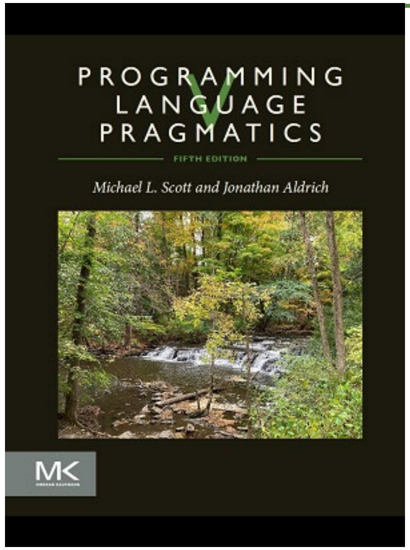
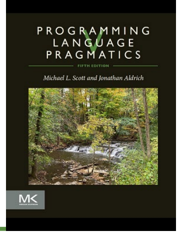
Ownership in Rust

This material is based heavily on the Rust book, as adapted by Will Crichton et al.



Programming Language Pragmatics, Fifth Edition

Michael L. Scott and Jonathan Aldrich



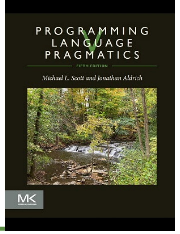
Safety in Rust

- Safety in Rust means a lack of undefined behavior
- Example of undefined behavior (from the Rust book):

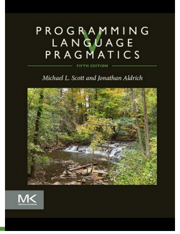
```
fn read(y: bool) {
    if y {
        println!("y is true!");
    }
}

fn main() {
    read(x); // oh no! x isn't defined!
    let x = true;
}
```

- It is undefined behavior to read a variable before it is defined
- Why is undefined behavior bad?
  - Well, it might execute just fine
  - But the program above could read garbage data, making results unpredictable
  - In general, undefined behavior can cause crashes or security vulnerabilities

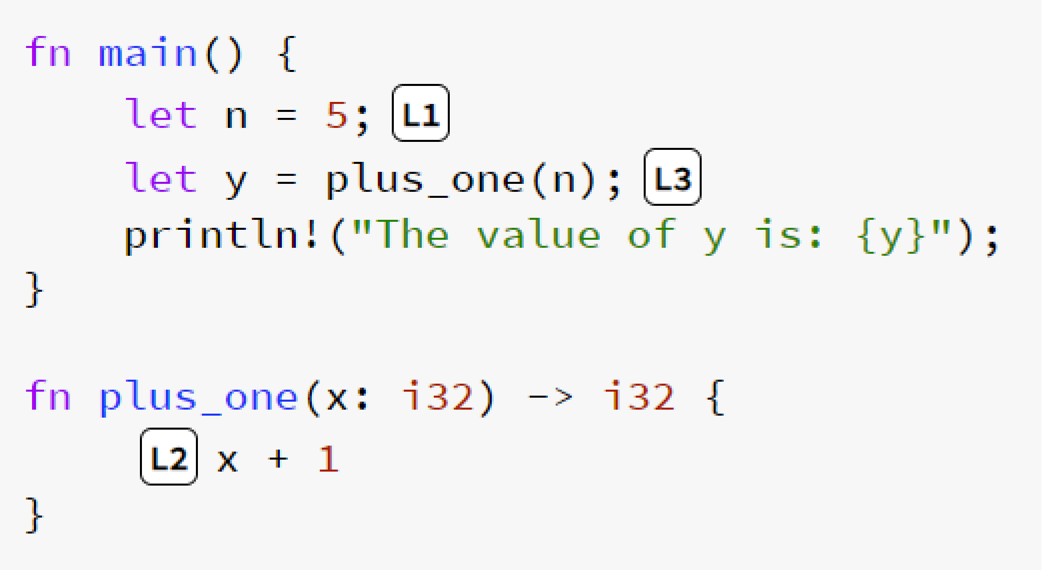
How Rust ensures safety



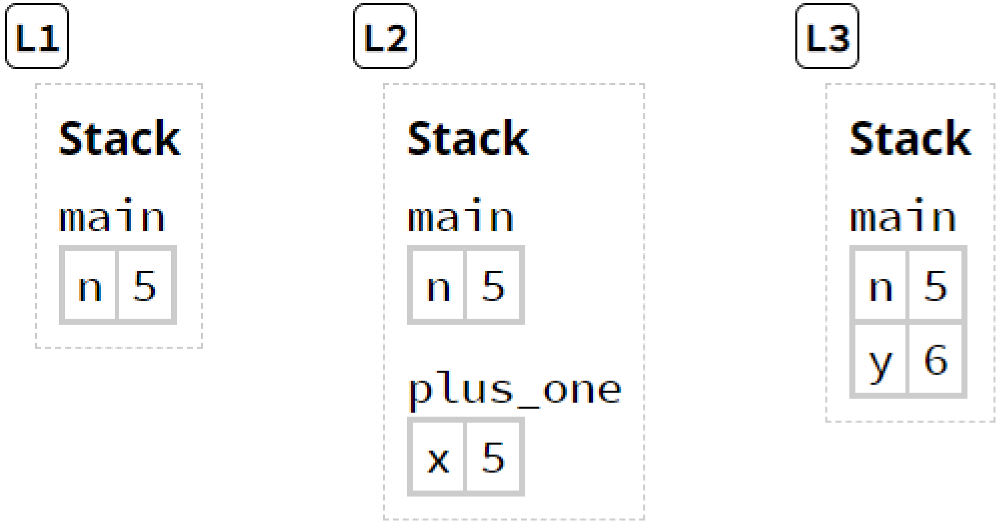
- Key goal of Rust: ensure program don’t have undefined behavior
  - Combination of static and dynamic checks
  - Check as much as possible statically
- Bugs are still possible! But certain kinds of bugs can’t happen.
- Ownership in Rust is a discipline for using memory
- Ownership prevents undefined behavior related to memory
  - Reading uninitialized memory
  - Using memory after it is freed
  - Freeing memory twice
  - Memory leaks (forgetting to free memory)



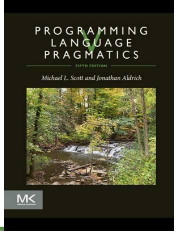
Conceptual model of variables on the stack



```
fn main() {
    let n = 5; L1
    let y = plus_one(n); L3
    println!("The value of y is: {y}");
}

fn plus_one(x: i32) -> i32 {
    L2 x + 1
}
```

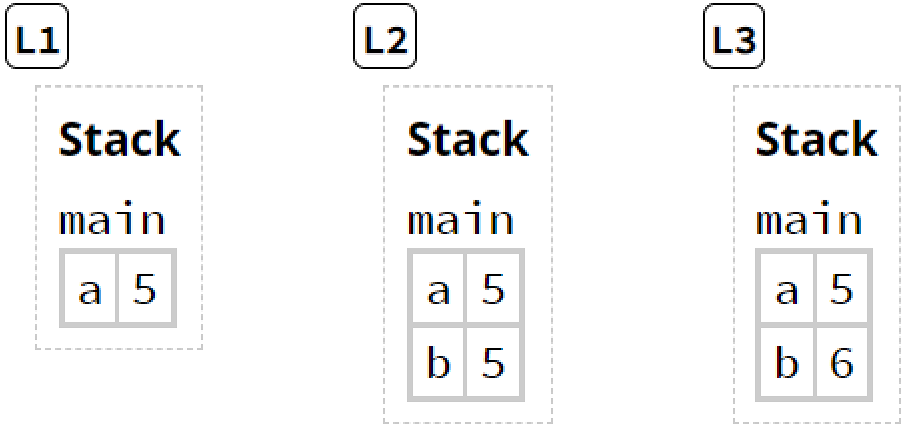


example from the Rust book (Brown version)



Assigning one variable to another copies the data

```
let a = 5; L1
let mut b = a; L2
b += 1; L3
```



