RUST BASICS

- Variables are bound with `let`:

You infer type of variable from declaration

```
let x = 17;
```

- Bindings are implicitly-typed: the compiler infers based on context.
- The compiler can't always determine the type of a variable, so sometimes you have to add type annotations.

```
let x: i16 = 17;
```

- Variables are inherently immutable:

```
let x = 5;
x += 1; // error: re-assignment of immutable variable x
let mut y = 5;
y += 1; // OK!
```

- (Almost!) everything is an expression: something which returns a value.
  - Exception: variable bindings are not expressions.

- Primitive types a lot
  - bool, char, numerics (i8, i16, …, u8, u16, …)
- Arrays
    - Arrays are generically of type [T; N].
    - N is a compile-time *constant*. Arrays cannot be resized.
    - Array access is bounds-checked at runtime.
    - Arrays are indexed with [] as in C

- Strings
- Tuples (fixed-sized heterogenous lists)
- Structs, unions, …
- Unit type (contains a single value - ())
- No void type

FUNCTIONS

```
// comment
fn main() {
    println!("Hello, world!");
}
```

Declared by keyword fn

Hello, world!

Recursive functions

```
fn fact(n:i32) -> i32
{
  if n == 0 { 1 }
  else {
    let x = fact(n-1);
    n * x
  }
}
```

- You can declare recursive functions normally

```
fn main() {
  let res = fact(6);
  println!("fact(6) = {}",res);
}
```

```
fact(6) = 720
```

let for f declaration is needed for Ocaml but there is a complete match

OCAML VS RUST

```
let f (m : int) (n : int) : int =
  let mm = 2 * m in
  mm + n    ;;

let main () =
  Printf.printf "%d" (f 3 5)  ;;
```

```
fn f(m: i32, n: i32) -> i32 {
  let mm = 2 * m;
  mm + n
}

fn main() {
  println!("{}", f(3, 5));
}
```

RUST is is expression-oriented. (short way of saying x is y in z)

The semicolon in **let x = y; z** has basically the role of the **in** keyword in Ocaml:

it defines an expression, not a statement as in C

- an expression is not a statement

When we say **Rust is expression-oriented**, we mean that **almost everything in Rust produces a value**, and that value can be used as part of a larger expression. This is in contrast to languages like C, where many things (like `if`, `while`, or even a `let` declaration) are **statements**, meaning they *do* something, but don't *return* a value you can use directly.

So what's the difference?

- A **statement** does something, like declaring a variable or calling a function, but it **doesn't return a value** you can use.
- An **expression**, on the other hand, **evaluates to a value**. You can use that value inside other expressions.

You're saying:

> Bind `x` to the value of `y`, then evaluate the expression `z`, **using that binding of x**. The whole thing is an *expression* that evaluates to the result of `z`.

IF expressions (not statements)

```
fn main() {
    let n = 5;
    if n < 0 {
        print!("{} is negative", n);
    } else if n > 0 {
        print!("{} is positive", n);
    } else {
        print!("{} is zero", n);
    }
}
```

5 is positive

LET

- By default, Rust variables are immutable
  - Usage checked by the compiler
- **`mut`** is used to declare a resource as mutable.

```
fn main() {
  let a: i32 = 0;
  a = a + 1;
  println!("{}" , a);
}
```

Compile error

```
fn main() {
  let mut a: i32 = 0;
  a = a + 1;
  println!("{}" , a);
}
```

LET BY EXAMPLES

```
{
  let x = 37;
  let y = x + 5;
  y
}//42
```

```
{
  let x = 37;
  x = x + 5;//err
  x
}
```

```
{ //err:
  let x:u32 = -1;
  let y = x + 5;
  y
}
```

```
{
  let x = 37;
  let x = x + 5;
  x
}//42
```

```
{
  let mut x = 37;
  x = x + 5;
  x
}//42
```

```
{
  let x:i16 = -1;
  let y:i16 =
x+5;
  y
}//4
```

Redefining a variable *shadows* it (like OCaml)

Assigning to a variable only allowed if `mut`

Type annotations must be consistent (may override defaults)

When you explicitly write type of variable

QUIZ: What does this evaluate to?

```
{ let x = 6;
  let y = "hi";
  if x == 5 { y } else { 5 };
  7
}
```

A. 6

B. 7

C. 5

D. Error

- ; means sth
- x and y are of different types.

Expressions can be placed in composition with other expressions, but the conditional expression cannot return either a string or an int. This is an expression oriented prog. language.

QUIZ #1: What does this evaluate to?

```
{ let x = 6;
  let y = "hi";
  if x == 5 { y } else { 5 };
  7
}
```

A. 6

B. 7

C. 5

D. Error – if and else have incompatible types

Mutation and Loop

```
fn fact(n: u32) -> u32 {
  let mut x = n;
  let mut a = 1;
  loop {
    if x <= 1 { break; }
    a = a * x;
    x = x - 1;
  }
  a
}
```

You can have loops of course. The break specifies how to exit from loop.

infinite loop
(break out)

Looping constructs

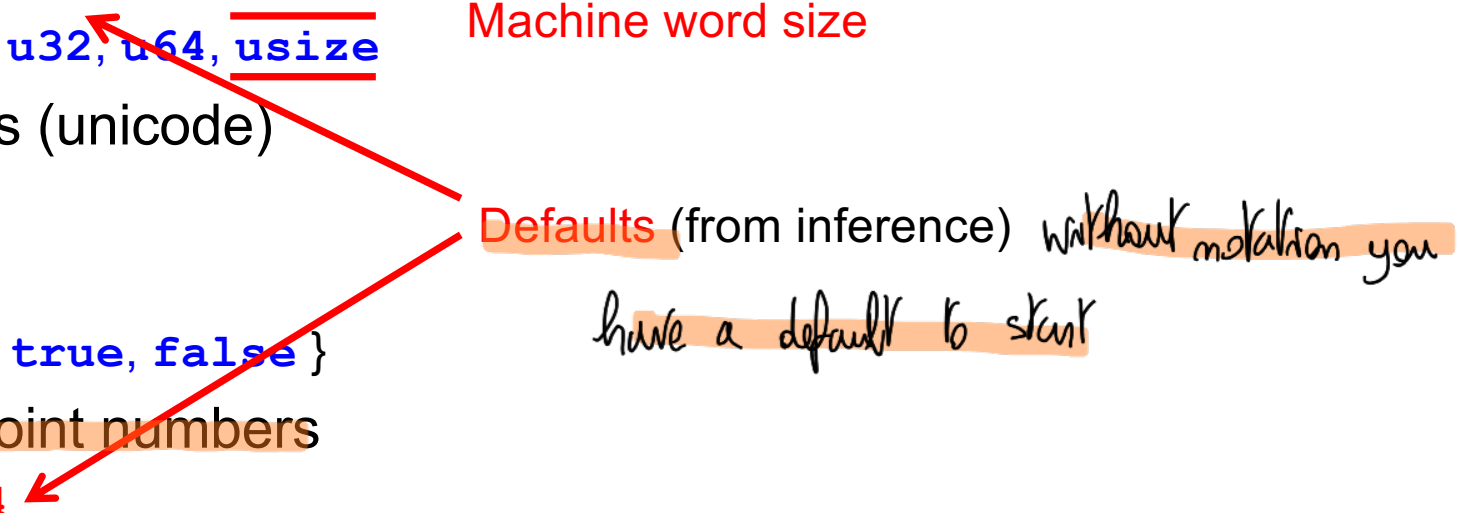
- **while ….**, **for** …, **loop** …
- These looping constructs are ***expressions***
  - They return the final computed value
  - unit, if none

`break` may take an expression argument, which is the final result of the loop

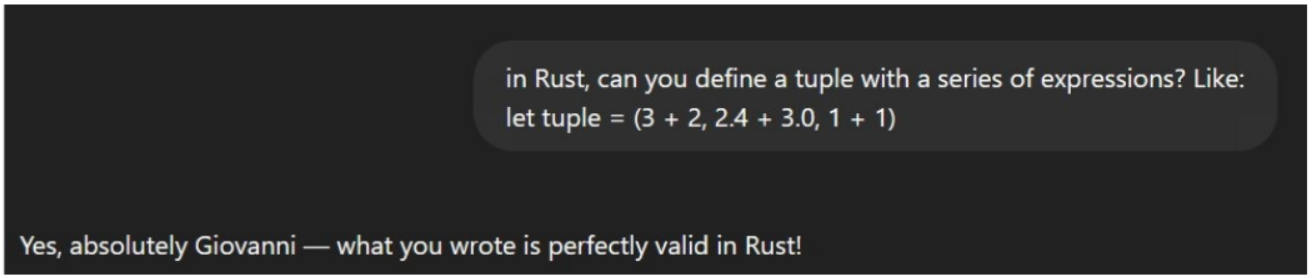
```
let mut x = 5;
let y = loop {
      x += x - 3;
      println!("{}", x);// 7 11 19 35
      x % 5 == 0 { break x; }
};
print!("{}",y); //35
```

Scalar type

- Integers
  - `i8`, `i16`, `i32`, `i64`, `isize`
  - `u8`, `u16`, `u32`, `u64`, `usize`
- Characters (unicode)
  - `char`
- Booleans
  - `bool` = { `true`, `false` }
- Floating point numbers
  - `f32`, `f64`
- Note: arithmetic operators (+, -, etc.) *overloaded*



tuples



Tuples

- n-tuple type `(t1, …, tn)`
  - `unit ()` is just the 0-tuple
- n-tuple expression `(e1, …, en)`
- Accessed by pattern matching or like a record field

sequence of expressions. You can use pattern matching to go inside tuple

```
fn main() {
    let tup = (500, 6.4, 1);

    let (x, y, z) = tup;

    println!("The value of y is: {}", y);
}
```

let (x, y, z) = tup; ⟶ x=500, y=6.4, z=1.

Can use a pattern to refer contents of the tuple directly

Type of tuple is cartesian product of components' type: in this case is Int x Real x Int. It does not need to be homogeneous.

```
fn main() {
    let x: (i32, f64, u8) = (500, 6.4, 1);

    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;

    println!("The value of five_hundred is: {}", five_hundred);

    println!("The value of six_point_four is: {}", six_point_four);

   println!("The value of one: {}", one);

}
```

pattern match to go inside tuple

RUN ▶ ··· DEBUG STABLE ···

```
    let five_hundred = x.0;

    let six_point_four = x.1;

    let one = x.2;

    println!("The value of five_hundred is: {}", five_hundred);

    println!("The value of six_point_four is: {}", six_point_four);

   println!("The value of one: {}", one);

}

```

Execution

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.57s
     Running `target/debug/playground`
```

Standard Output

```
The value of five_hundred is: 500
The value of six_point_four is: 6.4
The value of one: 1
```

Arrays

Standard operations

- Creating an array (can be mutable or not)
  - But must be of fixed length
- Indexing an array
- Assigning at an array index (only if mutable)

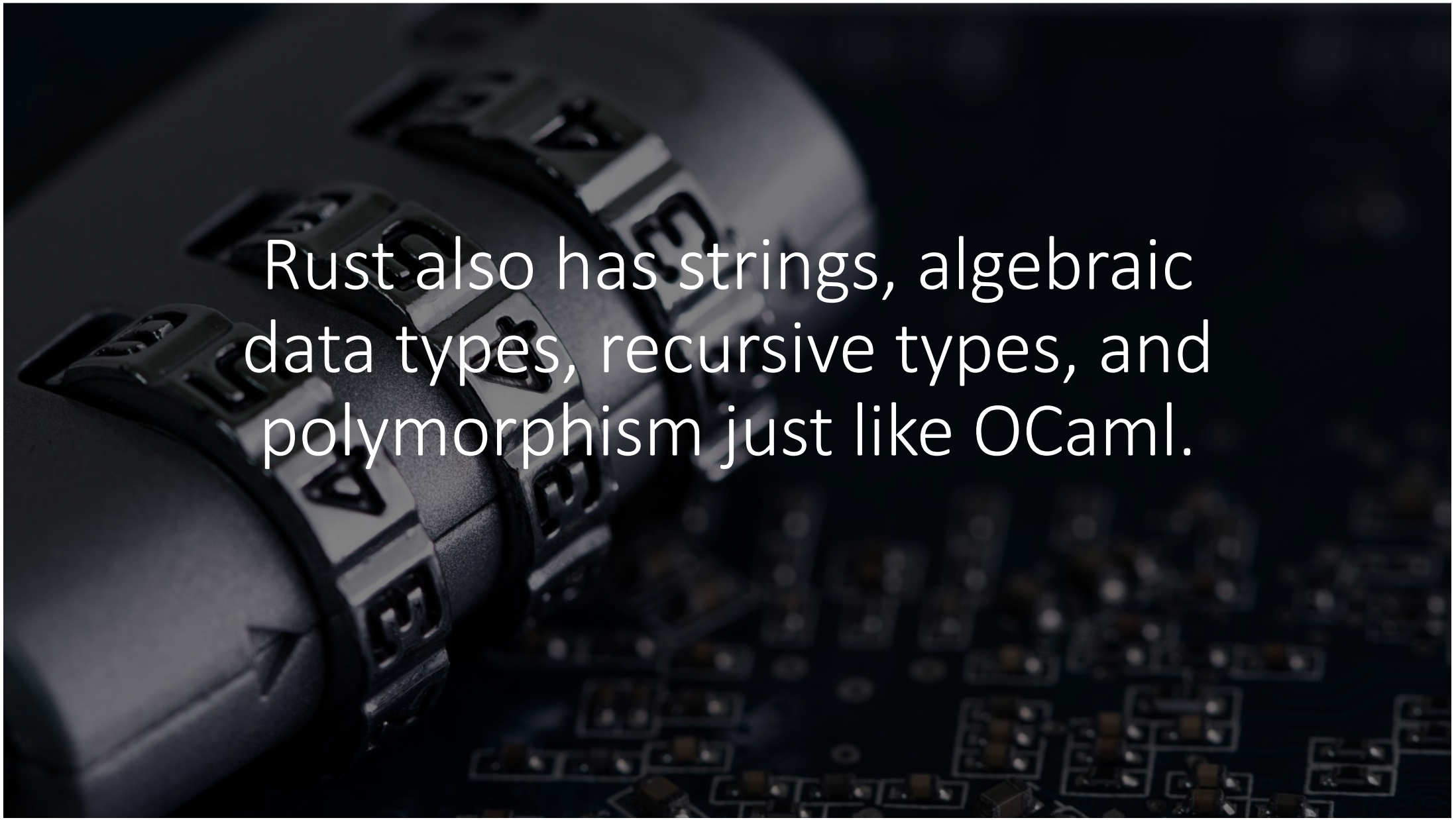
```
let nums = [1,2,3];
let strs = ["Monday","Tuesday","Wednesday"];
let x = nums[0]; // 1
let s = strs[1]; // "Tuesday"
let mut xs = [1,2,3];
xs[0] = 1; // OK, since xs mutable
let i = 4;
let y = nums[i]; //fails (panics) at run-time
```

↑ out of bounds: C doesn't give problems, no checks on size.
Here must gives failure at compile time. In C panic at run time.

- Rust provides a way to iterate over a collection
  - Including arrays

```
let a = [10, 20, 30, 40, 50];
for element in a.iter() {
  println!("the value is: {}", element);
}
```

  - `a.iter()` produces an iterator, like a Java iterator


Rust also has strings, algebraic data types, recursive types, and polymorphism just like OCaml.

We have polymorphic data structures

```
// Polymorphic record for points. Equivalent to type 'a point = {x : 'a; y : 'a}.

struct Point<T> {
  x: T, y: T
}

// Polymorphic function. T: ToString means "all types T where T has a to_string
// method." ToString is a trait, which we will discuss later.

fn point_print<T: ToString>(p: Point<T>) {
  println!("({}, {})", p.x.to_string(), p.y.to_string());
}
```

We define a generic structure of a pair of elements

base type T;

We can add constraints

We have a pair as input, Point<T> in which T is ToStringable.

Management of S and H is different:

Data allocation in memory: Stack vs Heap

The stack

- constant-time, automatic (de)allocation
- Data size and lifetime must be known at compile-time
- Function parameters and locals of known (constant) size

CONTAINS

The heap

- Dynamically sized data, with non-fixed lifetime
- Slightly slower to access than stack via a reference

References

(not ptrs)

Reference *types* are written with an &: `&i32`.

References can be taken with & (like C/C++).

References can be *dereferenced* with * (like C/C++).

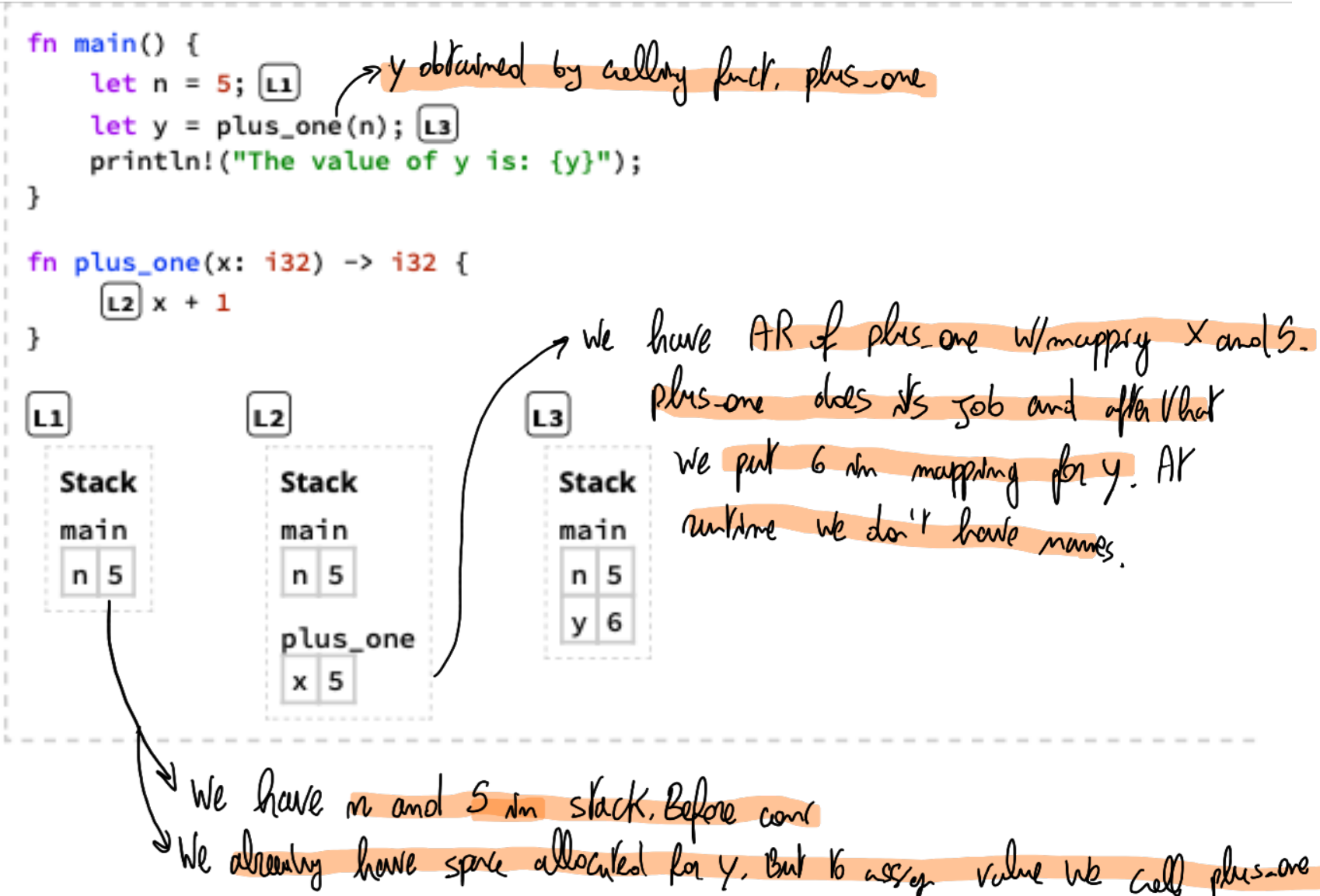
References are guaranteed to be valid.

- Validity is enforced through compile-time checks!

These are *not* the same as pointers!

