

A decorative graphic on the left side of the slide. It consists of a blue parallelogram and a light green parallelogram, both tilted at an angle. The blue shape is in the foreground, and the green shape is partially behind it. They are set against a dark blue background with faint, lighter blue diagonal stripes.

RUSTikales Rust for beginners



Plan for today



Plan for today

1. Recap



Plan for today

1. Recap
2. Lifetimes



Plan for today

1. Recap
2. Lifetimes
3. Functions



1. Recap



1. Recap

- `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

- `loop {}` to create an infinite loop
- `while condition {}` to create a conditional loop
- `for elem in collection {}` to create an iterator over a collection
- **Ownership-Model**
 - Every value in Rust has **exactly one owner**
 - **Values are dropped** (memory is freed) **when the owner is dropped**
 - **Ownership-Conflicts** are resolved by **moving ownership**, if we can't copy the value



1. Recap

- `loop {}` to create an infinite loop
- `while condition {}` to create a conditional loop
- `for elem in collection {}` to create an iterator over a collection
- Ownership-Model
- References
 - Way to `borrow data without moving or copying it`
 - Can be immutable or mutable



1. Recap

- `loop {}` to create an infinite loop
- `while condition {}` to create a conditional loop
- `for elem in collection {}` to create an iterator over a collection
- Ownership-Model
- References
- Borrow Checker
 - zero mutable references to a value → infinite immutable references allowed
 - one mutable reference to a value → zero immutable references allowed
 - two+ mutable references to a value → forbidden, compiler error