L1

Stack

main

a	5

L2

Stack

main

a	5
b	5

L3

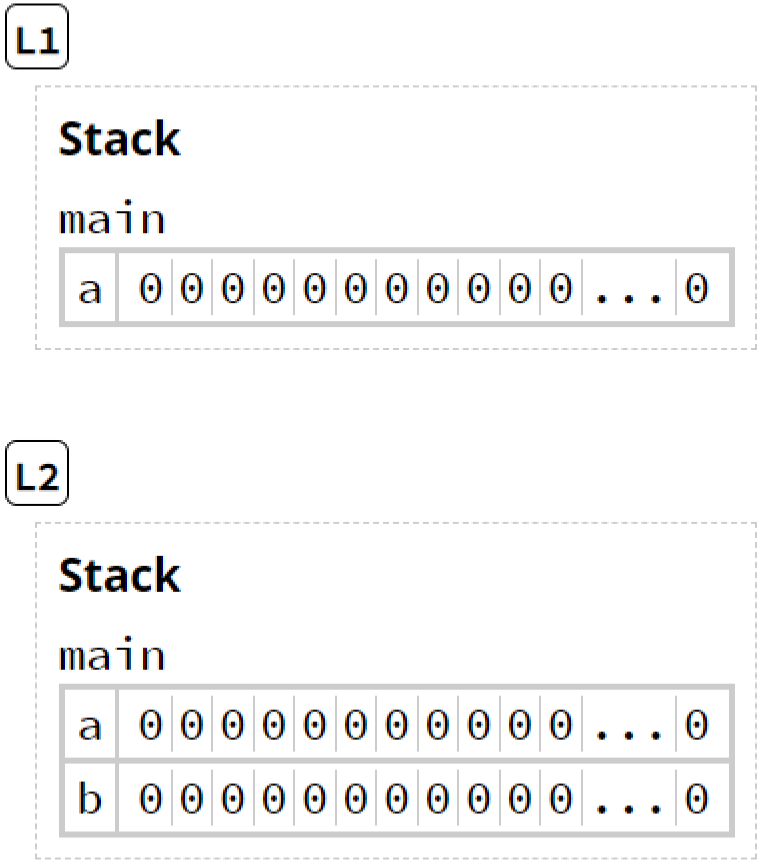
Stack

main

a	5
b	6

But this is very expensive if the data is large →

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



L1

Stack

main

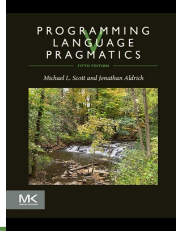
a	0	0	0	0	0	0	0	0	0	0	0	...	0

L2

Stack

main

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

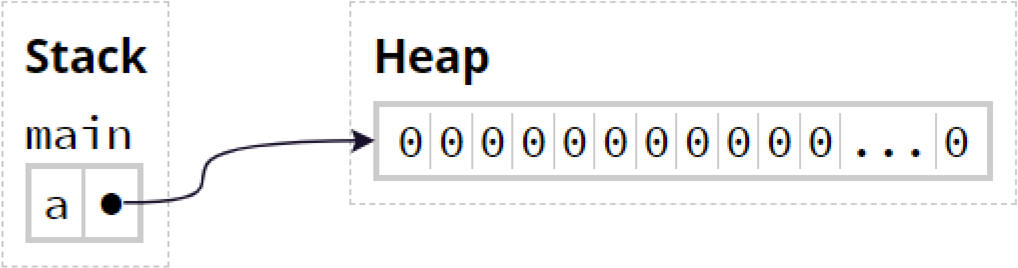


Boxes allocate memory in the heap

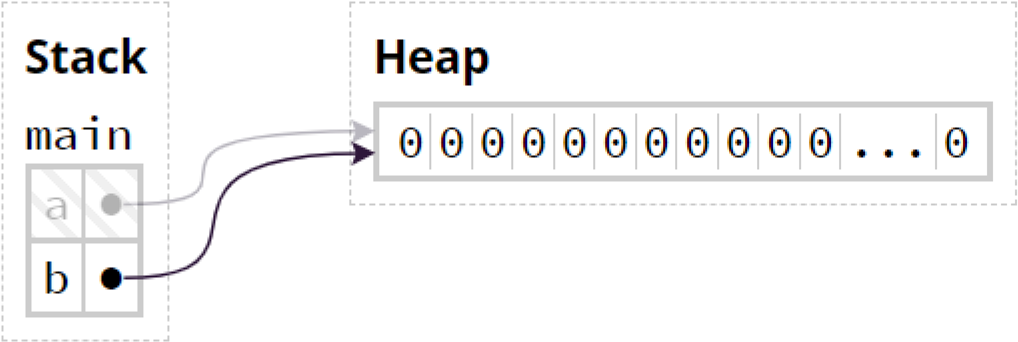
- Now a is of Box (pointer) type
- Assigning b to the value of a *moves* the pointer
- We say that a *owns* the data before the move, and b owns it afterward
- We cannot use a after the move

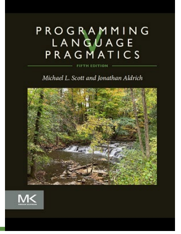
```
let a = Box::new([0; 1_000_000]); L1
let b = a; L2
```

L1



L2



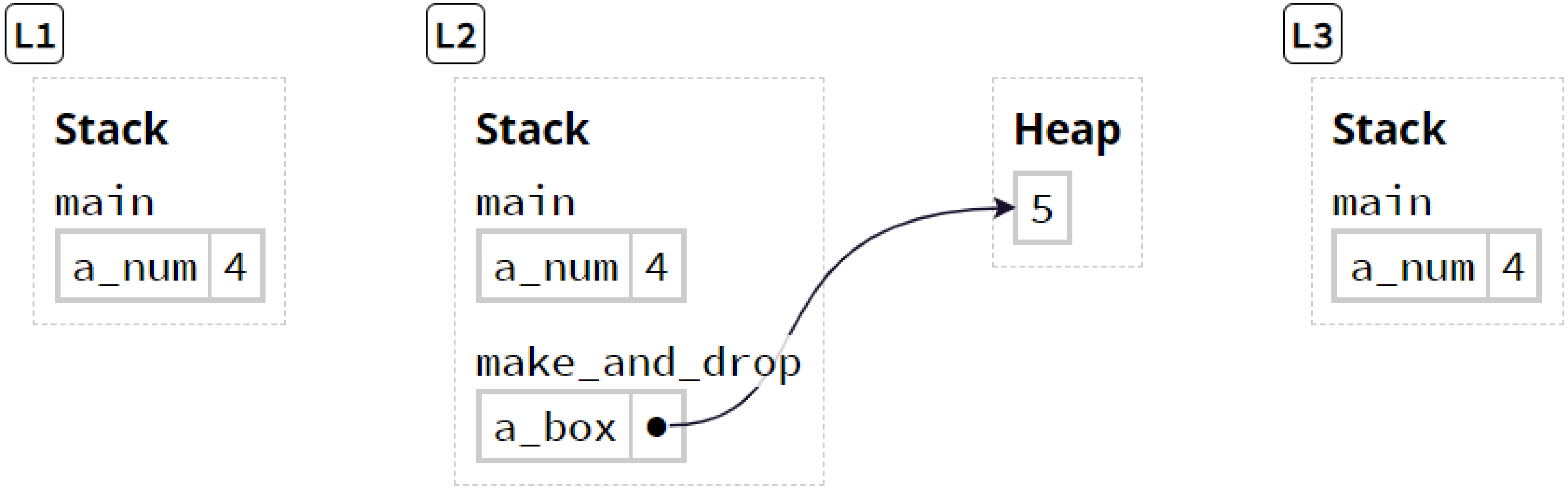


Owners deallocate boxes

If a variable owns a box, when Rust deallocates the variable’s frame, then Rust deallocates the box’s heap memory.

The box holding 5 is deallocated at the end of `make_and_drop`

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

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

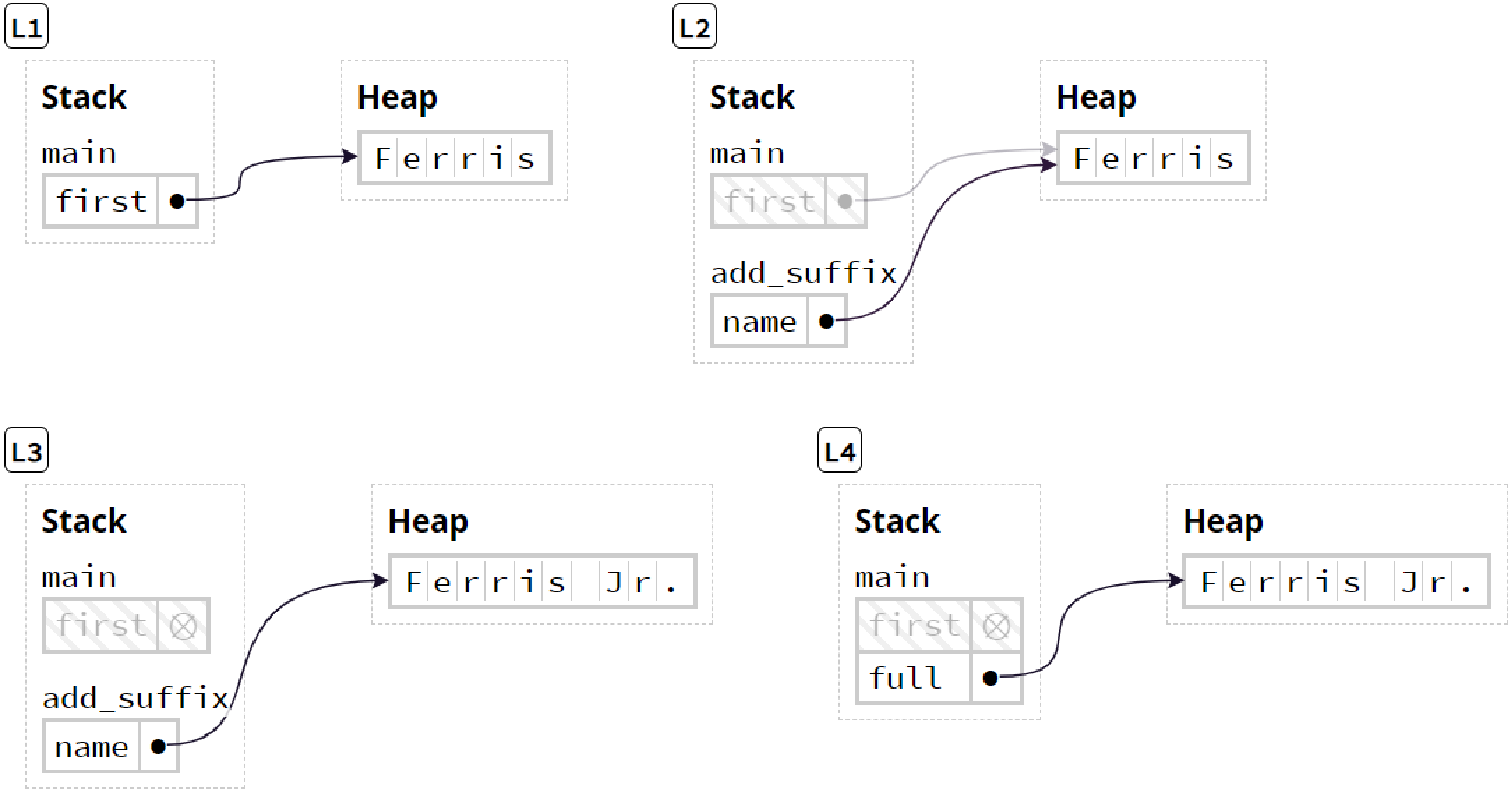


Ownership

Here's an example involving string manipulation

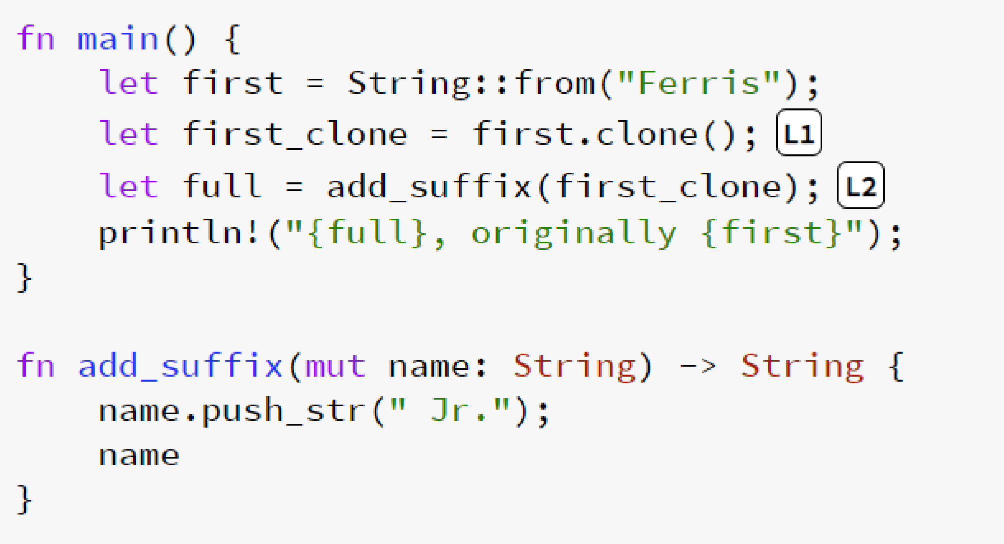
Note that it would be an error to use `first` after the pointer is moved in the call to `from`

```
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
}
```