```
let x = 12;
let ref_x = &x;
println!("{}", *ref_x); // 12
```

Stack allocation



Stack allocation

```
let a = [0; 1_000_000]; L1
let b = a; L2
```

L1

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0

stack contains 10^6 elements of array.

L2

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0
b	0	0	0	0	0	0	0	0	0	0	...	0

This wouldn't make any sense to manage array allocation

if you do:
let a = [0;10];
let b = a;

Stack allocation

Rust *moves* the value of `a` into `b` — or *copies* it, if the type implements the `Copy` trait.

Now, `[0; 10]` is an array of 10 integers, and since Rust arrays (of fixed size and simple types like `i32`) implement `Copy`, the whole array gets **copied**, not moved.

So after that line:

- `a` still exists and can be used.
- `b` now contains a *separate copy* of that array.

If instead it were something like a `Vec<i32>` (which is heap-allocated and does **not** implement `Copy`), you'd get a **move**, and trying to use `a` after assigning it to `b` would trigger a compiler error — unless you clone it manually.

```
let a = [0; 1_000_000]; L1
let b = a; L2
```

L1

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0

L2

Stack

main

a	0	0	0	0	0	0	0	0	0	0	...	0
b	0	0	0	0	0	0	0	0	0	0	...	0

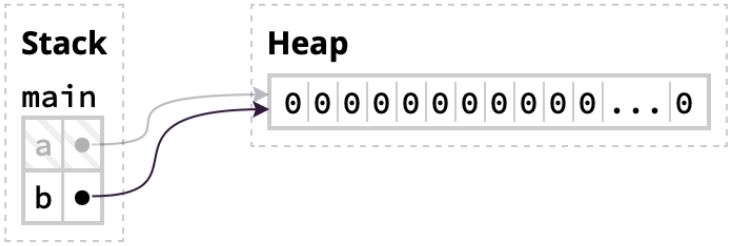
Copying a into b causes the main AR to contain 2 million elements.!!!!!

Boxes lives in the Heap

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```

L1

L2



The statement `let b = a` copies the pointer from `a` into `b`, but the pointed-to data is not copied.

Note that `a` is now grayed out because it has been *moved*.

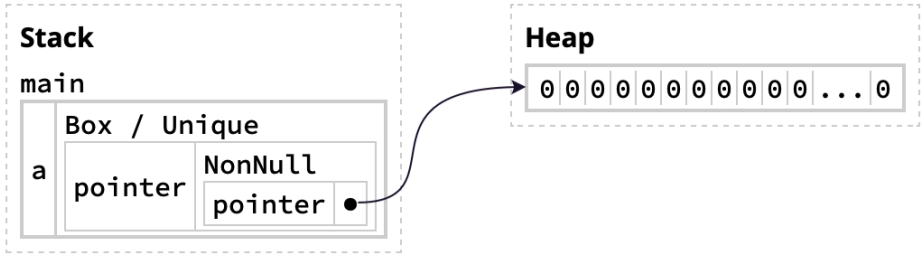
We have just a pointer movement.
The pointer from a to the heap is grayed (will see).

You cannot use a anymore

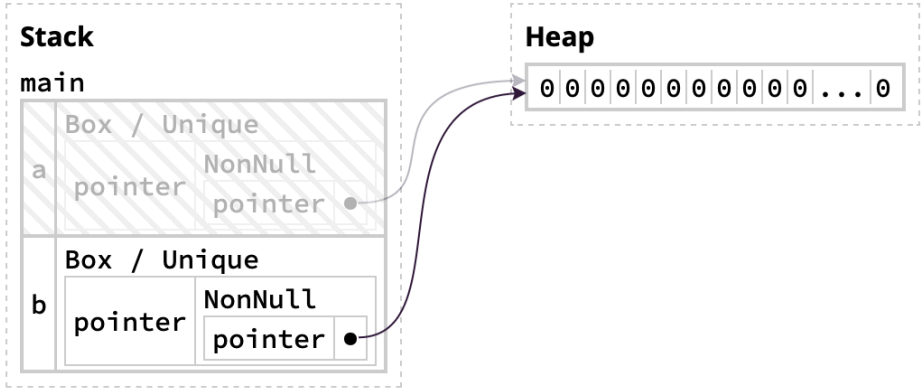
Boxes lives in the Heap

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```

L1



L2



In many prog. language you have metadata to perform checks: for arrays on heap, Box never entry allocated on the heap, a pointer (they specifies if ptr is null or non Null). In this case it is Non Null and content is actual pointer.

RUST: safe memory management

- Rust heap memory management: **without GC**
  - Type checking ensures **no dangling pointers**
  - unsafe idioms are disallowed

You have a ptr to heap: you either use a GC or manual management.
Rust: no GC but no manual management.
Management of heap does not rely on programer

entity over the heap (Rust adopts notation of Box)

IDEA

(if other conditions are met)

Box deallocation principle (almost correct):

If a variable is bound to a box, when Rust deallocates the variable's frame, then Rust deallocates the box's heap memory.

AR

↳ in which you find variable

Variable is in stack: there's ptr pointing element of stack to heap

Rust removes from heap Data Structure

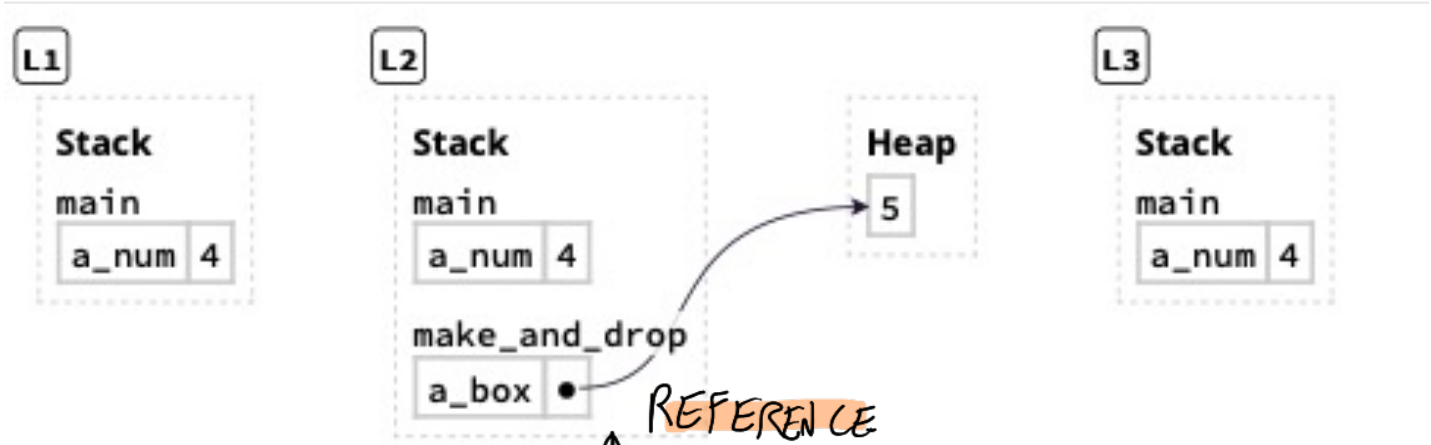
GARBAGE COLLECTOR:

- Mark and Sweep: you start from stack variables (active vars in a program) and you follow root to elements in heap and mark as live those elements.

```
fn main() {
    let a_num = 4;
     L1 make_and_drop(); L3
}

fn make_and_drop() {
    let a_box = Box::new(5); L2
}
```

int value a_num

allocates a box containing 5 over the heap



REFERENCE

bond between var and mem

and a ptr to heap

When make and drop terminates, we also remove 5 from heap.

Boxes data work well!! .... What if

```
let a = Box::new([0; 1_000_000]);
let b = a;
```

The boxed array has now been bound to both a and b (alias).

By applying the "almost correct" principle, Rust would try to free the box's heap memory *twice* on behalf of both variables.

b might also have been declared before

That's undefined behavior too!

Idea: OWNERSHIP: when you have a var bounded to a Data Structure (DS), that var is the owner of that DS. A piece of data might have at most 1 owner at a time.

Ownership

A variable binding takes ownership of its data.

A piece of data can only have one owner at a time.

When a binding goes out of scope, (1) the bound data is released automatically. Because there is only one owner

For heap-allocated data, this means heap-deallocation.

(1) Variable is removed from stack at end of its scope: variable on stack that is the owner of DS in heap

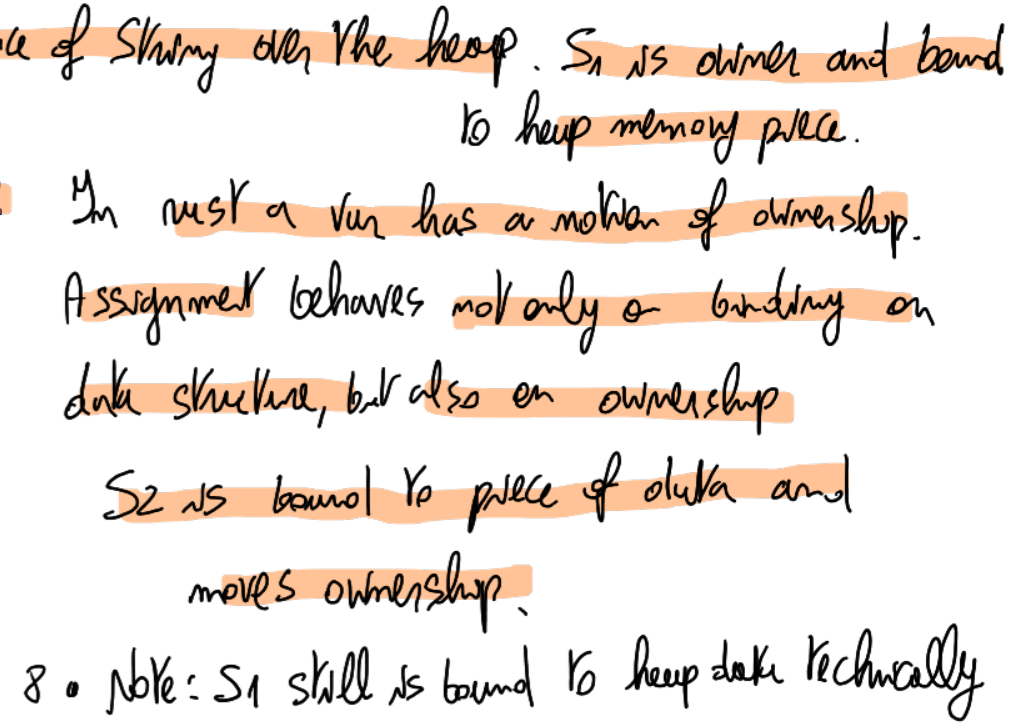
Ownership Example

```
fn foo() {
    // Creates a Vec object.
    // Gives ownership of the Vec object to v1.
    let mut v1 = vec![1, 2, 3];
    v1.pop();
    v1.push(4);
    // At the end of the scope, v1 goes out of scope.
    // v1 still owns the Vec object, so it can be cleaned up.
}
```

The notion of “move”

- When a variable is assigned to another variable, ownership is transferred: moved in Rust jargon

```
let s1 = String::from("hello");
let s2 = s1; // ownership moves to s2

// s1 is now invalid
```



Ownership: the move semantics

```
let v1 = vec![1, 2, 3];
let v2 = v1;   // Ownership of the Vec object moves to v2.
println!("", v1[2]); // error: use of moved value ‘v1‘
```

- `let v2 = v1;`
  - We don’t want to copy the data, since that’s expensive.
  - The data cannot have multiple owners.
  - **Solution:** move the Vec’s ownership into v2, and declare v1 invalid.

v1 is not anymore the owner

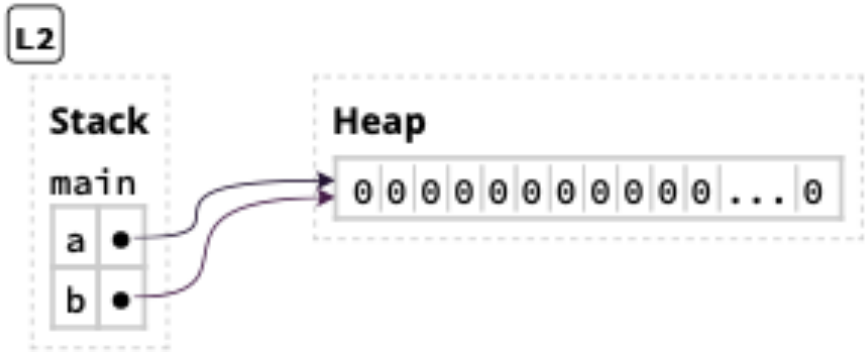
- `println!("{}", v1[2]);`
  - We know that v1 is no longer a valid variable binding, ∴ error!
  - Rust can reason about this at compile time, ∴ compiler error.
  - Moving ownership is a compile-time semantic.
    It doesn’t involve moving data during your program.

Move (of ownership) at run-time

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```

L1

L2

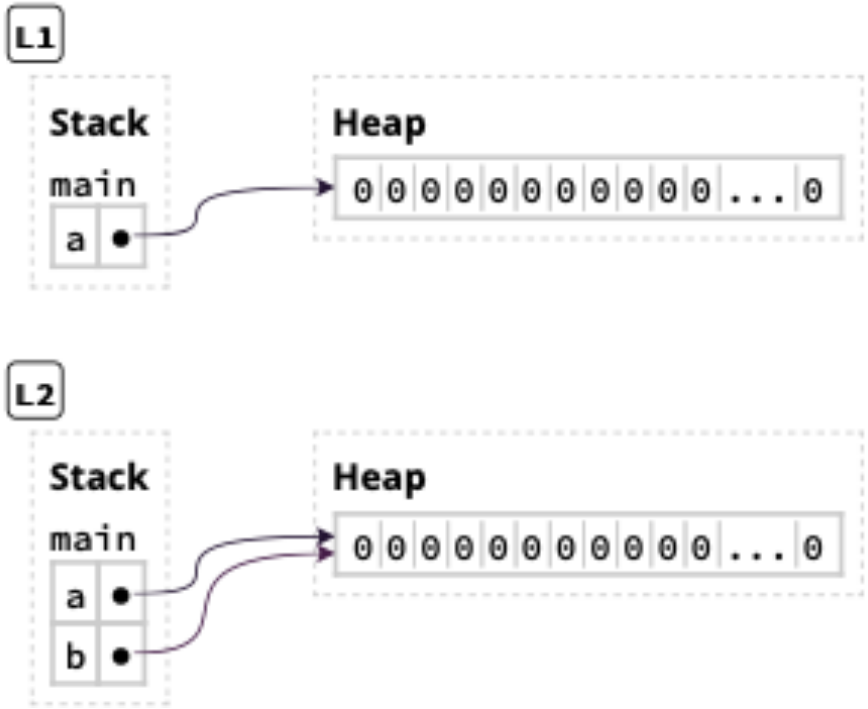


From implementation pov, reference is there, but after b=a, a is not the owner anymore and you cannot access vector through a. There is no ownership but at runtime because ownership is checked statically.

Idea: you don't care at runtime because you solve this problem statically.

Move at run-time

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```



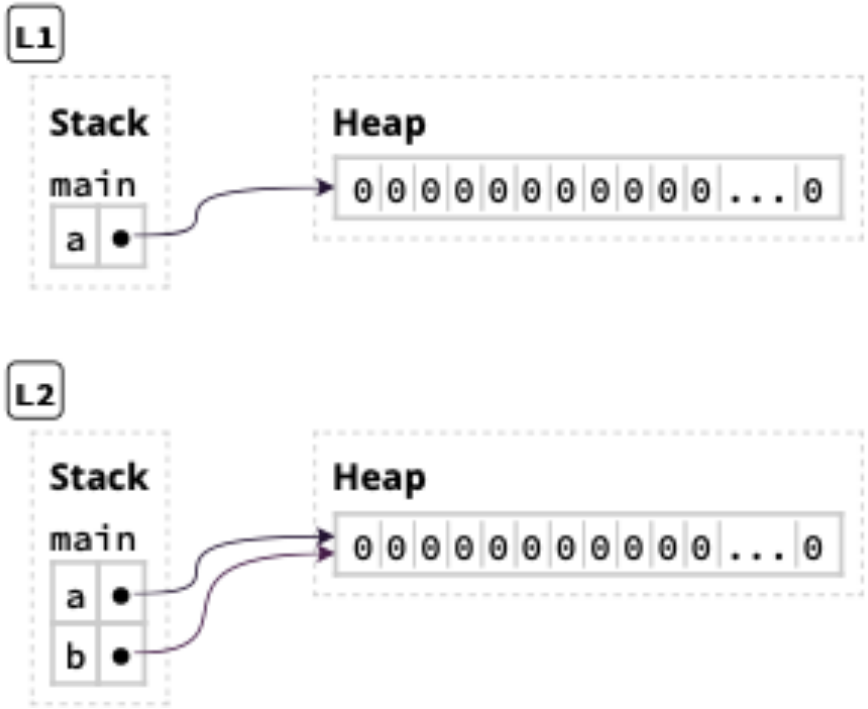
Common question:

if a is moved at L2, why is it still in the run-time?

Shouldn't a disappear, or gray out, or otherwise "move" somewhere?

Move at run-time

```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```



At runtime, nothing happens to a when it is moved.

