

The move behavior is inconvenient when programs need to use a data more than once.

```

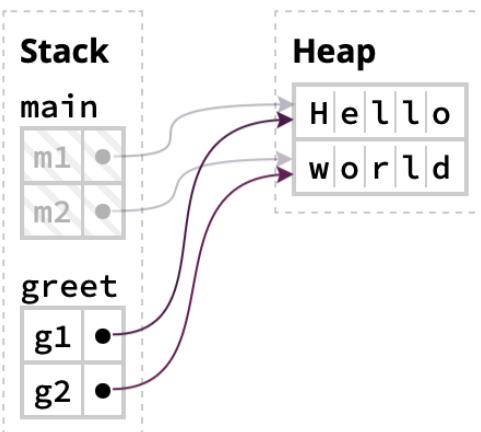
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world");
    greet(m1, m2); L2
    let s = format!("{} {}", m1, m2); L3 // Error: m1 and m2 are moved
}

fn greet(g1: String, g2: String) {
    println!("{} {}", g1, g2); L1
}

```



L1

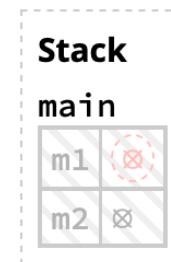


L2



L3

undefined behavior: pointer used after its pointee is freed

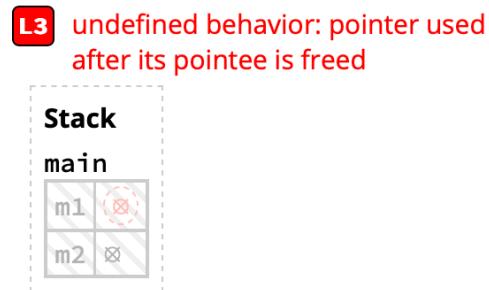
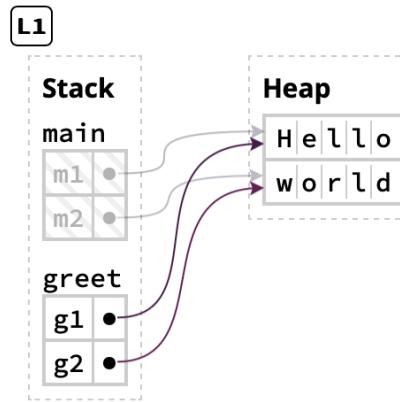


```

fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world");
    greet(m1, m2); L2
    let s = format!("{} {}", m1, m2); L3 // Error: m1 and m2 are moved
}

fn greet(g1: String, g2: String) {
    println!("{} {}", g1, g2); L1
}

```



The strings are dropped at the end of the call of greet, and therefore cannot be used within main. If we try to read them like in the operation format!(..), then that would be undefined behavior. The Rust compiler rejects this program.

# IDEA

Programs often need to use a value more than once.

An alternative approach could return ownership of the parameter.



In our example the alternative greet could return ownership of the strings passed as actual parameters.

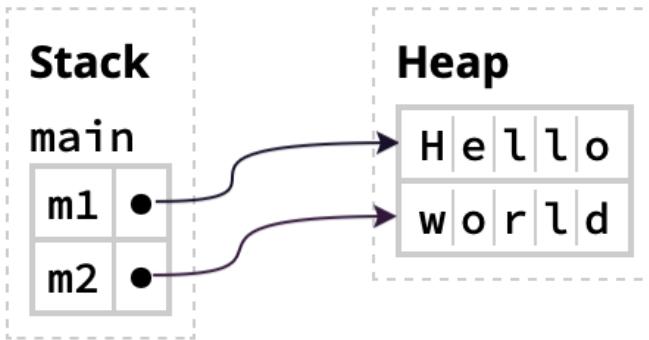
```

fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    let (m1_again, m2_again) = greet(m1, m2);
    let s = format!("{} {}", m1_again, m2_again); L2
}

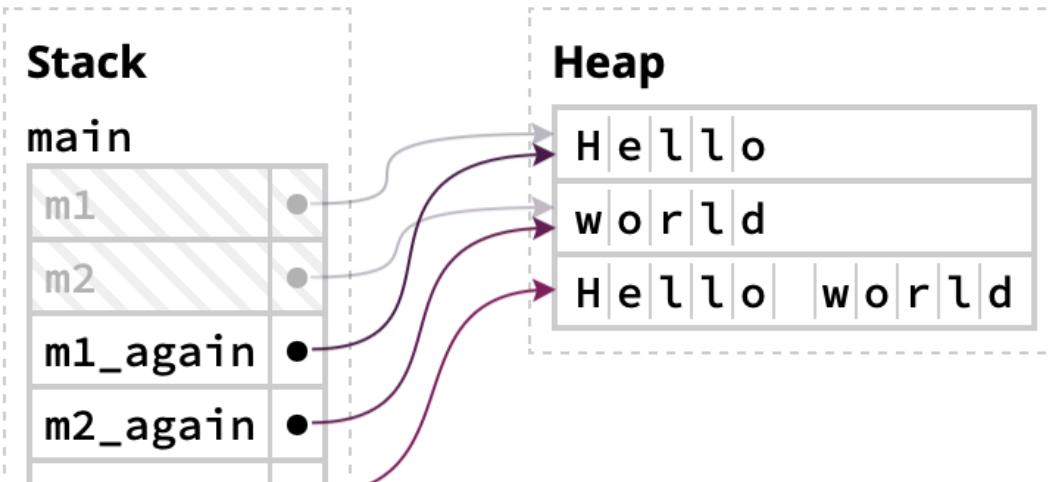
fn greet(g1: String, g2: String) -> (String, String) {
    println!("{} {}", g1, g2);
    (g1, g2)
}

```

**L1**



**L2**



You have a **main** and **associated references** of strings to **m<sub>1</sub>** and **m<sub>2</sub>**.

Because of call by move, ownership of strings is passed to formal parameters  $g_1$  and  $g_2$ . When we return and try to use  $m_1, m_2$  again we get an error. This is bad. Difficult to parameterize functions in the correct way. We could use temporary variables to take back ownership of the strings after greet;  $m_1$ -again and  $m_2$ -again are owners of strings.

But this paradigm for calls is not reasonable, you cannot duplicate inputs like this. Too verbose!

We'd like to have pren. constructs and paradigms to manage param passing.

# Evaluation

It works!!! Good new.

However, this style of program is quite verbose.

Rust solution: the language provides a concise style of reading and writing without moves through references.

Rust approach: references are non-owning  
pointers: another form: you can't take formal param with ampersand. When you  
call function, you have to feed greet with two references for input.

```
fn main() {  
    let m1 = String::from("Hello");  
    let m2 = String::from("world");  
    greet(&m1, &m2); // note the type of parameter passing ownership. References are non-owning  
    let s = format!("{} {}", m1, m2);  
}  
  
fn greet(g1: &String, g2: &String) { // note the ampersands  
    println!("{} {}", g1, g2);  
}
```

You pass references. In rust this operation does not move  
ownership. References are non-owning  
pointers-

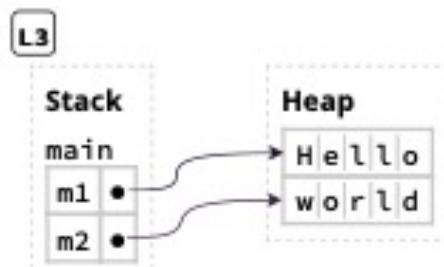
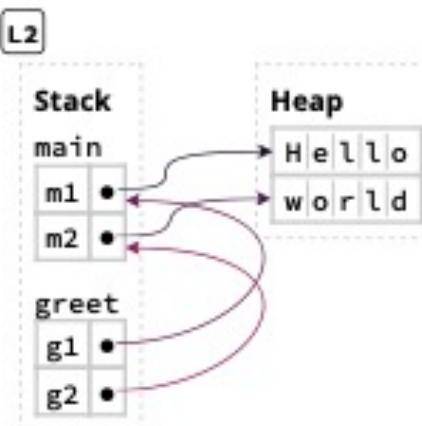
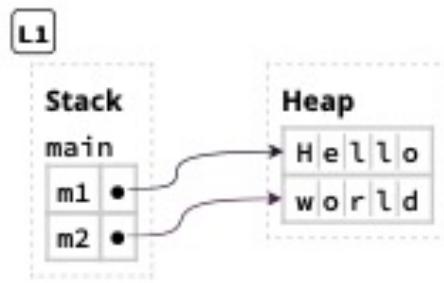
The expression **&m1** exploit the ampersand operator to create a reference to (or  
"borrow") **m1**. The type of the greet parameter **g1** is changed to **&String**, meaning "a  
reference to a String".

```

fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    greet(&m1, &m2); L3 // note the ampersands
    let s = format!("{} {}", m1, m2);
}

fn greet(g1: &String, g2: &String) { // note the ampersands
    L2 println!("{} {}", g1, g2);
}

```



## References Are Non-Owning Pointers

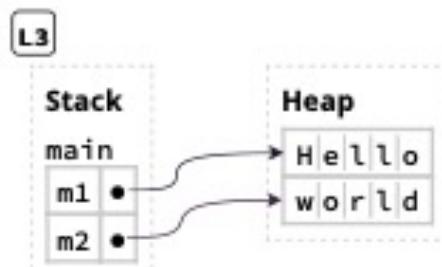
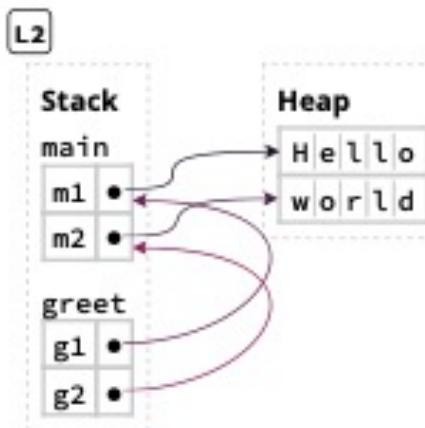
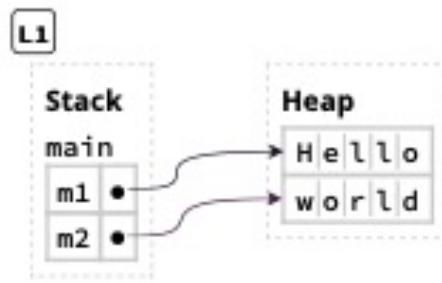
The expression `&m1` creates a reference to (or "borrow") `m1`. The type of the parameter `g1` is changed to `&String`, meaning "a reference to a String".

```

fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    greet(&m1, &m2); L3 // note the ampersands
    let s = format!("{} {}", m1, m2);
}

fn greet(g1: &String, g2: &String) { // note the ampersands
    L2 println!("{} {}", g1, g2);
}

```



## References Are Non-Owning Pointers

The expression `&m1` creates a reference to (or "borrow") `m1`. The type of the parameter `g1` is changed to `&String`, meaning "a reference to a String".

*g1 and g2 are references to position of stack where m1 and m2 are stored.*

`m1` owns the heap data "Hello", `g1` does *not* own either `m1` or "Hello".

*[m1 doesn't own reference stored in m1]*

When the function terminates the program reaches L3, no heap data has been deallocated.

This code was present on my slides to explain borrowing in Rust:

```
fn main() {  
    let m1 = String::from("Hello");  
    let m2 = String::from("world");  
    greet(&m1, &m2); // note the type of parameter passing  
    let s = format!("{} {}", m1, m2);  
}  
fn greet(g1: &String, g2: &String) { // note the ampersands  
    println!("{} {}", g1, g2);  
}
```

So basically, in this case m1 owns both the string in the heap "hello" and the reference placed in the stack? Right? While g1 doesn't own neither the reference in the stack nor the string in the heap?

