CS 241: Systems Programming
Lecture 12. References

Fall 2025

Prof. Stephen Checkoway

Boxes are pointers that own their data

Recall a Box is a pointer to valid data in the heap

When a Box is dropped, the heap memory is deallocated

When a Box is assigned to another variable (or passed to a function), ownership of the Box moves to the new variable

When a Box is moved, the data cannot be accessed from the old variable

Strings and Vecs have a Box* holding their contents

*Not really a Box, but similar

Does this code compile? If it does, what is its output?

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

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

A. Compile-time error

B. Compiles but run-time error

C. Prints “Hello World!”

D. Prints “Hello World!” then panics

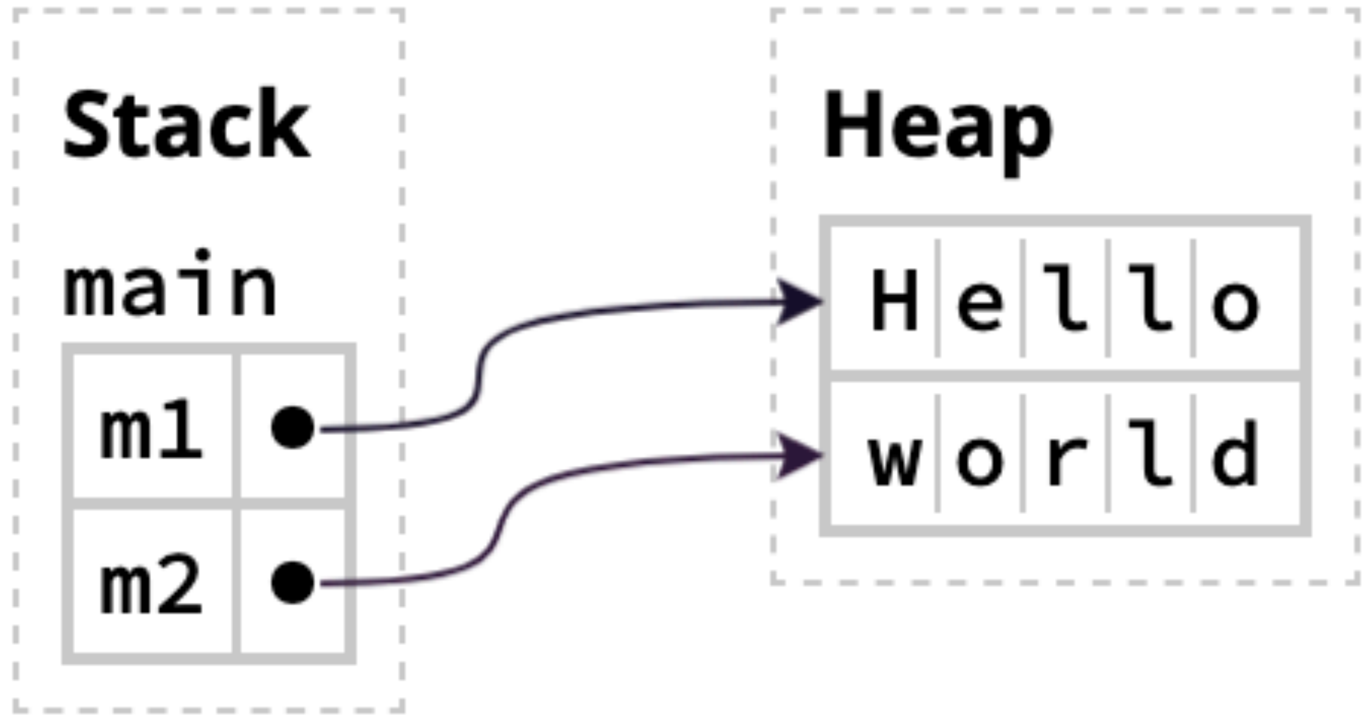
References are non-owning pointers

References are a way to lend an object to some code without making a clone/moving the data