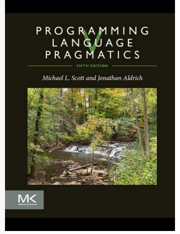


Cloning

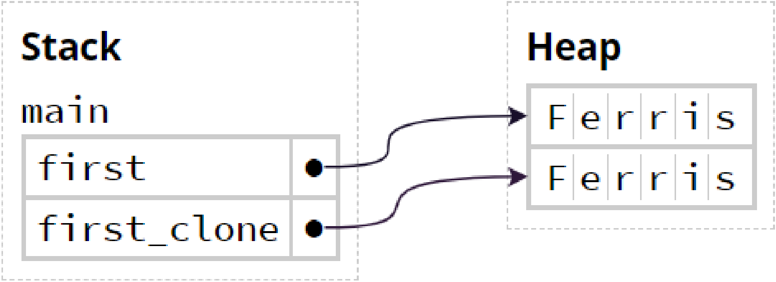
- If we want to continue to use the `first` string, we can *clone* it before moving the pointer
- The `clone` method makes a *deep copy* of the string (the data on the heap is copied, not just the pointer)



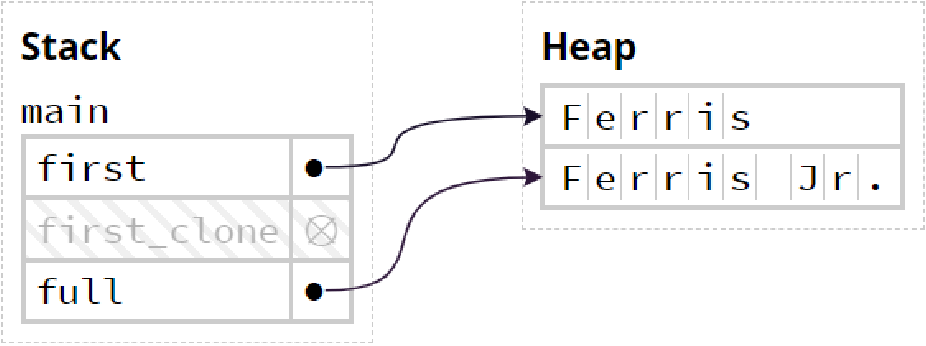
```
fn main() {
    let first = String::from("Ferris");
    let first_clone = first.clone(); L1
    let full = add_suffix(first_clone); L2
    println!("{full}, originally {first}");
}

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



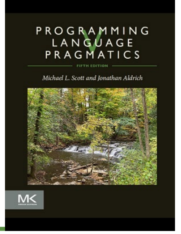
L1



L2

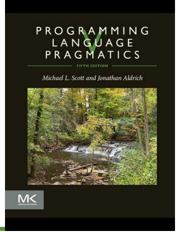


Ownership quiz



- Does this program compile? Why or why not?
- If it compiles, what is the result when it runs?

```
fn main() {
    let s = String::from("hello");
    let s2;
    let b = false;
    if b {
        s2 = s;
    }
    println!("{}", s);
}
```

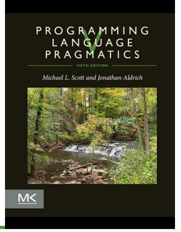


Ownership quiz (SOLUTION)

- Does this program compile? Why or why not?
- Answer: no, it does not compile, because `s` might be moved to `s2` inside the `if` statement, so `s` cannot be used in the `println!` call.
- Rust doesn't try to figure out whether if statements will execute (that's undecidable in general)

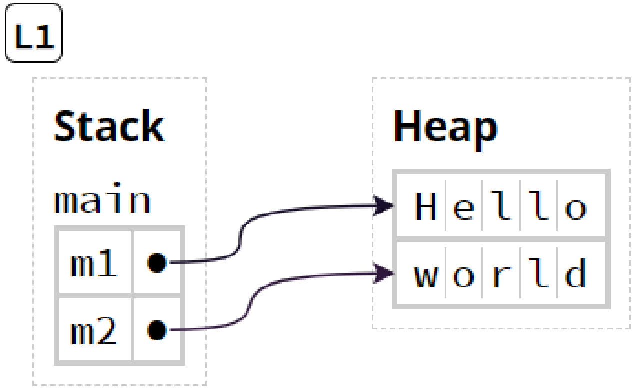
```
fn main() {
    let s = String::from("hello");
    let s2;
    let b = false;
    if b {
        s2 = s;
    }
    println!("{}", s);
}
```

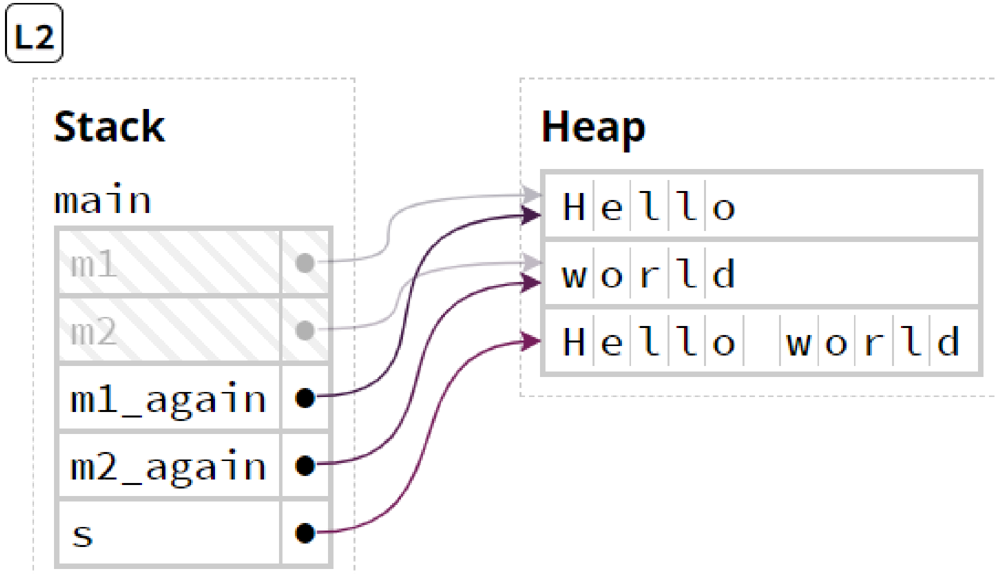
Using pointers after passing them to a function


PROGRAMMING LANGUAGE PRAGMATICS
Michael L. Scott and Jonathan Aldrich
MK

Moving owned pointers can be inconvenient

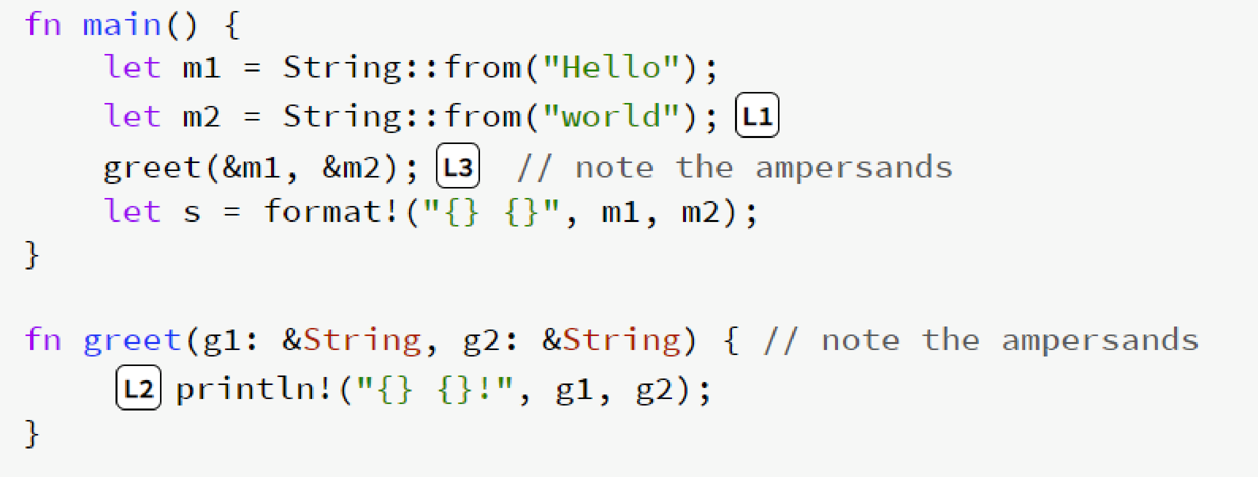
```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    let (m1_again, m2_again) = greet(m1, m2);
    let s = format!("{} {}", m1_again, m2_again); L2
}

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





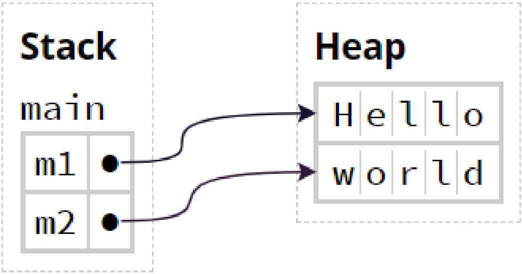
References

- A *reference* is a non-owning pointer
- The expression `&m1` *borrows* `m1`
- `g1` and `g2` are not deallocated at the end of `greet`, because they are not owned