Exactly, Giovanni — you're spot on, and you're thinking about ownership and references in Rust the right way. Let's walk through it together to make it crystal clear.

# Borrowing

Instead of transferring ownership, we can borrow data.

We borrow data to formal param  
w/out losing ownership. So we talk  
about pass by borrowing

A variable's data can be borrowed by taking a reference to the variable (i.e., aliasing); but ownership doesn't change.

When a reference goes out of scope, the borrow is over.

The original variable retains ownership throughout

↳ Is this consistent with Rust approach? Yes. You shall have one owner so principle marked

# Ownership vs Borrowing

---

- Is borrowing consistent with Rust *Moved Heap Data Principle*?
- The idea is that references are **non-owning pointers**, because they do not own the data they point to.
  - In our example variable g1 did not own "Hello", Rust did not deallocate "Hello" on behalf of g1.



# Parameter passing: borrowing

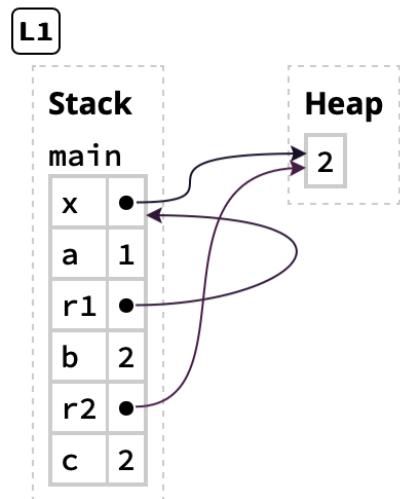
8

**functions may take a reference to an object  
as a parameter instead of taking ownership  
of the value.**

# Remark on (de)references

→ mutable var on the heap. X has reference

```
let mut x: Box<i32> = Box::new(1);  
let a: i32 = *x; // *x reads the heap value, so a = 1  
*x += 1; // *x on the left-side modifies the heap value,  
// so x points to the value 2  
  
let r1: &Box<i32> = &x; // r1 points to x on the stack  
let b: i32 = **r1; ② // two dereferences get us to the heap value  
  
let r2: &i32 = &*x; ③ // r2 points to the heap value directly  
let c: i32 = *r2; L1 // so only one dereference is needed to read it
```



- ① Takes a reference of x; the memory location in the stack where x is present.

- ② • First dereference is x, second brings you to 2.

- ③ • Location of the dereference of x, so 2. So you get address of 2.

Okay tell me if those assertions are correct:

1. `a` is an integer and dereferences the reference of `x` (so 1, and we store 1 in the stack for the variable)
2. `*x += 1`: since `x` is mutable, we are directly modifying the referenced value in the heap. `a` is not changed.
3. `r1` is a reference to a variable which is a reference to an integer in the heap (`&Box<i32>`), and because of that, we assign the reference of `x` (`&x`), which is indeed associated to a reference to an integer in the heap
4. `b` is an integer, and we associate to it the dereference to the dereference of `r1`. The first dereference operator gets us to `x` in the stack. The second dereferenciation gets us in the heap, in which we have 2. So `b` is 2.
5. `r2` is a reference of an integer. We assign `&*x`, so we dereference `x` and get the value in the heap pointed by `x`. With the `&` operator we retrieve its address, so we practically get the reference stored in `x`. `r2` points now to the heap.
6. `c` is an integer, and by dereferencing `r2`, we go in the heap and assign the integer value from the heap to `c`. 2 will be stored in the stack.

When we perform "let `r1: &Box<i32> = &x;`", are we borrowing `x`? Are there limitations to what we can do with `x` after this assignment? What about limitations for `r1`? And the ownership is not changed because of the borrow, correct?

Yes Giovanni, when you do this line:

rust

Copia Modifica

```
let r1: &Box<i32> = &x;
```

you are indeed **borrowing** `x` immutably. Let's unpack all the consequences of that, and you're asking all the right questions, so this is going to be fun.

#### ◆ Are there limitations to what we can do with `x` after this?

Yes but **only while** `r1` is alive and **only with regards to mutability**. Since `r1` is an immutable borrow, you cannot **mutably borrow** `x` or **mutate** `x` itself while `r1` is in scope.

So:

- You can **read** from `x` or do operations that don't require mutating it.
- **You cannot do** `*x += 1` while `r1` is still being used because that would require a mutable borrow.
- **You also can't assign to** `x` (e.g., `x = Box::new(10)`) while `r1` is active.

#### ◆ What are the limitations for `r1`?

`r1` is an **immutable reference**, so:

- You can **read** through it (e.g., `**r1` to get the value from the heap)
- You **cannot mutate** the value via `r1` (e.g., you can't do `**r1 += 1`)

Even though the original `x` is `mut`, `r1` is just a read-only view — it can't change anything.

Let's rework the line you're asking about:

```
rust
```

```
let mut x: Box<i32> = Box::new(1);
let r1: &mut Box<i32> = &mut x; // mutable borrow of x
*x += 1; // ✖ can we still do this??
```

Copia Modifica

The answer is: ✖ No, we can't.

Once `x` is **mutably borrowed** via `r1`, you can no longer use `x` directly — not even to mutate the thing it owns — until the mutable borrow ends.

✓ Yes — the **only way** to mutate `x` is through `r1`

If you do:

```
rust
```

```
let mut x = Box::new(1);
let r1: &mut Box<i32> = &mut x;
```

Copia Modifica

From this point on, until `r1` is no longer used, you must use `r1` to interact with `x`.

That means:

```
rust
```

```
**r1 += 1; // okay: we're going through the mutable reference
*x += 1; // ✖ error: cannot use `x` directly, it's already mutably borrowed
```

Copia Modifica

## ✖ No other borrows allowed while `r1` is alive

While `r1` (a mutable reference) is alive and possibly used:

- ✖ You can't create another mutable reference to `x`
- ✖ You can't create an immutable reference to `x`
- ✖ You can't even use `x` directly

This is Rust's golden rule:

Either one mutable reference OR many immutable references — never both at the same time.

This prevents data races and gives you safety without needing a garbage collector.

## 💡 But what about references *based on* `r1`?

Great question. Let's say you do something like:

```
rust
```

```
let mut x = Box::new(1);
let r1: &mut Box<i32> = &mut x;
let r2: &mut i32 = &mut **r1; // mutable ref to the value inside the Box
*r2 += 1;
```

Copia Modifica

✓ This is allowed. Why? Because:

- You're still going through `r1` — you're not touching `x` directly.
- `r2` is a **sub-borrow** of `r1`, meaning it doesn't break the exclusivity rule — it's just narrowing the focus to the heap value that `r1` already has exclusive access to.

BUT you have to be careful not to split mutable references. For example:

```
rust
```

```
let r2 = &mut **r1;
let r3 = &mut **r1; // ✖ cannot borrow `**r1` mutable more than once at a time
```

Copia Modifica

## 👉 Only if it's an immutable borrow.

If you immutably borrow a variable, you can still read from the original variable.

But if you **mutably** borrow it, then **you can't even read** the original variable while the mutable borrow is alive.

# Mutable references

```
fn main() {  
    let mut s = String::from("hello");  
  
    change(&mut s);  
    println!("The value of s is '{}'", s);  
}  
  
fn change(some_string: &mut String) {  
    some_string.push_str(", world");  
}
```

→ reference to a mut string

↳ we append world

RUN ▶

...

DEBUG ▼

STABLE ▼

...

```
1 fn main() {  
2     let mut s = String::from("hello");  
3  
4     change(&mut s);  
5     println!("The value of s is '{}'", s);  
6 }  
7  
8 fn change(some_string: &mut String) {  
9     some_string.push_str(", world");  
10 }  
11  
12  
13
```

⋮⋮⋮

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)  
Finished dev [unoptimized + debuginfo] target(s) in 3.15s  
Running `target/debug/playground`
```

The value of s is 'hello, world'

Main passes a reference Standard Output  
and we modify string accordingly.

RUN ▶

...

DEBUG ▼

STABLE ▼

...

```
1 fn main() {  
2     let mut s = String::from("hello");  
3  
4     let r1 = &mut s;  
5     let r2 = &mut s;  
6  
7     println!("{} , {}", r1, r2);  
8 }  
9  
10  
11  
12
```

We try to perform 2 mutable borrows from s. Not permitted

→ S is mutable, r<sub>1</sub> and r<sub>2</sub> are 2 different refs to a mutable string, we cannot have more than 1 reference for mutable obj.

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)  
error[E0499]: cannot borrow `s` as mutable more than once at a time  
--> src/main.rs:5:14
```

```
4 |     let r1 = &mut s;  
|         ----- first mutable borrow occurs here  
5 |     let r2 = &mut s;  
|         ^^^^^^ second mutable borrow occurs here  
6 |  
7 |     println!("{} , {}", r1, r2);  
|         -----
```

```
fn main() {  
    let mut s = String::from("hello");  
  
    let r1 = &mut s;  
    let r2 = &mut s;  
  
    println!("{{}, {}}", r1, r2);  
}
```

**Constraint on  
mutable  
references:** **we can  
have only one  
mutable reference  
to a particular  
piece of data in a  
particular scope**

RUN ▶ ... DEBUG ▾ STABLE ▾ ... SHARE TOOLS ▾ CONFIG ▾ ?

```
1 fn main() {
2     let s = String::from("hello");
3
4     // Multiple immutable borrows are allowed
5     let r1 = &s;
6     let r2 = &s;
7
8     println!("{} and {}", r1, r2);
9     // r1 and r2's scope ends here
10 }
```

Thws compiles because it is multiple references to immutable entities

Execution Standard Error Close

Compiling playground v0.0.1 (/playground)  
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.68s  
Running `target/debug/playground`

Standard Output

hello and hello

The screenshot shows the Rust playground interface. At the top, there's a toolbar with buttons for RUN, DEBUG, STABLE, SHARE, TOOLS, CONFIG, and HELP.

In the code editor, the following Rust code is written:

```
1 fn main() {  
2     let mut s = String::from("hello");  
3  
4     let r1 = &mut s;  
5  
6     println!("{}", r1);  
7 }  
8
```

A handwritten note "Mutable but just 1." is written over the line `let r1 = &mut s;`.

A modal window titled "Let's look at your example:" contains the following text:

rust

```
let x = String::from("hello");
println!("{}", x);
```

Yes, you can still use `x` after this!

Why? Because `println!("{}", x)` borrows `x` immutably behind the scenes.

