RUSTikales Rust for beginners

Plan for today

1. Recap
2. Ownership
3. Borrow Checker

1. Recap

- Primitive Types in Rust: i8, u8, …, i128, u128, bool

1. Recap

- Primitive Types in Rust: i8, u8, …, i128, u128, bool
- let → immutable variable
- let mut → mutable variable

1. Recap

- Primitive Types in Rust: i8, u8, …, i128, u128, bool
- let → immutable variable
- let mut → mutable variable
- Arrays: [type; size]
- Vectors: Vec<type>
- var[index] to access an element at a given index

1. Recap

- Primitive Types in Rust: i8, u8, …, i128, u128, bool
- let → immutable variable
- let mut → mutable variable
- Arrays: [type; size]
- Vectors: Vec<type>
- var[index] to access an element at a given index
- loop {} to create an infinite loop
- while condition {} to create a conditional loop
- for elem in collection {} to create an iterator over a collection

1. Recap

- Primitive Types in Rust: i8, u8, …, i128, u128, bool
- let → immutable variable
- let mut → mutable variable
- Arrays: [type; size]
- Vectors: Vec<type>
- var[index] to access an element at a given index
- loop {} to create an infinite loop
- while condition {} to create a conditional loop
- for elem in collection {} to create an iterator over a collection
- loop, while and for are equally powerful, but often certain loops are better
  - Infinite loops using for is convoluted
  - Iterating over a collection with loop is a lot of work

1. Recap

- Important to know:
  - for n in x..y → n stops before y
  - for n in x..=y → n includes y

1. Recap

- Important to know:
  - for n in x..y → n stops before y
  - for n in x..=y → n includes y
  - use break to exit out of a loop early

1. Recap

- Important to know:
  - for n in x..y → n stops before y
  - for n in x..=y → n includes y
  - use break to exit out of a loop early
  - use continue to skip one loop pass

1. Recap

```rust
fn main() {
    for n: i32 in 0..10 {
        if n < 5 {
            continue;
        }
        if n == 7 {
            break;
        }
        println!("n: {}", n);
    }
}
```

1. Recap

```
1  fn main() {
2      for n: i32 in 0..10 {
3          if n < 5 {
4              continue;
5          }
6          if n == 7 {
7              break;
8          }
9          println!("n: {}", n);
10     }
11 }
```

Jump to line 2 if n is less than 5

1. Recap

```
1  fn main() {
2          for n: i32 in 0..10 {
3                  if n < 5 {
4                          continue;
5                  }
6                  if n == 7 {
7                          break;
8                  }
9                  println!("n: {}", n);
10         }
11 }
```

Jump to line 10 if n is equal to 7

1. Recap

1/3

```rust
fn main() {
    for n: i32 in 0..10 {
        if n < 5 {
            continue;
        }
        if n == 7 {
            break;
        }
        println!("n: {}", n);
    }
}
```

What do we print in the console?

1. Recap

1/3

```rust
fn main() {
    for n: i32 in 0..10 {
        if n < 5 {
            continue;
        }
        if n == 7 {
            break;
        }
        println!("n: {}", n);
    }
}
```

What do we print in the console?

```
n: 5
n: 6
```

1. Recap

- Important to know:
  - for n in x..y → n stops before y
  - for n in x..=y → n includes y
  - use break to exit out of a loop early
  - use continue to skip one loop pass

1. Recap

- Important to know:
  - for n in x..y → n stops before y
  - for n in x..=y → n includes y
  - use break to exit out of a loop early
  - use continue to skip one loop pass
  - you can nest loops
    - Nesting means putting a structure inside itself
    - Similar to nested arrays

1. Recap

- Important to know:
  - for n in x..y → n stops before y
  - for n in x..=y → n includes y
  - use break to exit out of a loop early
  - use continue to skip one loop pass
  - you can nest loops
  - Loops allow us to control the flow of the program
    - We can now implement simple algorithms, such as:
      - The factorial of a number → n!
      - Primality test → Is n a prime?

