RUSTikales Rust for advanced coders

Plan for today

Plan for today

1. Introduction

Plan for today

1. Introduction
2. Recap on Rust Basics

Plan for today

1. Introduction
2. Recap on Rust Basics
3. General Info

1. Introduction

– Welcome to this Rust course!

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
    - Do lifetimes refer to memory locations? Do they refer to points in time?

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
    - Do lifetimes refer to memory locations? Do they refer to points in time?
    - When is cloning really necessary?

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
    - Do lifetimes refer to memory locations? Do they refer to points in time?
    - When is cloning really necessary?
    - How can we tell the compiler that our code would just work™, if it wasn‘t for the Borrow Checker?

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
    - Rust has a really strong macro system, allowing us to do all sorts of stuff

1. Introduction

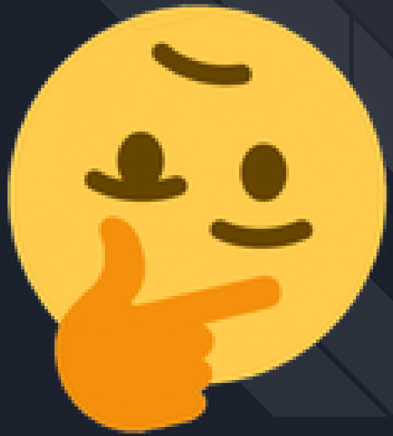
- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
    - Rust has a really strong macro system, allowing us to do all sorts of stuff
    - It‘s so powerful, it‘s turing-complete*

Source: https://mainisusuallyafunction.blogspot.com/2015/02/turing-tarpits-in-rusts-macro-system.html

```rust
/// http://esolangs.org/wiki/Bitwise_Cyclic_Tag
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));
    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));
    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));
    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    bct!(0, 0, 1, 1, 1 ; 1, 0, 1);
}
```

Source: https://mainisusuallyafunction.blogspot.com/2015/02/turing-tarpits-in-rusts-macro-system.html

```rust
/// http://esolangs.org/wiki/Bitwise_Cyclic_Tag
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));
    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));
    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));
    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    bct!(0, 0, 1, 1, 1 ; 1, 0, 1);
}
```



What

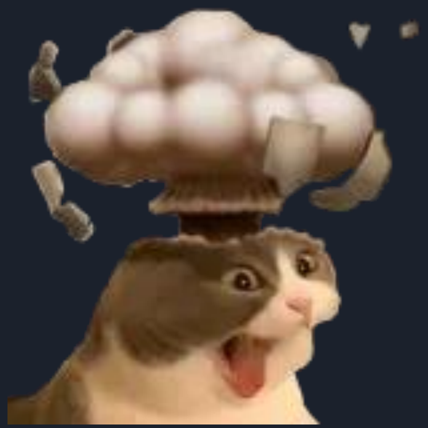
Source: https://mainisusuallyafunction.blogspot.com/2015/02/turing-tarpits-in-rusts-macro-system.html

```rust
/// http://esolangs.org/wiki/Bitwise_Cyclic_Tag
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));
    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));
    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));
    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    bct!(0, 0, 1, 1, 1 ; 1, 0, 1);
}
```

Proof that macros are turing complete

→ Macros can emulate BCT
→ BCT can emulate Cyclic Tag Systems
→ CTS can emulate Tag Systems
→ Turing Machines can be transformed into a TS (Minsky, 1961)
→ Rust macros can emulate TM → Turing Complete
→ We could write a Rust compiler using Rust macros :^) (please don't)