The playground output shows:

```
Compiling playground v0.0.1 (/playground)
Finished `dev` profile [unoptimized + debuginfo]
Running `target/debug/playground`
```

Output: hello

The screenshot shows a Rust IDE interface with the following components:

- Toolbar:** Includes buttons for RUN ▶, ..., DEBUG ▾, STABLE ▾, ..., SHARE, TOOLS ▾, CONFIG ▾, and ?.
- Code Editor:** Displays the following Rust code:

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     let r1 = &s; // Immutable borrow
5     let r2 = &s; // Another immutable borrow
6
7     // This would cause a compile error:
8     let r3 = &mut s;
9     // Error: cannot borrow `s` as mutable because it is also borrowed as immutable
10
11    println!("{} and {}", r1, r2);
12    // r1 and r2's scope ends here
13
14    // Now we can create a mutable borrow
15    let r3 = &mut s;
16    r3.push_str(", world");
17    println!("{}", r3);
```
- Execution Output:** Shows the error message:

```
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
--> src/main.rs:8:14
|         let r3 = &mut s;
|                 ^^^^^^ mutable borrow occurs here
...
8 |         let r3 = &mut s;
|                 ^^^^^^ mutable borrow occurs here
...
```

RUN ▶ ... DEBUG ▼ STABLE ▼ ... SHARE TOOLS ▼ CONFIG ▼ ?

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     let r1 = &s; // Immutable borrow
5     let r2 = &s; // Another immutable borrow
6
7
8
9     println!("{} and {}", r1, r2); r1 and r2 are not passed by reference, so their scope ends
10    // r1 and r2's scope ends here
11
12    // Now we can create a mutable borrow
13    let r3 = &mut s;
14    r3.push_str(", world");
15    println!("{}", r3);
16 }
```

Execution Close

Compiling playground v0.0.1 (/playground)  
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.62s  
Running `target/debug/playground`

---

Standard Output

```
hello and hello
hello, world
```

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

let r1 = &s; //immutable borrow
let r2 = &s; //immutable borrow

println!("{} and {}", r1, r2);

let r3 = &mut s;
r3.push_str(", world");
println!("{}", r3);
}
```

Listen to this reasoning and tell me if it is well explained:

We can borrow `s` mutably after `r1` and `r2`, because the variables are now out of scope, not being used anymore. The compiler can tell that the borrowing of `r1` and `r2` has ended because they are not used anymore after the print and because of this doesn't launch an error.

Giovanni, that's exactly right — and your reasoning is very well explained. Rust's borrow checker is smart,

### Your Reasoning (put simply):

- We can make `r3` a mutable reference after `r1` and `r2` are done being used, because the compiler can see that they're not used anymore.

And yes, Rust knows that `r1` and `r2` are not alive past the `println!`, so it lets you go ahead with the mutable borrow afterward.

In older Rust (pre-NLL), this code would have been rejected, because the compiler thought lifetimes ended with the **scope**, not with the **last actual usage**. But with NLL, it now understands that:

rust

`println!("{} and {}", r1, r2); // ← this is their final moment`

Copia Modifica

So from this point on, Rust says:

"Okay, they're not used anymore — it's safe to allow a mutable borrow."

# Mutable data and aliasing

- References are a powerful programming abstractions ... but they enable aliasing.
- Aliasing is accessing the same data through different variables.
- On its own, aliasing is harmless. But combined with mutation, we have a recipe for disaster.

8

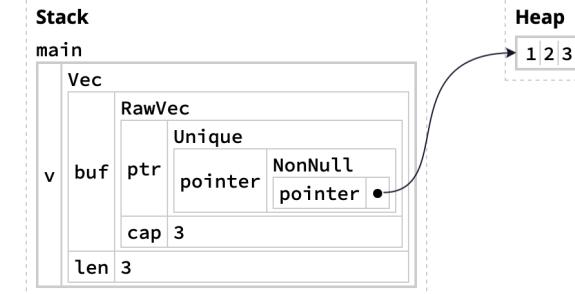
```
let mut v: Vec<i32> = vec![1, 2, 3]; L1  
v.push(4); L2
```



As a result of push you have 1, 2, 3, 4

```
let mut v: Vec<i32> = vec![1, 2, 3]; L1
```

L1



Similar to what we represent :-

Vector has a length (3), where is a ptr, initialised (non-null) that points to the heap. We have a lot of metadata associated.

```
let mut vec: Vec<i32> = vec![1, 2, 3];
```

```
let num: &i32 = &vec[2];
```

```
vec.push(4);
```

```
println!("Third element is {}", *num);
```

Thus in C++ would be unsafe, we have defined  
a ptr to an element on the heap.  
If I push, we are resizing structure  
of vector, which is placed somewhere  
else now.

C, C++ are not memory safe here. For an iterator this could invalidate it.

RUN ▶ ... DEBUG ▼ STABLE ▼ ...

SHARE

TOOLS ▾

CONFIG ▾

```
1 fn main() {  
2     let mut vec: Vec<i32> = vec![1, 2, 3];  
3     let num: &i32 = &vec[2];  
4     vec.push(4);  
5     println!("Third element is {}", *num);  
6 }  
7  
8
```

Exercises

### Standard Error

## ⚠ Why is this an error?

Because you're trying to:

1. Hold an immutable reference to part of `vec` (`&vec[2]`),
2. Then mutate the vector using `vec.push(4)`,
3. While still holding that immutable reference.

Rust says:

"Hey! You're still holding on to a read-only reference to `vec`, and now you're trying to change it. That's dangerous — what if `vec.push()` reallocates memory and invalidates your previous reference?"

## 🧠 Bonus: Is `vec` an object?

Sort of! `vec` is an instance of `Vec<i32>`, which is a **struct** in Rust's standard library — so you can think of it like an object in OOP terms. It has methods like `.push()`, `.len()`, `.pop()`, etc., which are just functions implemented via `impl Vec<T>`.

So yeah, `vec.push(4)` is calling the method `push` on the `Vec<i32>` object, just like you'd do in an object-oriented language — except with Rust's strict borrowing rules behind the scenes keeping it safe.

So, if you have a mutable variable `a`, and then have an immutable borrow to another variable `b`, you can read both `a` and `b`, but not write on `a`, neither `b`. Correct?

Yes Giovanni, **spot on!** You've just nailed a key rule in Rust's borrowing system.

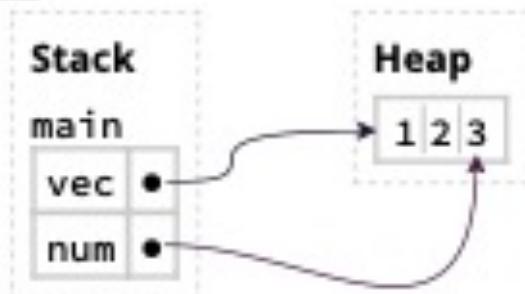
Let me rephrase and confirm your thought:

If you have a **mutable variable** `a`, and you create an **immutable borrow** `b = &a`, then:

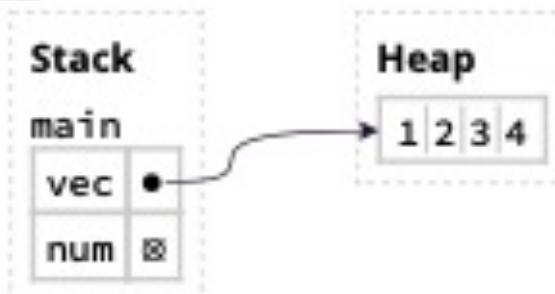
- "✓ You can read from both `a` and `b`,"
- "✗ You cannot write to `a` while `b` is alive,"
- "✗ And of course, you cannot write to `b`, because it's an immutable reference."

```
let mut vec: Vec<i32> = vec![1, 2, 3];
let num: &i32 = &vec[2]; L1
vec.push(4); L2
println!("Third element is {}", *num); L3
```

L1

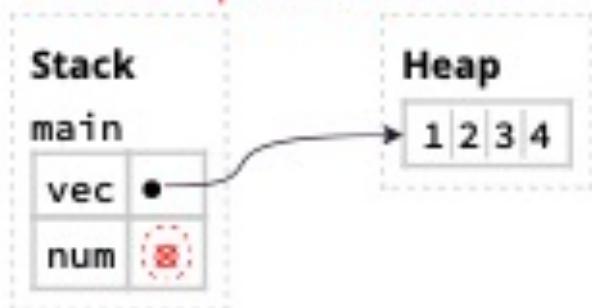


L2



L3

undefined behavior: pointer used  
after its pointee is freed



# **Pointer Safety Principle:**

data should never be  
aliased and mutated at the  
same time.

→ 2 immutable aliases, but not multiple immutables ones.

ALIASING: Multiple references to the same data exist in the program

- The compiler manages this. Compilers called borrow checker is in charge to describe of this is done well

# Pointer safety: borrow checker

- By design, references are meant to temporarily create aliases.
- Rust ensures the safety of references through the **borrow checker**.

# Static information

- Rust variables have three kinds of **type attributes** (sometimes called **permissions**) on their data:
  - **Read (R)**: data can be copied to another location.
  - **Write (W)**: data can be mutated in-place.
  - **Own (O)**: data can be moved or dropped.
- These attributes are static: They describe how the compiler "operates" over programs before the program is executed.

In rust we introduced the resource environment for assignments. Statically you don't only have possibility to be owner. There are immutable environments (only attribute Read), mutable environments (RW attributes). Of course you have ownership attribute and others. Those attributes are available statically to know if an op is possible.

## Three Kinds of Type Attributes

Rust tracks how it can interact with variables and their data through **three permissions or attributes**: **Read (R)**, **Write (W)**, and **Own (O)**. These attributes dictate what can be done to the data, and they are checked at **compile time**. They are called **static** because they exist before the program even runs, letting Rust's borrow checker enforce safety.

### 1. Read (R): Data Can Be Copied

- **What it means:** If a variable's data has the "Read" attribute, it means you can **copy** the data to other locations (like another variable), but you cannot **modify** it. This applies to **immutable references**.
- **Example:**

```
rust                                     ⌂ Copia ⌂ Modifica

let x = 10; // 'x' is read-only.
let y = x; // Copying the value of x to y is allowed.
println!("{}", x); // Accessing x is allowed.
```

- **Important to note:** This doesn't mean you're allowed to change the data; you are **only reading** it.

### 2. Write (W): Data Can Be Mutated

- **What it means:** If a variable has the "Write" attribute, it means you can **mutate** the data in-place. This applies to **mutable references**.

- **Example:**

```
rust                                     ⌂ Copia ⌂ Modifica

let mut x = 10; // 'x' is mutable.
x += 5; // You can change the value of x (write to it).
println!("{}", x); // Accessing x after mutation is allowed.
```

- **Important to note:** A variable can be either **read-only** or **mutable**, but **not both** at the same time in a given scope (no mutable and immutable references to the same data simultaneously).

### 3. Own (O): Data Can Be Moved or Dropped

- **What it means:** If a variable has the "Own" attribute, it means it **owns the data** and can **move or drop** it. This is about **ownership** in Rust.

- **Moving** means transferring the ownership of the data to another variable, which makes the original variable unable to access it anymore.
- **Dropping** means deallocating the memory when the variable goes out of scope.
- **Example:**

```
rust                                     ⌂ Copia ⌂ Modifica

let s1 = String::from("Hello");
let s2 = s1; // Ownership of the data moves from s1 to s2.
println!("{}", s1); // Error: s1 no longer owns the data, so we can't access it.
```

- **Important to note:** When ownership is transferred, the original variable is no longer usable, because it doesn't own the data anymore.

## Static Nature of These Attributes

The key idea here is that these attributes are **static**. This means:

- The **compiler** analyzes your program and determines the permissions for each variable **before** the program actually runs.
- These attributes are determined at **compile time**, which allows Rust to enforce strict rules about **memory safety** and **data races** without needing to rely on runtime checks.

The **borrow checker** uses these attributes to ensure:

- You can't **mutate** data while it's **borrowed immutably**.
- You can't have **multiple mutable references** to the same data.
- Once ownership is **moved**, the original owner can't access the data anymore.

### ● To Borrow Immutably (&T):

The variable must have:

- Read (R) permission

You don't need Write or Own because you're not changing or moving anything — you're just reading.

### ● To Borrow Mutably (&mut T):

The variable must have:

- Write (W)
- Own (O)
- And crucially, no one else is currently borrowing it !

That means:

- You're allowed to mutate it.
- You still own it (no one else has taken it away).
- You must be the only one borrowing it at the time (i.e., no other mutable or immutable borrows are active).

# Mutable vs Immutable revisited

- Immutable variables have read/own type attributes (**RO**) on their data.
- Mutable variables are also annotated the write the write attribute (**W**).
- The key idea: **references can temporarily remove these type information.**



Mutable vector. Info we pickup statically from this declaration:

- Name: vec
- Return: mutation represents fact that vec is initialised because it is mutable, R, W. And Vec is owner of data, so O attribute.