1. Recap

```rust
fn main() {
    let n: u32 = 10;
    let mut result: u32 = 1;
    for i: u32 in 1..=n {
        result *= i;
    }
    println!("  {}! is {}", n, result);
}
```

n! = n * (n – 1) * (n – 2) * ... * 2 * 1

1. Recap

```rust
fn main() {
    let n: u32 = 10;
    let mut result: u32 = 1;
    for i: u32 in 1..=n {
        result *= i;
    }
    println!("  {}! is {}", n, result);
}
```

n! = n * (n – 1) * (n – 2) * ... * 2 * 1

1. Recap

$n! = n * (n – 1) * (n – 2) * \dots * 2 * 1$

```rust
fn main() {
    let n: u32 = 10;
    let mut result: u32 = 1;
    for i: u32 in 1..=n {
        result *= i;
    }
    println!("  {}! is {}", n, result);
}
```

1. Recap

```rust
fn main() {
    let n: u32 = 10;
    let mut result: u32 = 1;
    for i: u32 in 1..=n {
        result *= i;
    }
    println!("  {}! is {}", n, result);
}
```

n! = n * (n – 1) * (n – 2) * ... * 2 * 1

```
10! is 3628800
```

1. Recap

WolframAlpha:

Input

10!

Result

3 628 800

```rust
fn main() {
    let n: u32 = 10;
    let mut result: u32 = 1;
    for i: u32 in 1..=n {
        result *= i;
    }
    println!("{}! is {}", n, result);
}
```

```
10! is 3628800
```

2. Ownership

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop

```rust
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3];
    for elem: i32 in vec {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

2. Ownership

– Last time: Using a Vector in a for-loop made the variable invalid after the loop

```rust
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3];
    for elem: i32 in vec {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

red wiggly lines are never good!

2. Ownership

– Last time: Using a Vector in a for-loop made the variable invalid after the loop

```rust
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3];
    for elem: i32 in vec {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

Dots mean the mistake lies here

red wiggly lines are never good!

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop
- Let‘s take a closer look at the error

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop
- Let's take a closer look at the error

```
error[E0382]: borrow of moved value: `vec`
   --> src\main.rs:6:52
    |
2   |     let vec: Vec<i32> = vec![1, 2, 3];
    |         --- move occurs because `vec` has type `Vec<i32>`, which does not implement the `Copy` trait
3   |     for elem in vec {
    |                 --- `vec` moved due to this implicit call to `.into_iter()`
...
6   |     println!("Now we can't use vec anymore! {:?}", vec);
    |                                                    ^^^ value borrowed here after move
    |
note: `into_iter` takes ownership of the receiver `self`, which moves `vec`
```

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop
- Let‘s take a closer look at the error

```
error[E0382]: borrow of moved value: `vec`
   --> src\main.rs:6:52
    |
2   |     let vec: Vec<i32> = vec![1, 2, 3];
    |         --- move occurs because `vec` has type `Vec<i32>`, which does not implement the `Copy` trait
3   |     for elem in vec {
    |                 --- `vec` moved due to this implicit call to `.into_iter()`
...
6   |     println!("Now we can't use vec anymore! {:?}", vec);
    |                                                    ^^^ value borrowed here after move
    |
note: `into_iter` takes ownership of the receiver `self`, which moves `vec`
```

A lot of words, let‘s focus on the important bits

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop
- Let‘s take a closer look at the error

A lot of words, let‘s focus on the important bits

```
error[E0382]: borrow of moved value: `vec`

        let vec: Vec<i32> = vec![1, 2, 3];
            --- move                          `Vec<i32>`,          not implement the `Copy` trait
        for elem in vec {
                    --- `vec` moved due to this implicit call to `.into_iter()`

        println!("Now we can't use vec anymore! {:?}", vec);
                                                        ^^^ value borrowed here after move

note: `into_iter` takes ownership                          , which moves `vec`
```

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop
- Let's take a closer look at the error