1. Recap

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

1. Recap

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

Immutable borrow

1. Recap

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

Type: Reference to i32

1. Recap

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

Dereference **b** to get the original value



1. Recap

Addr	Stack
------	-------


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

1. Recap

Addr	Stack	
0x4000	a	10

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


1. Recap



Addr	Stack	
0x4000	a	10
0x4004	b	0x4000

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

1. Recap



Addr	Stack
0x4000	a 10
0x4004	b 0x4000

Read the value at that memory location

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

1. Recap



Addr	Stack	
0x4000	a	10
0x4004	b	0x4000

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

10

1. Recap

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

1. Recap

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

Immutable borrows

1. Recap

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

Implicit dereference



1. Recap

Addr	Stack
------	-------

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

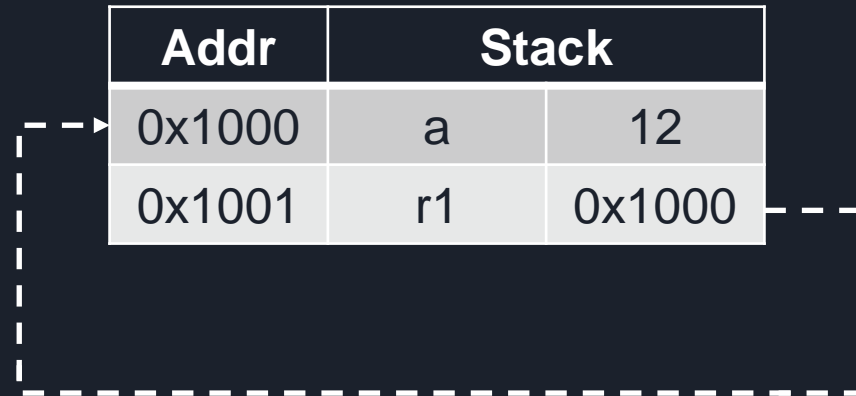


1. Recap

Addr	Stack	
0x1000	a	12

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

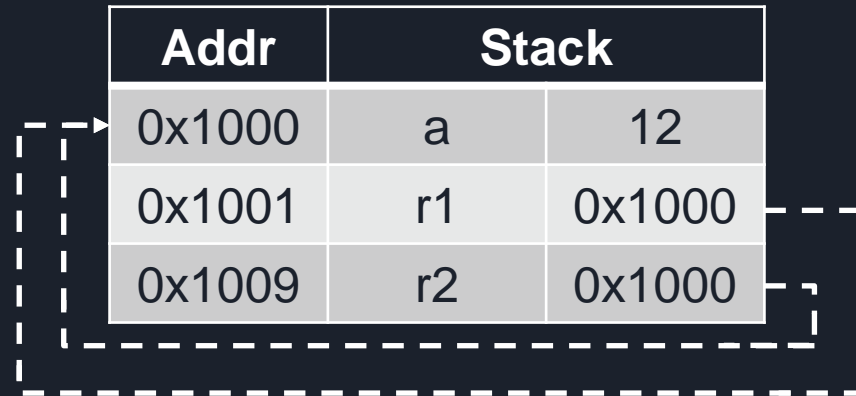

1. Recap



Addr	Stack	
0x1000	a	12
0x1001	r1	0x1000

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

1. Recap

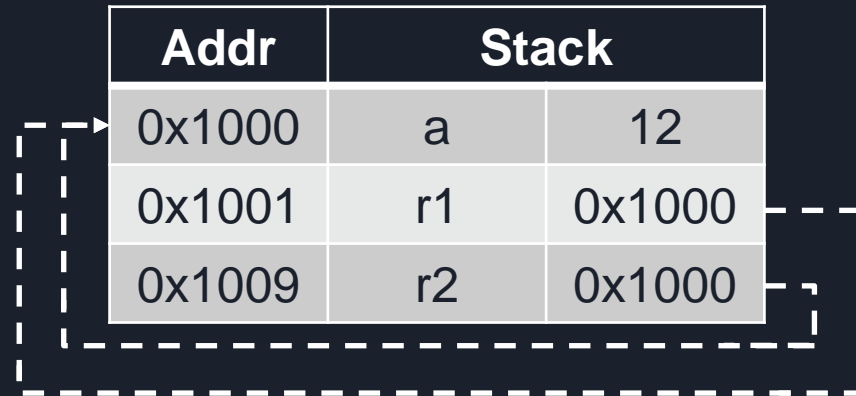


A diagram of a stack frame. It consists of a table with three rows and two columns: 'Addr' and 'Stack'. The first row has '0x1000' and 'a' with the value '12'. The second row has '0x1001' and 'r1' with the value '0x1000'. The third row has '0x1009' and 'r2' with the value '0x1000'. A dashed white box surrounds the entire table, and a dashed arrow points from the left towards the first row.

Addr	Stack
0x1000	a 12
0x1001	r1 0x1000
0x1009	r2 0x1000

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

1. Recap



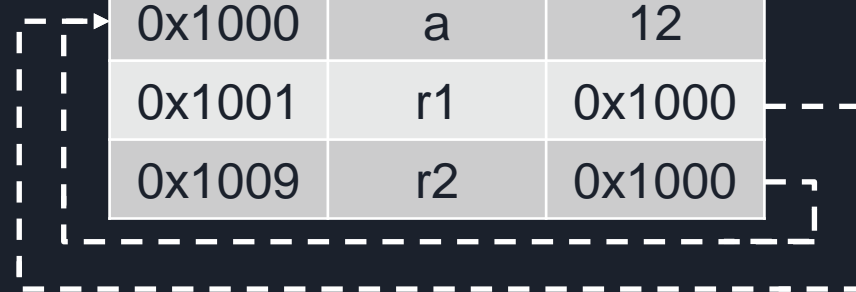
A diagram illustrating a stack frame. It consists of a table with three rows and three columns. The first column is labeled 'Addr' and the second and third columns are grouped under the label 'Stack'. The rows contain the following data: (0x1000, a, 12), (0x1001, r1, 0x1000), and (0x1009, r2, 0x1000). A dashed white box surrounds the entire table, and a dashed white arrow points from the left towards the first row.

Addr	Stack	
0x1000	a	12
0x1001	r1	0x1000
0x1009	r2	0x1000

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

`println!` automatically dereferences for us

1. Recap



Addr	Stack	
0x1000	a	12
0x1001	r1	0x1000
0x1009	r2	0x1000

```
fn main() {  
    let a: u8 = 12;  
    let r1: &u8 = &a;  
    let r2: &u8 = &a;  
    println!("r1 = {}", r1);  
    println!("r2 = {}", r2);  
}
```

r1 = 12
r2 = 12

println! automatically dereferences for us

1. Recap

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```

1. Recap

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a; Mutable borrow  
    *b = 1337;  
    println!("a = {}", a);  
}
```

1. Recap

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```

Mutable borrow only possible when value itself is mutable

Mutable borrow



1. Recap

Addr	Stack
------	-------

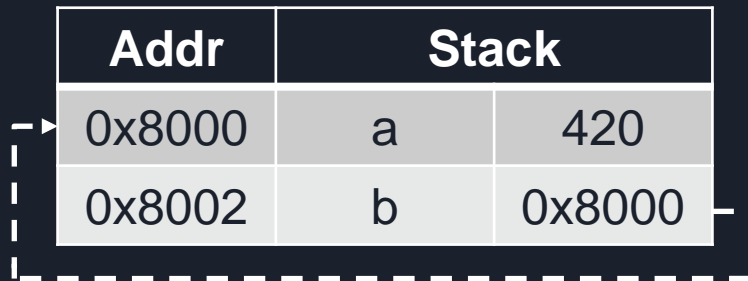
```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```