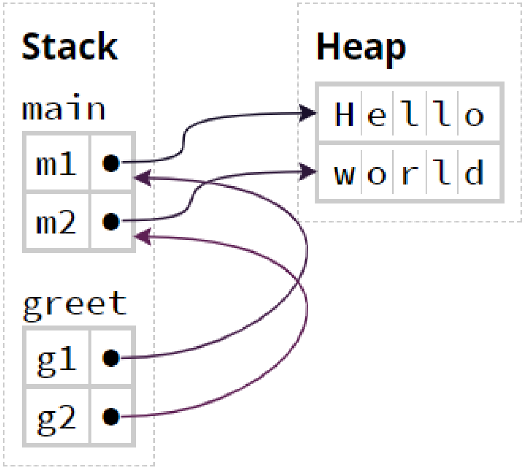


```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    greet(&m1, &m2); L3  // note the ampersands
    let s = format!("{} {}", m1, m2);
}

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

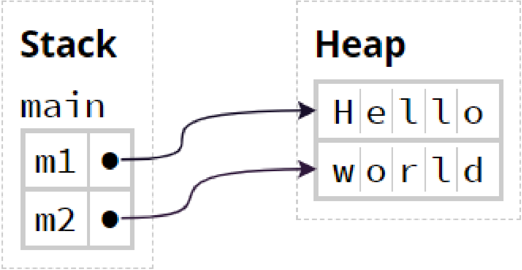
L1

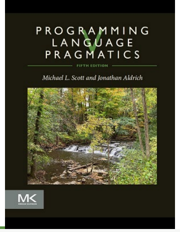


L2



L3

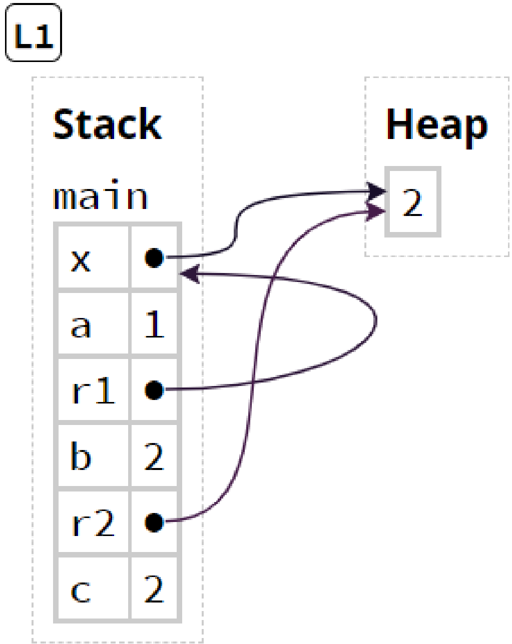




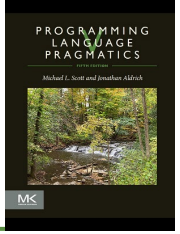
Dereferencing pointers

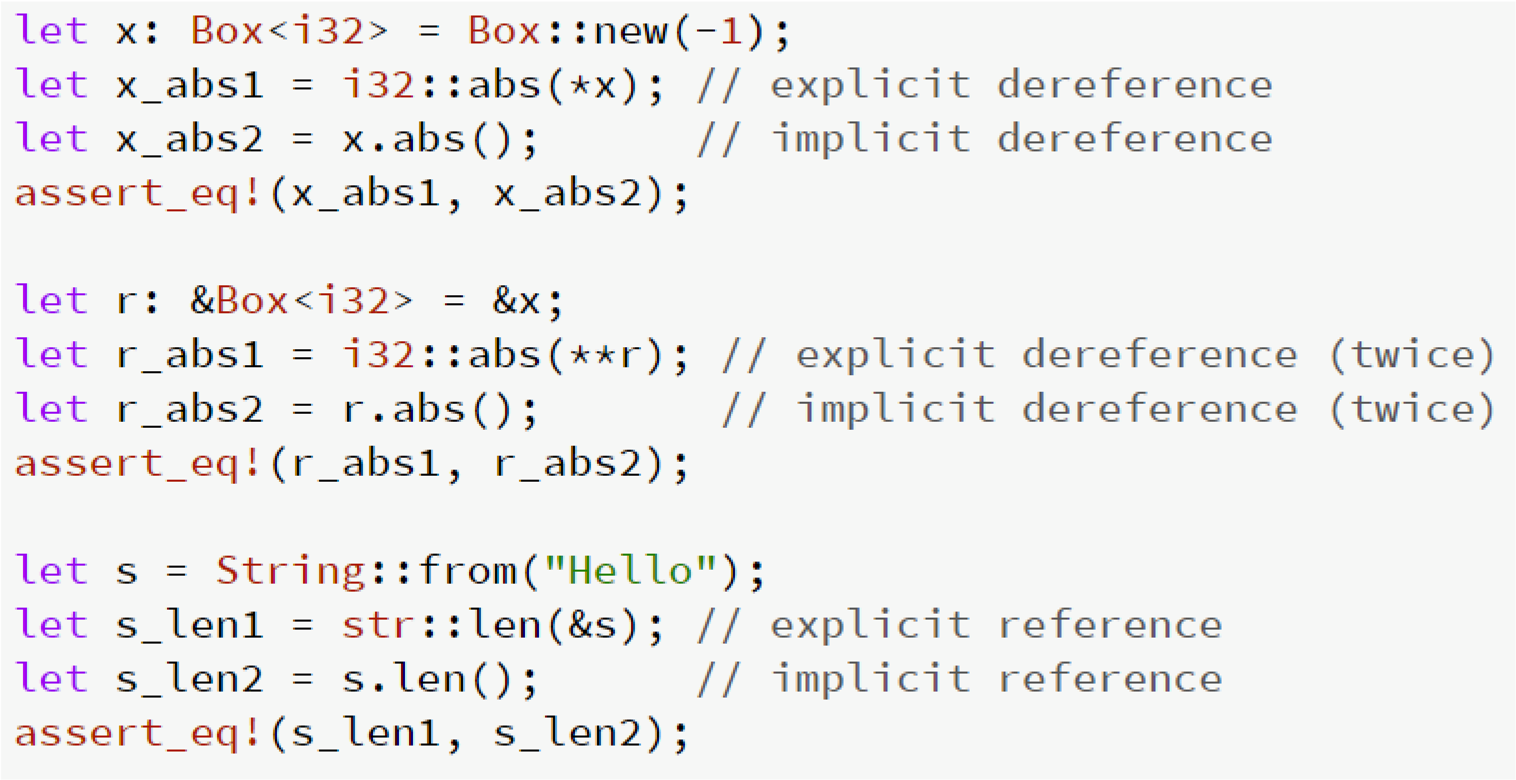
The * operator is used to access the data a pointer refers to

```
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
```

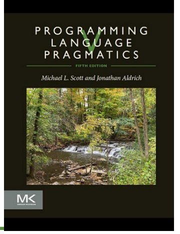


Rust inserts some (de)references automatically





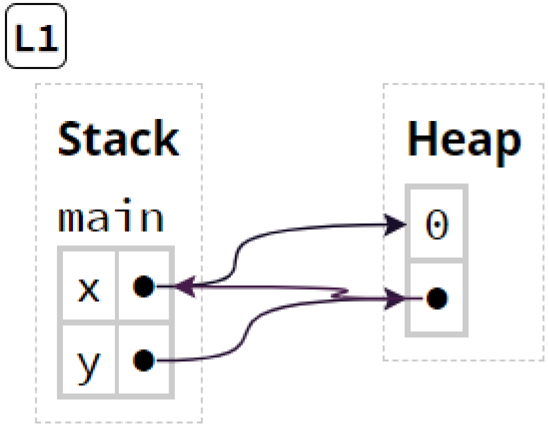
```
let x: Box<i32> = Box::new(-1);
let x_abs1 = i32::abs(*x); // explicit dereference
let x_abs2 = x.abs();      // implicit dereference
assert_eq!(x_abs1, x_abs2);

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);
```



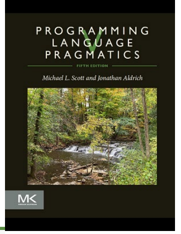
(De)referencing quiz

- Consider the following program, showing the state of memory after the last line:

```
let x = Box::new(0);
let y = Box::new(&x); L1
```

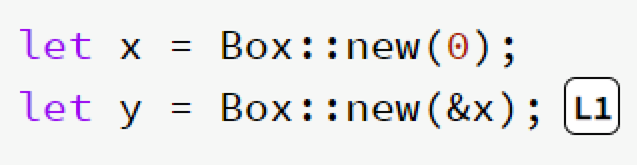


- If you wanted to copy out the number 0 through y, how many dereferences would you need to use?
  - For example, if the correct expression is *y, then the answer is 1.

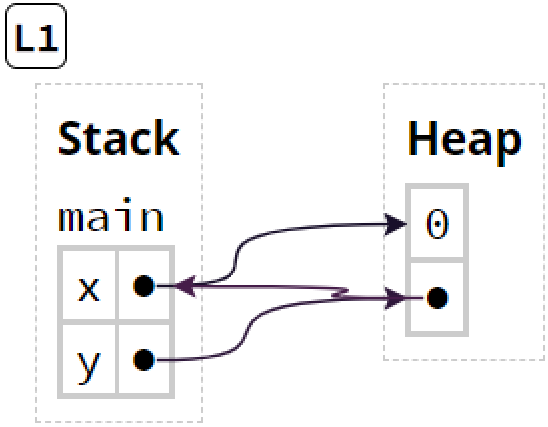


(De)referencing quiz (SOLUTION)

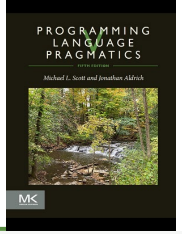
- Consider the following program, showing the state of memory after the last line:



```
let x = Box::new(0);
let y = Box::new(&x); L1
```



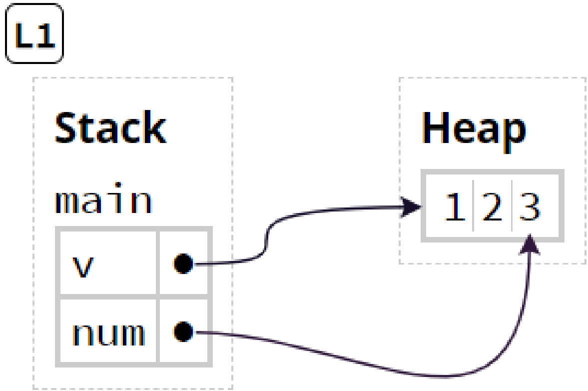
- If you wanted to copy out the number 0 through y, how many dereferences would you need to use?
  - For example, if the correct expression is *y, then the answer is 1.
- Answer: 3 (***y)
  - One dereference for each pointer in the diagram
  - Also: one for each new, one for each &

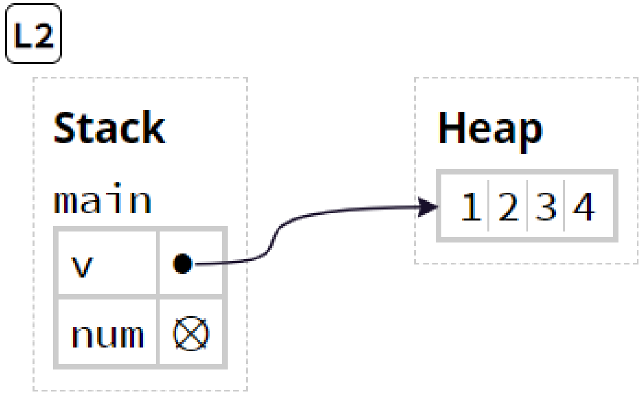


Rust avoids simultaneous aliasing and mutation

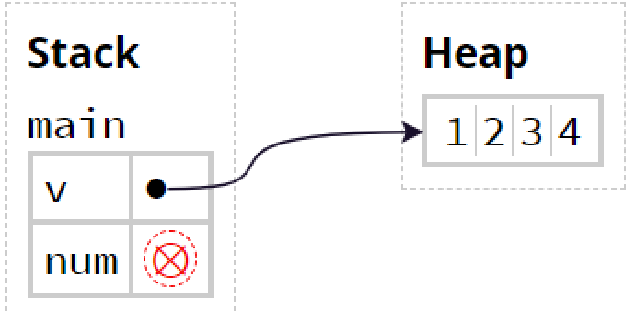
- At L2, the alias num points to v[2]
- We write to v at L2 and read from num at L3
- This is a problem because the Vec's memory is re-allocated at L2, so the pointer used at L3 points to deallocated memory. Undefined behavior!

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



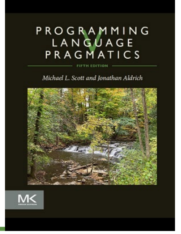


L3 undefined behavior: pointer used after its pointee is freed

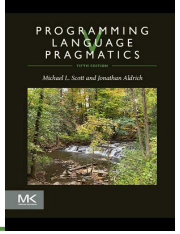


"Puzzled Ferris" means this code does not compile

Rust’s ***borrow checker*** ensures reference safety



- Ensures that data is never aliased and mutated at the same time
- Tracks the permissions associated with each variable:
  - **Read** (**R**): data can be copied to another location.
  - **Write** (**W**): data can be mutated in-place (`let mut` vars)
  - **Own** (**O**): data can be moved or dropped.
- Creating a reference can temporarily remove these permissions

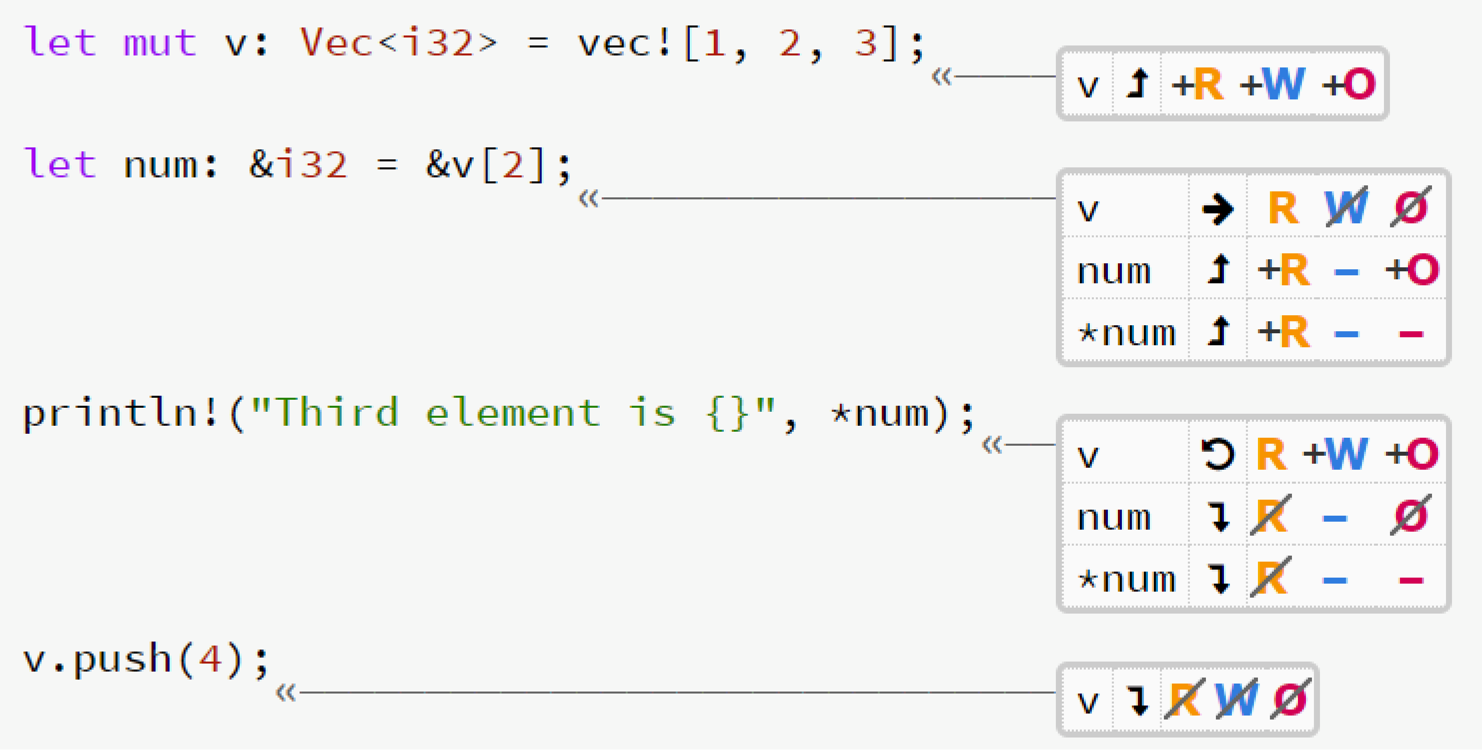

PROGRAMMING LANGUAGE PRAGMATICS

Michael L. Scott and Jonathan Aldrich

Example: how borrow checking works

Notes:

- Different permissions for num and *num
  - manipulating the reference vs. accessing the data
- Permissions are defined on *paths*
  - num, *num, `v[2]`, `a.field`, `*((*a)[0].1)`
- Permissions are lost when a mutually exclusive permission must be used
  - e.g. **W** on v is needed at `v.push(4)` so **R** on num is lost



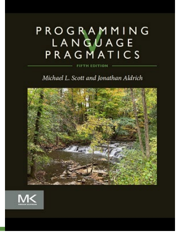
```
let mut v: Vec<i32> = vec![1, 2, 3];

let num: &i32 = &v[2];

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

v.push(4);
```

v	↥	+R	+W	+O

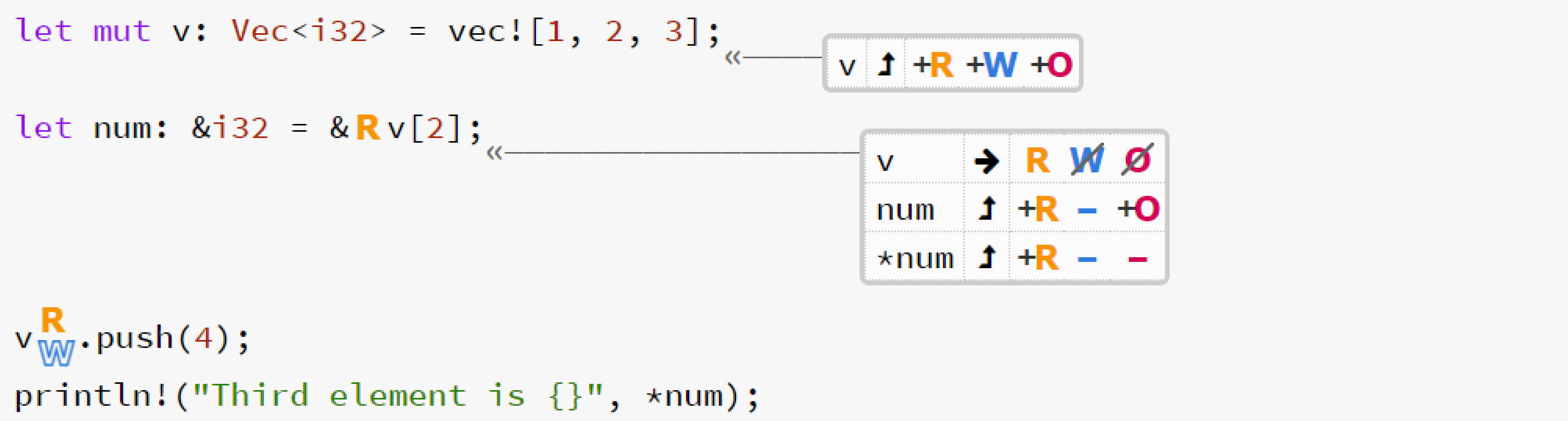
v	→	R	~~W~~	Ø
num	↥	+R	–	+O
*num	↥	+R	–	–

v	↻	R	+W	+O
num	↧	~~R~~	–	Ø
*num	↧	~~R~~	–	–

v	↧	~~R~~	~~W~~	Ø



A borrow checking error

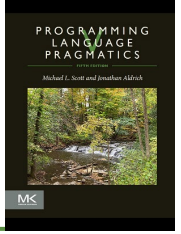