There is no "ownership bit" that gets flipped in memory.

Ownership only exists at compile-time.

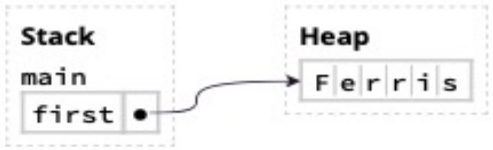
The picture does not show how the compiler "thinks" about the program. It shows how the program actually executes at runtime. At runtime, a move is just a copy. At compile-time, a move is a transfer of ownership.

```rust
fn main() {
    let first = String::from("Ferris"); L1
    let full = add_suffix(first); L4
    println!("{full}");
}

fn add_suffix(mut name: String) -> String {
    L2 name.push_str(" Jr."); L3
    name
}
```

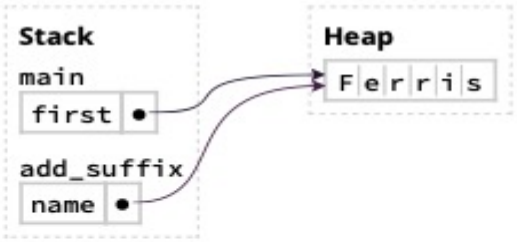
we take a mutable name and add Jr.

At L1, the string "Ferris" has been allocated on the heap. It is owned by first.

L1

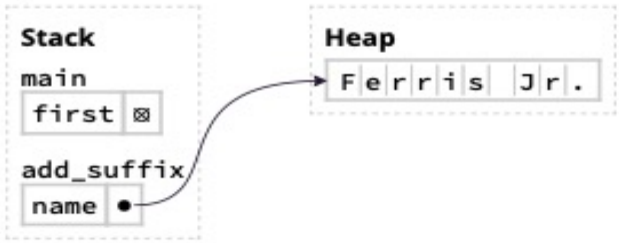


L2



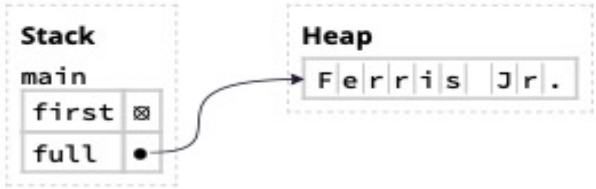
Share mem on the heap because of param passing

L3



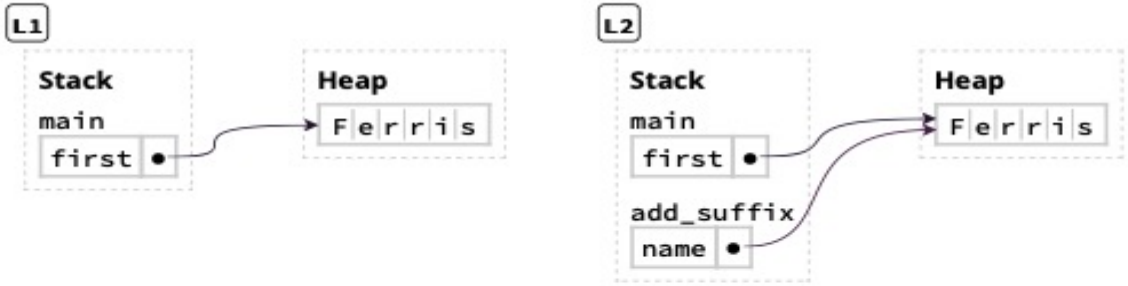
First is not anymore the owner

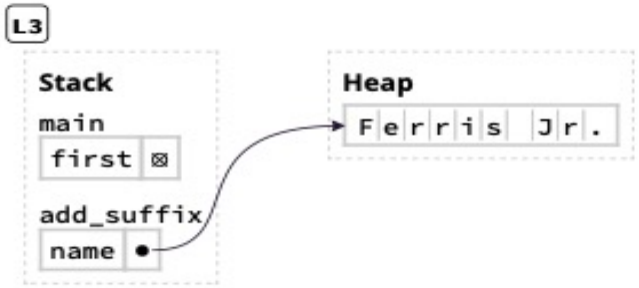
L4

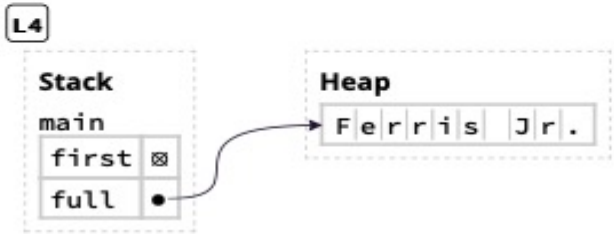


Full contains element correctly.

```rust
fn main() {
    let first = String::from("Ferris"); L1
    let full = add_suffix(first); L4
    println!("{full}");
}

fn add_suffix(mut name: String) -> String {
    L2 name.push_str(" Jr."); L3
    name
}
```



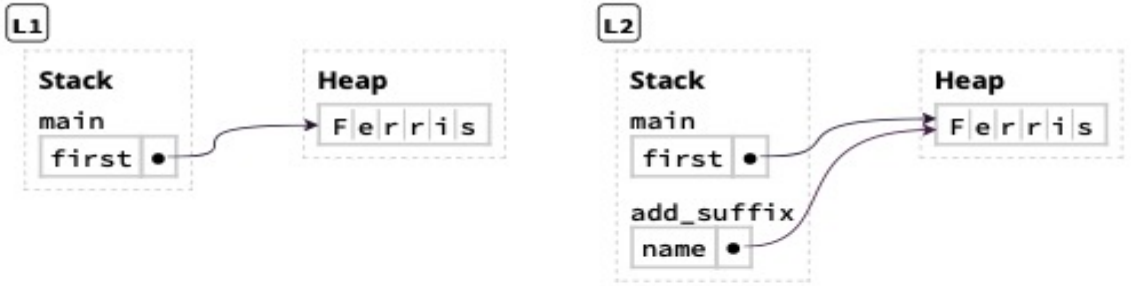


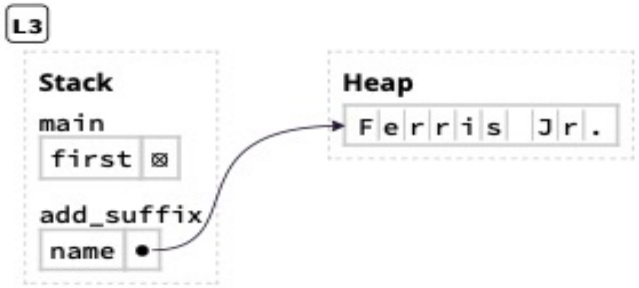


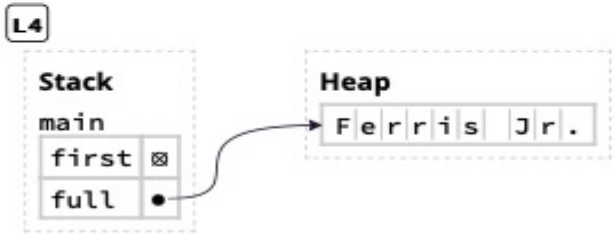
At L1, the string "Ferris" has been allocated on the heap.
It is owned by first.

At L2, the function add_suffix(first) has been called.
This moves ownership of the string from first to name.
The string data is not copied, but the pointer
to the data is copied.

```rust
fn main() {
    let first = String::from("Ferris"); L1
    let full = add_suffix(first); L4
    println!("{full}");
}

fn add_suffix(mut name: String) -> String {
    L2 name.push_str(" Jr."); L3
    name
}
```







At L1, the string "Ferris" has been allocated on the heap.
It is owned by first.

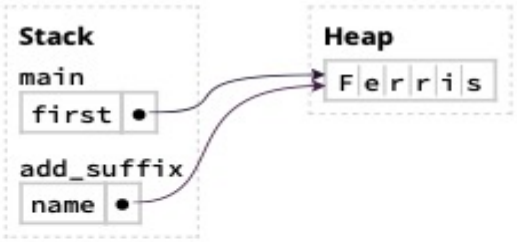
At L2, the function add_suffix(first) has been called.
This moves ownership of the string from first to name.
The string data is not copied, but the pointer
to the data is copied.

At L3, the function name.push_str(" Jr.") resizes
the string's heap allocation.
A new larger allocation is created
The original heap memory is dropped.

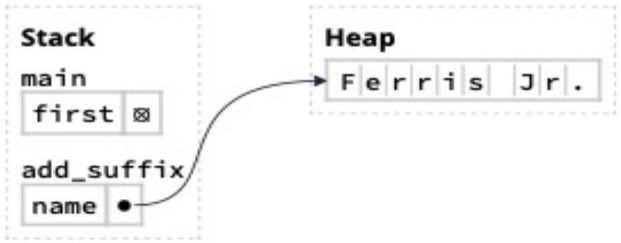
```rust
fn main() {
    let first = String::from("Ferris"); L1
    let full = add_suffix(first); L4
    println!("{full}");
}

fn add_suffix(mut name: String) -> String {
    L2 name.push_str(" Jr."); L3
    name
}
```

L1

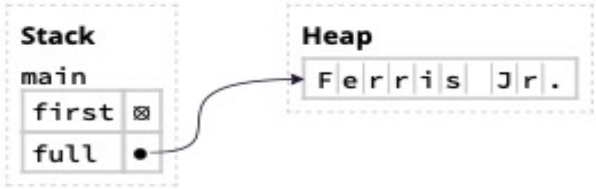
L2



L3



L4



At L1, the string "Ferris" has been allocated on the heap.
It is owned by first.

At L2, the function add_suffix(first) has been called.
This moves ownership of the string from first to name.
The string data is not copied, but the pointer
to the data is copied.

At L3, the function name.push_str(" Jr.") resizes
the string's heap allocation.
A new larger allocation is created
The original heap memory is dropped.

At L4 the ownership of the result is moved to full.

rust Copia Modifica

```rust
let first = String::from("Ferris");
```

Here, `first` owns a `String`. It's **immutable**, which just means you **can't change what `first` points to** or call mutating methods on `first` *while it's still `first`*.

But when you do:

rust Copia Modifica

```rust
let full = add_suffix(first);
```

You are **moving** ownership of the `String` from `first` into the `name` parameter of `add_suffix`.

At that point, `first` is **gone** — it's invalid, done, dead to the world — and `name` is now the new owner of the `String`. And guess what? Inside the function, `name` **is declared as `mut`**, so *within that scope*, you're allowed to mutate it.

It doesn't matter that `first` wasn't `mut` — it only matters that `name` is.

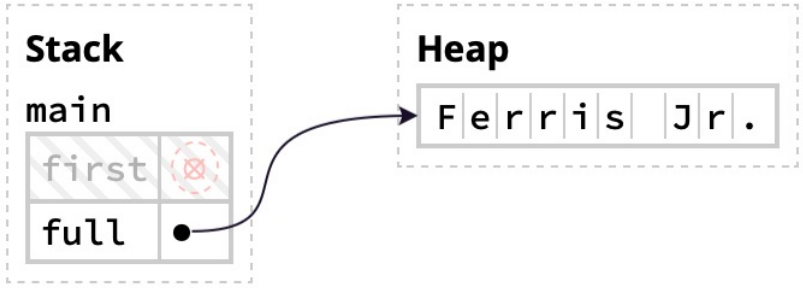
- You don't need `first` to be `mut` because you're not mutating `first`; you're moving it.
- Once `first` is moved into `name`, the function owns it.
- `name` is mutable, so the function can change it.

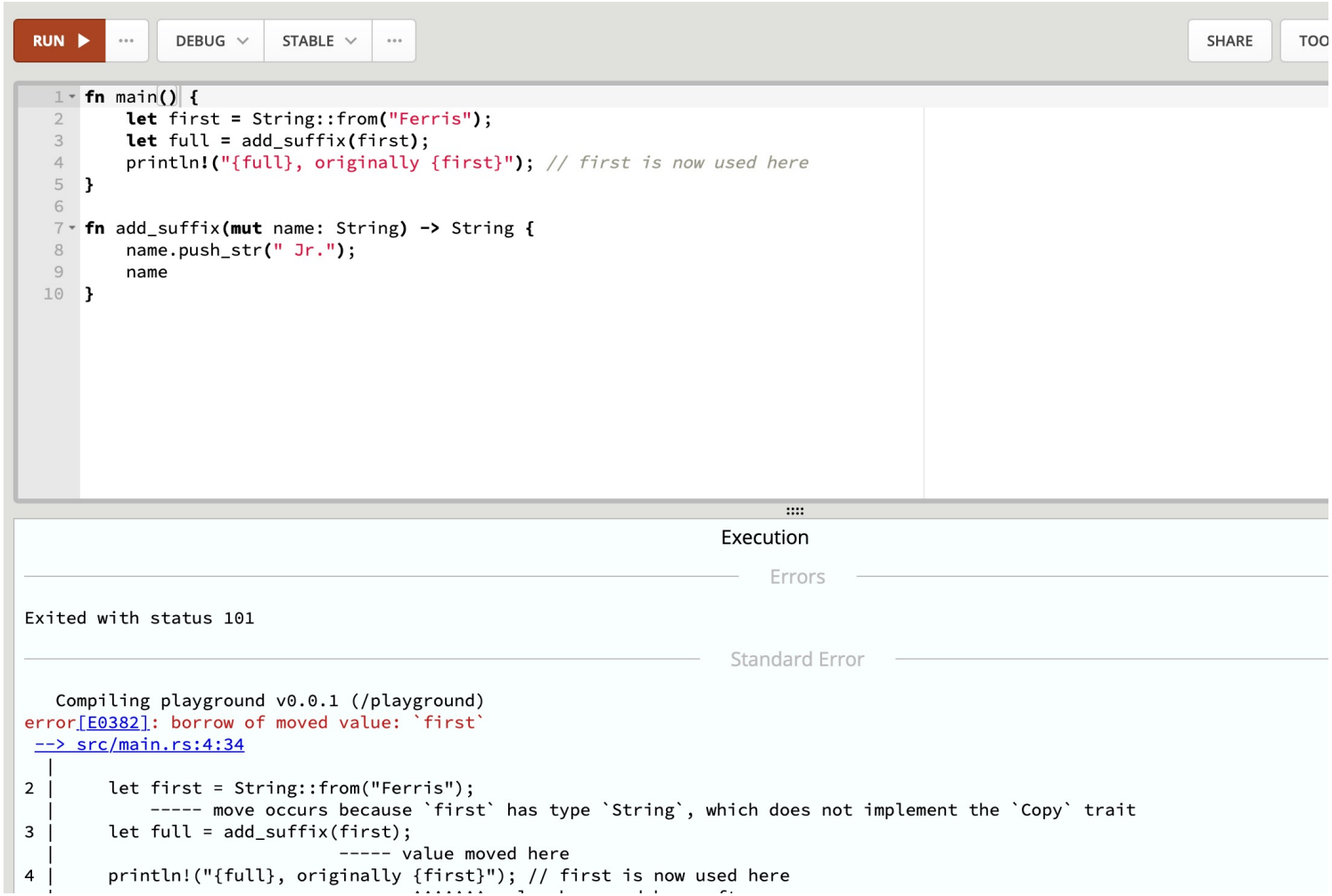
You are just saying that your formal parameter is mutable

Variables cannot be used after being moved

```
fn main() {
    let first = String::from("Ferris");
    let full = add_suffix(first);
    println!("{full}, originally {first}"); L1  // first is now used here
}

fn add_suffix(mut name: String) -> String {
    name.push_str(" Jr.");
    name
}
```

L1 undefined behavior: pointer used after its pointee is freed




RUN
DEBUG
STABLE
SHARE
TOO
1 fn main() {
2 let first = String::from("Ferris");
3 let full = add_suffix(first);
4 println!("{full}, originally {first}"); // first is now used here
5 }
6
7 fn add_suffix(mut name: String) -> String {
8 name.push_str(" Jr.");
9 name
10 }
Execution
Errors
Exited with status 101
Standard Error
Compiling playground v0.0.1 (/playground)
error[E0382]: borrow of moved value: `first`
--> src/main.rs:4:34
2 | let first = String::from("Ferris");
----- move occurs because `first` has type `String`, which does not implement the `Copy` trait
3 | let full = add_suffix(first);
----- value moved here
4 | println!("{full}, originally {first}"); // first is now used here

8

Moved heap data principle: if a variable x moves ownership of heap data to another variable y, then x cannot be used after the move.

✅ Example 2: Move of a purely stack value (like a struct)

rust Copia Modi

```rust
struct Point { x: i32, y: i32 }

fn main() {
    let p1 = Point { x: 3, y: 4 };
    let p2 = p1; // move!
    // p1 is no longer usable
}
```

Even though `Point` is stored entirely on the stack, its **ownership** still gets moved from `p1` to `p2`.