1. Recap

Addr	Stack	
0x8000	a	420

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```

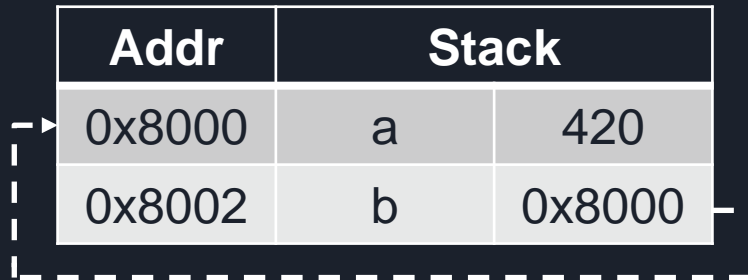
1. Recap



Addr	Stack	
0x8000	a	420
0x8002	b	0x8000

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```

1. Recap

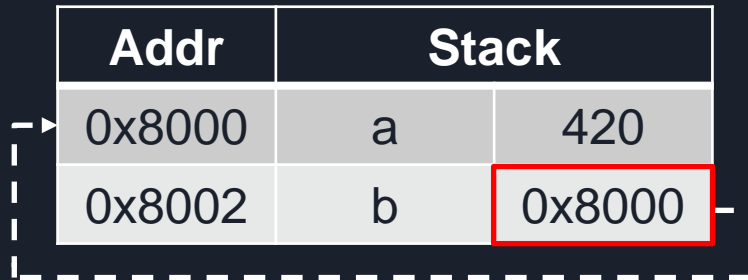


Addr	Stack	
0x8000	a	420
0x8002	b	0x8000

Write 1337 into location
pointed to by b

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```

1. Recap



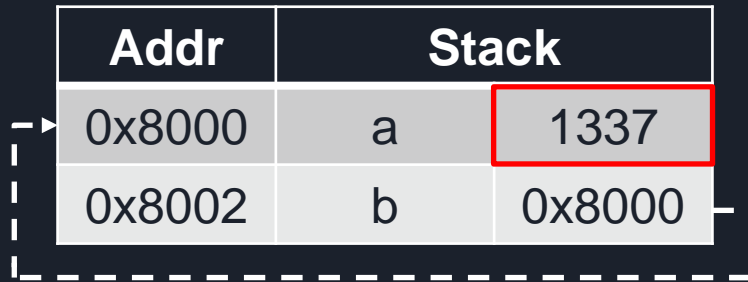
A diagram showing a memory stack layout. It consists of a table with two columns: 'Addr' and 'Stack'. The first row has '0x8000' in the 'Addr' column and 'a' and '420' in the 'Stack' column. The second row has '0x8002' in the 'Addr' column and 'b' and '0x8000' in the 'Stack' column. A dashed white box encloses the entire table. A dashed white arrow points from the left to the first row. The cell containing '0x8000' in the second row is highlighted with a red border.

Addr	Stack	
0x8000	a	420
0x8002	b	0x8000

Write 1337 into location
pointed to by b

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```

1. Recap

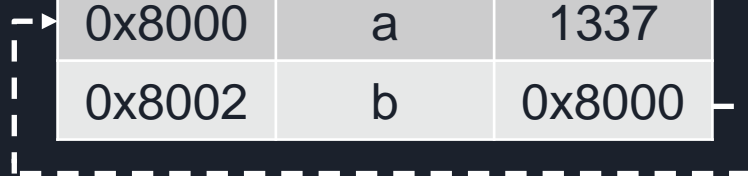


Addr	Stack	
0x8000	a	1337
0x8002	b	0x8000

Write 1337 into location
pointed to by b

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```

1. Recap



Addr	Stack	
0x8000	a	1337
0x8002	b	0x8000

A dashed white box encloses the two rows of the table. A dashed white arrow points from the left edge of the box to the first row.

a = 1337

```
fn main() {  
    let mut a: i16 = 420;  
    let b: &mut i16 = &mut a;  
    *b = 1337;  
    println!("a = {}", a);  
}
```



2. Lifetimes

2. Lifetimes

The problem: Memory safety

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```


2. Lifetimes

The problem: Memory safety

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

Compiler moves the value of **a** into **b**

2. Lifetimes

The problem: Memory safety

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{}", a);  
}
```

a	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...

2. Lifetimes

The problem: Memory safety

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

a	
content	
length	2
capacity	4

b	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...

2. Lifetimes

The problem: Memory safety

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

a	
content	
length	2
capacity	4

b	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...

Ownership violation!
Memory would be freed twice at the end!

2. Lifetimes

The problem: Memory safety

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{}", a);  
}
```

a	
content	???
length	???
capacity	???

b	
content	
length	2
capacity	4

heap	
0xabc0	1
0xabc4	2
...	...