```
let mut v: Vec<i32> = vec![1, 2, 3];

let num: &i32 = &v[2];


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

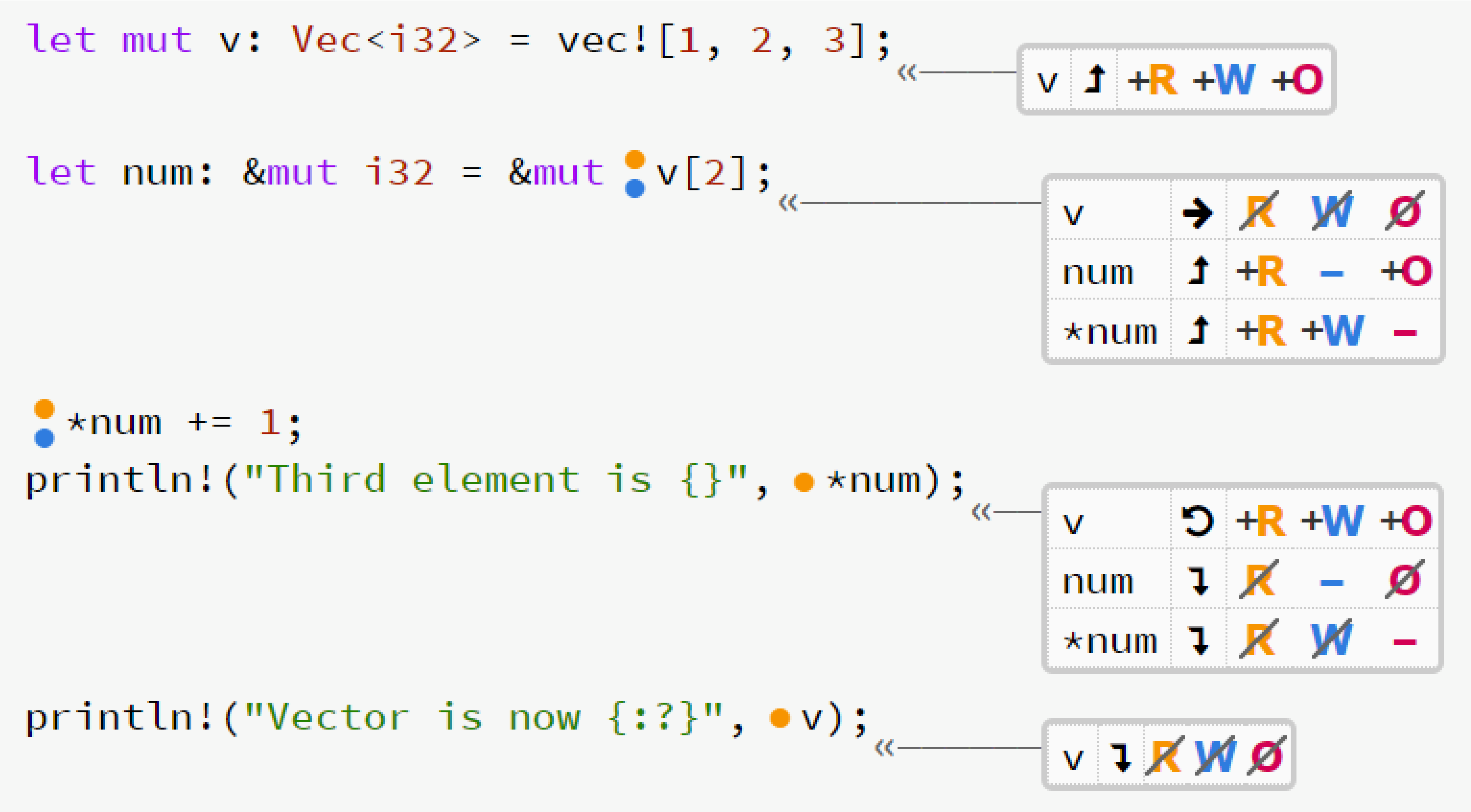
v	↥	+R	+W	+O

v	→	R	~~W~~	~~O~~
num	↥	+R	–	+O
*num	↥	+R	–	–

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



We can also borrow mutably with &mut



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

let num: &mut i32 = &mut ●●v[2];

●●*num += 1;
println!("Third element is {}", ●*num);

println!("Vector is now {:?}", ●v);
```

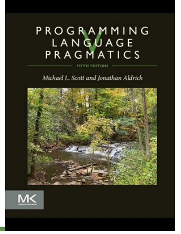
v	↥	+R	+W	+O

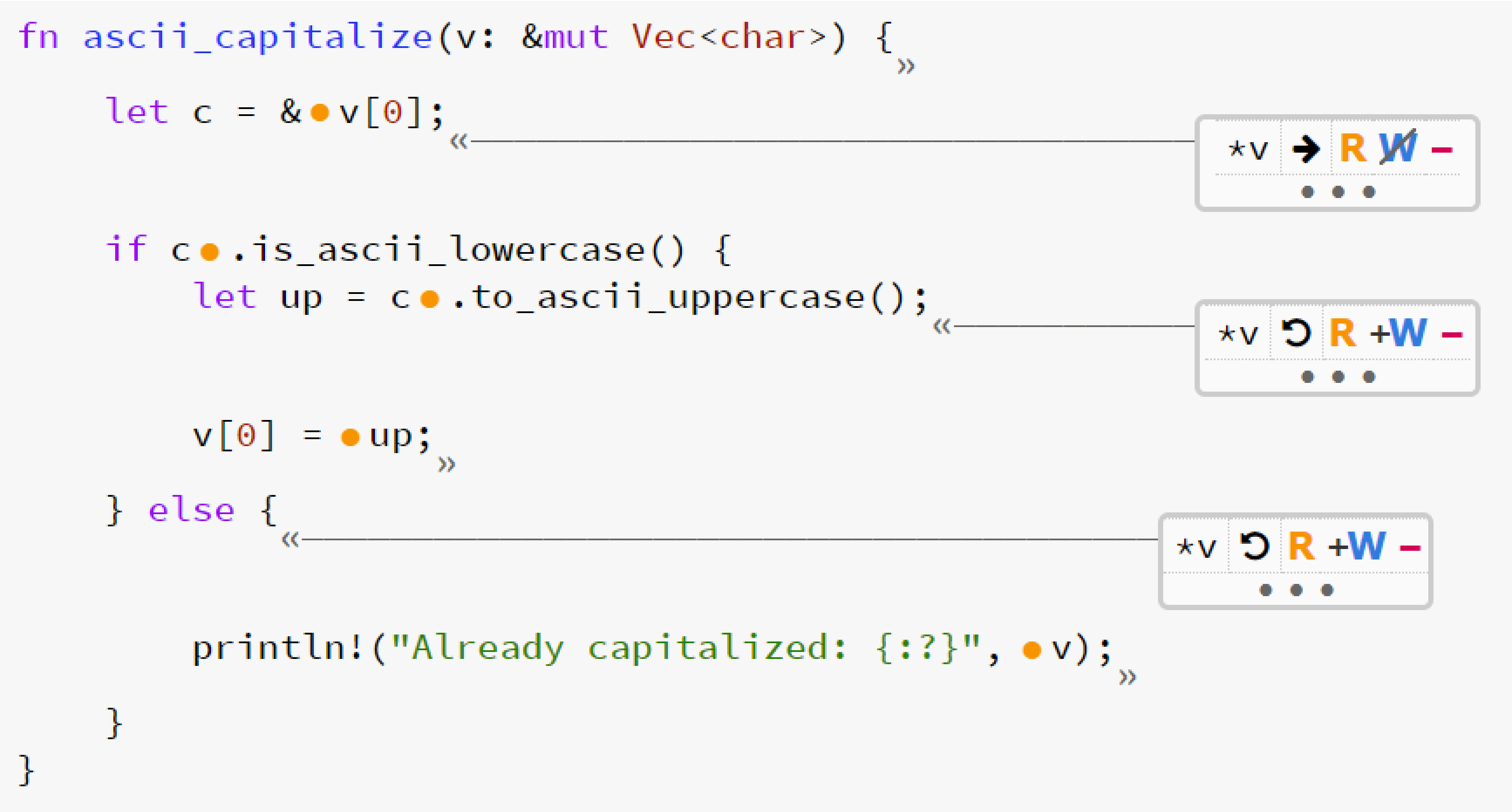
v	→	~~R~~	~~W~~	~~O~~
num	↥	+R	–	+O
*num	↥	+R	+W	–

v	↻	+R	+W	+O
num	↧	~~R~~	–	~~O~~
*num	↧	~~R~~	~~W~~	–

v	↧	~~R~~	~~W~~	~~O~~

Permissions are returned when a reference’s lifetime ends



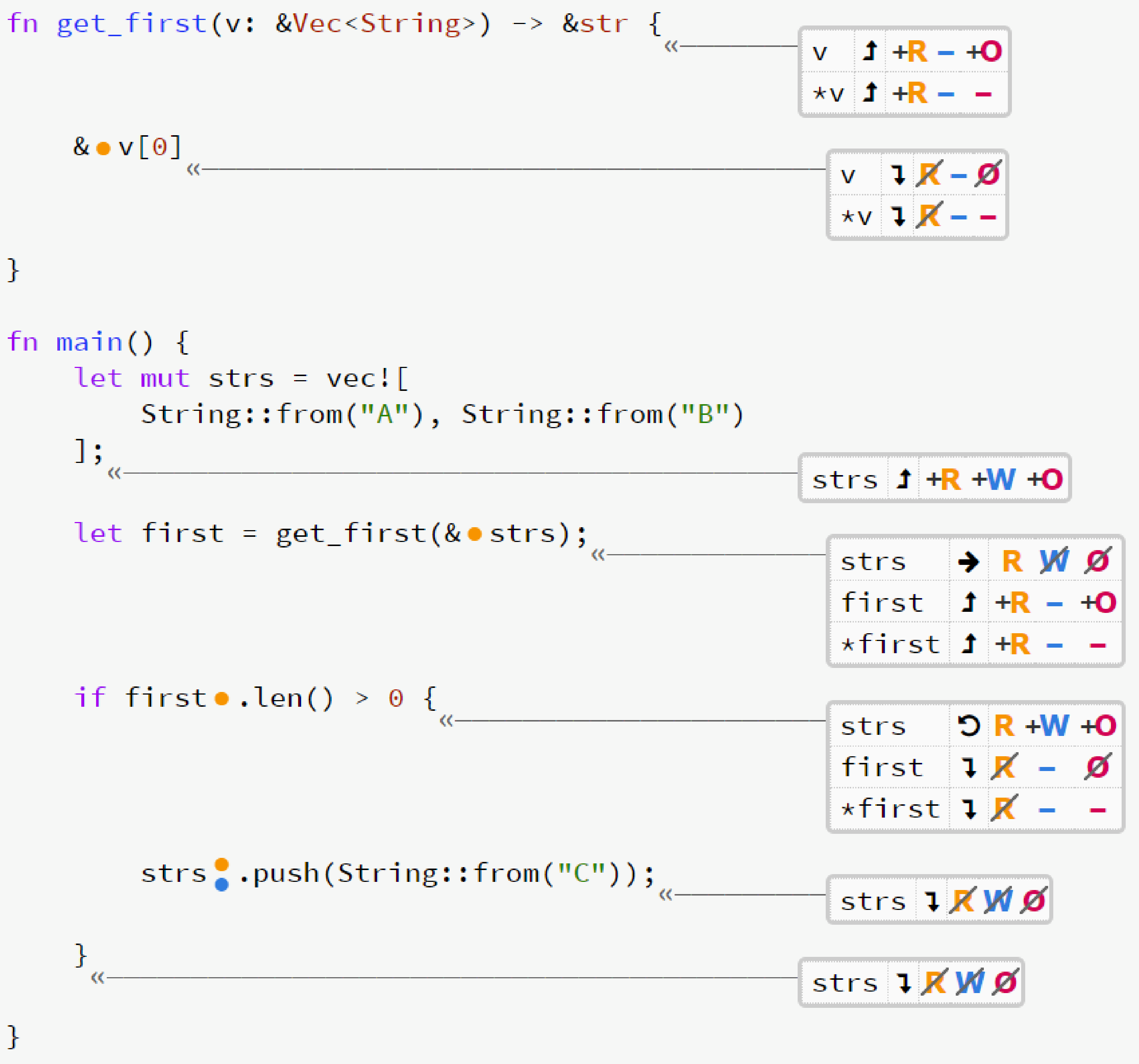