It's not about heap vs. stack. It's about:

> *Who gets to "own" this value, and is that value allowed to be duplicated (via `Copy`) or not?*

if not copiable, it is moved

Typing Rust: the basic principle

- The key principle of the Rust type system is the *exclusion principle*, also known as *mutability XOR aliasing*:
- Data can *either* be mutated exclusively through *one* unique pointer, *or* it can be *immutably* shared amongst many parties—but **not both at the same time**.

🔹 Mutability XOR Aliasing

- **Mutability** means being able to change (or mutate) the data.
- **Aliasing** means having **multiple references** to the same data (i.e., the data is shared by multiple parts of the program).

The key idea in Rust is that you **cannot have both** mutability and aliasing (multiple references to the same data) at the same time.

You can have:

- **Exclusive mutability**: One reference to data, and you can change it (mutate it).
- **Immutable sharing**: Multiple references to the same data, but **no one** is allowed to mutate it.

🔹 Why does Rust enforce this?

Rust's ownership and borrowing rules are designed to **prevent data races** and **undefined behavior** in concurrent environments.

- **If a piece of data is mutable and shared** (i.e., multiple threads are modifying it at the same time), this can lead to unpredictable behavior or corruption of data.
- **If data is immutable and shared**, there's no risk of modification, so the data can be safely accessed by multiple parties.

- Ownership is modified by assignment

Typing Rust: ownership

- The type of a variable not only represents information about the values it can take—it also represents *ownership* of resources related to the variable such as memory or file descriptors.
- When a variable gets passed between functions or threads, the associated ownership is *transferred* or “moved” with it.

Rule of ownership

1. Each value in Rust has a variable that's its **owner**
2. There can only be **one owner at a time**
3. When the **owner goes out of scope** the value will be **dropped** (freed)

Box deallocation principles

If a variable **owns** a box, when Rust deallocates the variable's frame, then Rust deallocates the box's heap memory (**drop**).

Scope rules … as usual

```
fn main() {
            // s is not valid here, it's not yet declared
  let s = "hello";
            // s is valid from this point forward
  // do stuff with s
}
            // this scope is now over, and s is no longer valid
```

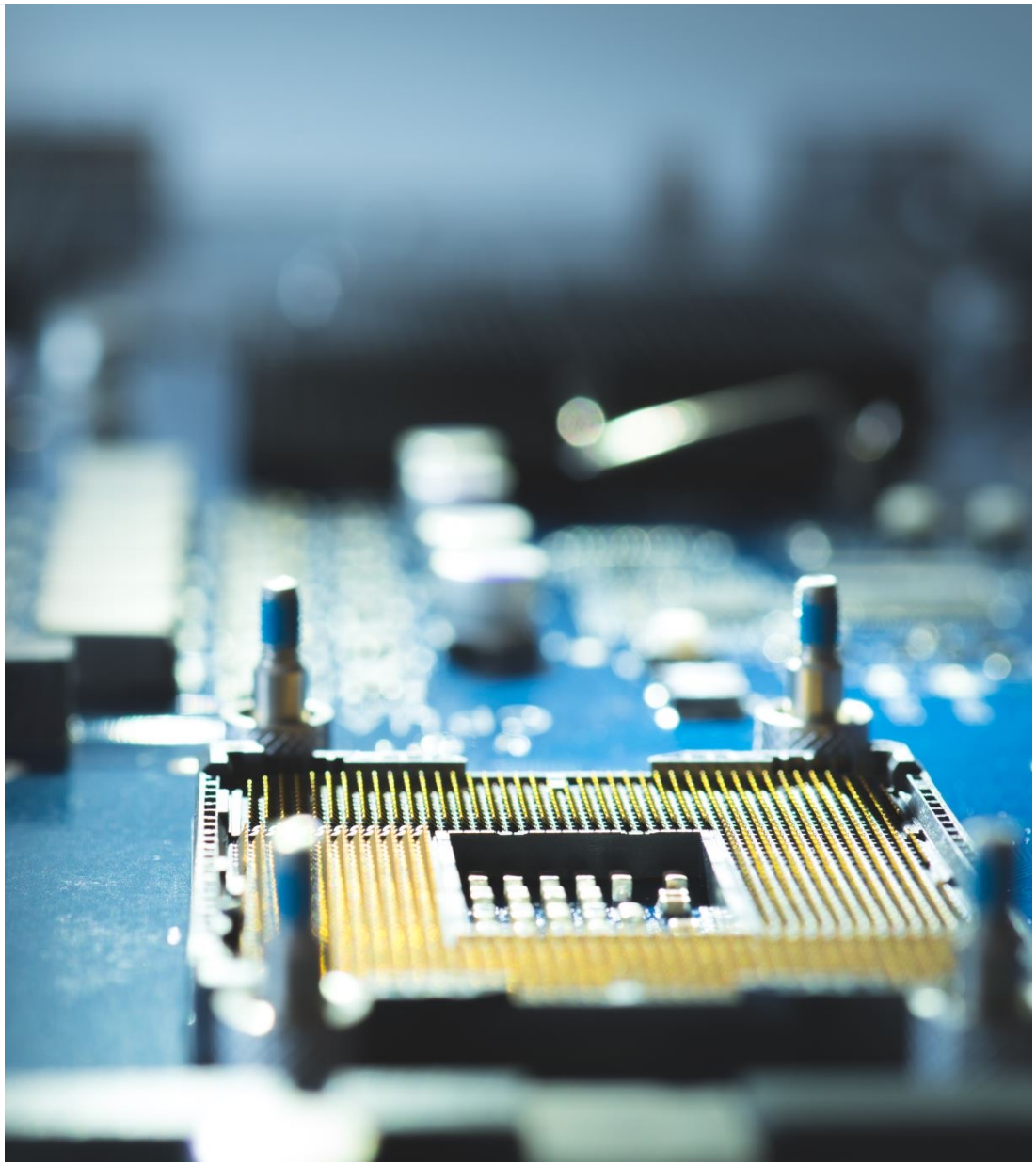
When s comes *into scope*, it is valid.

It remains valid until it goes *out of scope*.

Memory usage (1)

The memory must be requested from the memory allocator at runtime (standard in PL!!!)

We need a way of returning this memory to the allocator when we're done with our dynamic data.



Memory usage (2)

Returning memory is done differently in programming languages.

Memory usage (2)

Returning memory is done differently in programming languages.

**In languages with a *garbage collector (GC)*, the GC keeps track and cleans up memory that isn't being used anymore,
and the programmer don't need to think about it.**

Memory usage (2)

Returning memory is done differently in programming languages.

In languages with a *garbage collector (GC)*, the GC keeps track and cleans up memory that isn't being used anymore, and the programmer don't need to think about it.

Without a GC, it's responsibility of the programmer to identify when memory is no longer being used and call code to explicitly return it, just as we did to request it.

Memory usage (2)

Returning memory is done differently in programming languages.

In languages with a *garbage collector (GC)*, the GC keeps track and cleans up memory that isn't being used anymore, and the programmer don't need to think about it.

Without a GC, it's responsibility of the programmer to identify when memory is no longer being used and call code to explicitly return it, just as we did to request it.

!!Doing this correctly has historically been a difficult programming problem!!

If we forget, we'll waste memory (**garbage**).

If we do it too early, we'll have an invalid variable (**dangling references**).

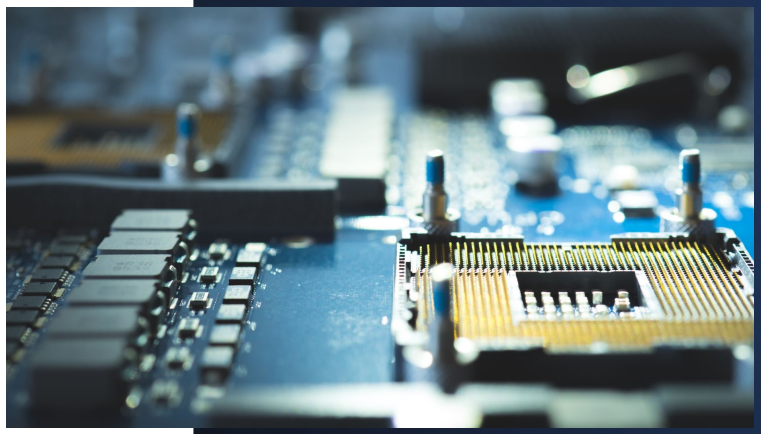
What about RUST?

RUST: drop

When a variable goes out of scope, Rust calls the special run-time function **drop**

(roughly a free())

Rust calls the drop function automatically at the closing of blocks.



String: dynamically sized mutable data

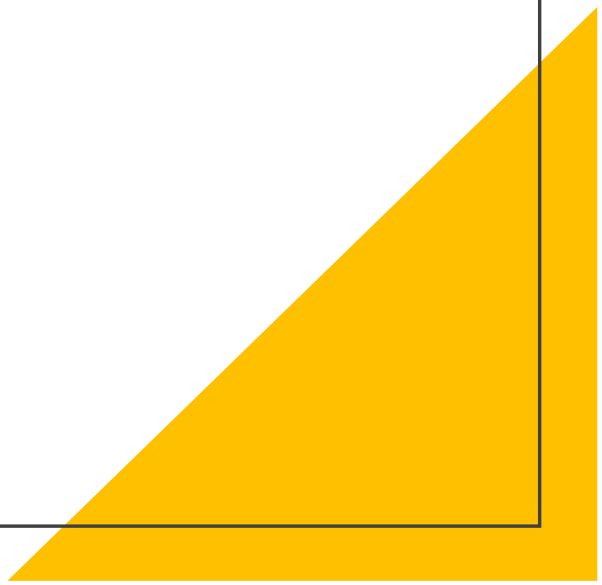
```
{
 let mut s = String::from("hello");
 s.push_str(", world!"); //appends to s
 println!("{}", s); //prints hello, world!
} //s's data is freed by calling s.drop()
```

namespace
(like Java package)

- The contents of string **s** is allocated on the heap
  - Pointer to data is internal to the **String** object rep
  - When appending to **s**, the old data is freed and new data is allocated
- **s** is the *owner* of this data
  - When **s** goes out of scope, its **drop** method is called, which frees the data

Intermezzo

Deep vs shallow copy



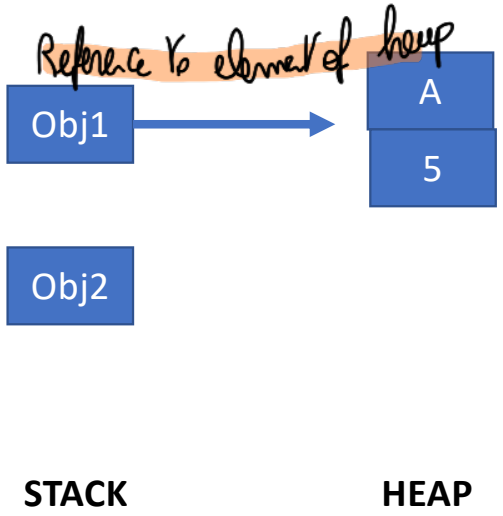
Memory usage: heap allocate data

Imagine you have class A and two objs. of type A.

HEAP ALLOCATED DATA

```
A Obj1 = new A(5);
A Obj2 = null;
:
```

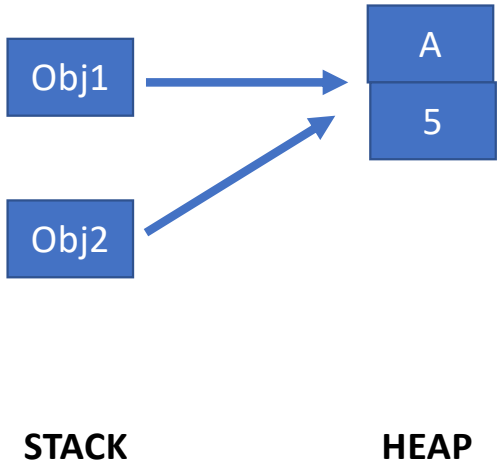
No reference to elements of heap



Memory usagae: heap allocate data

HEAP ALLOCATED DATA ARE USUALLY COPIED BY REFERENCE (AKA SHALLOW COPY)

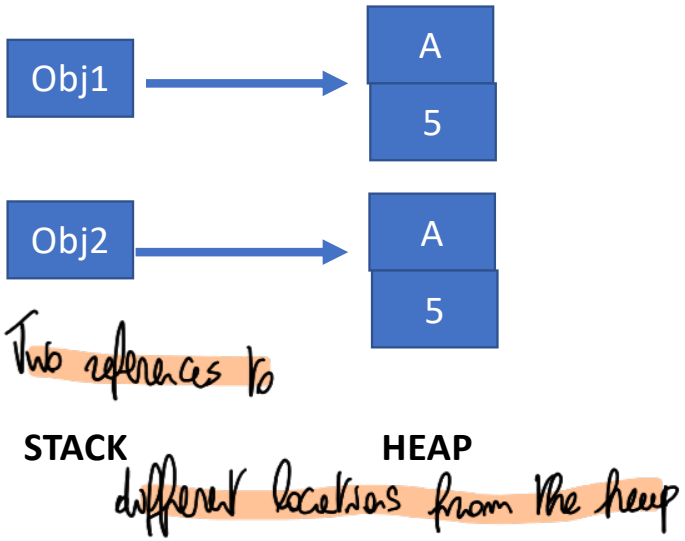
```
A Obj1 = new A(5);
A Obj2 = null;
:
Obj2 = Obj1;
```



Memory usage: heap allocate data

HEAP ALLOCATED DATA: DEEP COPY (via cloning)

```
A Obj1 = new A(5);
A Obj2 = null;
:
Obj2 = Obj1.clone();
```



What happens in RUST?

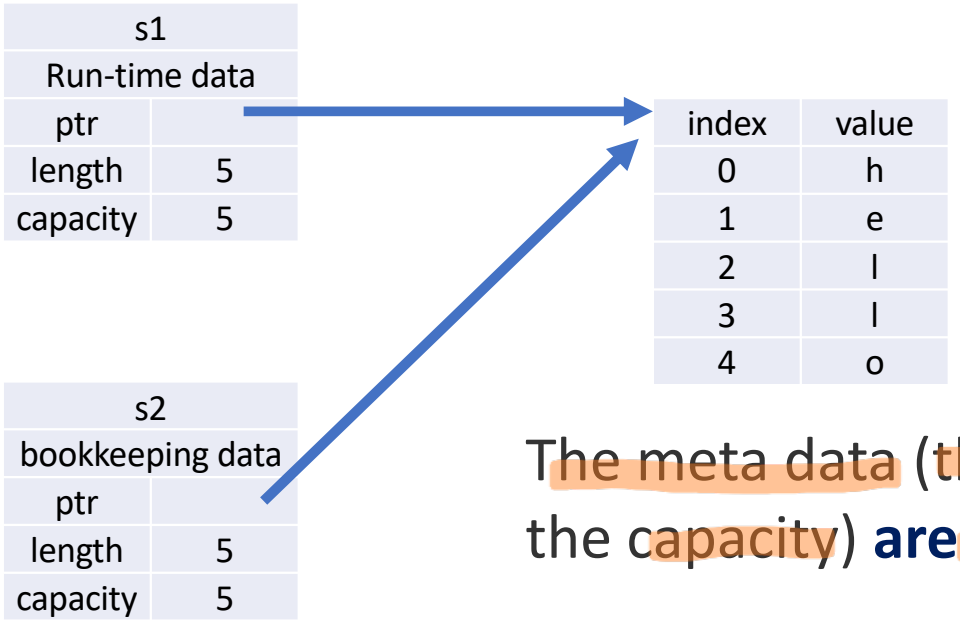
Run time representation

s1	
Run-time data	
ptr	
length	5
capacity	5

index	value
0	h
1	e
2	l
3	l
4	o

S1 and S2 were strings.
S1 in stack has a bit of metadata.
ptr is heap, length and capacity.
Capacity might be < length if you have space.

Run time representation

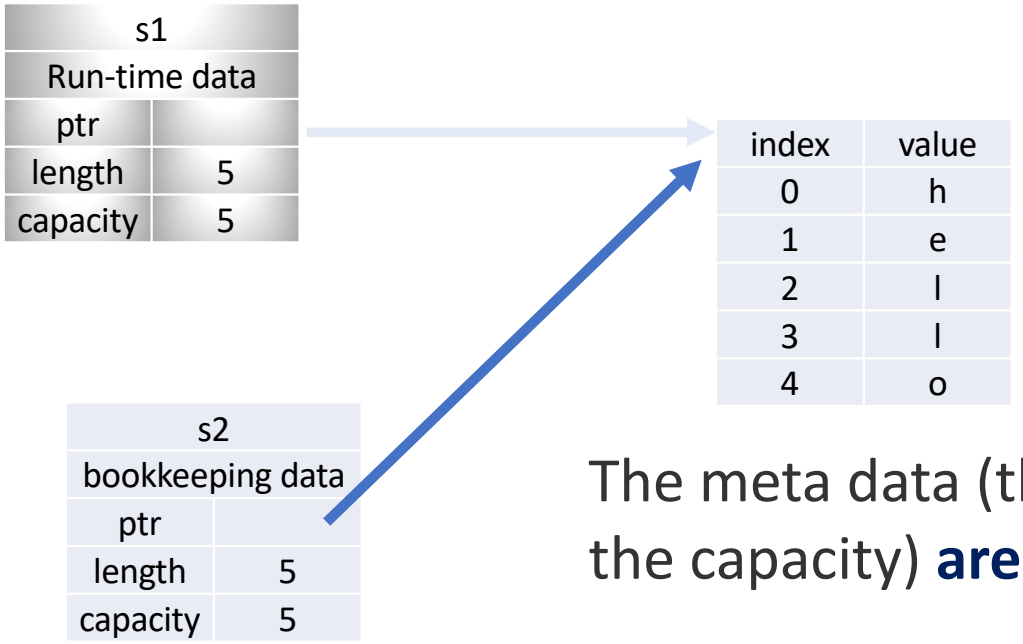


If you do $s_2 = s_1$ you do deep copy of metadata (ptr, length, capacity) but data on the heap (for references) are shallow copied.