Solution:

- Move a into b
- Invalidate a

2. Lifetimes

The problem: Memory safety

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: Vec<i32> = a;  
    println!("{:?}", a);  
}
```

a	
content	???
length	???
capacity	???

Error: a is not initialized, can't use it – It was moved



2. Lifetimes

The solution: References

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", a);  
}
```

2. Lifetimes

The solution: References

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", a);  
}
```



2. Lifetimes

The solution: References

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", a);  
}
```



No ownership violation:

- **a** still owns the memory on the heap
- **b** does not own **a**, it just points to it

2. Lifetimes

The solution: References

```
fn main() {  
    let a: Vec<i32> = vec![1, 2];  
    let b: &Vec<i32> = &a;  
    println!("{:?}", a);  
}
```



a still valid here, no data was moved



2. Lifetimes

- References in its simplest form are memory addresses
 - can point to any memory, to the stack, to the heap



2. Lifetimes

- References in its simplest form are memory addresses
 - can point to any memory, to the stack, to the heap
- Pointers are easy to mishandle
 - dangling pointers
 - race conditions

2. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

2. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

What does this C code print?

2. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

What does this C code print?

- We can't know.
- The first `printf()` prints 10
- `somethingElse()` may **overwrite** the memory the pointer is pointing to

2. Lifetimes

```
int *f() {  
    int x = 10;  
    return &x;  
}  
  
int main(void) {  
    int *hehe = f();  
    printf("%d\n", *hehe);  
    somethingElse();  
    printf("%d\n", *hehe);  
}
```

3/3

What does this C code print?

- We can't know.
- The first `printf()` prints 10
- `somethingElse()` may **overwrite** the memory the pointer is pointing to

```
void somethingElse() {  
    int a = 420;  
}
```

```
>main.exe  
10  
420
```




2. Lifetimes

- References in its simplest form are memory addresses
- Pointers are easy to mishandle
- More is required to make them memory safe, and fit for Rust's goals



2. Lifetimes

- References in its simplest form are memory addresses
- Pointers are easy to mishandle
- More is required to make them memory safe, and fit for Rust's goals
- The compiler needs to analyze when and how references are valid
 - Part One: Borrow Checker
 - Part Two: Lifetimes



2. Lifetimes

- Lifetimes are a construct of the compiler
 - Technically *everything* is a construct of the compiler



2. Lifetimes

- Lifetimes are a construct of the compiler
 - Technically *everything* is a construct of the compiler
- Lifetimes allow the compiler to analyze and optimize the final code
 - Lifetimes don't get into the final executable
 - Memory Safety guarantees



2. Lifetimes

- A lifetime describes two things



2. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the reference may point into

2. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

2. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

region of code where **&a** must be valid

2. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

region of code where **&a** must be valid

region of code where **&c** must be valid

2. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

region of code where **&a** must be valid

ref	value	var	value
b	→	a	12
	→	c	40

we use **b** here

2. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

region of code where **&a** must be valid

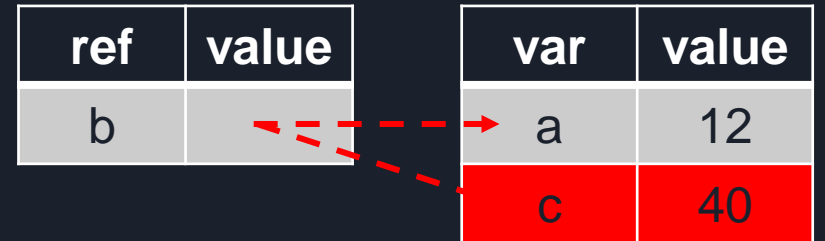
ref	value	var	value
b		a	12
		c	40

Depending on **RNG**, **b** may point to either **a** or **c**
→ When we ***b**, both **memory locations must be alive**

2. Lifetimes

```
pub fn main() {  
    let a: i32 = 12;  
    let mut b: &i32 = &a;  
    if rng().gen_bool(0.5) {  
        let c: i32 = 40;  
        b = &c;  
    }  
    println!("{}", *b);  
}
```

c is alive here and then dropped



region of code where c must be valid

2. Lifetimes

```
pub fn main()
let a: i32
let mut b
if rng().gen() < 0.5 {
    let c = 40;
    b = &c;
}
println!("{}", *b);
}
```

error[E0597]: `c` does not live long enough
--> src\ex2.rs:10:13

let c = 40;
- binding `c` declared here
b = &c;
^^ borrowed value does not live long enough
}
- `c` dropped here while still borrowed
println!("{}", *b);
-- borrow later used here



2. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the original value must live in
- The regions of code don't have to be contiguous



2. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the original value must live in
- The regions of code don't have to be contiguous
 - References must **only be valid between uses**



2. Lifetimes

- A lifetime describes two things
 - A **region of code** where the reference must be valid
 - A **region of memory** where the original value must live in
- The regions of code don't have to be contiguous
 - References must **only be valid between uses**
 - **Non-Lexical Lifetimes**
 - The compiler is very good at figuring out the shortest required lifetimes for references