Source: https://mainisusuallyafunction.blogspot.com/2015/02/turing-tarpits-in-rusts-macro-system.html



```
expanding `bct! { 0, 0, 1, 1, 1 ; 1, 0, 1 }`
to `bct! (0, 1, 1, 1, 0 ; 0, 1)`
expanding `bct! { 0, 1, 1, 1, 0 ; 0, 1 }`
to `bct! (1, 1, 1, 0, 0 ; 1)`
expanding `bct! { 1, 1, 1, 0, 0 ; 1 }`
to `bct! (1, 0, 0, 1, 1; 1, 1)`
expanding `bct! { 1, 0, 0, 1, 1; 1, 1 }`
to `bct! (0, 1, 1, 1, 0 ; 1, 1, 0)`
expanding `bct! { 0, 1, 1, 1, 0 ; 1, 1, 0 }`
to `bct! (1, 1, 1, 0, 0 ; 1, 0)`
expanding `bct! { 1, 1, 1, 0, 0 ; 1, 0 }`
to `bct! (1, 0, 0, 1, 1; 1, 0, 1)`
expanding `bct! { 1, 0, 0, 1, 1; 1, 0, 1 }`
to `bct! (0, 1, 1, 1, 0 ; 1, 0, 1, 0)`
expanding `bct! { 0, 1, 1, 1, 0 ; 1, 0, 1, 0 }`
to `bct! (1, 1, 1, 0, 0 ; 0, 1, 0)`
expanding `bct! { 1, 1, 1, 0, 0 ; 0, 1, 0 }`
to `bct! (1, 0, 0, 1, 1; 0, 1, 0)`
expanding `bct! { 1, 0, 0, 1, 1; 0, 1, 0 }`
to `bct! (0, 1, 1, 1, 0 ; 0, 1, 0)`
expanding `bct! { 0, 1, 1, 1, 0 ; 0, 1, 0 }`
to `bct! (1, 1, 1, 0, 0 ; 1, 0)`
expanding `bct! { 1, 1, 1, 0, 0 ; 1, 0 }`
to `bct! (1, 0, 0, 1, 1; 1, 0, 1)`
```

```
System evolution:

Commands    Data-
Executed    String
--------    -------
    0       101
    0        01
   11         1
   10         11
    0         110
   11          10
   10          101
    0          1010
   11           010
   10           010
    0           010
   11            10
  ...           ...
```

Source: https://mainisusuallyafunction.blogspot.com/2015/02/turing-tarpits-in-rusts-macro-system.html

```rust
/// http://esolangs.org/wiki/Bitwise_Cyclic_Tag
macro_rules! bct {
    // cmd 0:  d ... => ...
    (0, $($ps:tt),* ; $_d:tt)
        => (bct!($($ps),*, 0 ; ));
    (0, $($ps:tt),* ; $_d:tt, $($ds:tt),*)
        => (bct!($($ps),*, 0 ; $($ds),*));
    // cmd 1p:  1 ... => 1 ... p
    (1, $p:tt, $($ps:tt),* ; 1)
        => (bct!($($ps),*, 1, $p ; 1, $p));
    (1, $p:tt, $($ps:tt),* ; 1, $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; 1, $($ds),*, $p));
    // cmd 1p:  0 ... => 0 ...
    (1, $p:tt, $($ps:tt),* ; $($ds:tt),*)
        => (bct!($($ps),*, 1, $p ; $($ds),*));
    // halt on empty data string
    ( $($ps:tt),* ; )
        => (());
}
▶ Run | Debug
fn main() {
    trace_macros!(true);
    bct!(0, 0, 1, 1, 1 ; 1, 0, 1);
}
```

```
error: recursion limit reached while expanding `bct!`
  --> src\main.rs:19:13
   |
19 |         => (bct!($($ps),*, 0 ; $($ds),*));
   |             ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
...
34 |     bct!(0, 0, 1, 1, 1 ; 1, 0, 1);
   |     ------------------------------ in this macro invocation
   |
   = help: consider increasing the recursion limit by adding a
```

Downside: The Rust compiler has a recursion limit on macros :^)

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
    - Rust has a really strong macro system, allowing us to do all sorts of stuff
    - It's so powerful, it's turing-complete*
    - Allows us to work on Rust code itself as inputs and outputs
      - Code Generators

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
    - Rust idiomatic code does not like for

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
    - Rust idiomatic code does not like for
    - Instead, we have powerful Iterators which we can work with
      - filter()
      - map()

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
    - Rust idiomatic code does not like for
    - Instead, we have powerful Iterators which we can work with
    - Allows for lazy-evaluation and more optimizations

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
    - How can we create systems working on the same data, using multiple threads?
      - Example: You have a list of 1 billion elements, and want to concurrently work on them

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
    - How can we create systems working on the same data, using multiple threads?
    - How do we prevent race conditions and deadlocks without tanking performance?

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
    - How can we find hotspots in our code?

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
    - How can we find hotspots in our code?
    - How do we know if a certain implementation of a function is better?

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
- The focus will be more on the problems, and what you can do to solve them in Rust

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
- The focus will be more on the problems, and what you can do to solve them in Rust
  - There is no „single trick to beat them all", everything has pros and cons
    - Technically, cloning also solves the Borrow Checker :^)

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
- The focus will be more on the problems, and what you can do to solve them in Rust
  - There is no „single trick to beat them all“, everything has pros and cons
  - By looking at the problems, you can extrapolate solutions to other programming languages
    - Rust has procedural macros to do X, can Python do something similar?
    - In Rust Y is recommended for multithreading, does C++ also support that?
    - Rust has Rc<T>, that's basically what Java does internally! → Now we can also optimize our Java code ☺

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
- The focus will be more on the problems, and what you can do to solve them in Rust
- Each session will follow the same pattern

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
- The focus will be more on the problems, and what you can do to solve them in Rust
- Each session will follow the same pattern
  1. Try something that will not work (or at least takes a lot of effort) with what we know so far

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
- The focus will be more on the problems, and what you can do to solve them in Rust
- Each session will follow the same pattern
  1. Try something that will not work (or at least takes a lot of effort) with what we know so far
  2. Introduce the problem, why does our initial attempt not work?