```
error[E0382]: borrow of moved value: `vec`

        for elem in vec {
                    --- `vec` moved

                                                vec);
                                                ^^^ value borrowed

              takes ownership                  , which moves `vec`
```

A lot of words, let's focus on the important bits, boiling it down even more

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop
- Let's take a closer look at the error
- Error boiled down to three keywords:
  - Borrowed
  - Moved
  - Ownership

2. Ownership

- Last time: Using a Vector in a for-loop made the variable invalid after the loop
- Let's take a closer look at the error
- Error boiled down to three keywords:
  - Borrowed
  - Moved
  - Ownership
- Today, we'll dive deep down into the world of Rust

2. Ownership

- At some point, every compiler constructor has to address the elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
  - Manual Management, like in C or Assembly

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
  - Manual Management, like in C or Assembly
  - Garbage Collection, like in Java or Python

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
  - Manual Management, like in C or Assembly
  - Garbage Collection, like in Java or Python
  - Automatic Reference Counting, like in Swift

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
  - Manual Management, like in C or Assembly
  - Garbage Collection, like in Java or Python
  - Automatic Reference Counting, like in Swift
  - Ownership-Model, like in Rust or… Well, so far only Rust has really pulled it off (and maybe C++)

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
- The Ownership-Model is the technique used in Rust, it controls everything

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
- The Ownership-Model is the technique used in Rust, it controls everything
- Set of rules

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
- The Ownership-Model is the technique used in Rust, it controls everything
- Set of rules
  - Each value has an owner
    - Your 5 on the stack has an owner
    - Your elements on the heap have an owner

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
- The Ownership-Model is the technique used in Rust, it controls everything
- Set of rules
  - Each value has an owner
  - There can only be exactly one owner at any given time
    - This is related to the Vector-problem we faced earlier

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
- The Ownership-Model is the technique used in Rust, it controls everything
- Set of rules
  - Each value has an owner
  - There can only be exactly one owner at any given time
  - When the owner is dropped, the value is dropped (memory is freed)

2. Ownership

- The elephant in the room: Memory Management
  - How do you handle data structures, how do you handle heap allocations, etcetc
- Many different techniques exist
- The Ownership-Model is the technique used in Rust, it controls everything
- Set of rules
  - Each value has an owner
  - There can only be exactly one owner at any given time
  - When the owner is dropped, the value is dropped (memory is freed)
    - Almost always initiated by a variable going out of scope
    - Drop is recursive
      - Scope drops variable → variable drops Vector → Vector drops elements, which can drop other things

2. Ownership

```rust
fn main() {
    let a: i32 = 0;
    {
        let b: i32 = 1;
        if b == 1 {
            let v: Vec<i32> = vec![1];
            println!("{:?}", v);
        }
    }
    println!("{}", a);
}
```

2. Ownership

```rust
fn main() {
    let a: i32 = 0;
    {
        let b: i32 = 1;
        if b == 1 {
            let v: Vec<i32> = vec![1];
            println!("{:?}", v);
        } v dropped here → Vector dropped here → Vector elements dropped here
    }
    println!("{}", a);
}
```

2. Ownership

```rust
fn main() {
    let a: i32 = 0;
    {
        let b: i32 = 1;
        if b == 1 {
            let v: Vec<i32> = vec![1];
            println!("{:?}", v);
        } v dropped here → Vector dropped here → Vector elements dropped here
    } b dropped here → Value 1 dropped here
    println!("{}", a);
}
```

2. Ownership

```rust
fn main() {
    let a: i32 = 0;
    {
        let b: i32 = 1;
        if b == 1 {
            let v: Vec<i32> = vec![1];
            println!("{:?}", v);
        } v dropped here → Vector dropped here → Vector elements dropped here
    } b dropped here → Value 1 dropped here
    println!("{}", a);
} a dropped here → Value 0 dropped here
```

2. Ownership

– Ownership-Conflicts are resolved by moving or copying the data

2. Ownership

- Ownership-Conflicts are resolved by moving or copying the data
- Whether a value is copied or moved is based on the trait system

2. Ownership

- Ownership-Conflicts are resolved by moving or copying the data
- Whether a value is copied or moved is based on the trait system
  - Types implementing the Copy-trait are copied
  - Otherwise, they are moved

2. Ownership