2. Lifetimes

```
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;  
}
```

2. Lifetimes

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

2. Lifetimes

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

→ No overlap, everything is fine

2. Lifetimes

```
pub fn main() {  
    let mut r: &Vec<i32>;  
    {  
        let x: Vec<i32> = vec![2];  
        r = &x;  
        println!("{:?}", *r);  
    }  
    {  
        let y: Vec<i32> = vec![2];  
        r = &y;  
        println!("{:?}", *r);  
    }  
}
```

2. Lifetimes

```
pub fn main() {  
    let mut r: &Vec<i32>;  
    {  
        let x: Vec<i32> = vec![2];  
        r = &x;  
        println!("{:?}", *r);  
    }  
    {  
        let y: Vec<i32> = vec![2];  
        r = &y;  
        println!("{:?}", *r);  
    }  
}
```

region of code where **&x** must be valid

region of code where **&y** must be valid

2. Lifetimes

```
pub fn main() {  
    let mut r: &Vec<i32>;  
    {  
        let x: Vec<i32> = vec![2];  
        r = &x;  
        println!("{:?}", *r);  
    }  
    {  
        let y: Vec<i32> = vec![2];  
        r = &y;  
        println!("{:?}", *r);  
    }  
}
```

region of code where **&x** must be valid

We don't care about this section
→ **r** refers to a dropped value, but we don't use **r**

region of code where **&y** must be valid



2. Lifetimes

- Lifetimes get complicated when we cross the function border
 - But as we don't know what functions are, we will cover that later :^)



2. Lifetimes

- Lifetimes get complicated when we cross the function border
 - But as we don't know what functions are, we will cover that later :^)
- Extra details about lifetimes will be introduced when needed



3. Functions



3. Functions

- Programming languages come with **many control flow structures**



3. Functions

- Programming languages come with **many control flow structures**
 - **if**
 - **loops, break, continue**



3. Functions

- Programming languages come with **many control flow structures**
- Functions are another way of controlling your code



3. Functions

- Programming languages come with **many control flow structures**
- Functions are another way of controlling your code
- Functions allow us to **isolate pieces of logic**, instead of writing one big blob of code



3. Functions

- Programming languages come with **many control flow structures**
- Functions are another way of controlling your code
- Functions allow us to **isolate pieces of logic**, instead of writing one big blob of code
 - We can write a **function that calculates prime numbers**
 - We can write a **function that checks if a vector contains a number**
 - We can write a **function that checks if a vector contains a prime number**



3. Functions

- Programming languages come with **many control flow structures**
- Functions are another way of controlling your code
- Functions allow us to **isolate pieces of logic**, instead of writing one big blob of code
- Functions in programming are almost* identical to mathematical functions
 - We **take in some inputs** (called **parameters**)
 - We **do something** with those inputs
 - We **return some output** (called **return value**)



3. Functions

- Programming languages come with **many control flow structures**
- Functions are another way of controlling your code
- Functions allow us to **isolate pieces of logic**, instead of writing one big blob of code
- Functions in programming are almost* identical to mathematical functions
 - We **take in some inputs** (called **parameters**)
 - We **do something** with those inputs
 - We **return some output** (called **return value**)
- Functions are declared using the keyword **fn**
 - The most important one – **fn main()** – always needs to be declared



3. Functions

```
fn square(x: i16) -> i16 {  
    return x * x;  
}
```



3. Functions

Parameters

```
fn square(x: i16) -> i16 {  
    return x * x;  
}
```

3. Functions

Parameters

Return Type (optional)

```
fn square(x: i16) -> i16 {  
    return x * x;  
}
```

3. Functions

Parameters

Return Type (optional)

```
fn square(x: i16) -> i16 {  
    return x * x;  
}
```

Function body

3. Functions

```
fn square(x: i16) -> i16 {  
    return x * x;  
}
```

Parameters

Return Type (optional)

return keyword
→ we return the provided value (here $x * x$) from the function

3. Functions

```
fn square(x: i16) -> i16 {  
    return x * x;  
}
```

Function name

Parameters

Return Type (optional)

return keyword

→ we return the provided value (here $x * x$) from the function

3. Functions

```
fn many(mut a: i16, bla: u32, x: f32) {  
    a = 5;  
    println!("{}", bla);  
    println!("{}", x);  
}
```

3. Functions

Parameters are comma-separated name-type pairs

```
fn many(mut a: i16, bla: u32, x: f32) {  
    a = 5;  
    println!("{}", bla);  
    println!("{}", x);  
}
```


3. Functions

```
fn many(mut a: i16, bla: u32, x: f32) {  
    a = 5;  
    println!("{}", bla);  
    println!("{}", x);  
}
```

Parameters are **treated like normal variables**:

- Mutability rules apply
- Ownership rules apply
- Borrow Checker rules apply



3. Functions

Functions can also take zero parameters

```
fn none() {  
}
```



Intermission - Floating Point

- Integers are good, your computer loves them, but they have their limits



Intermission - Floating Point

- Integers are good, your computer loves them, but they have their limits
 - We can **only represent whole numbers**
 - We **can't represent big numbers**