The meta data (the pointer, the length, and the capacity) **are deep copied**

The data on the heap that the pointer refers to **are shallow copied**

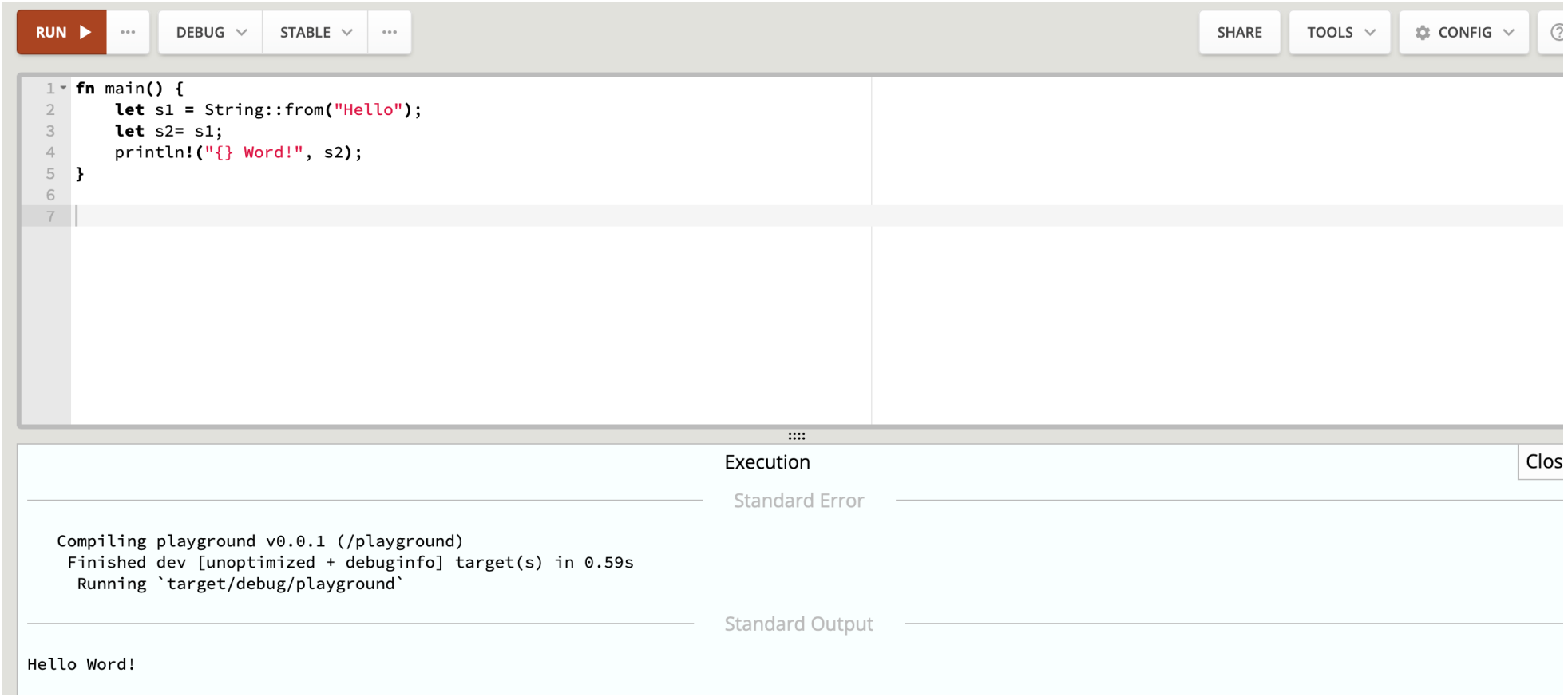
Run time representation



The meta data (the pointer, the length, and the capacity) **are deep copied**

The data on the heap **are moved from s1 to s2**

Ownership is transferred


RUN
DEBUG
STABLE
SHARE
TOOLS
CONFIG
1 fn main() {
2 let s1 = String::from("Hello");
3 let s2= s1;
4 println!("{} Word!", s2);
5 }
6
7
Execution
Clos
Standard Error
Compiling playground v0.0.1 (/playground)
Finished dev [unoptimized + debuginfo] target(s) in 0.59s
Running `target/debug/playground`
Standard Output
Hello Word!

BORROWING

Our old friend: assignment

X = Y

Our old friend: assignment (statically)

The type of X is τ **X = Y** **Y has type τ**

$$\frac{\Gamma \vdash X : \tau, \Gamma \vdash Y : \tau}{\Gamma \vdash X = Y : unit}$$

To say that $X := Y$ as assignment, this works if types are compatible

↑ result is unit: state transformation, it does not have any value

Our old friend: assignment (statically)

The type of X is τ **X = Y** **Y has type $\tau_1 < \tau$**

$$\frac{\Gamma \vdash X : \tau, \Gamma \vdash Y : \tau_1 \; \tau_1 < \tau}{\Gamma \vdash X = Y : unit}$$

This also works if τ_1 is a subtype of τ.

Our old friend: assignment (statically)

X = Y

$$\frac{\Gamma \vdash X : \tau, \quad \Gamma \vdash y : \tau_1 \quad \tau_1 < \tau}{\Gamma \vdash X = Y : unit}$$

Flow from Y (β) to X (α)

We can devise what assignment has a flow

$$\frac{flow(Y) = \beta. \qquad \text{flow}(X) = \alpha}{\beta \leq \alpha}$$

We represent statically a flow from y to x.

Security level included: standard type system + security level

Our old friend: assignment (statically)

X = op(X)

X is mutable

$$\frac{\Gamma \vdash X : \tau, X : (R, W) \quad \Gamma \vdash op(X) : \tau}{\Gamma \vdash X = op(X) : unit}$$

X is immutable

$$\frac{\Gamma \vdash X : \tau, \quad X : \{R\} \quad \Gamma \vdash op(X) : \tau}{\Gamma \vdash X = Y : error}$$

Rust assignment (statically)

Context: map from PVars to types (whatever type system). This is enough for type checking

$\Gamma: PVar \rightarrow Types$ — Γ = typing context mapping variables to types (e.g., String, i32, etc.)

Manage notion of ownership with this mapping

$\Sigma: PVar \rightarrow Status$ — Σ = *resource context* mapping variables to ownership status (e.g. Owned, Moved, etc.)

Rust assignment (statically)

$\Gamma: PVar \rightarrow Types$ — Γ = typing context mapping variables to types (e.g., String, i32, etc.)

$\Sigma: PVar \rightarrow Status$ — Σ = *resource context* mapping variables to ownership status (e.g. Owned, Moved, etc.)

$$\frac{\Gamma(x) = \tau_1 \quad \Gamma(y) = \tau \quad \tau_1 < \tau \quad \Sigma(x) = Owned}{\Gamma \vdash y = x \quad \Sigma(x) = Moved \quad \Sigma(y) = Owned,}$$

① Types need to be compatible and x must be owner

② to be able to perform assignment

is y=x well typed? ① If x is owner yes, and x is not owner anymore

② Data structure associated to x is owned by x.

Rust assignment (statically)

$\Gamma: PVar \rightarrow Types$ — Γ = typing context mapping variables to types (e.g., String, i32, etc.)

$\Sigma: PVar \rightarrow Status$ — Σ = *resource context* mapping variables to ownership status (e.g. Owned, Moved, etc.)

$$\frac{\Gamma(x) = \tau_1 \quad \Gamma(y) = \tau \quad \tau_1 < \tau \quad \Sigma(x) = Owned}{\Gamma \vdash y = x \quad \Sigma(x) = Moved \quad \Sigma(y) = Owned,}$$

"Oxide: The Essence of Rust" by Matthew Flatt et al..

Move Safety Invariant

A **move-safe program** satisfies the property;

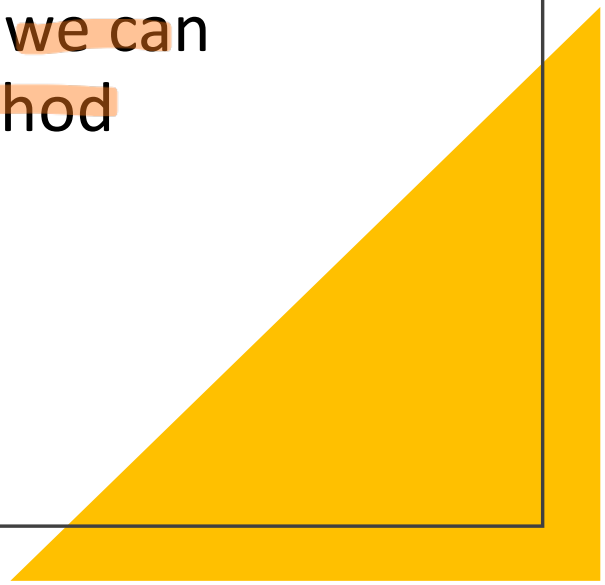
For any variable v, once Σ(v) = Moved, there are no future dereferences or usages of v.

This ensures:

- No double-free.
- No use-after-move.

Clone

If we do want to **deeply copy** the heap data, not just the stack data, we can use a common method called **clone**.



Deep copy retails ownership

Make clones (copies) to avoid ownership loss

```
let x = String::from("hello");
let y = x.clone(); //x no longer moved
println!("{}, world!", y); //ok
println!("{}, world!", x); //ok
```

RUN ▶ ··· DEBUG STABLE ···

```
1  fn main() {
2      let s1 = String::from("hello");
3      let s2 = s1.clone();
4
5      println!("s1 = {}, s2 = {}", s1, s2);
6  }
```

Execution

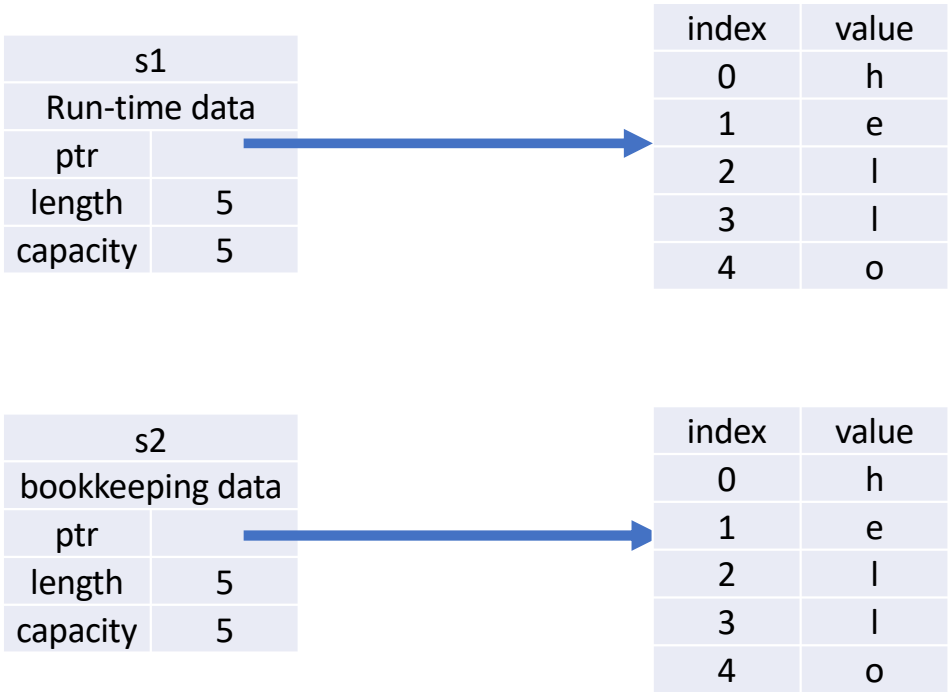
Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.42s
     Running `target/debug/playground`
```

Standard Output

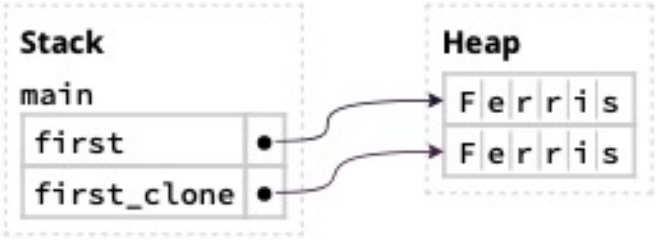
```
s1 = hello, s2 = hello
```

Run time representation (copy -- clone)

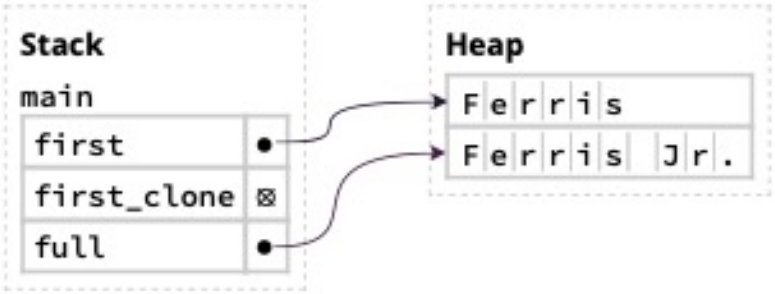


```rust
fn main() {
    let first = String::from("Ferris");
    let first_clone = first.clone(); L1
    let full = add_suffix(first_clone); L2
    println!("{full}, originally {first}");
}

fn add_suffix(mut name: String) -> String {
    name.push_str(" Jr.");
    name
}
```

L1



L2



Parameter passing

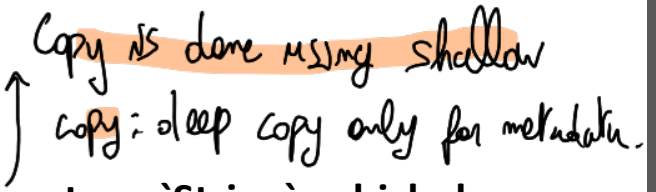
- Ownership of DS is moved to the formal parameter

The parameter passing is an assigment of the value of the actual parameter to the formal parameter

The move and copy concepts apply as well!!

```
fn main() {
  let s1 = String::from("hello");
  println!("{}",s1);
  let s2 = id(s1); //s1 moved to id param
                   //id's result moved to s2
  println!("{}",s2);
  println!("{}",s1); //fails
}
fn id(s:String) -> String {
  s
}
```

```
fn main() {
  let s1 = String::from("hello");
  println!("{}",s1);
  let s2 = id(s1); //s1 moved to id param
                   //id's result moved to s2
  println!("{}",s2);
  println!("{}",s1); //fails
}
fn id(s:String) -> String {
  s
}
```



let s1 = String::from("hello");
| -- move occurs because `s1` has type `String`, which does not implement the `Copy` trait 3
| let s2 = id(s1);
| -- value moved here ...
| println!("{}",s1);
^^ value borrowed here after move

Pass by moving

Extension of call by value

Call by moving =

Call by value +

Transfer of the ownership

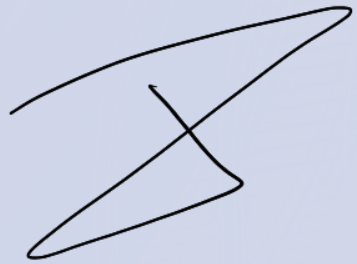
from the actual argument to the formal argument

We learned (hopefully)

Ownership is a discipline of heap management

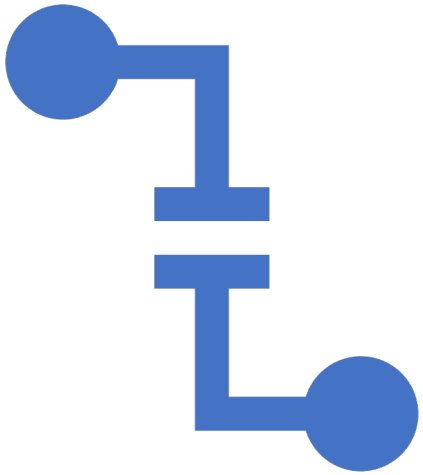
- All heap data must be owned by exactly one variable.
- Rust deallocates heap data once its owner goes out of scope.
- Ownership can be transferred by moves, which happen on assignments and function calls.
- Heap data can only be accessed through its current owner, not a previous owner.

These notions should help you with interpreting Rust's error messages. They should also help you design more Rustic APIs.



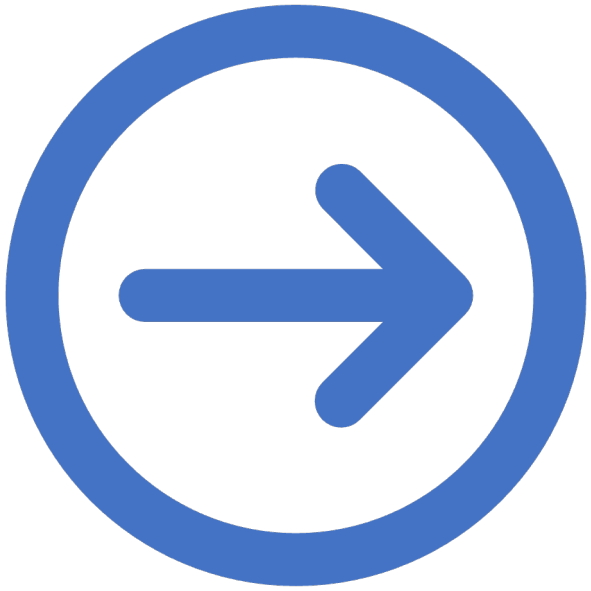
You supply a routine some data and supply data back,

8



Parameter passing (again)

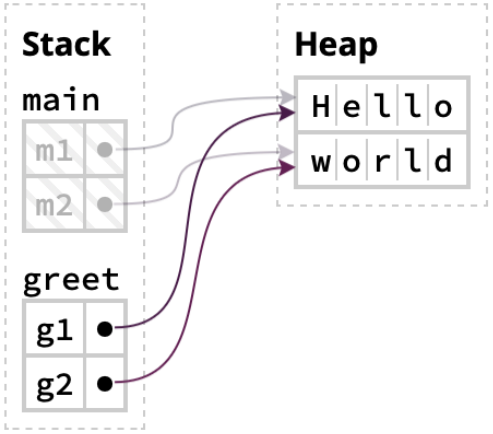
- Parameter passing allows the values of local variables within the caller to be shared and used within multiple the callee without the need to create or use global variables.
- One fundamental difference among programming languages concerns the mechanisms adopted for parameter passing
  - C, C++ and Java only support call by value.



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