```
fn main() {  
    let mut vec: Vec<i32> = vec![1, 2, 3];           « vec ↗ +R +W +O  
    let num: &i32 = &vec[2];                          « vec → R W O  
                                                num ↗ +R - +O  
                                                *num ↗ +R - -  
    println!("Third element is {}", *num);             « vec ↗ R W O  
    vec.push(4); @                                num ↗ X - O  
                                                *num ↗ X - -  
}
```

What about second assignment? num is a reference to a position of declared vector.

We have two more names: num and \*num, automatically declared because num is a reference. num is the owner now because of assignment, so W and O are eliminated. But you can read, " $\rightarrow$ " means moved.

Num has been initialised, you can read. Num is not mutable, so no W, but it has ownership.

So no one can write in that position, because `vec` is no owner anymore. Plus, even if num is still to position, you lose ownership.

## ① Reserving ownership and attributes

(a) Can you use `vec` after pushing? "↑ invalid. No R, W, O attributes."

Why can we still read `vec`?

To change pointed data, declare a mutable & variable

### ✓ After program point 1

You're totally correct here:

- `vec` is mutable, owns the heap-allocated data (`vec[1,2,3]`), and is fully initialized.
- So yes, statically `vec` has R, W, and O permissions.

Perfect.

### 🟡 After program point 2

This is mostly great, but a few tweaks:

"`vec` cannot be modified anymore because of the immutable borrow."

✓ True, as long as that borrow (`num`) is still active, we can't mutate `vec`.

"Ownership is currently in the hands of `num` for the vector..."

✗ Not quite. This is the key part to correct.

- `num` does not own the data. It's borrowing it.
- The ownership of the vector and its elements stays with `vec`. That's why it all gets dropped when `vec` goes out of scope.
- `num` just has an immutable reference to one element (`&vec[2]`), so `num` only has R, not O.
- Because of that, `vec` temporarily loses the ability to be mutably borrowed — but its ownership is never transferred.

↳ think of loss of O like plus  
↳ You can't either move it.

"vec has lost the W,O attributes"

- Close, but not exactly how Rust thinks about it.
- Rust doesn't *remove* the W or O statically.
- Instead, the **borrow checker enforces** that no one can use W or O operations (like mutation or moving) **while** an active immutable borrow exists.
- So think of it like this: vec still owns the data, but it's **under restrictions** until num's lifetime ends.

"num is an immutable variable..."

- Yes. Immutable, and it can only read. So R only, no W, no O.

\*"num has just the R attribute"

- Absolutely. You can read the value pointed to, but not change it.

### ✓ After program point 3

"Lifetime of num is over. Every attribute is restored to vec."

Yes! This is why the call to `vec.push(4)` is allowed here — because the last use of num was in that `println!`.

So vec regains full freedom to be mutated.

### ✓ After program point 4

Totally fine — program ends, memory is dropped, vec is out of scope.

CASE 2:

rust

✖ Copia   ✖ Modifica

```
let num: &mut i32 = &mut vec[2];
```

- ✓ num: &mut i32

- num is a **mutable reference**, meaning:
  - You can **mutate** the data it points to (`*num = 99`)
  - But you **cannot change** num itself to point somewhere else (because num is immutable as a binding).

Attributes on num itself:

- R : yes, you can read what it points to.
- W : no — not on num itself, because you can't reassign it or make num point to a new location.
- O : no — it doesn't own the data; it's just a borrow.

Attributes on \*num (i.e., the actual value it points to):

- R : yes, read allowed
- W : yes, write allowed! Because num is a &mut, it enables **mutation** of what it points to.
- O : no — still just a borrow.

**Code:**

rust

[Copiar](#) [Modificar](#)

```
let mut num: &mut i32 = &mut vec[2];
```

**Now: what does this *really* give us?**

Thing	R	W	O	Explanation
num	✓	✓	✗	You can <b>reassign</b> num to point to a different &mut i32, because the binding itself is mutable.
*num	✓	✓	✗	You can <b>read and mutate</b> the value it points to — classic &mut behavior.

So YES — you're right! With `mut num: &mut i32`, you have W on both num (the reference) and \*num (the data).

```

fn main() {
let mut vec: Vec<i32> = vec![1, 2, 3];           «-- vec ↗ +R +W +O
let num: &i32 = &vec[2];                         «-- vec → R W O
                                                num ↗ +R - +O
                                                *num ↗ +R - -
println!("Third element is {}", *num);           «-- vec ↤ R +W +O
                                                num ↴ X - O
                                                *num ↴ X - -
vec.push(4);                                     «-- vec ↴ X W O
}

```

The execution of `let mut vec = (...)`, initializes variable `vec` (graphically indicated by the return arrow). It gains `+R+W+O` attributes (the plus sign indicates gain).

```

fn main() {
    let mut vec: Vec<i32> = vec![1, 2, 3];
    let num: &i32 = &vec[2];
    println!("Third element is {}", *num);
    vec.push(4);
}

```

The diagram illustrates the state of variables and memory after each line of code is executed:

- Initial State:** The variable `vec` has attributes `+R +W +O`. It contains three elements: 1, 2, and 3.
- After `let num: &i32 = &vec[2];`:** The variable `vec` now has attributes `R W O`. The variable `num` has attributes `+R - +O`, pointing to the value 2. The variable `*num` has attributes `+R - -`.
- After `println!("Third element is {}", *num);`:** The variable `vec` has attributes `O R +W +O`. The variable `num` has attributes `I R - O`. The variable `*num` has attributes `I R - -`.
- Final State:** The variable `vec` has attributes `I R W O`.

The execution of `num = &vec[2]`, makes the data in `vec` borrowed by `num` (indicated by the righarrow).  
The borrow removes **WO** attributes (`vec` cannot be written or owned, but it can still be read).  
The variable `num` has gained **RO** attributes but `num` is not writable  
The access path `*num` has gained the **R** permission.

```
fn main() {
```

```
let mut vec: Vec<i32> = vec![1, 2, 3];
```

vec ↑ +R +W +O

```
let num: &i32 = &vec[2];
```

vec	→	R	W	O
num	↑	+R	-	+O
*num	↑	+R	-	-

↳?

```
println!("Third element is {}", *num);
```

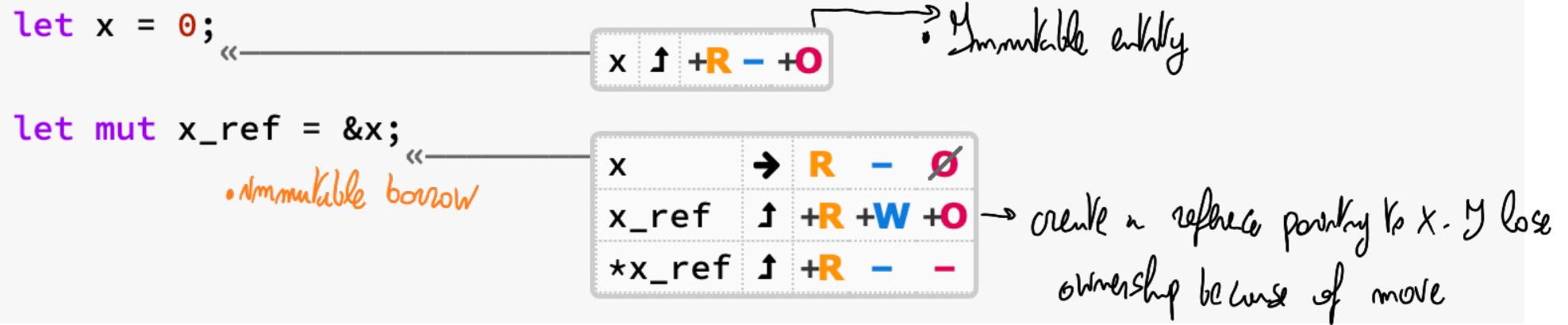
vec	○	R	+W	+O
num	↓	X	-	O
*num	↓	X	-	-

```
vec.push(4);
```

vec ↓ X W O

```
}
```

After the execution of `vec.push(4)`, then `vec` is no longer in use, and it loses all of its attributes.



x\_ref has the **W** attribute, while \*x\_ref does not.

Overall:

we can assign a different reference to the x\_ref variable (e.g. x\_ref = &y), but we cannot mutate the data it points to (e.g. \*x\_ref += 1).

so a mutable borrow is always done like this:

```
let x = &mut y;
```

and not like

```
let mut x = &y;
```

Exactly, Giovanni — you've nailed it. 🎉⚙️

```
fn main() {  
    let x=0;  
    let mut xref = &x;  
    println!(" x value = {}", x);  
    println!("x via xref {}", *xref);  
    *xref+=1;  
}
```

error[E0594]: cannot assign to `\*xref`, which is behind a `&` reference -->  
[src/main.rs:6:1](#)

```
| *xref+=1;  
| ^^^^^^^^^ `xref` is a `&` reference, so the data it refers to cannot be written  
| help: consider changing this to be a mutable reference  
| let mut xref = &mut x;
```



```
let x = 0;
```

x	↑	+R	-	O
---	---	----	---	---

```
let mut x_ref = &x;
```

x	→	R	-	∅
x_ref	↑	+R	+W	O
*x_ref	↑	+R	-	-

x\_ref has the W attribute, while \*x\_ref does not.

3

x\_ref = &y is a type correct assignment

\*x\_ref += 1 is not type correct: we cannot mutate the pointed data.

# Static annotations: paths

- A path is anything that can put on the left-hand side of an assignment
  - 1. Variables (a)
  - 2. Dereferences of paths ( $*a$ )
  - 3. Array accesses of paths ( $a[0]$ )
  - 4. Fields of paths for tuples or field for structs
  - 5. Any combination of the (1-4) above, ( $*((\ast a)[0].1)$ )

# Borrow Checking (on path attributes)

Paths are used to check if we have unsafe/unpermitted operations. Compliers by exploiting notion of path can say if operation can be performed or not.

The *Pointer Safety Principle*: data should not be aliased and mutated.

The borrow checker looks for potentially unsafe operations involving references (by exploiting path attributes)

# Back to our example: path checking

```
fn main() {  
    let mut vec: Vec<i32> = vec![1, 2, 3];           «→ vec ↑ +R +W +O  
    let num: &i32 = &R vec[2];                         «→  
    vec R W.push(4);  
    println!("Third element is {}", *num);  
}
```

The diagram illustrates the ownership state for variables in the code. It consists of two parts: a row of boxes at the top and a 3x3 grid below it.

Row of boxes:

vec	↑	+R	+W	+O
-----	---	----	----	----

3x3 Grid:

vec	→	R	W	∅
num	↑	+R	-	+O
*num	↑	+R	-	-

Annotations with arrows point from the code to the corresponding boxes and grid cells. Arrows point from the 'vec' variable in the first two lines of code to the 'vec' box and the 'vec' cell in the grid. An arrow points from the 'num' variable in the third line of code to the 'num' box and the 'num' cell in the grid. Another arrow points from the 'vec' in the fourth line of code to the 'vec' cell in the grid.

# Back to our example: path checking