```rust
fn main() {
    let x: i32 = 0;
    let y: i32 = x;
}
```

2. Ownership

```rust
fn main() {
    let x: i32 = 0;
    let y: i32 = x;
}
```

i32 is a simple primitive type that implements the Copy-trait
→ Value 0 is copied, we can still use x after assigning to y

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

Vec<T> does not implement the Copy-trait:

→ Vec is generic, meaning we can put *any* type in it

→ But we can‘t guarantee that we can copy every type we put in there!

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

Vec<T> does not implement the Copy-trait:
→ Vec is generic, meaning we can put *any* type in it
→ But we can‘t guarantee that we can copy every type we put in there!

However, we still copy *some* data

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

Vector is a bit more complex: The data itself is located on the Heap, there‘s only Metadata on the Stack:
→ Pointer to the Heap
→ Length of Vector
→ Capacity of Vector

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

Stack	
ptr	
len	3
capacity	4

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

Stack	
ptr	0x1000
len	3
capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

Even if values are moved, the data on the stack is still copied!

Stack	
ptr	0x1000
len	3
capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

2. Ownership

Stack		
v1	ptr	0x1000
	len	3
	capacity	4
v2	ptr	0x1000
	len	3
	capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

2. Ownership

Stack		
v1	ptr	0x1000
	len	3
	capacity	4
v2	ptr	0x1000
	len	3
	capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

This is very bad! v1 and v2 now point to the same heap location, and when dropped will both free the same memory!

2. Ownership

Stack		
v1	ptr	0x1000
	len	3
	capacity	4
v2	ptr	0x1000
	len	3
	capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

This is very bad! v1 and v2 now point to the same heap location, and when dropped will both free the same memory!
Very bad for many reasons!

2. Ownership

Stack		
v1	ptr	0x1000
	len	3
	capacity	4
v2	ptr	0x1000
	len	3
	capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

What does the compiler do?

It invalidates v1, and moves the data into v2

2. Ownership

Stack		
v1	ptr	???
	len	???
	capacity	???
v2	ptr	0x1000
	len	3
	capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

What does the compiler do?

It invalidates v1, and moves the data into v2

2. Ownership

Stack		
v1	ptr	???
	len	???
	capacity	???
v2	ptr	0x1000
	len	3
	capacity	4

Heap	
0x1000	1
0x1004	2
0x1008	3
0x100C	90123987

What does the compiler do?

It invalidates v1, and moves the data into v2

By doing that, the data on the heap will only be freed once, everything is fine!
Downside is that you can‘t use v1 anymore, until you re-assign a value to it.

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1;
}
```

After the 2nd line, v1 was moved into v2 and can‘t be used anymore, until you re-assign to v1.

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1.clone();
}
```

We can still copy explicitly using .clone()

2. Ownership

```rust
fn main() {
    let v1: Vec<i32> = vec![1, 2, 3];
    let v2: Vec<i32> = v1.clone();
}
```

We can still copy explicitly using .clone()
The Clone-trait implementation also clones the underlying elements, so everything is fine

2. Ownership

– Let’s go back to the Vector example:

```
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3];
    for elem: i32 in vec {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

2. Ownership

- Let's go back to the Vector example:

```
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3];
    for elem: i32 in vec {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

We now understand that for moves the value

2. Ownership

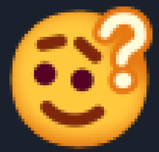
Easy fix, clone it!!

- Let's go back to the Vector example:

```rust
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3];
    for elem: i32 in vec.clone() {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

2. Ownership

Easy fix, clone it!?