Intermission - Floating Point

- Integers are good, your computer loves them, but they have their limits
- For those situations, we have to use **floating point numbers**



Intermission - Floating Point

- Integers are good, your computer loves them, but they have their limits
- For those situations, we have to use **floating point numbers**
- Rust has **f32** and **f64**



Intermission - Floating Point

- Integers are good, your computer loves them, but they have their limits
- For those situations, we have to use **floating point numbers**
- Rust has **f32** and **f64**
- As with integers, the number **specifies the size in bits**
 - f32 → **32bit float**
 - f64 → **64bit float**



Intermission - Floating Point

- Integers are good, your computer loves them, but they have their limits
- For those situations, we have to use **floating point numbers**
- Rust has **f32** and **f64**
- As with integers, the number **specifies the size in bits**
- Floating Point numbers are numbers in **Scientific Notation (in base 2)**
 - Example in base 10: $5e7 = 5 \cdot 10^7 = 50.000.000$



Intermission - Floating Point

- Integers are good, your computer loves them, but they have their limits
- For those situations, we have to use **floating point numbers**
- Rust has **f32** and **f64**
- As with integers, the number **specifies the size in bits**
- Floating Point numbers are numbers in **Scientific Notation (in base 2)**
 - Example in base 10: $5e7 = 5 \cdot 10^7 = 50.000.000$
 - Numbers are made of two numbers:
 - **Mantissa** → Number before the **e**, here **5**
 - **Exponent** → Number after the **e**, here **7**



Intermission - Floating Point

- Floating Point numbers are numbers in **Scientific Notation (in base 2)**
 - Example in base 10: $5e7 = 5 \cdot 10^7 = 50.000.000$
 - Numbers are made of two numbers:
 - **Mantissa** → Number before the **e**, here **5**
 - **Exponent** → Number after the **e**, here **7**
- Floats get a **few bits for the mantissa**, and a **few for the exponent**
 - f32 → 1 bit sign, 8 bits exponent , 23 bits mantissa
 - f64 → 1 bit sign, 11 bits exponent , 52 bits mantissa



Intermission - Floating Point

- Floating Point numbers are numbers in **Scientific Notation (in base 2)**
- Floats get a **few bits for the mantissa**, and a **few for the exponent**
- This way, **we can represent fractions**
 - `f32` → **~8 decimal digits**
 - `f64` → **~16 decimal digits**



Intermission - Floating Point

- Floating Point numbers are numbers in **Scientific Notation (in base 2)**
- Floats get a **few bits for the mantissa**, and a **few for the exponent**
- This way, **we can represent fractions**
 - f32 → **~8 decimal digits**
 - f64 → **~16 decimal digits**
- This way, **we can represent big numbers**
 - f32 → **MAX = $3.4 \cdot 10^{38}$ ← 38 zeroes!** We'd need **u128 to represent** that!
 - f64 → **MAX = $1.8 \cdot 10^{308}$ ← 308 zeroes!** We'd need **u1024 to represent** that!



Intermission - Floating Point

- There's a problem → Limited precision



Intermission - Floating Point

- There's a problem → Limited precision
 - We can only store 7 decimal digits at all times in the mantissa
 - At big exponents, we skip numbers
 - $5.400000000 \cdot 10^{30} + 1 == 5.400000000 \cdot 10^{30}$



Intermission - Floating Point

- There's a problem → Limited precision
 - We can only store 7 decimal digits at all times in the mantissa
 - At big exponents, we skip numbers
 - $5.400000000 \cdot 10^{30} + 1 == 5.400000000 \cdot 10^{30}$
 - We only have limited space for numbers
 - Just like we can't represent $1/3$ in base 10, we can't represent some numbers in base 2



Intermission - Floating Point

- There's a problem → **Limited precision**
 - We can only **store 7 decimal digits at all times in the mantissa**
 - At big exponents, we skip numbers
 - $5.400000000 \cdot 10^{30} + 1 == 5.400000000 \cdot 10^{30}$
 - We only have **limited space for numbers**
 - Just like we can't represent $1/3$ in **base 10**, we can't represent **some numbers in base 2**

```
>>> 0.1 + 0.2 == 0.3
False
```




Intermission - Floating Point

- There's a problem → Limited precision
 - We can only store 7 decimal digits at all times in the mantissa
 - At big exponents, we skip numbers
 - $5.400000000 \cdot 10^{30} + 1 == 5.400000000 \cdot 10^{30}$
 - We only have limited space for numbers
 - Just like we can't represent $1/3$ in base 10, we can't represent some numbers in base 2

```
>>> 0.1 + 0.2 == 0.3
False
>>> 0.1 + 0.2
0.30000000000000004
```