```
fn main() {  
    let mut vec: Vec<i32> = vec![1, 2, 3];           «→ vec ↑ +R +W +O  
  
    let num: &i32 = &R vec[2];                         «→  
  
    vec R W.push(4);  
    println!("Third element is {}", *num);  
}
```

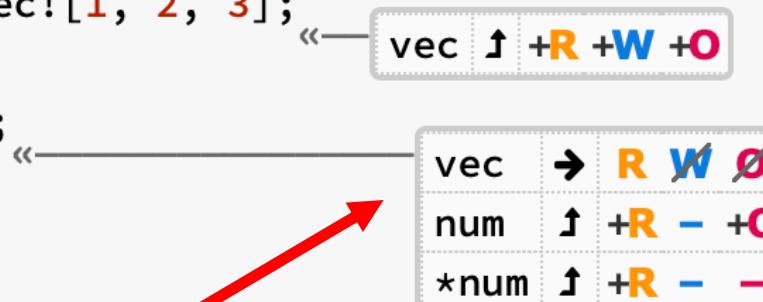


vec	→	R	W	∅
num	↑	+R	-	+O
*num	↑	+R	-	-

the borrow **&vec[2]** requires that **vec** is readable.

# Back to our example: path checking

```
fn main() {  
    let mut vec: Vec<i32> = vec![1, 2, 3];           «-- vec ↑ +R +W +O  
  
    let num: &i32 = &R vec[2];                         «-- num ↑ +R - -O  
  
    vec R W.push(4);  
    println!("Third element is {}", *num);  
}
```



vec	→	R	W	∅
num	↑	+R	-	+O
*num	↑	+R	-	-

`vec.push(4)` requires that `vec` is readable and writable. However, `vec` does not have write attribute (it is borrowed by `num`): that the write attribute is expected but `vec` does not have it.

The screenshot shows a Rust playground interface. At the top, there are tabs for 'RUN' (highlighted in red), '...', 'DEBUG', 'STABLE', and '...'. Below the tabs is a code editor window containing the following Rust code:

```
1 fn main() {  
2     let mut vec: Vec<i32> = vec![1, 2, 3];  
3     let num: &i32 = &vec[2];  
4     vec.push(4);  
5     println!("Third element is {}", *num);  
6 }  
7  
8 |
```

⋮

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)  
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable  
--> src/main.rs:4:1  
|  
3 |     let num: &i32 = &vec[2];  
|         --- immutable borrow occurs here  
4 |     vec.push(4);  
|     ^^^^^^^^^^ mutable borrow occurs here  
5 |     println!("Third element is {}", *num);  
|             ---- immutable borrow later used here
```

The screenshot shows the Rust playground interface. At the top, there's a toolbar with buttons for RUN, DEBUG, STABLE, and other options. Below the toolbar is a code editor containing the following Rust code:

```
1 fn main() {
2     let mut vec: Vec<i32> = vec![1, 2, 3];
3     let num: &i32 = &vec[2];
4     vec.push(4);
5     println!("Third element is {}", *num);
6 }
7
8
```

The line `vec.push(4);` is highlighted with a yellow background. Below the code editor is a terminal window divided into three tabs: Execution, Standard Output, and Standard Error. The Standard Error tab contains the compilation output:

```
Compiling playground v0.0.1 (/playground)
error[E0502]: cannot borrow 'vec' as mutable because it is also borrowed as immutable
--> src/main.rs:4:1
|
3 |     let num: &i32 = &vec[2];
|             --- immutable borrow occurs here
4 |     vec.push(4);
|             ^^^^^^^^^^ mutable borrow occurs here
5 |     println!("Third element is {}", *num);
|             ---- immutable borrow later used here
```

**The error message explains that v cannot be mutated while the reference num is in use. The underlying issue is that num could be invalidated by push. Hence, Rust catches that potential violation of memory safety.**

# Immutable references

- Read-only **immutable references** (also called **shared references**): permit aliasing but disallow mutation



# However ....

It is useful to temporarily provide mutable access to data without moving it.

The Rust mechanism for this is **mutable references** (also called **unique references**).

# Mutable references

```
let mut vec: Vec<i32> = vec![1, 2, 3];           vec |↑ +R +W +O
let num: &mut i32 = &mut vec[2];                  vec → ✘ W O
                                                num ↑ +R - +O
                                                *num ↑ +R +W -
*num += 1;
println!("Third element is {}", *num);            vec ○ +R +W +O
                                                num ↓ ✘ - ✘
                                                *num ↓ ✘ W -
println!("Vector is now {:?}", vec);             vec |↑ ✘ W O
```

```
1 fn main() {  
2     let mut vec: Vec<i32> = vec![1, 2, 3];  
3     let num: &mut i32 = &mut vec[2];  
4     *num += 1;  
5     println!("Third element is {}", *num);  
6     println!("Vector is now {:?}", vec);  
7 }
```

:::

Execution

Standard Error

---

```
Compiling playground v0.0.1 (/playground)  
Finished dev [unoptimized + debuginfo] target(s) in 0.59s  
Running `target/debug/playground`
```

---

Standard Output

```
Third element is 4  
Vector is now [1, 2, 4]
```

```

let mut vec: Vec<i32> = vec![1, 2, 3];
    ↪ vec ↑ +R +W +O

let num: &mut i32 = &mut vec[2];
    ↪ num ↑ +R - +O
    ↪ *num ↑ +R +W -
    ↪ num ↑ +R - +O
    ↪ *num ↑ +R +W - +O

*num += 1;
println!("Third element is {}", *num);
    ↪ num ↑ +R - +O
    ↪ *num ↑ +R +W - +O

println!("Vector is now {:?}", vec);
    ↪ vec ↑ +R +W +O

```

The diagram consists of three rows of tables, each representing the state of variables and memory at a different point in the code execution.

- Row 1:** Shows the initial state. The variable `vec` contains the elements [1, 2, 3]. The pointer `num` points to the second element (2). The dereference `*num` also points to the second element (2). All operations are marked as Read (+R), Write (+W), or Ownership (+O).
- Row 2:** Shows the state after `*num += 1;`. The value at `*num` has been updated to 3. The pointer `num` still points to the second element (2). The ownership of the vector is transferred from `vec` to `*num`.
- Row 3:** Shows the final state after `println!`. The variable `vec` no longer contains the original elements. The pointer `num` still points to the second element (2). The ownership of the vector is transferred back to `vec`.

Why mutable references are *safe*? Mutable references allow mutation but prevent aliasing.

```

let mut vec: Vec<i32> = vec![1, 2, 3];
    <<- vec ↗ +R +W +O

let num: &mut i32 = &mut vec[2];
    <<- vec ↗ X W O
    <<- num ↗ +R - +O
    <<- *num ↗ +R +W -

* *num += 1;
println!("Third element is {}", *num);
    <<- vec ↗ C +R +W +O
    <<- num ↗ X - O
    <<- *num ↗ X W - 

println!("Vector is now {:?}", vec);
    <<- vec ↗ X W O

```

The borrowed path `vec` becomes temporarily unusable, so effectively not an alias.

Why mutable references are *safe*. Mutable references allow mutation but prevent aliasing.

```

let mut vec: Vec<i32> = vec![1, 2, 3];
let num: &mut i32 = &mut vec[2];
*num += 1;
println!("Third element is {}", *num);
println!("Vector is now {:?}", vec);

```

Diagram illustrating the state of variables and memory after each step:

- Initial State:** The variable `vec` is a vector with elements [1, 2, 3]. The pointer `num` points to the second element (index 1). The pointer `*num` also points to the second element.
- After `*num += 1;`:** The value at index 1 is mutated to 4. The pointer `num` still points to index 1, but the value at index 1 is now 4. The pointer `*num` also points to index 1, which is now 4.
- After `println!("Third element is {}", *num);`:** The output is "Third element is 4". The pointer `num` is now pointing to index 2 (the original value 3). The pointer `*num` is pointing to index 1 (the mutated value 4).
- Final State:** The output is "Vector is now [1, 4, 3]". The pointer `num` is pointing to index 2. The pointer `*num` is pointing to index 1.

The borrowed path `vec` becomes temporarily unusable, so effectively not an alias.

`vec[2]` can be mutated through `*num`. `*num` has the `W` attributed, but `num` does not..

Why mutable references are *safe*. Mutable references allow mutation but prevent aliasing.

# Quiz

## ILLUSTRATE THE PERMISSIONS OF X

```
fn main() {  
    let mut x = 1;           X has R,W,O  
    let y = &x;              y has R,O  
    let z = *y;             *y  
    x += z;                z has R,O  
}
```

# Quiz

## ILLUSTRATE THE PERMISSIONS OF X

```
fn main() {  
    let mut x = 1;      

|   |   |   |   |   |
|---|---|---|---|---|
| x | ↑ | R | w | o |
|---|---|---|---|---|

  
    let y = &x;  
    let z = *y;  
    x += z;  
}
```

# Quiz

## ILLUSTRATE THE PERMISSIONS OF X

```
fn main() {  
    let mut x = 1;  
    let y = &x;  
    let z = *y;  
  
    x += z;  
}
```

x	↑	R	w	o
---	---	---	---	---

x	→	R	-	-
---	---	---	---	---

X becomes moved

# Quiz

## ILLUSTRATE THE PERMISSIONS OF X

```
fn main() {  
    let mut x = 1;  
    let y = &x;  
    let z = *y;  
    x += z;  
}
```

X	↑	R	W	O
---	---	---	---	---

X	→	R	-	-
---	---	---	---	---

X	5	R	W	O
---	---	---	---	---

# Quiz

## ILLUSTRATE THE PERMISSIONS OF X

```
fn main() {  
    let mut x = 1;      

|   |   |   |   |   |
|---|---|---|---|---|
| X | ↑ | R | W | O |
|---|---|---|---|---|

  
    let y = &x;  
    let z = *y;  
    x += z;           

|   |   |   |   |   |
|---|---|---|---|---|
| X | → | R | - | - |
|---|---|---|---|---|

  
}  


|   |   |   |   |   |
|---|---|---|---|---|
| X | ↖ | R | W | O |
|---|---|---|---|---|


```

The **W** permission on x is returned to x after the lifetime of y has ended,

## Program Point 1:

rust

Copia Modifica

```
let mut x = 1;
```

- `x` is a mutable variable bound to an integer value.

- So it has:
  - **R** (Read): we can access its value.
  - **W** (Write): we can change the value.
  - **O** (Own): it owns the integer in the stack.

`x : R, W, O`

## Program Point 2:

rust

Copia Modifica

```
let y = &x;
```

We borrow `x` immutably, binding a reference `&x` to `y`.

This means:

- For the lifetime of `y`, we cannot mutate `x`, because it is being *aliased immutably*.
- `x` becomes **frozen** for writing.
- But we can still read from it.

So:

- `x : R` only (temporarily loses W and O for safety)
- `y : R` (it's an immutable reference — it can read, but not write)

`x : R`  
 `y : R`



## Program Point 3:

rust

Copia Modifica

```
let z = *y;
```

We dereference `y` and copy the value `1` into `z`.

- This is just a **read** of the value.
- `y` is no longer used after this line, so its lifetime ends here. That means:
  - The borrow is done.
  - `x` regains its full permissions.

`z : R, O` (owns the copied value `1`)  
 `x : gets back R, W, O`

## Program Point 4:

rust

Copia Modifica

```
x += z;
```

Now we **mutate** `x`, which is fine because:

- The immutable borrow (`y`) is dead.
- We're back to full access.

`x : R, W, O`

### Recap of Permissions at Each Program Point:

Program Point	x	y	z
1	RWO	-	-
2	R	R	-
3	RWO	R	RO
4	RWO	-	RO

You can say Y "acquires" the O  
of X, but it is not the actual  
owner.