– Let's go back to the Vector example:

```
fn main() {
    let vec: Vec<i32> = vec![1, 2, 3];
    for elem: i32 in vec.clone() {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

2. Ownership

```
fn main() {
    let vec: Vec<i32> = vec![1; 100_000_000];
    for elem: i32 in vec.clone() {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

2. Ownership

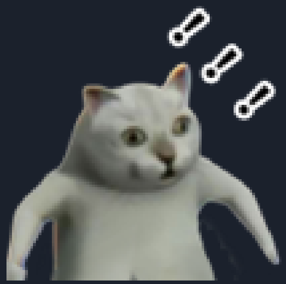
Cloning might take a while...

```
fn main() {
    let vec: Vec<i32> = vec![1; 100_000_000];
    for elem: i32 in vec.clone() {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

2. Ownership

```rust
fn main() {
    let vec: Vec<i32> = vec![1; 100_000_000];
    for elem: i32 in vec.clone() {
        println!("We're doing something with {}", elem);
    }
    println!("Now we can't use vec anymore! {:?}", vec);
}
```

Cloning might take a while...



Back to our original problem anyway!! We want to modify vec, not any copies of it!

Intermission - References

– References offer an additional way of accessing values

Intermission - References

- References offer an additional way of accessing values
- References do not involve copying or moving, and do not invalidate original variables

Intermission - References

- References offer an additional way of accessing values
- References do not involve copying or moving, and do not invalidate original variables
- A reference in programming is similar to a real life reference:

Intermission - References

- References offer an additional way of accessing values
- References do not involve copying or moving, and do not invalidate original variables
- A reference in programming is similar to a real life reference:
  - When you're referring to something, you don't own it, but simply point to it

Intermission - References

- References offer an additional way of accessing values
- References do not involve copying or moving, and do not invalidate original variables
- A reference in programming is similar to a real life reference:
  - When you’re referring to something, you don‘t own it, but simply point to it
  - A reference in a book might point to another book, written by another author
  - That other book might change in the meantime, but the reference still points to it

Intermission - References

- References offer an additional way of accessing values
- References do not involve copying or moving, and do not invalidate original variables
- A reference in programming is similar to a real life reference:
  - When you're referring to something, you don't own it, but simply point to it
  - A reference in a book might point to another book, written by another author
  - That other book might change in the meantime, but the reference still points to it
- References in Rust do the same, they simply point to a value

Intermission - References

- References offer an additional way of accessing values
- References do not involve copying or moving, and do not invalidate original variables
- A reference in programming is similar to a real life reference:
  - When you're referring to something, you don't own it, but simply point to it
  - A reference in a book might point to another book, written by another author
  - That other book might change in the meantime, but the reference still points to it
- References in Rust do the same, they simply point to a value
- In the context of Ownership, a reference is called borrowing:
  - „As in real life, if a person owns something, you can borrow it from them. When you're done, you have to give it back. You don't own it." – Rustdocs

Intermission - References

```rust
fn main() {
    let a: i32 = 0;
    let b: &i32 = &a;
}
```

Intermission - References

```rust
fn main() {
    let a: i32 = 0;
    let b: &i32 = &a;
}
```

Reference to a

Intermission - References

```rust
fn main() {
    let a: i32 = 0;
    let b: &i32 = &a;
}
```

Type: Reference to i32

Intermission - References

```rust
fn main() {
    let a: i32 = 0;
    let b: &i32 = &a;
}
```

That means: b contains the memory address of a

Intermission - References

```rust
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &Vec<i32> = &v1;
}
```

Intermission - References

```rust
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &Vec<i32> = &v1;
}
```

Reference to v1, no copy or move involved!
v1 is still valid after this line!

Intermission - References

```rust
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &Vec<i32> = &v1;
}
```

Addr	Stack		
0x4000	v1	ptr	0x1000
		len	100_000_000
		cap	100_000_000

Heap	
0x1000	1
0x1004	1
0x1008	1
...	...

Intermission - References