1. Introduction

- Welcome to this Rust course!
- Learning Rust is easy, Mastering it is hard
- In this course, we will attempt to solve problems that arise from time to time
  - Lifetimes and Borrowing
  - Metaprogramming
  - Functional Programming
  - Multithreading
  - Profiling
- The focus will be more on the problems, and what you can do to solve them in Rust
- Each session will follow the same pattern
  1. Try something that will not work (or at least takes a lot of effort) with what we know so far
  2. Introduce the problem, why does our initial attempt not work?
  3. What does Rust offer to solve the problem?

2. Recap of Rust Basics

- Warning: I will now proceed to dump a semester worth of content into a single slide :^)

2. Recap of Rust Basics

– Rust is a statically typed, compiled language

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
  - Every variable, every literal, everything has a type which can't be changed once determined
    - Type inference allows us to omit the type annotation

```
▶ Run | Debug
fn main() {
    let a: i32 = 10;
    let b: i32 = a; // also i32
}
```

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
  - Every variable, every literal, everything has a type which can't be changed once determined
  - Static typechecker will catch errors at compile time

```
error[E0308]: mismatched types
  --> src\main.rs:14:17
   |
14 |     let b = a + 5u8; // also i32
   |                 ^^^ expected `i32`, found `u8`

error[E0277]: cannot add `u8` to `i32`
  --> src\main.rs:14:15
   |
14 |     let b = a + 5u8; // also i32
   |               ^ no implementation for `i32 + u8`
```

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
  - Every variable, every literal, everything has a type which can‘t be changed once determined
  - Static typechecker will catch errors at compile time
  - Compiled means machine code, which runs directly on your CPU → Faster than interpretation

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values

```rust
fn main() {
    let mut a: i32 = 10;
    a = 15;
    let b: i32 = 20;
    b = 30;
}
```

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops

```rust
fn main() {
    loop { println!("Whheeeee!!!"); }
    let cond: bool = true;
    while cond {}
    for number: i32 in [1, 2, 3] {
        println!("{}", number);
    }
}
```

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
  - Every variable, every value, everything in Rust has exactly one owner
  - Values are dropped when the owner is dropped
    - Variables are dropped at the end of the scope they are defined in
  - Ownership-Conflicts are resolved by moving ownership, if the underlying value can't be copied

2. Recap of Rust Basics

```rust
fn main() {
    let v: Vec<i32> = vec![1, 2];
    let v1: Vec<i32> = v; // v moved here
    {
        let v2: Vec<i32> = v1; // v1 moved here
    } // v2 dropped here
    let a: i32 = 10;
    let b: i32 = a; // i32 is copied
    println!("v1: {:?}", v1); // can't use v1 here
}
```

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
  - Way of not moving or copying data when it’s not needed
  - References
  - Can be mutable `&mut` or immutable `&`

2. Recap of Rust Basics

```rust
fn main() {
    let v: Vec<i32> = vec![1, 2];
    let v1: &Vec<i32> = &v; // ref to v
    {
        let v2: &&Vec<i32> = &v1; // ref to v1
    } // v2 dropped here, v1 still valid
    let a: i32 = 10;
    let b: i32 = a; // i32 is copied
    println!("v1: {:?}", v1); // can use v1 here
}
```

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
  - Mechanism to guarantee memory safety at compile time
  - 0 mutable references to a value → infinite immutable references allowed
  - 1 mutable reference to a value → zero immutable references allowed
  - more than 1 mutable reference to a value → illegal, compiler error
  - References must outlive original value → no dangling references
  - References only count when they are used → Lifetimes

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
}
```

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
}
```

Lifetime* of original value

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
}
```

Lifetime* of original value

Lifetime of mutable reference

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
}
```

Lifetime* of original value
Lifetime of mutable reference
Reference does not outlive original, everything is okay

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
    ref_v.push(4);
}
```

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
    ref_v.push(4);
}
```

Lifetime* of original value

2. Recap of Rust Basics

```
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
    ref_v.push(4);
}
```

Lifetime* of original value

Lifetime of mutable reference

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
    ref_v.push(4);
}
```

Lifetime* of original value
Lifetime of mutable reference
println creates an immutable reference

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
    ref_v.push(4);
}
```