```
fn ascii_capitalize(v: &mut Vec<char>) {
    let c = &v[0];

    if c.is_ascii_lowercase() {
        let up = c.to_ascii_uppercase();

        v[0] = up;
    } else {

        println!("Already capitalized: {:?}", v);
    }
}
```

- Control flow can make this interesting!

Borrowing quiz

In the example, explain why `strs` loses and regains write (W) permissions



```
fn get_first(v: &Vec<String>) -> &str {
```

v	↥	+R	–	+O
*v	↥	+R	–	–

```
    &v[0]
```

v	↧	~~R~~	–	~~O~~
*v	↧	~~R~~	–	–

```
}

fn main() {
    let mut strs = vec![
        String::from("A"), String::from("B")
    ];
```

strs	↥	+R	+W	+O

```
    let first = get_first(&strs);
```

strs	→	R	~~W~~	~~O~~
first	↥	+R	–	+O
*first	↥	+R	–	–

```
    if first.len() > 0 {
```

strs	↻	R	+W	+O
first	↧	~~R~~	–	~~O~~
*first	↧	~~R~~	–	–

```
        strs.push(String::from("C"));
```

strs	↧	~~R~~	~~W~~	~~O~~

```
    }
```

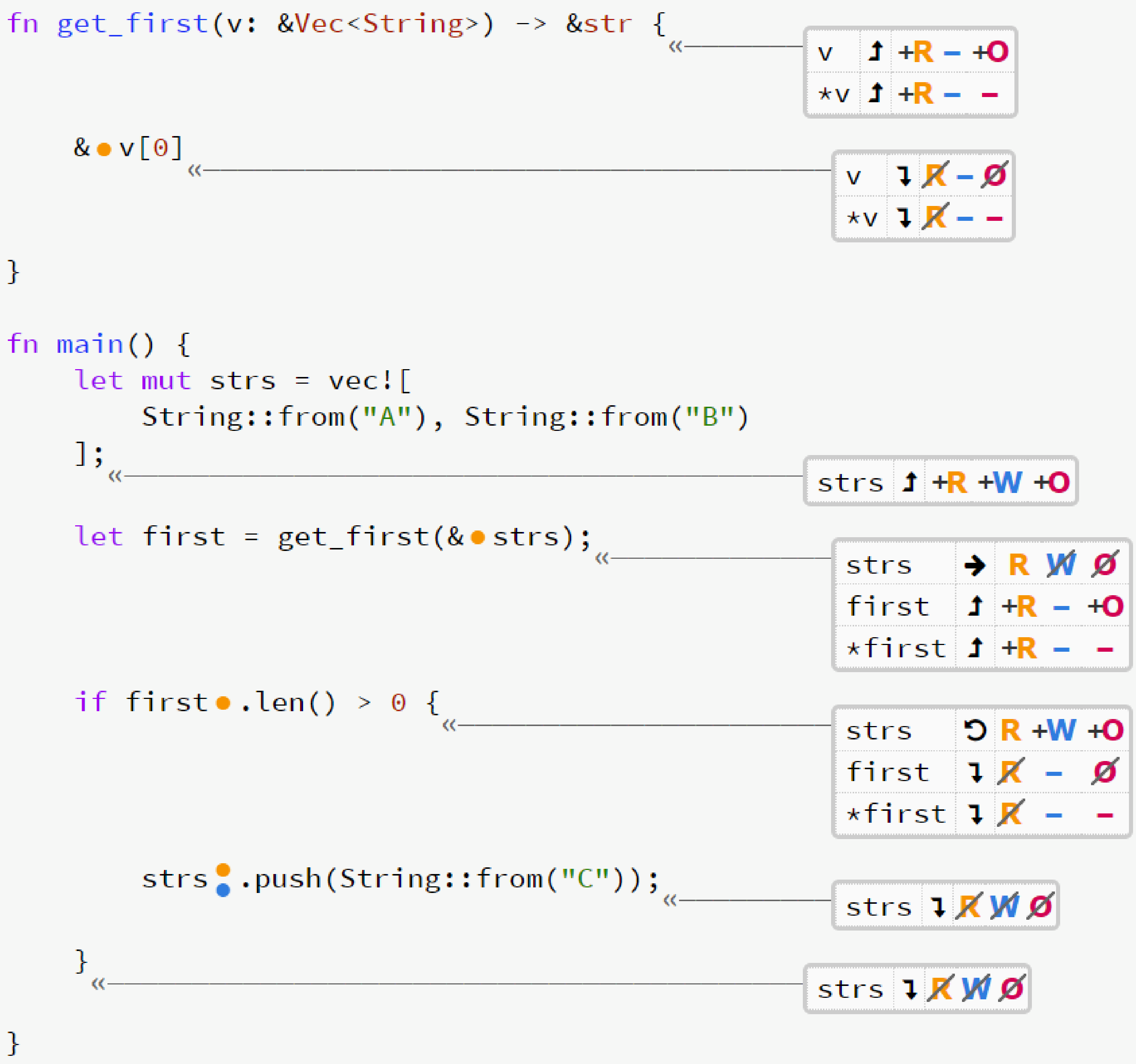
strs	↧	~~R~~	~~W~~	~~O~~

```
}
```

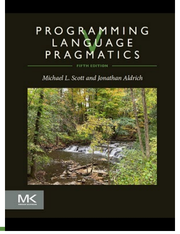
Borrowing quiz

In the example, explain why `strs` loses and regains write (W) permissions

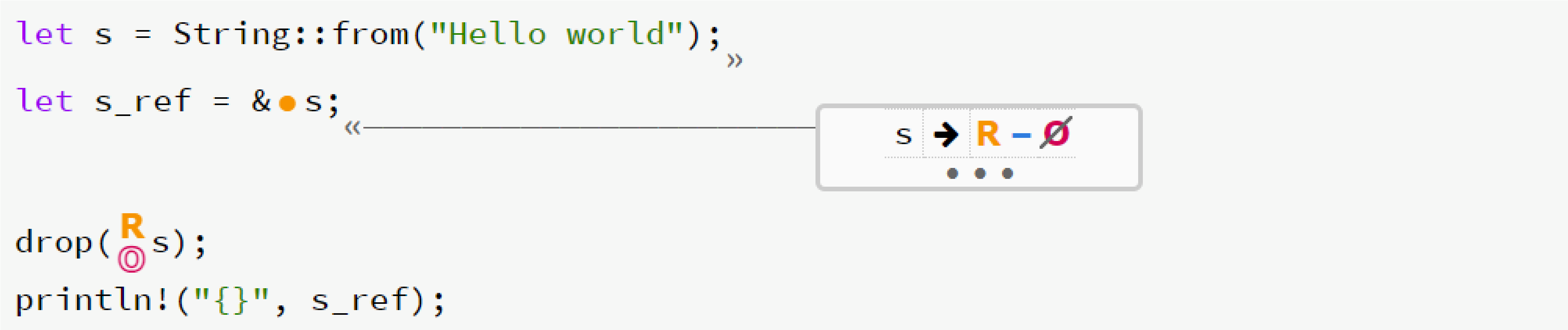
ANSWER: `get_first` returns an immutable reference to data within `strs`, so `strs` is not writable while `first` is live



```
fn get_first(v: &Vec<String>) -> &str {
    &v[0]
}

fn main() {
    let mut strs = vec![
        String::from("A"), String::from("B")
    ];
    let first = get_first(&strs);
    if first.len() > 0 {
        strs.push(String::from("C"));
    }
}
```



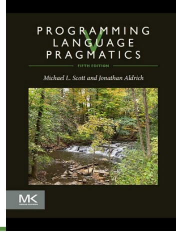
Data must outlive its references