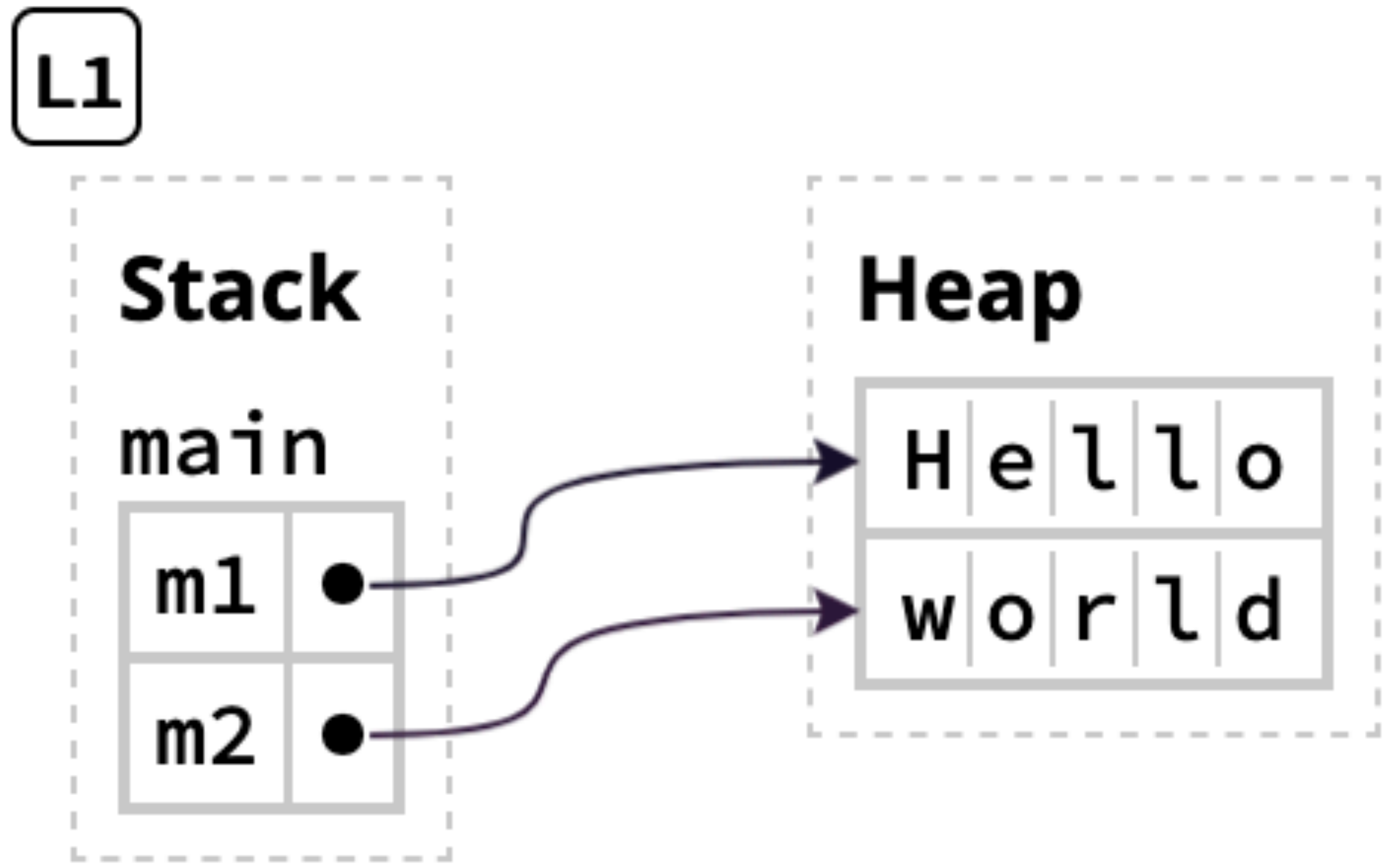
We use `&var` to create a reference to the variable `var`