Lifetime* of original value
Lifetime of mutable reference
println creates an immutable reference

Mutable and Immutable overlap → Compilation error

2. Recap of Rust Basics

```rust
fn main() {
    let mut v: Vec<i32> = vec![1, 2];
    let ref_v: &mut Vec<i32> = &mut v;
    ref_v.push(3);
    println!("v: {:?}", v);
    ref_v.push(4);
}
```

```
error[E0502]: cannot borrow `v` as immutable because it is also borrowed as mutable
  --> src\main.rs:16:25
   |
14 |     let ref_v = &mut v;
   |                 ------ mutable borrow occurs here
15 |     ref_v.push(3);
16 |     println!("v: {:?}", v);
   |                         ^ immutable borrow occurs here
17 |     ref_v.push(4);
   |     ----- mutable borrow later used here
   |
```

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions fn
  - Take in parameters, and may return values
  - Overloading functions does not exist in Rust
  - No default arguments

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions fn
- Structs struct
  - User-created data types
    - As such, can be used everywhere where types are required
      - Variable types
      - Function parameters
      - Struct fields
  - Made out of fields
    - Field names must be unique
  - Values of structs are called instances

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions `fn`
- Structs `struct`
- Associated functions `impl`
  - Associate a function with a given type
  - Used by calling `<type_name>::<fn_name>()`
  - `Vec::new()` is different from `String::new()`, which is also different from `new()`

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions fn
- Structs struct
- Associated functions impl
- Methods
  - Associated functions where the first parameter is one of self, &self or &mut self
  - Can be called on instances of structs using <instance>.<method_name>()

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions `fn`
- Structs `struct`
- Associated functions `impl`
- Methods
- Traits `trait`
  - Traits are contracts
  - Implemented for a type by `impl <trait_name> for <type_name> { <functions> }`
  - Allow us to generalize our code
    - Instead of requiring a specific type, we can accept anything that implements a given trait

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions fn
- Structs struct
- Associated functions impl
- Methods
- Traits trait
- Enums enum
  - Enum values are part of a finite set of Enum variants
    - Types of dog breeds is finite
    - Token types in a compiler is a finite, known set

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions fn
- Structs struct
- Associated functions impl
- Methods
- Traits trait
- Enums enum
- Pattern Matching match
  - match allows us to control the flow of the program → if-else on steroids
  - Allows us to access the data behind enum variants
  - Allows us to bind values to variables
  - Very powerful, almost everything in Rust uses it

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions fn
- Structs struct
- Associated functions impl
- Methods
- Traits trait
- Enums enum
- Pattern Matching match
- Generics <T>
  - Allow us to generalize our code by writing data structures and functions that work with any type
    - Can be restricted using trait boundaries <T: Display + Debug>
  - Most famous: Option<T> and Result<T, E>

2. Recap of Rust Basics

- Rust is a statically typed, compiled language
- Rust allows us to specify the mutability of values
- Rust has three types of loops
- Ownership-Model
- Borrowing
- Borrow-Checker
- Functions fn
- Structs struct
- Associated functions impl
- Methods
- Traits trait
- Enums enum
- Pattern Matching match
- Generics <T>

3. General Info

- Slides are available on the Github repository

3. General Info

- Slides are available on the Github repository
  - I recommend cloning the repo, and doing regular pulls
  - Alternatively, Github also supports PDF rendering
  - Slides for the next session are always uploaded the weekend before

3. General Info

- Slides are available on the Github repository
- Every session will be split into three parts
  - Recap of last session
  - New topics
  - Exercises at the end

3. General Info

- Slides are available on the Github repository
- Every session will be split into three parts
- Exercises and Code snippets in future slides will be colored coded
  - Green → 0/3 → We have covered the topic already, should be easy enough
  - Yellow → 1/3 → We have just covered the topic, may be hard
  - Red → 2/3 → Same as Yellow, but trickier
  - Purple → 3/3 → We have not covered the topic, but challenges are always fun

3. General Info

- Slides are available on the Github repository
- Every session will be split into three parts
- Exercises and Code snippets in future slides will be colored coded
- Participation and Feedback is very important
  - Basic program stands, but my goal is to teach you Rust the best I can
  - Don’t understand something? Am I too fast? Did I make any mistake?
    - Just raise your hand, we can discuss a topic for a while! ☺
    - Slides are more of a guideline, technical conversations are always appreciated ☺

4. Next time

- So, what *are* Lifetimes?
- Slices