Dangling references are still  
possible?

RUN ▶

...

DEBUG ▼

STABLE ▼

...

```
1 fn main() {  
2     let reference_to_nothing = dangle();  
3 }  
4  
5 fn dangle() -> &String {  
6     let s = String::from("hello");  
7     &s  
8 }  
9  
10  
11  
12
```

Create a string and provide back reference. When dangle is finished  
"hello" is removed from heap.

- My ns stopped at compile time.

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)  
error[E0106]: missing lifetime specifier  
--> src/main.rs:5:16  
5 | fn dangle() -> &String {  
|     ^ expected named lifetime parameter  
  
= help: this function's return type contains a borrowed value, but there is no value for it to be borrowed from  
help: consider using the `<static>` lifetime  
|
```

## ⚠ What this code does:

```
rust
```

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String {
    let s = String::from("hello");
    &s
} // <-- `s` goes out of scope here
```

Copia Modifica

You're trying to return a reference to a local variable (`s`) that is created inside `dangle()`. But here's the issue:

- `s` is created on the stack inside `dangle()`
- At the end of the function, `s` is dropped
- You're returning a reference to a value that no longer exists
- That's a dangling reference

## 💡 So what's happening?

Rust statically forbids you from:

- Returning references to local variables
- Creating references that could ever dangle

The error about "missing lifetime specifier" is kind of the surface-level message, but the real reason is:

! You're trying to return a reference to something that's already been dropped

## ✓ Correct version (own the value instead of borrowing it):

```
rust
```

```
fn no_dangle() -> String {
    let s = String::from("hello");
    s // ownership is moved, not borrowed
}
```

Copia Modifica

Now you're moving ownership instead of trying to return a reference. So no dangling reference — and no lifetime needed.

# Use after free

```
fn main() {  
    let x = "Hello".to_string();  
    drop(x);  
    println!("{}", x);  
}
```

RUN ▶

...

DEBUG ▼

STABLE ▼

...

```
1 use std::mem::drop; // equivalent to free()
2
3 fn main() {
4     let x = "Hello".to_string();
5     drop(x);
6     println!("{}", x);
7 }
8
9
```

:::

Execution

Standard Error

Compiling playground v0.0.1 (/playground)

error[E0382]: borrow of moved value: `x`  
--> src/main.rs:6:18

*Not possible: value has been removed after the drop and you can't use it anymore*

```
4     let x = "Hello".to_string();
5         - move occurs because `x` has type `String`, which does not implement the `Copy` trait
6     drop(x);
7         - value moved here
8     println!("{}", x);
9             ^ value borrowed here after move
```

# Use after free

```
fn main() {  
    let x = "Hello".to_string();  
    drop(x);  
    println!("{}", x);  
}
```

8

Rust's ownership: when we give up ownership on a value, then subsequent attempts to use the value are no longer valid.

This also protects against double frees, since two calls to drop would encounter a similar ownership type error.

# Flow Permissions

What happens, when references are either input to a function, or output from a function?

```
fn first(strings: &Vec<String>) -> &String {  
    let s_ref = &strings[0];  
    s_ref  
}
```

Takes a reference to a vector of strings and provides a reference to a string that is the first element of the vector.

Statically we need a new kind of permission, the flow permission F.

Allows to pass as a result reference values

The F permission is expected whenever an expression uses an input reference (like `&strings[0]`), or returns an output reference (like `return s_ref`).

## First, look at this function:

```
rust
```

```
fn first(strings: &Vec<String>) -> &String {  
    let s_ref = &strings[0];  
    s_ref  
}
```

Copia Modifica

We're doing two things here:

1. We're taking in a reference: `&Vec<String>` — an *input reference*.
2. We're returning a reference: `&String` — an *output reference*.

Now this raises a question:

"How do we track the *safety* of borrowing across function boundaries?"

That's where **flow permissions (F)** come in.



### So what is this Flow Permission (F)?

You can think of it like this:

- **Read (R)**: you can read the data
- **Write (W)**: you can mutate the data
- **Own (O)**: you can drop or move the data
- **Flow (F)**: you can pass a reference in or out of a function

F is a kind of *meta-permission*. It doesn't describe what the variable can do on its own, but what you can do with the reference that flows through it.

### Why do we need F?

Because when a reference *enters* or *leaves* a function, the compiler needs to track how long it's valid and make sure no illegal borrow happens.

In your function:

```
rust
```

```
fn first(strings: &Vec<String>) -> &String {  
    let s_ref = &strings[0]; // uses the input reference  
    s_ref // returns it (flows it out)  
}
```

Copia Modifica

We're:

- Using the input reference (`strings`) to create a new reference
- Returning that new reference (`s_ref`) — which depends on the lifetime of `strings`

→ Rust has to make sure that the **returned reference is safe** — it can't outlive `strings`.

So we say: this function has **flow permission F** — because it *uses and returns* a reference.

### Flow Permission (F) doesn't *by itself* enforce lifetimes —

Instead, it *signals* to the compiler that:

"Hey, this function is *moving a reference around*, either taking one in, or returning one out — so pay attention to the lifetime!"

In other words, using `F` doesn't *allow* anything, but rather *alerts the compiler* to activate lifetime tracking logic over that reference.

What's going on here:

1. `strings: &Vec<String>` is a reference input, so Rust must track how long `strings` lives.
2. You're returning a reference (`&String`) that depends on `strings`. Specifically, it points inside of it.
3. So the compiler says:

"This return value's lifetime is tied to `strings`. If the caller lets `strings` go, this reference is invalid."

Now here's the magic of Rust:

If the compiler sees a function returning a reference that was taken from an input reference, and F is required, it will tie their lifetimes together — usually via lifetime annotations (like `'a`).

A screenshot of a web-based Rust playground interface. The top navigation bar includes 'RUN' (highlighted in red), 'DEBUG', 'STABLE', and 'SHARE/TOOLS' buttons. The main area displays the following Rust code:

```
1 fn main() {  
2     let v:Vec<String> = vec![String::from("aa"), String::from("bb"), String::from("cc")];  
3     let y = first( &v);  
4     println!("{}" , *y);  
5 }  
6  
7 fn first(strings: &Vec<String>) -> &String {  
8     let s_ref = &strings[0];  
9     s_ref  
10 }  
11  
12
```

A handwritten note 'A lot of static info in the back' is written over the code area.

The bottom section shows the compilation and execution results:

```
Compiling playground v0.0.1 (/playground)  
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.52s  
Running `target/debug/playground`
```

Execution and Standard Error tabs are visible above the output. The Standard Output tab is selected, showing the output 'aa'.

Go back to our example

```
fn main() {  
    let mut x = 1;  
    let y = &x;  
    let z = *y;  
    x += z;  
    println!("the value of x is {}", x);  
}
```

8

```
1 fn main() {  
2     let mut x = 1;  
3     let y = &x;  
4     let z = *y;  
5     x += z;  
6     println!("the value of x is {}", x);  
7 }  
8 |
```

8

:::

Execution

Standard Error

Compiling playground v0.0.1 (/playground)  
Finished dev [unoptimized + debuginfo] target(s) in 0.92s  
Running `target/debug/playground`

Standard Output

the value of x is 2

```
let mut x = 1;  
»
```

```
let y = &●x;  
»
```



```
let z = ●*y;  
»
```



```
●x += ●z;  
»
```

8

# DISCUSSION

```
fn main() {  
    let mut x = 1;  
    let y = &x;  
    let z = *y;  
    x += z;  
    println!("the value of x is {}", x);  
}
```

**Issue:** a reference changes attributes while it is "in use". What does it mean "in use"?

A new notion:

**lifetime of a reference:** the portion of code of spanning from its birth (where the reference is created) to its death (the last time(s) the reference is used).

The lifetime of y starts with `let y = &x` , and ends with `let z = *y` :

# LIFETIME



Lifetime is a notion associated to data statically. Corresponds to the portion of code where a certain variable lives.

# Lifetime

- A *lifetime* is the static notion the compiler exploit to ensure all borrows are valid.
- A variable's lifetime begins when it is created and ends when it is destroyed. While lifetimes and scopes are often referred to together, they are not the same.

# Static annotation: lifetimes

```
fn main() {  
    let r; // -----+-- 'a  
    //  
    //  
    {  
        let x = 5; // ---+-- 'b  
        r = &x; // |  
    } // --+  
    //  
    //  
    println!("r: {}", r); //-----+  
}
```

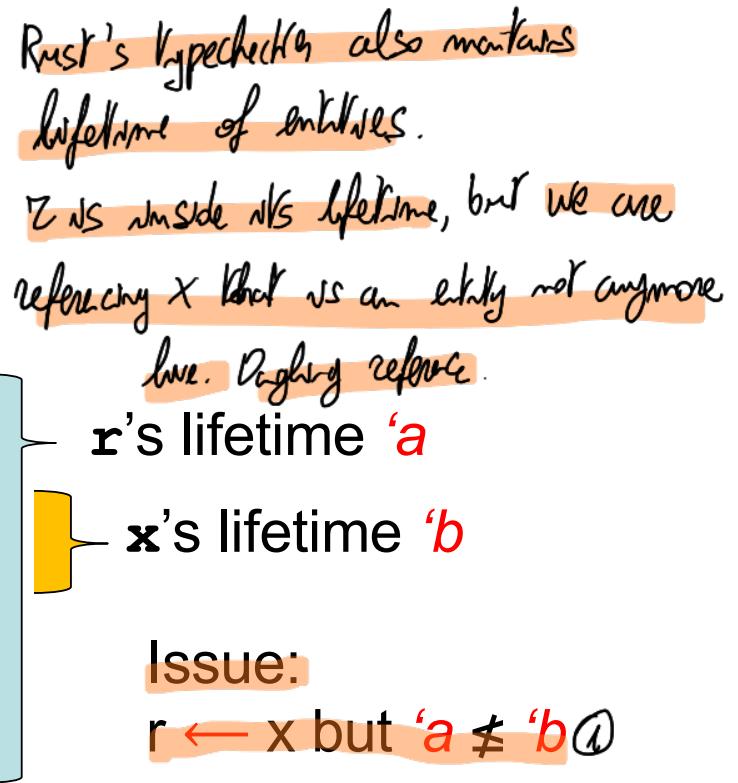
Lifetime: portion of code where a variable lives.

X has a different lifetime here.

Lifetime of r is called a, of X is called b.

# Static annotation: Lifetime

```
{  
    let r; // deferred init  
    {  
        let x = 5;  
        r = &x;  
    }  
    println!("r: {}", r); //fails  
}
```



- The Rust type checker observes that **x** goes out of scope while **r** still exists
  - A **lifetime** is a *type variable* that identifies a scope
  - **r's lifetime 'a exceeds x's lifetime 'b**

For correct assignment, lifetime of right hand side should be greater than lifetime of LHS.

RUN ▶ ... DEBUG ▾ STABLE ▾ ...

SHARE TOOLS ▾ CONFIG ▾ ?

```
1 fn main() {  
2     let r;  
3     {let x = 5;  
4         r = &x;  
5     }  
6     println!("r: {}", r);  
7 }  
8
```

Execution

Close

Standard Error

```
Compiling playground v0.0.1 (/playground)  
error[E0597]: `x` does not live long enough  
--> src/main.rs:4:9  
|  
4 |     r = &x;  
|         ^^ borrowed value does not live long enough  
5 | }  
|     - `x` dropped here while still borrowed  
6 | println!("r: {}", r);  
|             - borrow later used here  
  
For more information about this error, try 'rustc --explain E0597'.
```

Lifetime Apply to All Reset Background

# Static annotation: lifetimes



```
fn main() {  
    let x = 5;           // -----+--- 'b  
    //  
    let r = &x;          // ---+--- 'a |  
    // |  
    println!("r: {}", r); // |  
    // ---+ |  
}  
// -----+
```

```
r: 5
```

# Lifetime

Lifetimes are **named** regions of code that a reference must be valid for.

