    fn fmt(&self, f: &mut Formatter<'_>) -> Result<(), Error>;
}
```

The Debug Trait (continued)

```
#[derive(Debug)]
struct User {
    ...
}
fn main() {
    let tony = User {
        active: true,
        username: String::from("Tony"),
        email: String::from("OohBaby42@haskell.org"),
        sign_in_count: 12345
    };
    println!("{:?}", tony);
}
```

```
> User { active: true, username: "Tony",
    email: "OohBaby42@haskell.org", sign_in_count: 12345 }
```

The Default Trait

The `Default` trait allows you to quickly create new objects to save on duplication:

```
pub trait Default {
    fn default() -> Self;
}
```

The Default Trait (continued)

The `Default` macro can be derived if every associated type also implements `Default`:

```
#[derive(Default)]
struct User {
    ...
}
fn main() {
    let default_user = User::default();
    println!("{:?}", default_user);
}
```

```
> User { active: false, username: "", email: "", sign_in_count: 0 }
```

The Default Trait (continued)

The `Default` trait can also be useful for automatically filling in data:

```
fn main() {
    let tony = User {
        username: String::from("Tony"),
        email: String::from("OohBaby42@haskell.org"),
        ..Default::default()
    };
    println!("{:?}", tony);
}
```

```
> User { active: false, username: "Tony",
    email: "OohBaby42@haskell.org", sign_in_count: 0 }
```

The From/Into Traits

The `From`/`Into` traits can be used to explicitly convert between types:

```
pub trait From<T> {
    fn from(T) -> Self;
}
```

Whenever you implement `From`, you get `Into` for free as it is defined for all types that implement `From`!

The From/Into Traits (continued)

As you can see, we've been using from to convert `&str` to `String` this entire time:

```
let s1: &str = "Ohhh Baby!";
let s2: String = String::from(s1);
let s3: String = s1.into();
```

The Iterator/IntoIterator Traits

The `Iterator`/`IntoIterator` traits provide a huge amount of methods that allow you to traverse a data structure:

```
pub trait Iterator {
    type Item;
    // 70 more methods
}
pub trait IntoIterator {
    type Item;
    type IntoIter: Iterator;
    fn into_iter(self) -> Self::IntoIter;
}
```

`IntoIterator`, describes how you can convert an object into an iterator. Types that implement this trait can be used in for loops!

Highlight of Iterators

```rust
fn main() {
    let arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
    let product = arr.iter().fold(1, |acc, x| acc * x);
    let doubled = arr.iter().map(|x| x * 2).collect::<Vec<i32>>();
    let evens = arr.into_iter().filter(|x| *x % 2 == 0).collect::<Vec<i32>>();
    println!("{:?}\n{:?}\n{:?}", product, doubled, evens);
}
```

```
3628800
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
[2, 4, 6, 8, 10]
```

Some Homework

By now you know enough to start programming in Rust!

To get started, the Rustlings Project is a brilliantly-made beginner-friendly set of exercises.

You should be able to complete most of them (with some help from the compiler and documentation).

If you get stuck, then ask for help! The Rust community is a brilliant and kind source of knowledge and they'll always help you out!