```
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &Vec<i32> = &v1;
}
```

Addr	Stack		
0x4000	v1	ptr	0x1000
		len	100_000_000
		cap	100_000_000
0x4020	v2	ptr	0x4000

Heap	
0x1000	1
0x1004	1
0x1008	1
...	...

Intermission - References

```rust
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &mut Vec<i32> = &mut v1;
    v2.push(1);
    println!("{}", v1.len());
}
```

Intermission - References

```
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &mut Vec<i32> = &mut v1;
    v2.push(1);
    println!("{}", v1.len());
}
```

Mutable reference

Intermission - References

```rust
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &mut Vec<i32> = &mut v1;
    v2.push(1);
    println!("{}", v1.len());
}
```

Pushing a value to a reference doesn‘t make sense
→ Rust automatically dereferences a reference if necessary

Intermission - References

```rust
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &mut Vec<i32> = &mut v1;
    v2.push(1);
    println!("{}", v1.len());
}
```

Pushing a value to a reference doesn‘t make sense
→ Rust automatically dereferences a reference if necessary

→ This line pushes an element to v1

Intermission - References

```rust
fn main() {
    let mut v1: Vec<i32> = vec![1; 100_000_000];
    let v2: &mut Vec<i32> = &mut v1;
    v2.push(1);
    println!("{}", v1.len());
}
```

Pushing a value to a reference doesn‘t make sense
→ Rust automatically dereferences a reference if necessary

→ This line pushes an element to v1

```
100000001
```

3. Borrow Checker

3. Borrow Checker

- Rust is all about memory safety

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
  - Ask your local Java developer, he'll tell you a story or two

```java
public class Main {
    private static class Test {
        int a = 10;
    }
    private static void someFunction(Test t) {
        System.out.println(t.a);
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = t1;
        Test t3 = t1;
        t2.a = 5;
        someFunction(t1);
        someFunction(t3);
    }
}
```

```java
public class Main {
    private static class Test {
        int a = 10;
    }
    private static void someFunction(Test t) {
        System.out.println(t.a);
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = t1;
        Test t3 = t1;
        t2.a = 5;
        someFunction(t1);
        someFunction(t3);
    }
}
```

Object assignments in Java are references by default

```java
public class Main {
    private static class Test {
        int a = 10;
    }
    private static void someFunction(Test t) {
        System.out.println(t.a);
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = t1;
        Test t3 = t1;
        t2.a = 5;
        someFunction(t1);
        someFunction(t3);
    }
}
```

Objects are initialized with .a = 10

```java
public class Main {
    private static class Test {
        int a = 10;
    }
    private static void someFunction(Test t) {
        System.out.println(t.a);
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = t1;
        Test t3 = t1;
        t2.a = 5;
        someFunction(t1);
        someFunction(t3);
    }
}
```

Modifying t2 also modifies t1 and t3