Those regions may be fairly complex, as they correspond to paths of execution in the program



```
fn main() {  
    let x = 0;  
    let y = &x;  
    let z = &y;  
}.
```

Rust compiles creates lifetime  
of code.

## LIFETIME VIEW –DONE BY RUST COMPILER

```
'a: {  
    let x: i32 = 0;  
'b: {  
    let y: &'b i32 = &'b x;  
'c: {  
    let z: &'c &'b i32 = &'c y;  
}  
}  
}
```

# Why lifetime?

- The answer lies in Rust's ownership model. The borrow checker takes care of allocating and freeing memory and also ensures that no references point to memory that has been freed.
- Lifetimes are checked at compile time, which means that the program can't compile if the borrow checker deems the references invalid.

Lifetimes are checked by borrow checker: RHS should be compatible with LHS of the assignment.

# LIFETIME EXAMPLE

```
fn main() {
    let r;          // r is declared but not initialized
    {
        let x = 5;    // x is valid for this inner scope
        r = &x;      // r borrows a reference to x
    }              // x goes out of scope here

    // This would cause a compile-time error:
    println!("r: {}", r); // Error: `x` does not live long enough
}
```

Rust compiler creates what are called Region Inference. It infers for each var declaration the points in which variable lives.

# LIFETIME: DEEPER VIEW

---

The Rust compiler performs the borrow checking on this MIR representation, which makes analysis easier than working directly with the original syntax.

## Region Inference

The borrow checker uses "region inference" (also called lifetime inference) to determine the lifetimes of references:

1. It assigns each reference a region variable (a lifetime)
2. It collects constraints based on how these references are used
3. It solves these constraints to find concrete lifetimes

## Non-Lexical Lifetimes (NLL)

Rust uses "non-lexical lifetimes" (NLL), which means the lifetime of a borrow isn't just based on lexical scope (the block it appears in) but on the actual usage:

```
fn main() {  
    let mut x = 5;  
  
    let r1 = &x;          // Borrow starts here  
    println!("r1: {}", r1);  
    // r1 is no longer used after this point  Lifetime associated to r1 is over. Denied  
    // by compiler  
    let r2 = &mut x;    // Mutable borrow is allowed here because r1's  
                      // lifetime effectively ended at its last usage  
    *r2 += 1;  
  
    println!("x: {}", x);  
}
```

The screenshot shows the Rust playground interface with the following components:

- Top Bar:** Includes buttons for RUN ▶, ..., DEBUG ▾, STABLE ▾, ..., SHARE, TOOLS ▾, CONFIG ▾, and ?.
- Code Editor:** Displays the following Rust code:

```
1 fn main() {
2     let mut x = 5;
3
4     let r1 = &x;           // Borrow starts here
5     println!("r1: {}", r1);
6     // r1 is no longer used after this point
7
8     let r2 = &mut x;      // Mutable borrow is allowed here because r1's
9     // lifetime effectively ended at its last usage
10    *r2 += 1;
11
12    println!("x: {}", x);
13 }
```
- Execution Output:** A modal window titled "Execution" containing the output of the compilation and execution process:

```
Compiling playground v0.0.1 (/playground)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.61s
Running `target/debug/playground`
```
- Standard Output:** A text area showing the program's output:

```
r1: 5
x: 6
```

```
fn main() {  
    let mut s = String::from("hello");  
  
    let choice = true;  
  
    if choice {  
        let r = &s  
        println!("{}", r);  
    } else {  
        let r = &mut s;  
        r.push_str(", world");  
    }  
  
    println!("{}", s);  
}
```

## QUIZ

At the same time  
you have a mutable and  
immutable reference. You can work with flow sensitive static analysis.

The screenshot shows a Rust playground interface with the following components:

- Toolbar:** Includes buttons for RUN ▶, ..., DEBUG ▾, STABLE ▾, ..., SHARE, TOOLS ▾, CONFIG ▾, and ?.
- Code Editor:** Displays the following Rust code:

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     let choice = true;
5
6     if choice {
7         let r = &s;
8         println!("{}", r);
9     } else {
10        let r = &mut s;
11        r.push_str(", world");
12    }
13
14    println!("{}", s);
15 }
```
- Execution Output:** A modal window titled "Execution" containing the output of the compilation and execution process:

```
Compiling playground v0.0.1 (/playground)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.66s
Running `target/debug/playground`
```
- Standard Output:** A text area showing the output of the program's execution:

```
hello
hello
```

The screenshot shows a Rust playground interface. At the top, there's a toolbar with buttons for RUN, DEBUG, STABLE, SHARE, TOOLS, CONFIG, and HELP.

The code editor contains the following Rust code:

```
1 fn main() {
2     let mut s = String::from("hello");
3
4     let choice = true;
5
6     if choice {
7         let r = &s;
8         println!("{}", r);
9     } else {
10        let r = &mut s;
11        r.push_str(", world");
12    }
13
14    println!("{}", s);
15 }
16
```

In the center of the screen, the word "WHY???" is displayed in large red capital letters.

Below the code editor is a terminal window titled "Execution". It shows the compilation and execution process:

```
Compiling playground v0.0.1 (/playground)
Finished `dev` profile [unoptimized + debuginfo] target(s) in 0.66s
Running `target/debug/playground`
```

At the bottom of the terminal window, it says "Standard Output". The output itself shows:

```
hello
hello
```

# QUIZ

```
fn main() {  
    let mut s = String::from("hello");  
  
    let choice = true;  
  
    if choice {  
        let r = &s  
        println!("{}", r);  
    } else {  
        let r = &mut s;  
        r.push_str(", world");  
    }  
  
    println!("{}", s);  
}
```

The borrow checker performs a flow-sensitive analysis

The compiler understands that the immutable  
and mutable borrows occur in different  
branches and don't overlap in their lifetimes.

# RUST BORROW CHECKER: ISSUES

## Limitations and Complexity

Despite being sophisticated, the borrow checker has some limitations:

**Conservative Approach:** It sometimes rejects valid programs because it can't prove they're safe.

**Self-Referential Structures:** Creating certain data structures (like doubly-linked lists) is challenging ex: double linked lists: circular entities are hard to manage

**Learning Curve:** The rules can be complex to learn, especially for programmers coming from languages with garbage collection. *Think of ownership*

# Lifetime and functions

I associate to result the x or y parameter.

Are you sure the two params have  
the same lifetime?

We should modify function.

```
fn main() {  
    let string1 = String::from("abcd");  
    let string2 = "xyz";  
  
    let result = longest(string1.as_str(), string2);  
    println!("The longest string is {}", result);  
}  
  
fn longest(x: &str, y: &str) -> &str {  
    if x.len() > y.len() {  
        x  
    } else {  
        y  
    }  
}
```

## The Problem:

The function `longest` is supposed to take two string references, compare their lengths, and return a reference to the longest string. Here's the catch: **Rust doesn't know how long those references will be valid.** This is because one reference (`x`) is from a `String` which is **local** to the `main` function, and the other reference (`y`) is from a string literal, which has a **static lifetime**.

### The `main` function:

```
rust                               ⌂ Copia ⌂ Modifica

fn main() {
    let string1 = String::from("abcd"); // string1 is a heap-allocated String.
    let string2 = "xyz";              // string2 is a string literal, which is stored in the binary image.
    let result = longest(string1.as_str(), string2); // Passing string references to longest.
    println!("The longest string is {}", result); // This prints the longest string.
}
```

- `string1` is a `String` created on the heap with the value `"abcd"`.
- `string2` is a string literal `"xyz"`, which lives for the entire duration of the program (its lifetime is **static**).
- `string1.as_str()` gives a reference to the `String` as a `&str` (an immutable reference to the string data).
- `string2` is already a `&str`, so no need to convert it.

In the `longest` function, you're passing two references: one to `string1` (via `string1.as_str()`) and one to `string2`.

### The `longest` function:

```
rust                               ⌂ Copia ⌂ Modifica

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

- The function takes two references (`&str`), compares their lengths, and returns one of those references.
- The problem is that **Rust doesn't know** how long these references (`x` and `y`) will live.

## What's going wrong?

The issue is about **lifetimes**. Let's break down the two main pieces of the puzzle:

1. `string1.as_str()` — This gives an immutable reference (`&str`) to the contents of `string1`. The lifetime of this reference is tied to the lifetime of `string1`. When `string1` goes out of scope (at the end of `main()`), the reference `&string1` will no longer be valid.
2. `string2` — This is a **string literal**, and string literals in Rust have a **'static lifetime**, meaning they live for the entire duration of the program.

The function `longest` returns either `x` or `y`, but Rust can't figure out how long the returned reference should be valid for. If it returns `x`, it's valid as long as `x` is valid. But if `x` points to `string1`, and `string1` is a local variable, `x` will no longer be valid after `string1` goes out of scope at the end of `main()`. If it returns `y`, `y` is a string literal, which is **always valid**.

Rust needs more information to figure out the lifetime of the returned reference. In this case, it needs to know that the lifetime of the reference returned from `longest` depends on the shortest of the lifetimes of `x` and `y`.

## The Compiler's Error:

Without explicit lifetime annotations, Rust doesn't know how to track the lifetimes of the references. It doesn't know how long `x` and `y` are valid for in relation to each other, so it cannot guarantee that the reference returned from `longest` will remain valid.

## Solution: Adding Lifetime Annotations

To fix this, we need to explicitly tell the compiler the relationships between the lifetimes of `x`, `y`, and the returned reference.

We do this by adding **lifetime parameters** to the function signature:

```
rust Copiar Modificar
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

### Breakdown of `longest<'a>`:

- `'a` is a lifetime parameter that we're introducing. It tells the compiler that the reference returned by `longest` will be valid for as long as either `x` or `y` (whichever one has the shorter lifetime).
- `'a` is applied to both `x` and `y` (they must live at least as long as `'a`), and it's applied to the return type (the reference returned by `longest` will live at least as long as `'a`).

### What does this mean?

- Lifetime of `x: x` is a reference to `string1`. Its lifetime is tied to `string1` and will be valid for the duration of `string1`'s scope in `main`.
- Lifetime of `y: y` is a string literal, which has a **static lifetime** (it lives for the entire program).
- Lifetime of the return value: The reference returned by `longest` will be valid as long as the **shorter** of the two input references (`x` and `y`). Since `x` is tied to `string1`, it will live as long as `string1` does in `main()`. `y`, being a string literal, will live for the entire program, so the returned reference will be valid as long as `string1` is valid.

## 2. What happens if `x` and `y` have different lifetimes (like `string1` and `"xyz"`)?

Here's the thing: Rust does not allow references with different lifetimes to be used in ways that might cause one to outlive the other.

In the code you provided, you are passing two references with different lifetimes to the `longest` function:

- `string1.as_str()` is a reference to a `String` (on the heap), and it has a **temporary lifetime** tied to `string1` in `main()`. The reference will be invalid once `string1` goes out of scope, i.e., at the end of `main()`.
- `"xyz"` is a string literal, and it has a **static lifetime**, meaning it lives for the entire duration of the program.

### Will this cause an error?

The answer depends on what you're trying to do. When you define the function like this:

```
rust Copiar Modificar
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
```

You're saying that both `x` and `y` must have the same lifetime, specifically `'a`. This means that the references passed to `longest` must either live for the same duration (same lifetime).

So, if you try to pass one reference (`x` or `y`) that has a lifetime tied to a local variable (like `string1`) and another reference that has a static lifetime (like `"xyz"`), Rust will **not compile** the code because the function signature explicitly requires that both references have the same lifetime `'a`.

## You're Right: Choosing the Shortest Lifetime Is Sound

If you have:

rust

 Copia  Modifica

```
fn longest(x: &str, y: &str) -> &str
```

Then yes — **from a theoretical safety point of view**, returning a reference with the shorter lifetime between `x` and `y` would be perfectly valid. You're not dangling anything, you're not outliving your input lifetimes. It's a **safe lower bound**.

So Rust **could**, in theory, say:

"Cool, the shortest of `x` and `y` is `'s`, so I'll assign `'s` to the output."

This would never allow the output to outlive the input, so you're not introducing memory safety bugs.

## So Why Doesn't Rust Do It?

Because of how it wants lifetimes to behave: *not just safely*, but also transparently and predictably in the type system. Rust isn't *just* about safety—it's about **soundness + clarity + compiler reasoning**.

# Lifetime and functions

Rust will refuse to compile the example code. To get it to compile we need to tell the compiler more about the lifetimes of the references. This can be done by making the lifetimes explicit in the function declarations.