Intermission - Floating Point

```
fn main() {  
    let a: f32 = 0.0;  
    let b: f64 = 5.0 / 7.0;  
    let c: f32 = 0;  
}
```

Intermission - Floating Point

```
fn main() {  
    let a: f32 = 0.0;  
    let b: f64 = 5.0 / 7.0;  
    let c: f32 = 0;  
    ...  
}
```

Float literals always include a dot

Intermission - Floating Point

```
fn main() {  
    let a: f32 = 0.0;  
    let b: f64 = 5.0 / 7.0;  
    let c: f32 = 0;  
}
```

This is an **integer literal**
→ Can't assign integers to float



3. Functions

- Not all functions return values, some just do stuff with the inputs
 - It might just write to a file, or print something in the console



3. Functions

- Not all functions return values, some just do stuff with the inputs
 - It might just write to a file, or print something in the console
- The **return type is optional**, and declared using the **Arrow syntax**



3. Functions

- The **return type is optional**, and declared using the **Arrow syntax**

```
fn returns() -> i32 {  
    let a: i32 = 18;  
    return a;  
}
```

3. Functions

- The **return type is optional**, and declared using the **Arrow syntax**

Arrow Syntax

```
fn returns() -> i32 {  
    let a: i32 = 18;  
    return a;  
}
```




3. Functions

- The **return type is optional**, and declared using the **Arrow syntax**

This function **returns a value of type i32**

```
fn returns() -> i32 {  
    let a: i32 = 18;  
    return a;  
}
```



3. Functions

- The **return type is optional**, and declared using the **Arrow syntax**

```
fn returns() -> i32 {  
    let a: i32 = 18;  
    return a;  
}
```

All return statements **must return a value of the specified type**

3. Functions

```
fn invalid() -> i32 {  
    let a: i8 = 12;  
    return a;  
}
```

3. Functions

```
fn invalid() -> i32 {  
    let a: i8 = 12;  
    return a;  
}
```

returns a value of type i8
→ i8 is not i32
→ Compiler error



3. Functions

```
fn branch(a: i32) -> i32 {  
    if a < 0 {  
        println!("No return!");  
    } else {  
        return a;  
    }  
}
```

3. Functions

```
fn branch(a: i32) -> i32 {  
    if a < 0 {  
        println!("No return!");  
    } else {  
        return a;  
    }  
}
```

If a function is declared to return something, every possible control flow path must end with a return statement

3. Functions

```
fn branch(a: i32) -> i32 {  
    if a < 0 {  
        println!("No return!");  
    } else {  
        return a;  
    }  
}
```

This branch does not return a value,
not even after the if-block

If a function is declared to return something,
every possible control flow path must end
with a return statement

3. Functions

1/3

```
fn returns(a: i32) -> i32 {  
    if a < 0 { return 0; }  
    while a >= 0 {  
        if a == 5 { return 1; }  
        a -= 1;  
    }  
}
```


3. Functions

1/3

```
fn returns(a: i32) -> i32 {  
    if a < 0 { return 0; }  
    while a >= 0 {  
        if a == 5 { return 1; }  
        a -= 1;  
    }  
}
```

Do all possible paths lead to a return?

3. Functions

1/3

```
fn returns(a: i32) -> i32 {  
    if a < 0 { return 0; }  
    while a >= 0 {  
        if a == 5 { return 1; }  
        a -= 1;  
    }  
}
```

Do all possible paths lead to a return?

Nope. Imagine **a** was 4 in the beginning :^)

We get to the end of the function, but there is no return statement there!

3. Functions

1/3

```
fn returns(a: i32) -> i32 {  
    if a < 0 { return 0; }  
    while a >= 0 {  
        if a == 5 { return 1; }  
        a -= 1;  
    }  
}
```

Code wouldn't compile anyway, **a** is **not mutable** :^)

3. Functions

```
fn ret(mut a: i32) {  
    if a <= 0 { return; }  
    while a >= 0 {  
        if a == 5 { return; }  
        a -= 1;  
    }  
    println!("Hello!");  
}
```

3. Functions

We don't return anything

```
fn ret(mut a: i32)[] {  
    if a <= 0 { return; }  
    while a >= 0 {  
        if a == 5 { return; }  
        a -= 1;  
    }  
    println!("Hello!");  
}
```

3. Functions

```
fn ret(mut a: i32) {  
    if a <= 0 { return; }  
    while a >= 0 {  
        if a == 5 { return; }  
        a -= 1;  
    }  
    println!("Hello!");  
}
```

Now we simply return – without a value

3. Functions

```
fn ret(mut a: i32) {  
    if a <= 0 { return; }  
    while a >= 0 {  
        if a == 5 { return; }  
        a -= 1;  
    }  
    println!("Hello!");  
}
```

There's an **implicit return at the end of every function**,
not every control flow path needs to end in a return



Intermission - Exercises

- Time for exercises!