```java
public class Main {
    private static class Test {
        int a = 10;
    }
    private static void someFunction(Test t) {
        System.out.println(t.a);
    }
    public static void main(String[] args) {
        Test t1 = new Test();
        Test t2 = t1;
        Test t3 = t1;
        t2.a = 5;
        someFunction(t1);
        someFunction(t3);
    }
}
```

Objects are initialized with .a = 10
→ Reasonable to expect t1.a=10

Prints 5 for both t1.a and t3.a!
Depending on the situation, you may not have wanted that!

Do something with t1 and t3

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
  - Ask your local Java developer, he'll tell you a story or two
  - Ask your local Multithreading developer, he'll tell you a story or two

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
  - Ask your local Java developer, he'll tell you a story or two
  - Ask your local Multithreading developer, he'll tell you a story or two
- Race condition:
  - Multiple references access the same data at the same time
  - One reference writes data
  - At the same time, other reference reads data
    - Does it see the new value, or the old value?

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
- Race condition
- To prevent this, Rust has the Borrow Checker

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
- Race condition
- To prevent this, Rust has the Borrow Checker
- Guarantees at compile time that no data races or race conditions happen

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
- Race condition
- To prevent this, Rust has the Borrow Checker
- Guarantees at compile time that no data races or race conditions happen
- Set of rules that must be true at any point in your program

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
- Race condition
- To prevent this, Rust has the Borrow Checker
- Guarantees at compile time that no data races or race conditions happen
- Set of rules that must be true at any point in your program
  - Reference mutably borrowed twice+ → Not allowed, illegal
    - Race condition: What happens if both write at the same time?

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
- Race condition
- To prevent this, Rust has the Borrow Checker
- Guarantees at compile time that no data races or race conditions happen
- Set of rules that must be true at any point in your program
  - Reference mutably borrowed twice+ → Not allowed, illegal
  - Reference mutably borrowed once → No other references allowed, even immutable
    - Race condition

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
- Race condition
- To prevent this, Rust has the Borrow Checker
- Guarantees at compile time that no data races or race conditions happen
- Set of rules that must be true at any point in your program
  - Reference mutably borrowed twice+ → Not allowed, illegal
  - Reference mutably borrowed once → No other references allowed, even immutable
  - Reference immutably borrowed → Only other immutable borrows allowed
    - Immutable borrow is readonly, 100 Reads don't change the value

3. Borrow Checker

- Rust is all about memory safety
- References are powerful, but can also lead to all sorts of bugs if not treated carefully
- Race condition
- To prevent this, Rust has the Borrow Checker
- Guarantees at compile time that no data races or race conditions happen
- Set of rules that must be true at any point in your program
  - Reference mutably borrowed twice+ → Not allowed, illegal
  - Reference mutably borrowed once → No other references allowed, even immutable
  - Reference immutably borrowed → Only other immutable borrows allowed
  - Reference may not outlive borrowed data
    - If the original value was dropped in the meantime, we'd have dangling references

3. Borrow Checker

- The Borrow Checker checks those rules by evaluating the lifetimes of references
  - Will be covered next week

3. Borrow Checker

- The Borrow Checker checks those rules by evaluating the lifetimes of references
  - Will be covered next week
- TLDR for now:
  - References only fall into those categories for the duration they‘re used

3. Borrow Checker

```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &mut Vec<i32> = &mut vector;
    let v3: &mut Vec<i32> = &mut vector;
    let v4: &mut Vec<i32> = &mut vector;
}
```

3. Borrow Checker

```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &mut Vec<i32> = &mut vector;
    let v3: &mut Vec<i32> = &mut vector;
    let v4: &mut Vec<i32> = &mut vector;
}
```

Even though we have 4 mutable references, it‘s fine because we‘re not doing anything with them!

3. Borrow Checker

```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &mut Vec<i32> = &mut vector;
    let v3: &mut Vec<i32> = &mut vector;
    let v4: &mut Vec<i32> = &mut vector;
    v1.push(3);
}
```

3. Borrow Checker

```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &mut Vec<i32> = &mut vector;
    let v3: &mut Vec<i32> = &mut vector;
    let v4: &mut Vec<i32> = &mut vector;
    v1.push(3);
}
```

Mutable reference used in this range
→ No other references allowed here

3. Borrow Checker

```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &mut Vec<i32> = &mut vector;
    let v3: &mut Vec<i32> = &mut vector;
    let v4: &mut Vec<i32> = &mut vector;
    v4.push(3);
}
```

3. Borrow Checker

```rust
fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &mut Vec<i32> = &mut vector;
    let v3: &mut Vec<i32> = &mut vector;
    let v4: &mut Vec<i32> = &mut vector;
    v4.push(3);
}
```

Mutable reference used in this range
→ No other references allowed here

Intermission - Exercises

– Time for exercises!

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut a: i32 = 0;
    if a == 0 {
        let b: &mut i32 = &mut a;
        *b = 10;
    }
    println!("{}", a);
}
```

Intermission - Exercises

2/3

```
pub fn main() {
    let mut a: i32 = 0;
    if a == 0 {
        let b: &mut i32 = &mut a;
        *b = 10;
    }
    println!("{}", a);
}
```

Does the code compile?
If yes, what does it print?