```
Compiling playground v0.0.1 (/playground)
error[E0106]: missing lifetime specifier
--> src/main.rs:10:33
|
10 | fn longest(x: &str, y: &str) -> &str {
|     ----      ^ expected named lifetime parameter
|
= help: this function's return type contains a borrowed value, but the signature does not say whether it is borrowed from `x` or `y`
help: consider introducing a named lifetime parameter
10 | fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
|     +++++    ++      ++      ++
```

```
fn main() {
    let string1 = String::from("abcd");
    let string2 = "xyz";

    let result = longest(string1.as_str(), string2);
    println!("The longest string is {}", result);
}

fn longest(x: &str, y: &str) -> &str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

# Lifetime parameters



- Each reference of type  $t$  has a lifetime parameter
  - $\&t$  (and  $\&\text{mut } t$ ) – lifetime is implicit
  - $\&'\text{a } t$  (and  $\&'\text{a mut } t$ ) – lifetime ' $\text{a}$  is explicit
- Where do the lifetime names come from?
  - When left implicit, they are generated by the compiler
  - Global variables have lifetime ' $\text{static}$

# Lifetime and functions

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() { x }  
    else { y }  
}
```

x and y must have the same lifetime, and the returned reference shares it

The function signature also tells Rust that the string returned from the function will live at least as long as lifetime 'a.

Problem of compositionality

## 😱 The Setup

You wrote:

```
rust
```

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
```

Copia Modifica

This says:

"Both `x` and `y` must live at least as long as `'a`, and I'll return a reference that is valid for `'a`."

So you're asking:

But wait! `string1` is from a `String`, and `"xyz"` is a string literal—doesn't that mean they have *different lifetimes*?

Yup. They do. But here comes the trick.

## 🗣 What's Actually Happening?

Let's break it down line by line:

```
rust
```

```
let string1 = String::from("abcd");
let string2 = "xyz"; // ← string literal, has 'static lifetime
let result = longest(string1.as_str(), string2);
```

Copia Modifica

Now let's annotate the lifetimes from the compiler's POV.

- `string1.as_str()` → returns a `&str` with a lifetime tied to `string1`, let's call that `'x`
- `"xyz"` → a string literal, so it's `&'static str` → lifetime is `'static`

When you call:

```
rust
```

```
longest(string1.as_str(), string2)
```

Copia Modifica

The compiler is asked:

Can I find a single lifetime `'a` such that:

- `"string1.as_str()" → &'a str"`
- `""xyz"" → &'a str"`

Now here's the key insight:

👉 The compiler can pick `'a` to be `'x`, the lifetime of `string1.as_str()` — the shorter one.

That's perfectly valid, because `'static` (from `"xyz"`) lives *longer* than `'x`, so it's fine to shorten it.

This is called **lifetime subtyping**.

## 🗣 Bonus Thought: Lifetime Parameters Aren't Saying "Equal"

This is subtle but important:

```
rust
```

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str
```

Copia Modifica

This doesn't mean `x` and `y` must have the same exact lifetime.

It means:

Whatever `'a` is, both `x` and `y` must be valid for that lifetime, and I'll return a `&str` valid for `'a`.

So if `'a` is `'x`, and `'y` is longer (`'static`), that's still totally fine.

## 💡 Scenario:

```
rust                               ⌂ Copia ⌂ Modifica

fn main() {
    let result;
    {
        let string1 = String::from("abcd");           // 's1 begins
        let string2 = "xyz";                          // 'static lifetime

        result = longest(string1.as_str(), string2); // compiler infers 'a = lifetime of string1
    }                                                 // 's1 ends here

    println!("The longest string is {}", result);   // Uh-oh?
}
```

## 🧐 What Would Happen if Rust Allowed It?

Let's say `longest()` returns `string2` (which lives forever), but the compiler assigned the lifetime `'a = lifetime of string1`, the shorter one.

Now you're holding a reference (`result`) to a perfectly safe string literal...

But the compiler *thinks* its lifetime ended when `string1` went out of scope.

So when you try to use `result`, Rust panics?

No — Rust won't even let you compile that in the first place.

```
Compiling playground v0.0.1 (/playground)
error[E0597]: `string1` does not live long enough
--> src/main.rs:7:26
  |
4 |     let string1 = String::from("abcd");           // 's1 begins
  |     ----- binding `string1` declared here
...
7 |     result = longest(string1.as_str(), string2); // compiler infers 'a = lifetime of string1
  |             ^^^^^^ borrowed value does not live long enough
8 | }                                                 // 's1 ends here
  | - `string1` dropped here while still borrowed
9 |
10| println!("The longest string is {}", result);    // Uh-oh?
   |         ----- borrow later used here
```

```

1 fn main() {
2     let string1 = String::from("abcd");
3     let string2 = "xyz";
4
5     let result = longest(string1.as_str(), string2);
6     println!("The longest string is {}", result);
7 }
8
9
10 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
11     if x.len() > y.len() {
12         x
13     } else {
14         y
15     }
16 }
```

Compiling playground v0.0.1 (/playground)  
 Finished dev [unoptimized + debuginfo] target(s) in 0.58s  
 Running `target/debug/playground`

The longest string is abcd

Vímecké sm. listky  
 e sú s mychá demetká

But, what if you want  
 to define your own  
 implementation of a list?  
 You have complications  
 associated to lifetime of  
 lists, so all elements in a list  
 must have a lifetime greater than the  
 one in list.

```
fn main() {  
    let string1 = String::from("abcd");  
    let string2 = "xyz";  
  
    let result = longest(string1.as_str(), string2);  
    println!("The longest string is {}", result);  
}  
  
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {  
    if x.len() > y.len() { x } else { y }  
}
```

}

**The generic lifetime '*a* will get the concrete lifetime that is equal to the smaller of the lifetimes of the parameter *x* and *y*.**

**Because we annotated the returned reference with the same lifetime parameter '*a*, the returned reference will be valid for the length of the smaller of the lifetimes of *x* and *y*.**

RUN ►

DEBUG ▾

STABLE ▾

...

```
1 fn main() {
2     let string1 = String::from("long string is long");
3
4     {
5         let string2 = String::from("xyz");
6         let result = longest(string1.as_str(), string2.as_str());
7         println!("The longest string is {}", result);
8     }
9 }
10
11 fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
12     if x.len() > y.len() {
13         x
14     } else {
```

8

:::

Execution

Standard Error

```
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 1.02s
Running `target/debug/playground`
```

Standard Output

The longest string is long string is long

```
fn main() {
    fn always_return_x<'a, 'b: 'a>(x: &'a i32, y: &'b i32) -> &'a i32 {
        y
    }
    let x = 1;
    let z;
    {
        let y = 10;
        z = always_return_x(&x, &y);
    }
    println!("{}", z);
}
```

Rust compiler will try to return a 'b lifetime y.

8

```
fn main() {
    fn always_return_x<'a, 'b: 'a>(x: &'a i32, y: &'b i32) -> &'a i32 {
        y
    }
    let x = 1;
    let z;
    {
        let y = 10;
        z = always_return_x(&x, &y);
    }
    println!("{}", z);
}
```

8

```
Compiling playground v0.0.1 (/playground)
warning: unused variable: `x`
--> src/main.rs:3:36
3 |     fn always_return_x<'a, 'b: 'a>(x: &'a i32, y: &'b i32) -> &'a i32 {
|             ^ help: if this is intentional, prefix it with an underscore: `_x`
|
|= note: `#[warn(unused_variables)]` on by default

error[E0597]: `y` does not live long enough
.
```

```
fn main() {
    fn always_return_x<'a, 'b: 'a>(x: &'a i32, y: &'b i32) -> &'a i32 {
        y
    }
    let x = 1;
    let z;
    {
        let y = 10;
        z = always_return_x(&x, &y);
    }
    println!("{}", z);
}
```

✗

y's lifetime ('b) is required to be longer or equal to x and z's lifetime ('a). The maximum possible 'b region is within the block, and obviously, z's lifetime ('a) is strictly longer than y's ('b), which contradicts with 'b: 'a. Therefore, the code is rejected by the compiler

```
Compiling playground v0.0.1 (/playground)
warning: unused variable: `x`
--> src/main.rs:3:36
3 |     fn always_return_x<'a, 'b: 'a>(x: &'a i32, y: &'b i32) -> &'a i32 {
|          ^ help: if this is intentional, prefix it with an underscore: `_x`
|          = note: `#[warn(unused_variables)]` on by default
error[E0597]: `y` does not live long enough

```

# Reading

<https://practice.rs/lifetime/advance.html>

# Unsafe Rust

- All the code we've discussed so far has had Rust's memory safety guarantees enforced at compile time.
- Rust has a second language hidden inside it that doesn't enforce these memory safety guarantees: it's called *unsafe Rust*
- Rust works just like regular Rust, but gives programmers extra powers.

# Extra (Super)Powers

Dereference	Dereference a raw pointer
Call	Call an unsafe function or method
Access or modify	Access or modify a mutable static variable
Implement	Implement an unsafe trait

If you use unsafe component but remaining parts are safe, in most cases unsafety doesn't propagate to safe part of code. Same for APIs.

## ❖ So what is a trait?

A trait in Rust is kind of like an interface in other languages. It defines shared behavior that types can implement.

Imagine it as saying:

"Hey, any type that wants to be considered *Printable* must implement a `print()` method."

Here's a tiny example:

rust

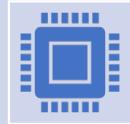
Copia

Modifica

```
trait Printable {  
    fn print(&self);  
}
```

Now if you want your own struct to be "Printable", you implement the trait:

# Unsafe Rust



The unsafe key word only gives you access to these features that are then not checked by the compiler for memory safety.



Programmers still get some degree of safety inside of an unsafe block.



The intent is that programmers will ensure the code inside an unsafe block will access memory in a valid way.

# RUST

## SUMMARY

Rust's type system is based on ownership:

1. Full ownership: T
2. Mutable borrowed reference: &'a mut T
3. Shared borrowed reference: &'a T

Lifetimes 'a decide how long borrows last.

Everything handled by compiler.

# Ownership, Move and Borrowing

The idea of *borrowing* takes a lot of inspiration from work on *region types*

M. Tofte and J. Talpin. Region-based memory management. *Information and Computation*, 132(2), 1997.

D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in *Cyclone*. In *PLDI*, 2002.

↑ Taken from Khue Ho