Intermission - Exercises

1/3

```
fn square(x: i32) -> i64 {  
    return x * x;  
}
```

Intermission - Exercises

Is this a valid function definition?

1/3

```
fn square(x: i32) -> i64 {  
    return x * x;  
}
```

Intermission - Exercises

Is this a valid function definition?

1/3

```
fn square(x: i32) -> i64 {  
    return x * x;  
}
```

Expected return type: i64

Intermission - Exercises

Is this a valid function definition?

1/3

```
fn square(x: i32) -> i64 {  
    return x * x;  
}
```

Expected return type: i64

Actual: $i32 * i32 = i32$

Intermission - Exercises

Is this a valid function definition?

1/3

```
fn square(x: i32) -> i64 {  
    return x * x;  
}
```

Expected return type: i64

Actual: $i32 * i32 = i32$

Rust does not perform type casts for us, this code does not compile

Intermission - Exercises

1/3

```
fn min(a: i32, b: i32) -> i32 {  
    if a < b {  
        return a;  
    } else {  
        return b;  
    }  
}
```

Intermission - Exercises

Is this a valid function definition?

1/3

```
fn min(a: i32, b: i32) -> i32 {  
    if a < b {  
        return a;  
    } else {  
        return b;  
    }  
}
```

Intermission - Exercises

Is this a valid function definition?

1/3

```
fn min(a: i32, b: i32) -> i32 {  
    if a < b {  
        return a;  
    } else {  
        return b;  
    }  
}
```

Yep!

→ All return statements return i32

→ All possible paths lead to a return

Intermission - Exercises

2/3

```
fn loop(start: i32, end: i32, mut counter: i32) {  
    if counter <= start {  
        counter = start;  
        return;  
    }  
    while counter < end {  
        println!("{}", counter);  
        counter += 1;  
    }  
    return counter;  
}
```

Intermission - Exercises

Is this a valid function definition?

2/3

```
fn loop(start: i32, end: i32, mut counter: i32) {  
    if counter <= start {  
        counter = start;  
        return;  
    }  
    while counter < end {  
        println!("{}", counter);  
        counter += 1;  
    }  
    return counter;  
}
```

Intermission - Exercises

Is this a valid function definition?

2/3

```
fn loop(start: i32, end: i32, mut counter: i32)[] {  
    if counter <= start {  
        counter = start;  
        return;  
    }  
    while counter < end {  
        println!("{}", counter);  
        counter += 1;  
    }  
    return counter;  
}
```

Function not declared to return anything

Intermission - Exercises

Is this a valid function definition?

2/3

```
fn loop(start: i32, end: i32, mut counter: i32) {}  
    if counter <= start {  
        counter = start;  
        return;  
    }  
    while counter < end {  
        println!("{}", counter);  
        counter += 1;  
    }  
    return counter;  
}
```

Function not declared to return anything

Yet, we're attempting to return i32.
→ Not a valid function

Intermission - Exercises

3/3

```
fn is_prime(candidate: i32) -> bool {  
    for number: i32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

Intermission - Exercises

Is this function *correct*?

3/3

```
fn is_prime(candidate: i32) -> bool {  
    for number: i32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

Intermission - Exercises

Is this function *correct*?

3/3

```
fn is_prime(candidate: i32) -> bool {  
    for number: i32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

The algorithm itself is correct, it correctly tells us if a given number is prime or not.

Intermission - Exercises

Is this function *correct*?

3/3

```
fn is_prime(candidate: i32) -> bool {  
    for number: i32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

The algorithm itself is correct, it correctly tells us if a given number is prime or not.

BUT: It **returns true for negative numbers, 0 and 1!** :^)

Those are not prime numbers :(

Intermission - Exercises

3/3

```
fn is_prime(candidate: i32) -> bool {  
    for number: i32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

The algorithm itself is correct, it correctly tells us if a given number is prime or not.

BUT: It returns true for negative numbers, 0 and 1! :^)

Those are not prime numbers :(

How do we fix that?

Intermission - Exercises

Step 1: Prevent negative numbers!

3/3

```
fn is_prime(candidate: u32) -> bool {  
    for number: u32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

Intermission - Exercises

3/3

Step 2: 0 and 1 are not prime

```
fn is_prime(candidate: u32) -> bool {  
    if candidate < 2 { return false; }  
    for number: u32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

Intermission - Exercises

3/3

```
fn is_prime(candidate: u32) -> bool {  
    if candidate < 2 { return false; }  
    for number: u32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

Is this now a correct function?

Intermission - Exercises

3/3

```
fn is_prime(candidate: u32) -> bool {  
    if candidate < 2 { return false; }  
    for number: u32 in 2..candidate {  
        if candidate % number == 0 {  
            return false;  
        }  
    }  
    return true;  
}
```

Is this now a correct function?

Yes :)



4. Next time

- Using functions