```rust
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world");
    greet(m1, m2); L2  You would like to use m1 and m2 again
    let s = format!("{} {}", m1, m2); L3  // Error: m1 and m2 are moved
}

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

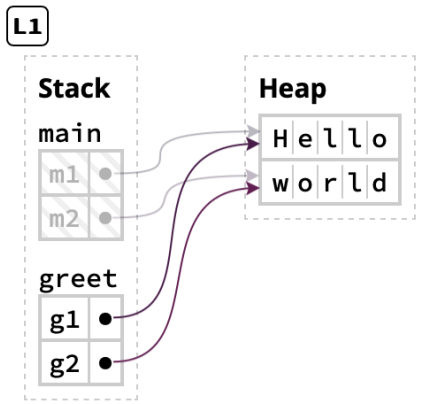
L1



L2

L3 undefined behavior: pointer used after its pointee is freed

```rust
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.

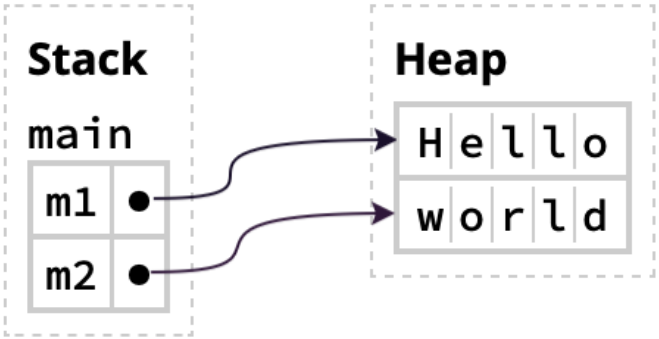
An alternative approach could return ownership of the parameter.

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

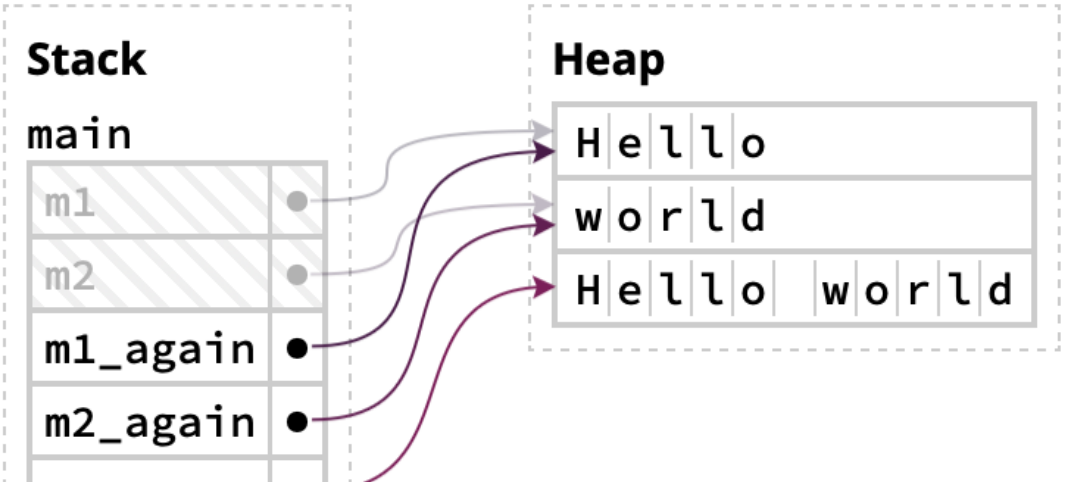
```rust
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    let (m1_again, m2_again) = greet(m1, m2);
    let s = format!("{} {}", m1_again, m2_again); L2
}

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

- You could work like this

L1



L2



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.

You want a notion of movement by using references, avoiding values.

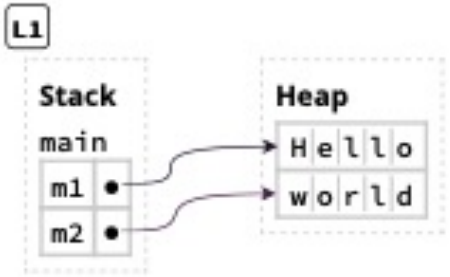
Rust uses concept of Borrowing to solve problem.

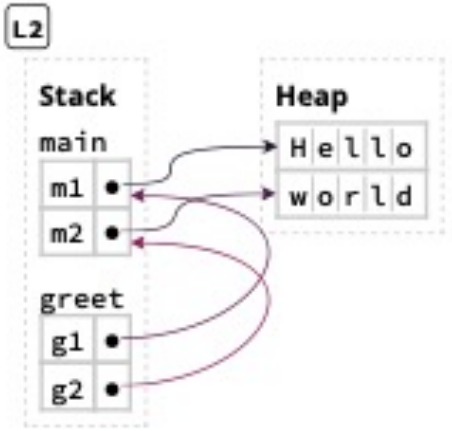
Rust approach: references are non-owning pointers

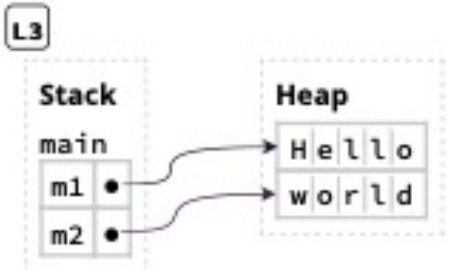
```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world");
    greet(&m1, &m2); // note the type of parameter passing
    let s = format!("{} {}", m1, m2);
}

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

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

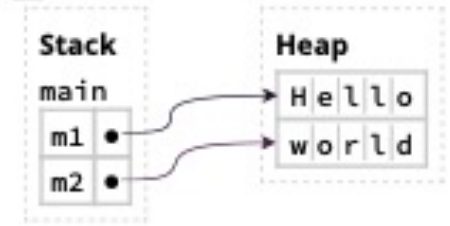
The expression &m1 creates a reference to (or "**borrow**") m1. The type of the parameter g1 is changed to &String, meaning "a reference to a String".

```
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);
}
```

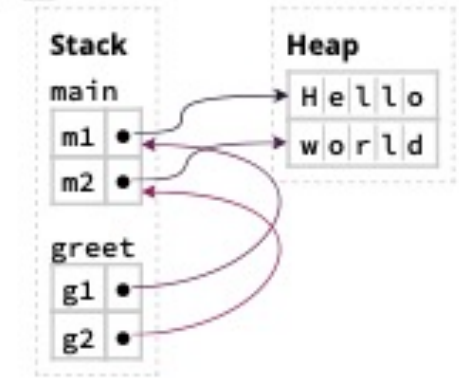
References Are Non-Owning Pointers

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".

L1

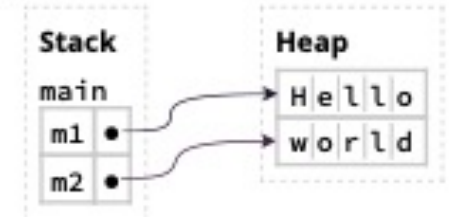


L2



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

L3



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

Borrowing

Instead of transferring ownership, we can borrow data.

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

Ownership vs Borrowing

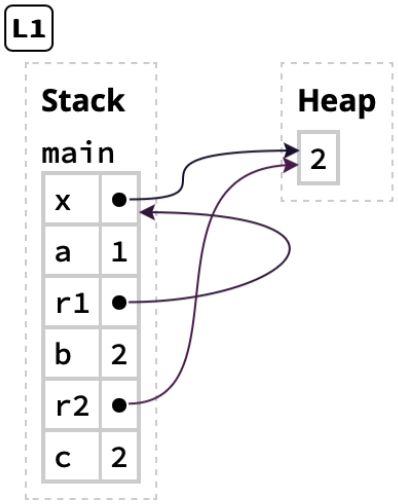
- Is borrowing consistent with Rust *Moved Heap Data Principle*?
- The idea is that references are **non-owning pointers**, because they do not own the data they point to.
  - In. our example variable g1 did not own "Hello", Rust did not deallocate "Hello" on behalf of g1.

Parameter passing: borrowing

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

Remark on (de)references

```
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;        // two dereferences get us to the heap value

let r2: &i32 = &*x;       // r2 points to the heap value directly
let c: i32 = *r2; L1      // so only one dereference is needed to read it
```



Inplicit/explicit (de)refrences

```
let x: Box<i32> = Box::new(-1);
let x_abs1 = i32::abs(*x); // explicit dereference
let x_abs2 = x.abs();      // implicit dereference
assert_eq!(x_abs1, x_abs2);

let r: &Box<i32> = &x;
let r_abs1 = i32::abs(**r); // explicit dereference (twice)
let r_abs2 = r.abs();       // implicit dereference (twice)
assert_eq!(r_abs1, r_abs2);

let s = String::from("Hello");
let s_len1 = str::len(&s); // explicit reference
let s_len2 = s.len();      // implicit reference
assert_eq!(s_len1, s_len2);
```

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");
}
```

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`
```

Standard Output

```
The value of s is 'hello, world'
```

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
```

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);
}
```

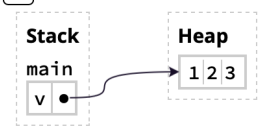
Constraint on mutable references: we can have only one mutable reference to a particular piece of data in a particular scope

Mutable data and aliasing

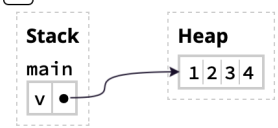
- References are a powerful programming abstractions … but they enable aliasing.
- Aliasing is accessing the same data through different variables.
- On its own, aliasing is harmless. But combined with mutation, we have a recipe for disaster.

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

L1

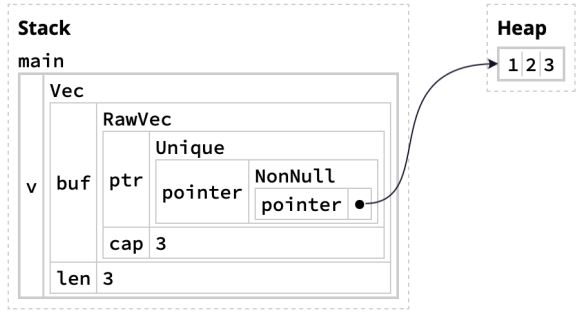


L2




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

L1



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

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
```

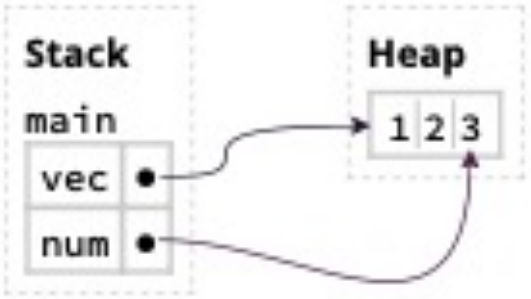
Execution

Standard Error

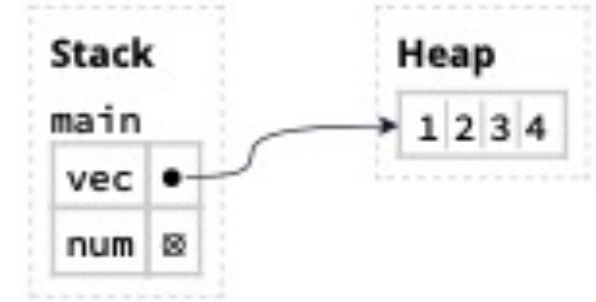
```
   Compiling playground v0.0.1 (/playground)
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:1
  |
3 | let num: &i32 = &vec[2];
  |                  --- immutable borrow occurs here
4 | vec.push(4);
  | ^^^^^^^^^^ mutable borrow occurs here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