```
let s = String::from("Hello world");

let s_ref = &s;


drop(s);
println!("{}", s_ref);
```

- The drop function explicitly frees a pointer
- But drop requires an ownership (O) permission and we do not have that for s while s_ref is live

Fixing borrow checking errors

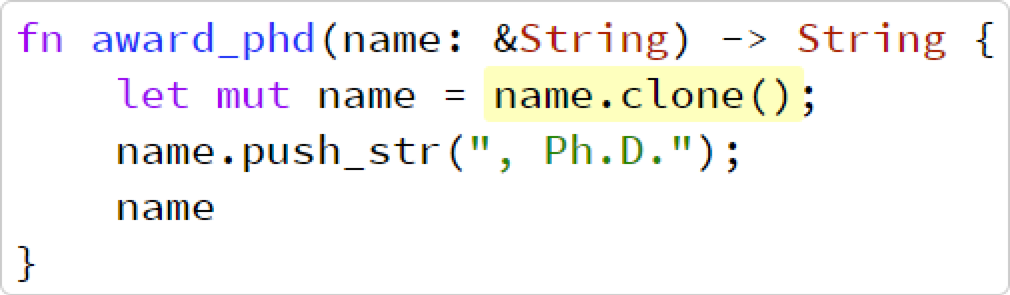


- The following code has a borrow error:

```
/// Returns a person's name with "Ph.D." added as a title
fn award_phd(name: &String) -> String {
    let mut name = *name;
    name.push_str(", Ph.D.");
    name
}
```

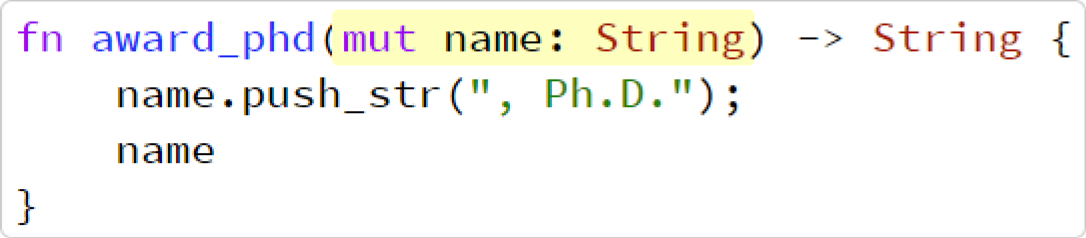
- What’s the best fix?

A



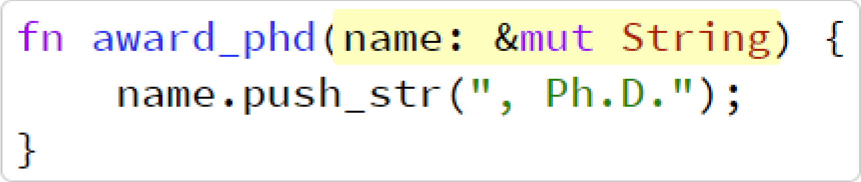
```
fn award_phd(name: &String) -> String {
    let mut name = name.clone();
    name.push_str(", Ph.D.");
    name
}
```

B



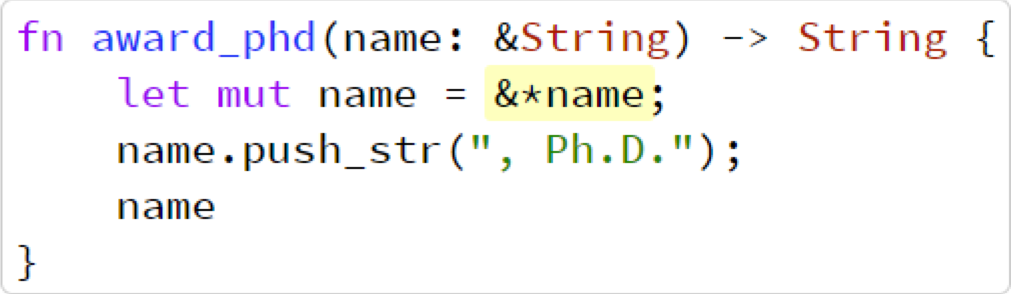
```
fn award_phd(mut name: String) -> String {
    name.push_str(", Ph.D.");
    name
}
```

C



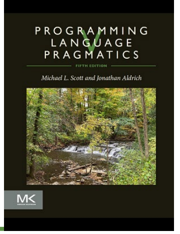
```
fn award_phd(name: &mut String) {
    name.push_str(", Ph.D.");
}
```

D



```
fn award_phd(name: &String) -> String {
    let mut name = &*name;
    name.push_str(", Ph.D.");
    name
}
```

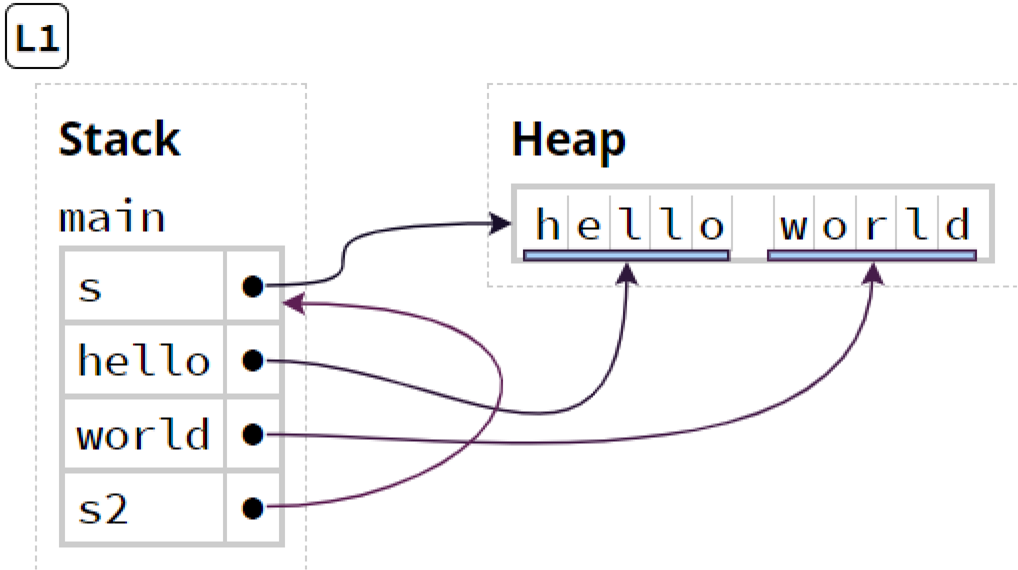
String slices



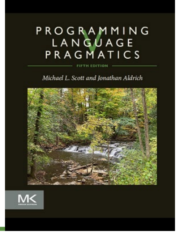
- A string slice of type `&str` points to a range of characters in a string
  - `&str` is the type of string literals in Rust!
- A slice knows its length—access beyond the length is a run time error
- Slices are references, so taking a slice changes the permission to the underlying data
  - If `s` were `mut` then we couldn't mutate it while `hello` is live
- You can also take slides of arrays

```
let a = [1, 2, 3, 4, 5];
let slice : &[i32] = &a[1..3];
```

```
let s = String::from("hello world");

let hello: &str = &s[0..5];
let world: &str = &s[6..11];
let s2: &String = &s; L1
```

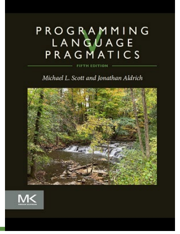


Sometimes Rust can’t tell the lifetime of a reference



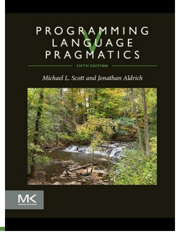
```rust
// does longest return x or y?
// unclear -- and it matters if they have different lifetimes
fn longest(x: &str, y: &str) -> &str {
   if x.len() > y.len() { x } else { y }
}

fn main() {
   let string1 = String::from("abcd");
   let string2 = "xyz";
   let result = longest(string1.as_str(), string2);
   println!("The longest string is {result}");
}
```

Lifetime annotations can help



```
&i32            // a reference
&'a i32         // a reference with an explicit lifetime
&'a mut i32 // a mutable reference with an explicit lifetime
```

- We don’t need to write lifetime annotations anywhere—just when we need to compare the lifetimes of different references (e.g. in a function signature)

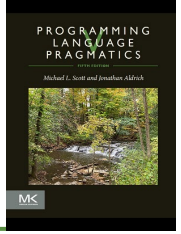


Using lifetime annotations

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

- This signature tells Rust that for some lifetime `'a`, the arguments must live at least as long as `‘a`.
- Also, the value returned by `longest` will live at least as long as `'a`.

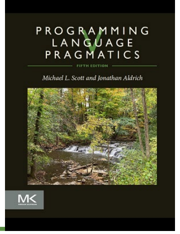
We can put lifetime annotations in structs



```
struct ImportantExcerpt<'a> { part: &'a str, }
fn main() {
  let novel = String::from("Call me Ishmael. Some years ago...");
  let first_sentence = novel.split('.').next().unwrap();
  let i = ImportantExcerpt { part: first_sentence, };
}
```

- The lifetime parameter of `ImportantExcerpt` tracks how long the `part` reference lives.
- Rust checks that `i` isn’t used after `novel` goes out of scope

Lifetime annotations can often be elided



- If you don’t provide them, Rust acts as if they were specified according to the following rules
  - Every lifetime in the input type gets its own lifetime parameter

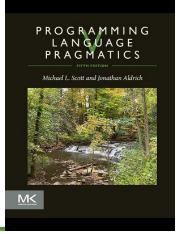
```
fn foo(x: &i32, y: &i32) → foo<'a, 'b>(x: &'a i32, y: &'b i32)
```

  - If there is exactly one lifetime parameter, that lifetime is assigned to all output lifetime parameters

```
fn foo(x: &i32) -> &i32 → fn foo<'a>(x: &'a i32) -> &'a i32
```

  - [Methods only]: If there are multiple input lifetime parameters, but one of them is `&self` or `&mut self`, that lifetime is used for all output lifetime parameters

The `'static` lifetime



- The `'static` lifetime is for things that live for the entire execution of the program
  - Example: string literals
- Only use it if you know the underlying data lives indefinitely!