Intermission - Exercises

2/3

```
pub fn main() {
    let mut a: i32 = 0;
    if a == 0 {
        let b: &mut i32 = &mut a;
        *b = 10;
    }
    println!("{}", a);
}
```

Does the code compile?
If yes, what does it print?

b contains the memory address of a

Intermission - Exercises

2/3

```
pub fn main() {
    let mut a: i32 = 0;
    if a == 0 {
        let b: &mut i32 = &mut a;
        *b = 10;
    }
    println!("{}", a);
}
```

Does the code compile?
If yes, what does it print?

Dereference b
→ Get the original memory address
→ Gets address of a

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut a: i32 = 0;
    if a == 0 {
        let b: &mut i32 = &mut a;
        *b = 10;
    }
    println!("{}", a);
}
```

Writes 10 into a

Does the code compile?
If yes, what does it print?

Intermission - Exercises

2/3

```
pub fn main() {
    let mut a: i32 = 0;
    if a == 0 {
        let b: &mut i32 = &mut a;
        *b = 10;
    }
    println!("{}", a);
}
```

Does the code compile?
If yes, what does it print?

Writes 10 into a

It compiles, and prints:

```
10
```

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &Vec<i32> = &vector;
    v1.push(1);
    println!("{:?}", vector);
}
```

Intermission - Exercises

2/3

```
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &Vec<i32> = &vector;
    v1.push(1);
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &Vec<i32> = &vector;
    v1.push(1);
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

Mutable borrow here

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &Vec<i32> = &vector;
    v1.push(1);
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

Immutable borrow here

Mutable borrow here

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &Vec<i32> = &vector;
    v1.push(1);
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

Immutable borrow here

Mutable borrow here

Regions overlap
→ Borrow Checker violation
→ Code does not compile!

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    let v1: &mut Vec<i32> = &mut vector;
    let v2: &Vec<i32> = &vector;
    v1.push(1);
    println!("{:?}", vector);
}
```

```
error[E0502]: cannot borrow `vector` as immutable because it is also borrowed as mutable
 --> src\ex2.rs:4:25
  |
3 |     let v1: &mut Vec<i32> = &mut vector;
  |                             ----------- mutable borrow occurs here
4 |     let v2: &Vec<i32> = &vector;
  |                         ^^^^^^^ immutable borrow occurs here
5 |     v1.push(1);
  |     -- mutable borrow later used here
```

Intermission - Exercises

2/3

```
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

It does compile!

Intermission - Exercises

2/3

```
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

It does compile!

We‘re finally at a point where we can reuse the same Vector in a for-loop!

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

It does compile!

We're finally at a point where we can reuse the same Vector in a for-loop!

Those loops modify the elements of our original vector!

Intermission - Exercises

2/3

```rust
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

Mutable borrow in this range

Mutable borrow in this range

Intermission - Exercises

2/3

```
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

Does the code compile?
If yes, what does it print?

Mutable borrow in this range

No overlap → No Borrow Checker violation!

Mutable borrow in this range

Intermission - Exercises

2/3

```
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

```
true
true
false
false
```

Does the code compile?
If yes, what does it print?

Intermission - Exercises

2/3

```
pub fn main() {
    let mut vector: Vec<i32> = vec![1, 2];
    for i: i32 in 3..=10 {
        vector.push(i);
    }
    for elem: &mut i32 in &mut vector {
        *elem *= 2;
    }
    for elem: &mut i32 in &mut vector {
        *elem += 1;
    }
    println!("{}", vector.contains(&3));
    println!("{}", vector.contains(&11));
    println!("{}", vector.contains(&14));
    println!("{}", vector.contains(&20));
    println!("{:?}", vector);
}
```

```
true
true
false
false
[3, 5, 7, 9, 11, 13, 15, 17, 19, 21]
```

Does the code compile?
If yes, what does it print?

4. Next time

- Lifetimes
- Functions