```rust
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
```

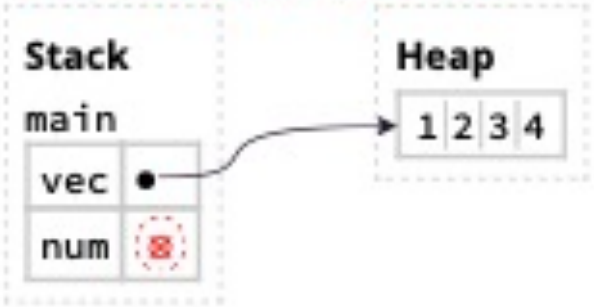
L1



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.

Pointer safety: borrow checker

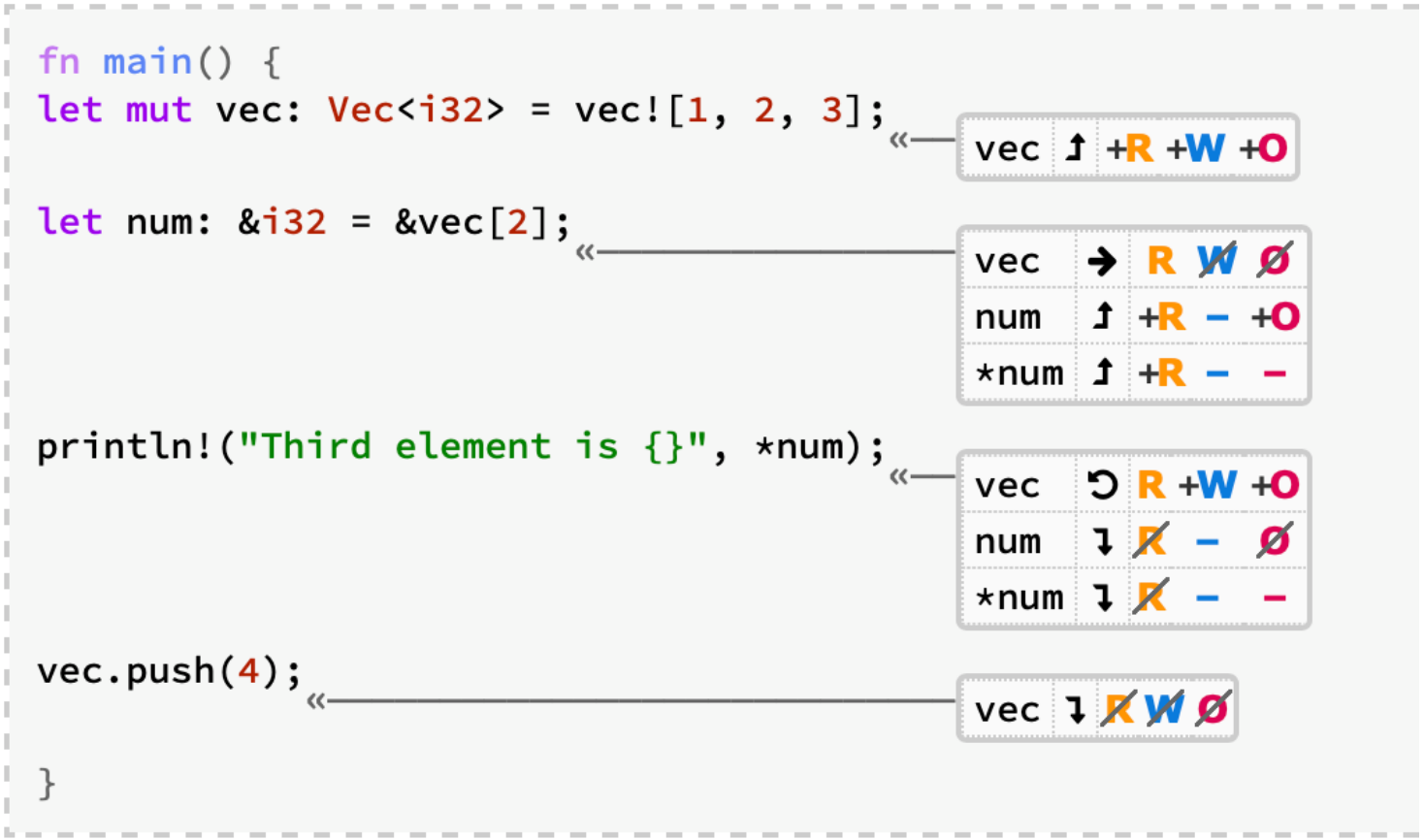
- By design, references are meant to temporarily create aliases.
- Rust ensures the safety of references through the **borrow checker.**

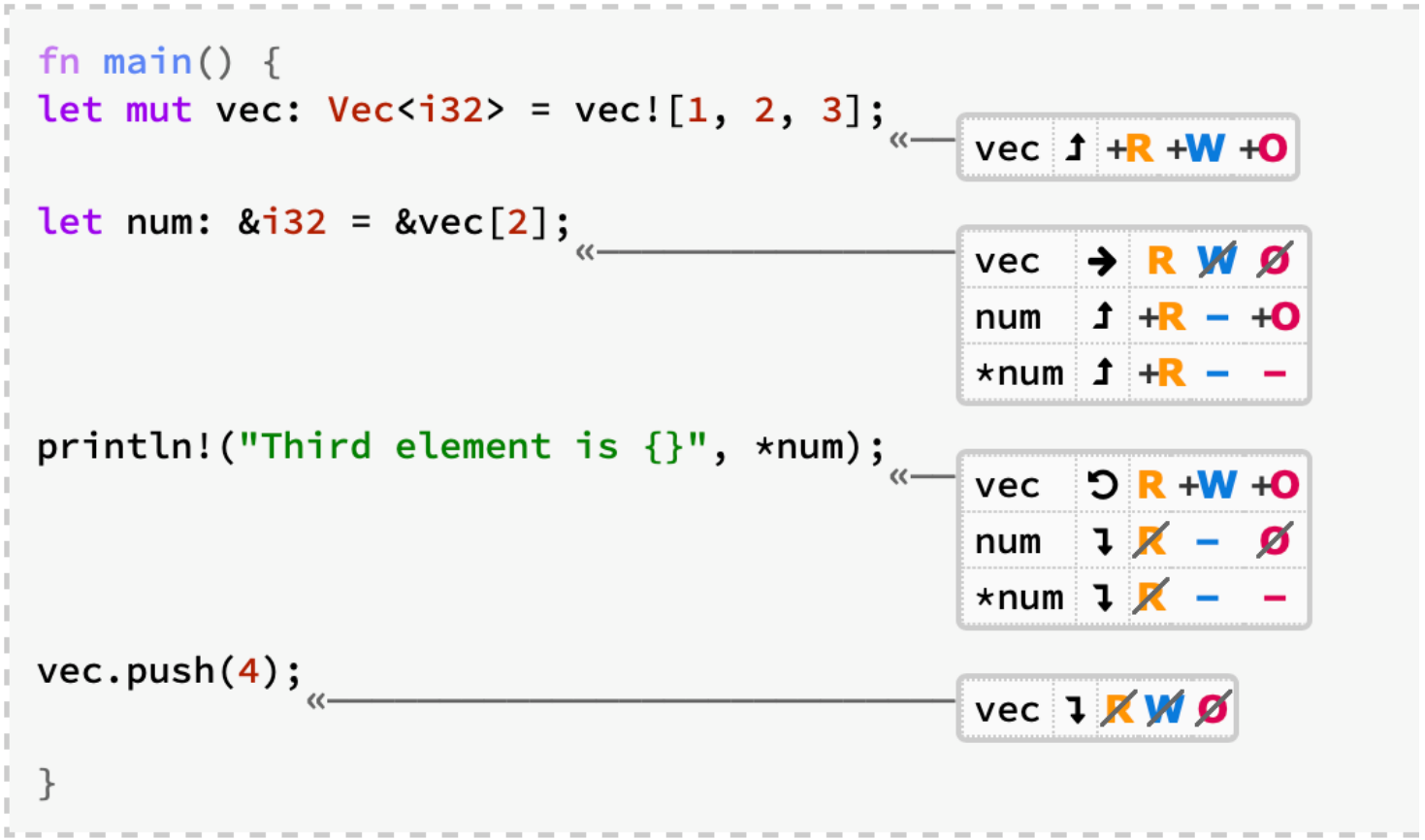
Static information

- 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 don't exist at runtime
  - They describe how the compiler ”operates" over programs before the program is executed.
-

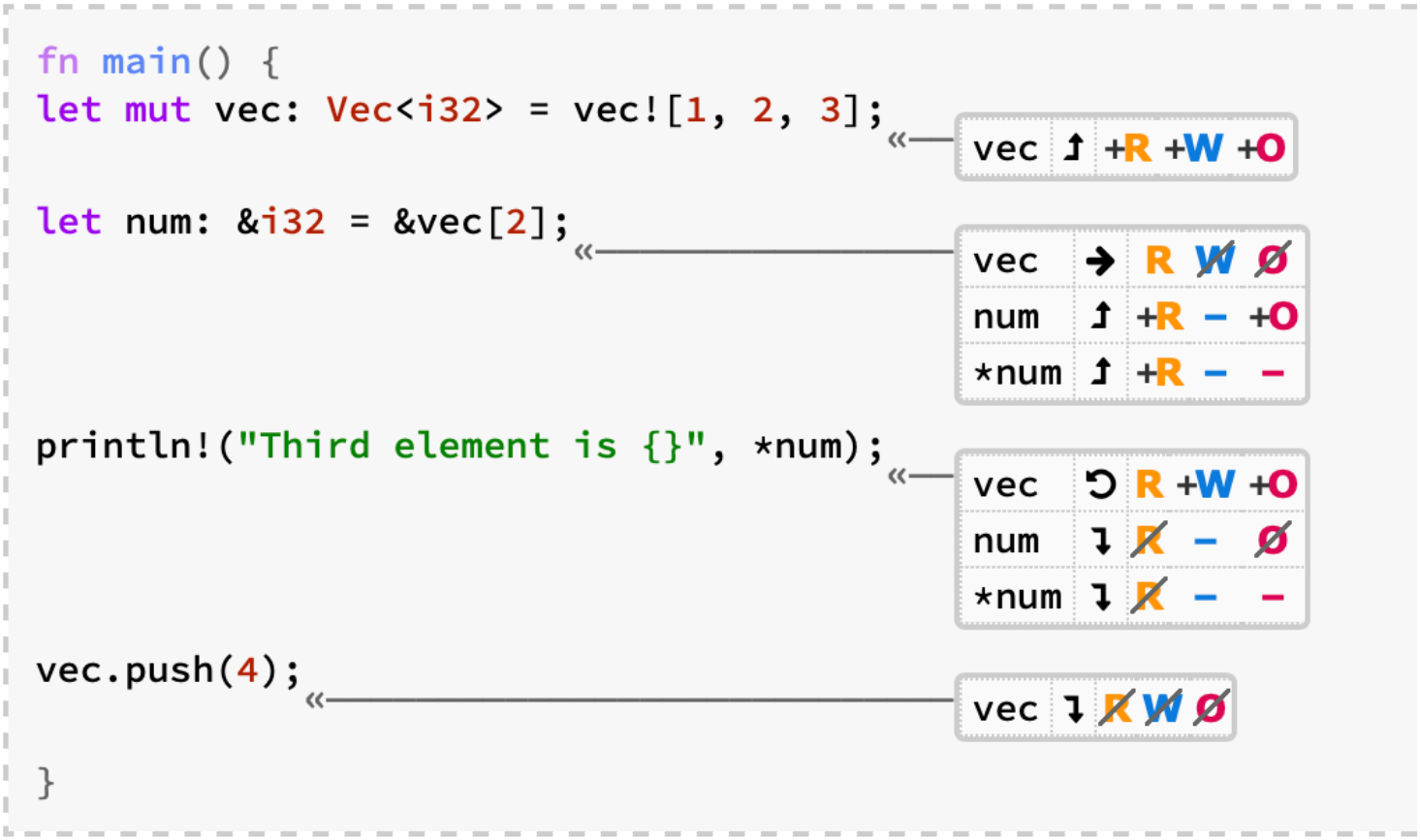
Mutable vs Immutable revisited

- 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.**

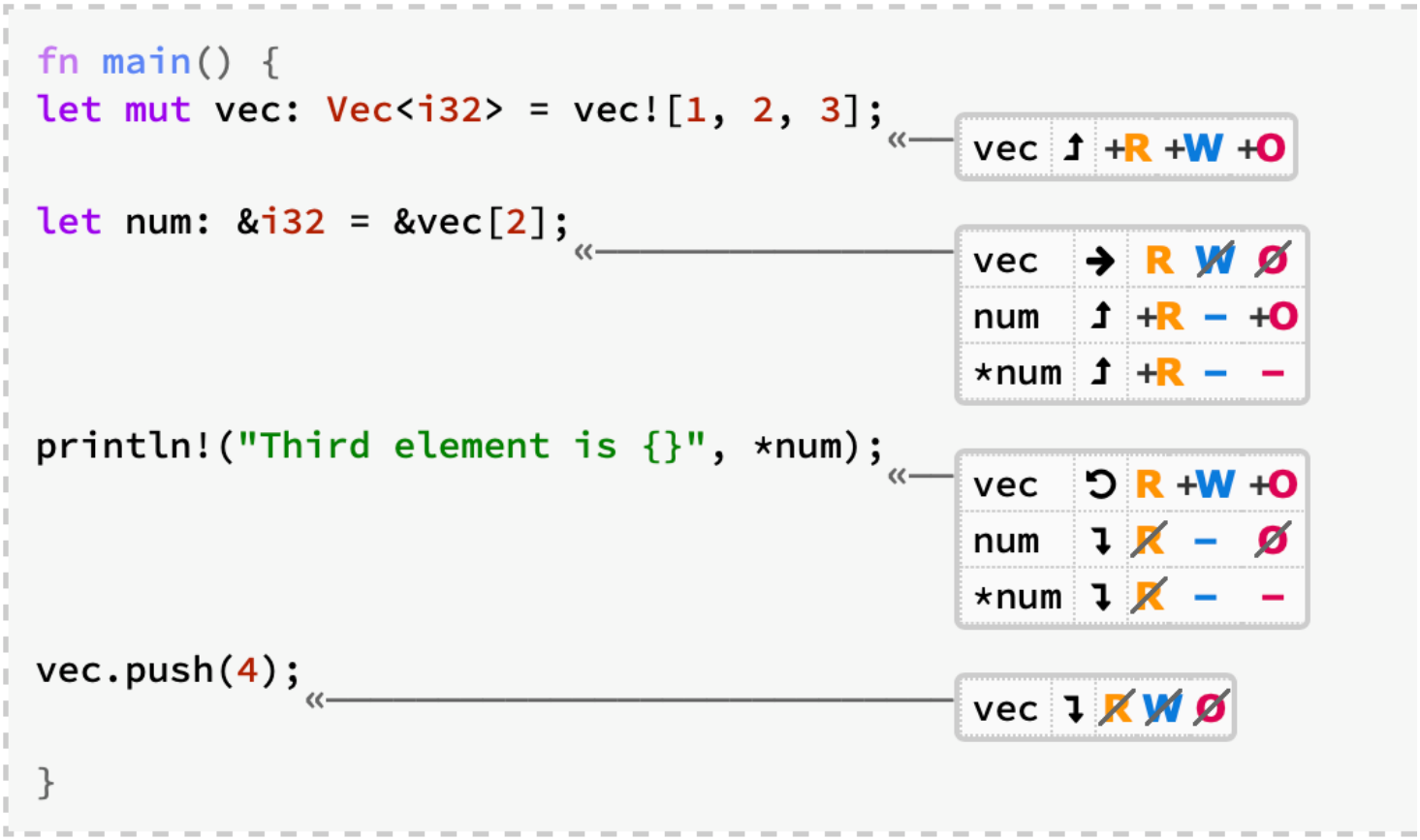

fn main() {
let mut vec: Vec<i32> = vec![1, 2, 3];
vec +R +W +O
let num: &i32 = &vec[2];
vec R W Ø
num +R – +O
*num +R – –
println!("Third element is {}", *num);
vec R +W +O
num R – Ø
*num R – –
vec.push(4);
vec R W Ø
}



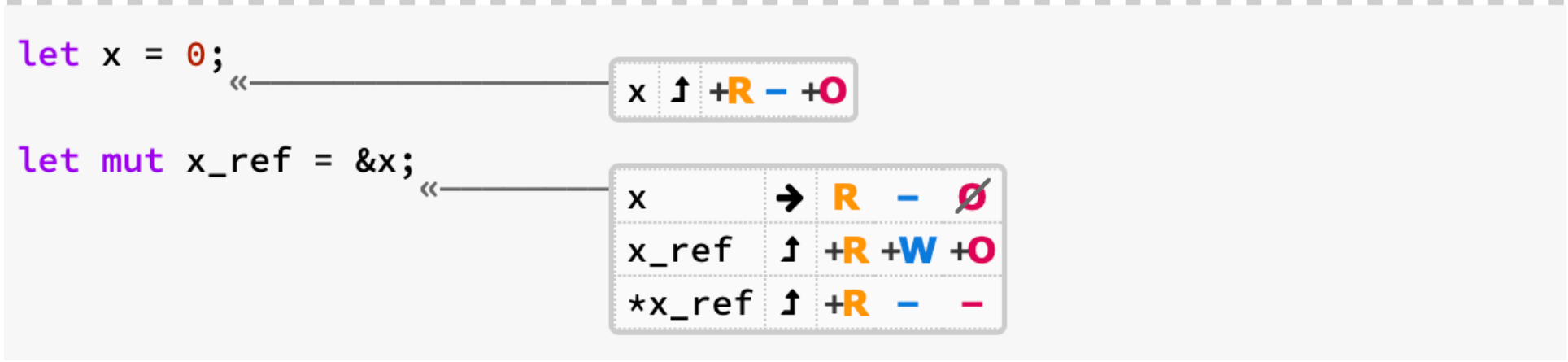
The execution of let mut vec = (...), initializes variable vec (graphically indicated by the return arrow). It gains +**R**+**W**+**O** attributes (the plus sign indicates gain).



The execution of num = &vec[2], makes the data in vec **borrowed** by num (indicated by the righrarrow).
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.



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

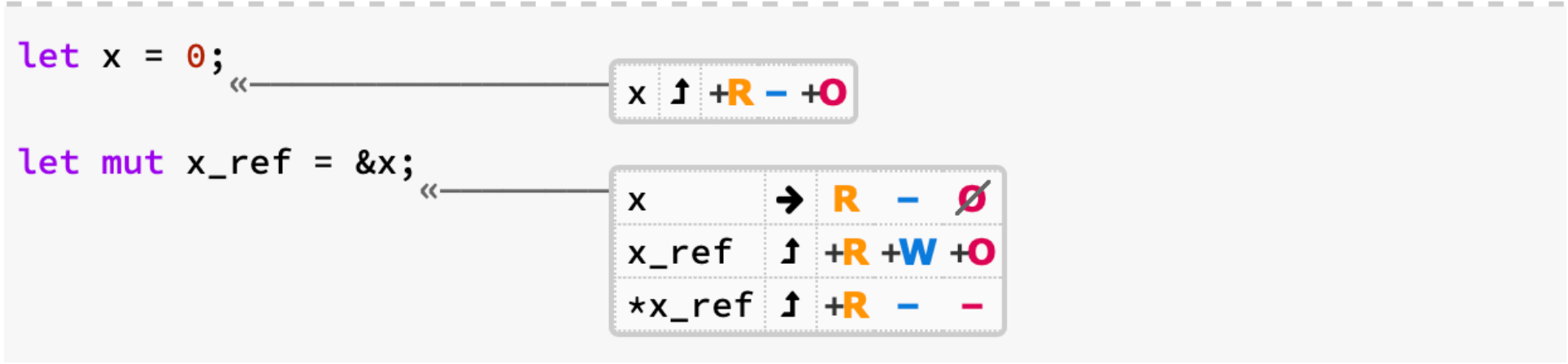


x_ref has the **W** attribute, while ***x_ref** does not.

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**).

```
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;
```



x_ref has the **W** attribute, while ***x_ref** does not.

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**, **(*((*a)[0].1))**

Borrow Checking (on path attributes)

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];

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

vec R W .push(4);
println!("Third element is {}", *num);
}
```

vec	↥	+R +W +O

vec	→	R ~~W~~ ~~Ø~~
num	↥	+R – +O
*num	↥	+R – –

Back to our example: path checking

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

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

vec R W .push(4);
println!("Third element is {}", *num);
}
```

vec	↥	+R	+W	+O

vec	→	R	~~W~~	~~Ø~~
num	↥	+R	–	+O
*num	↥	+R	–	–

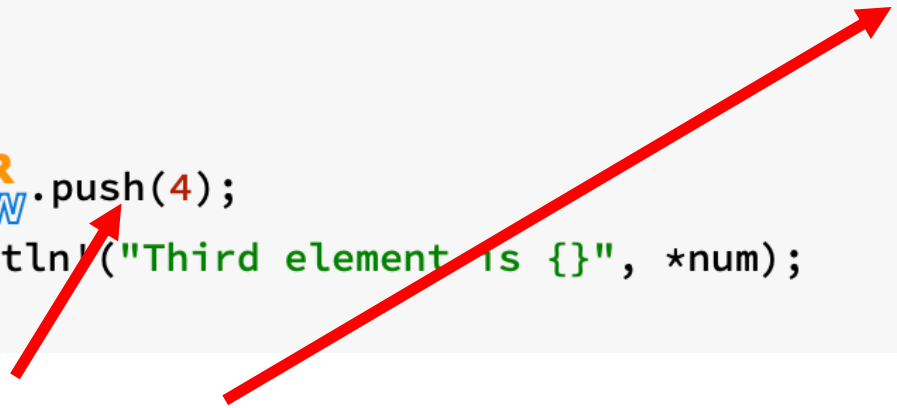
the borrow **&vec[2]** requires that **vec** is readable.

Back to our example: path checking

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

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



vec R W .push(4);
println!("Third element is {}", *num);
}
```

vec	↥	+R +W +O

vec	→	R ~~W~~ ~~O~~
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.

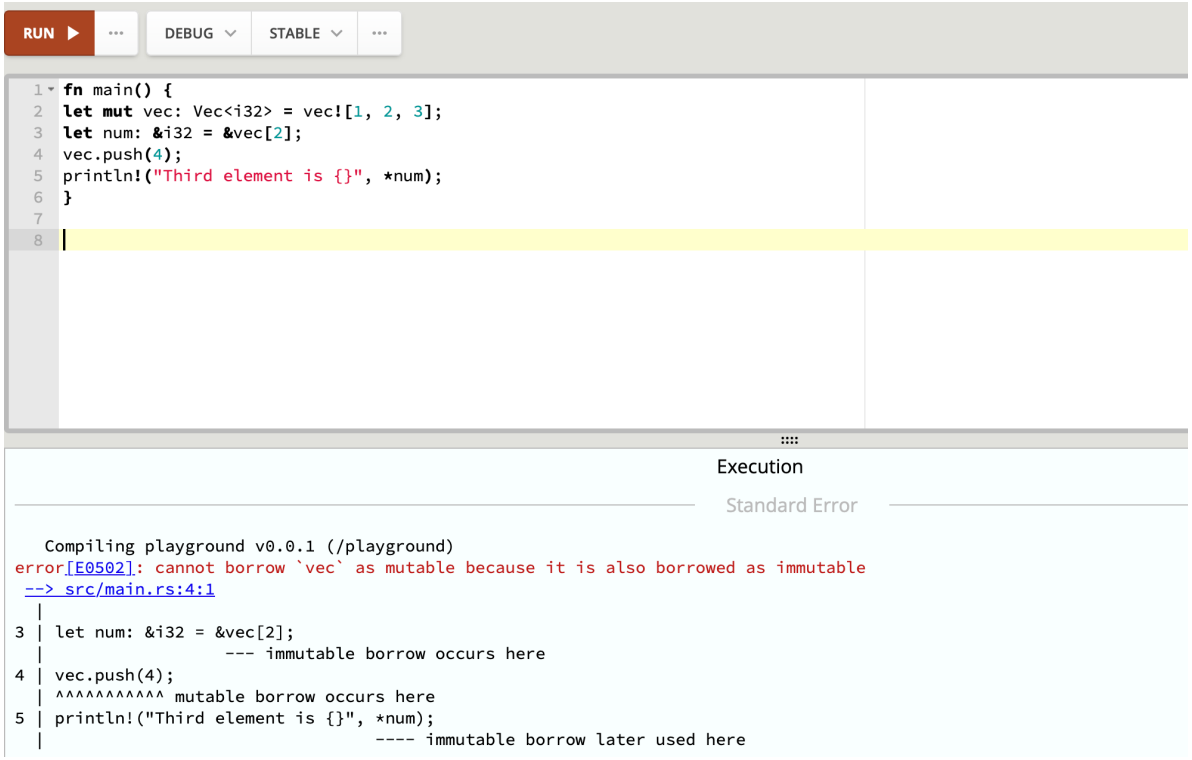
RUN ▶ ··· DEBUG STABLE ···

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

```

Execution

Standard Error

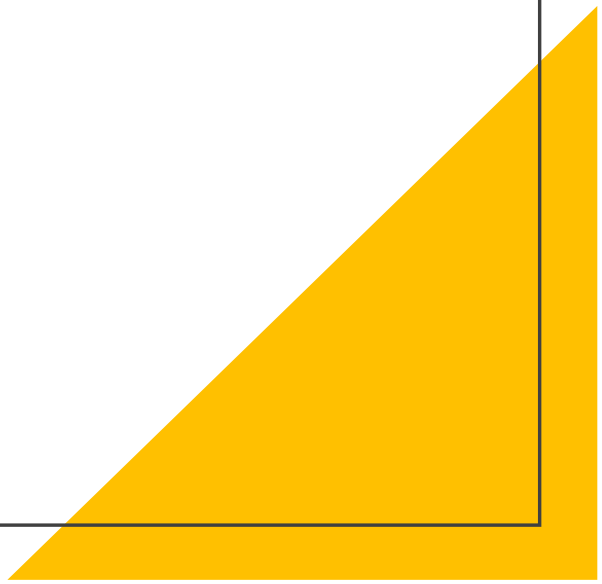
```
   Compiling playground v0.0.1 (/playground)
error[E0502]: cannot borrow `vec` as mutable because it is also borrowed as immutable
 --> src/main.rs:4:1
  |
3 | let num: &i32 = &vec[2];
  |                  --- immutable borrow occurs here
4 | vec.push(4);
  | ^^^^^^^^^^^ mutable borrow occurs here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



The 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

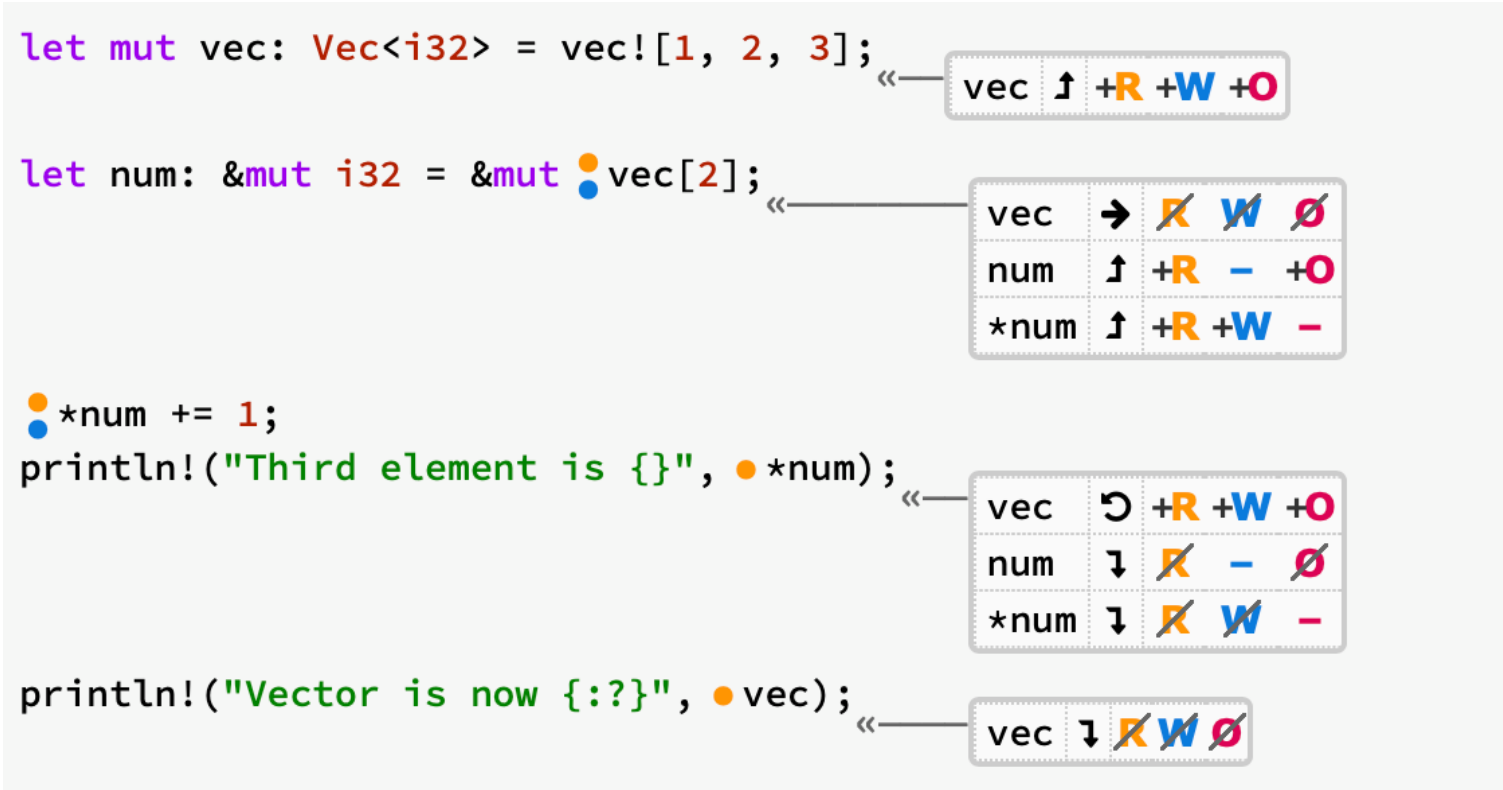


However ….

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



```rust
fn main() {
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);
}
```

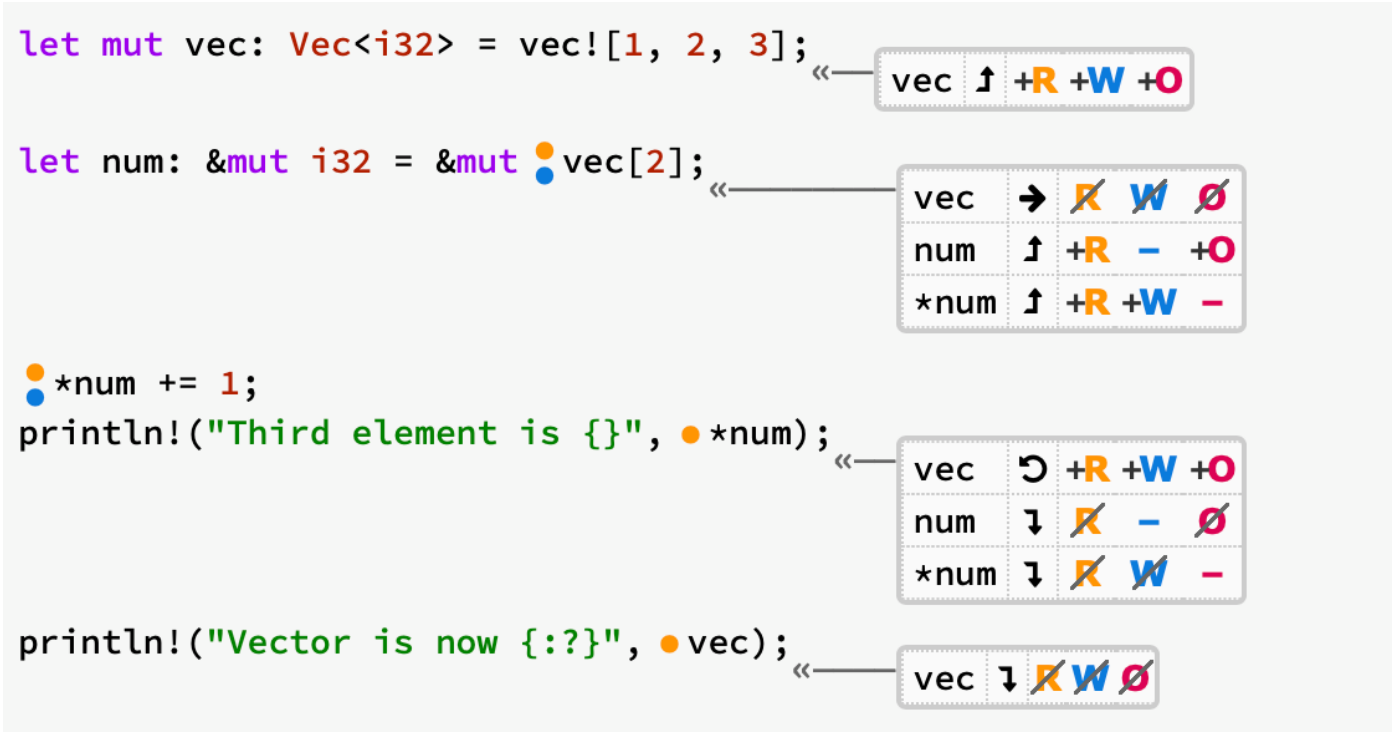
Execution

Standard Error

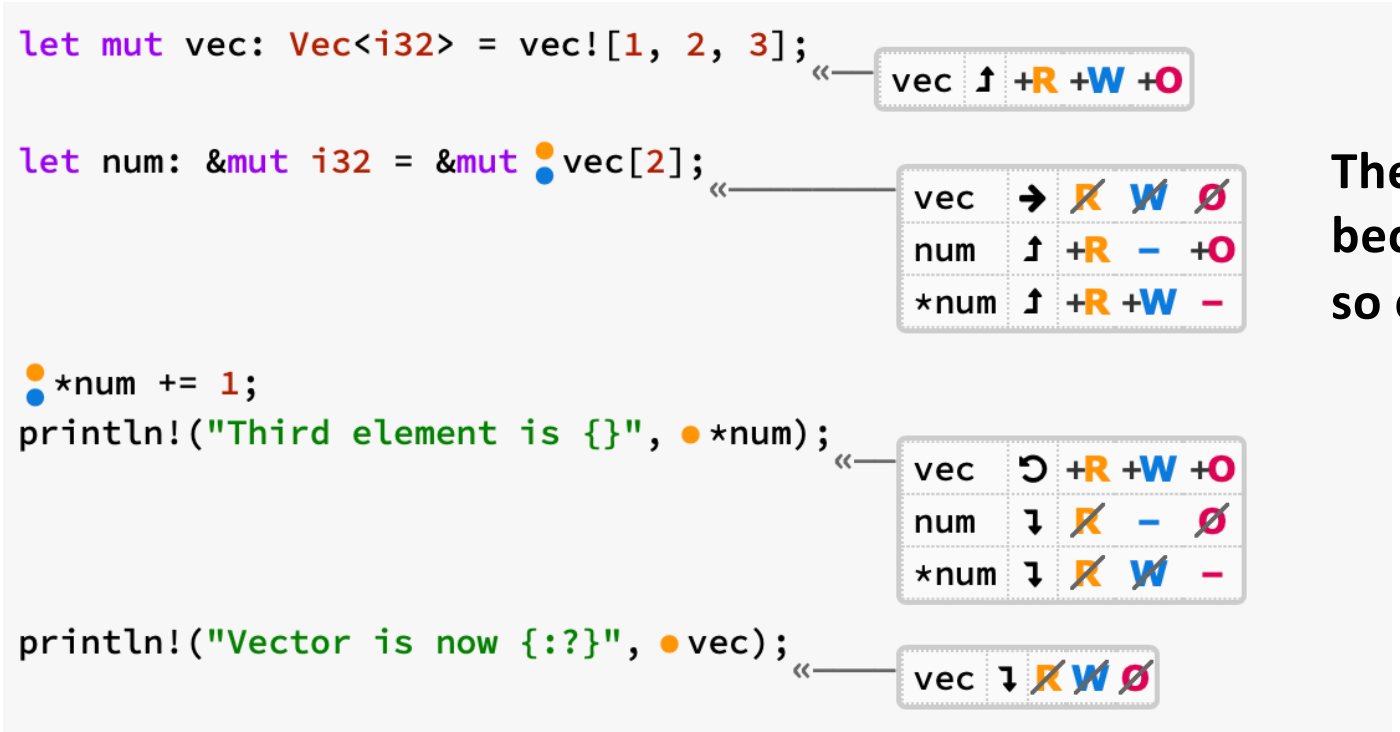
```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 0.59s
     Running `target/debug/playground`
```

Standard Output

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

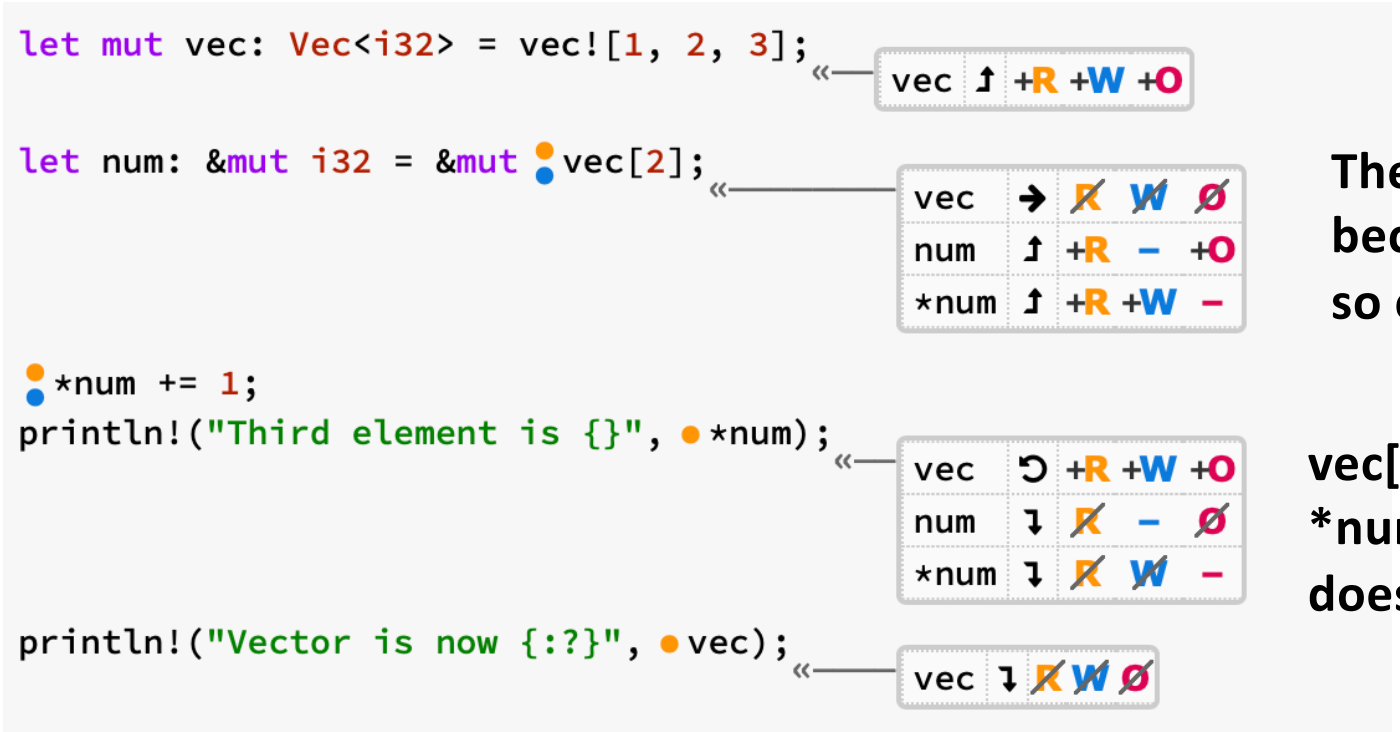


Why mutable references are *safe?*. Mutable references allow mutation but prevent aliasing.



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

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



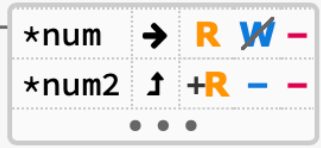
The borrowed path vec becomes temporarily unusable, so effectively not an alias.

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.

Downgrading mutable references

```
let mut v: Vec<i32> = vec![1, 2, 3];
let num: &mut i32 = &mut v[2];
let num2: &i32 = &*num;
println!("{} {}", *num, *num2);
```



Quiz

ILLUSTRATE THE PERMISSIONS OF X

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

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	-	-

Quiz

ILLUSTRATE THE PERMISSIONS OF X

```
fn main() {
    let mut x = 1;
    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	↖	R	W	O

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

QUIZ

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

    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s;

    println!("{}, {}, and {}", r1, r2, r3);
}
```

QUIZ

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

    let r1 = &s;
    let r2 = &s;
    let r3 = &mut s;

    println!("{}, {}, and {}", r1, r2, r3);
}
```

Execution

Standard Error

```
   Compiling playground v0.0.1 (/playground)
error[E0502]: cannot borrow `s` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:14
  |
4 |     let r1 = &s;
  |              -- immutable borrow occurs here
5 |     let r2 = &s;
6 |     let r3 = &mut s;
  |              ^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}, {}, and {}", r1, r2, r3);
  |                                -- immutable borrow later used here
```

QUIZ

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

    let r1 = &s;
    let r2 = &s;
    println!("{} and {}", r1, r2);
}
    let r3 = &mut s;

    println!("{}", r3);
}
```

RUN ▶ ··· DEBUG STABLE ···

```
1  fn main() {
2      let mut s = String::from("hello");
3
4      let r1 = &s; // no problem
5      let r2 = &s; // no problem
6      println!("{} and {}", r1, r2);
7      // r1 and r2 are no longer used after this point
8
9      let r3 = &mut s; // no problem
10     println!("{}", r3);
11 }
12
13
14
```

Execution

Standard Error

```
   Compiling playground v0.0.1 (/playground)
    Finished dev [unoptimized + debuginfo] target(s) in 1.89s
     Running `target/debug/playground`
```

Standard Output

```
hello and hello
hello
```