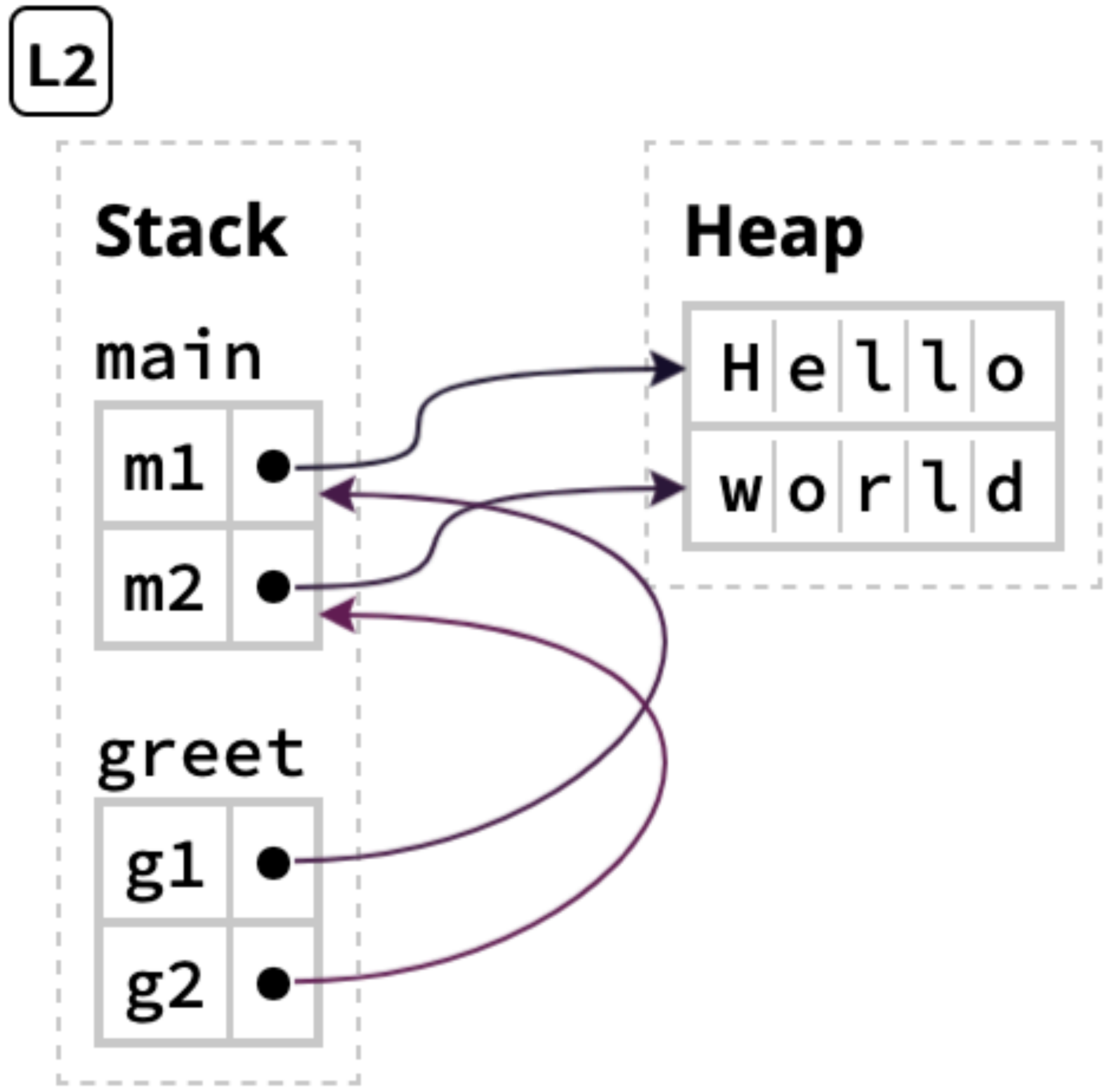
```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    greet(&m1, &m2); L3  // note the ampersands
    let s = format!("{} {}", m1, m2);
}

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

L1



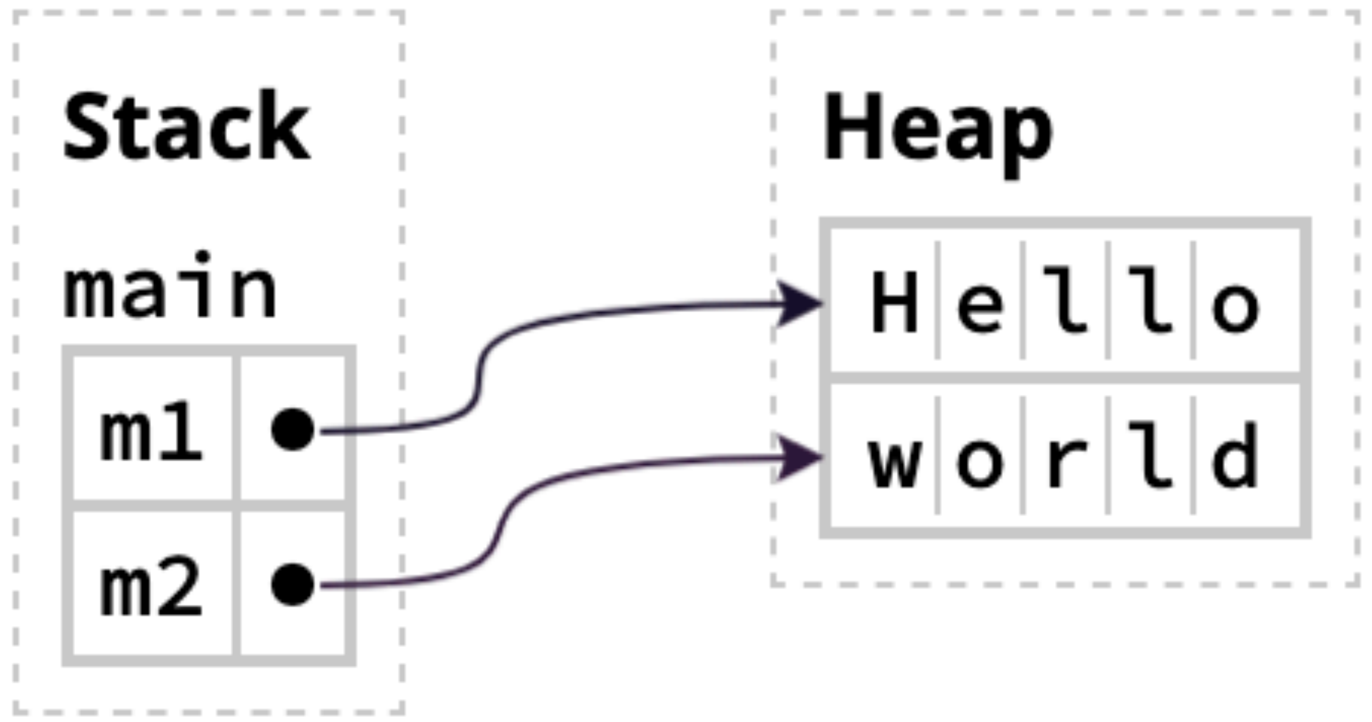
```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    greet(&m1, &m2); L3  // note the ampersands
    let s = format!("{} {}", m1, m2);
}

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



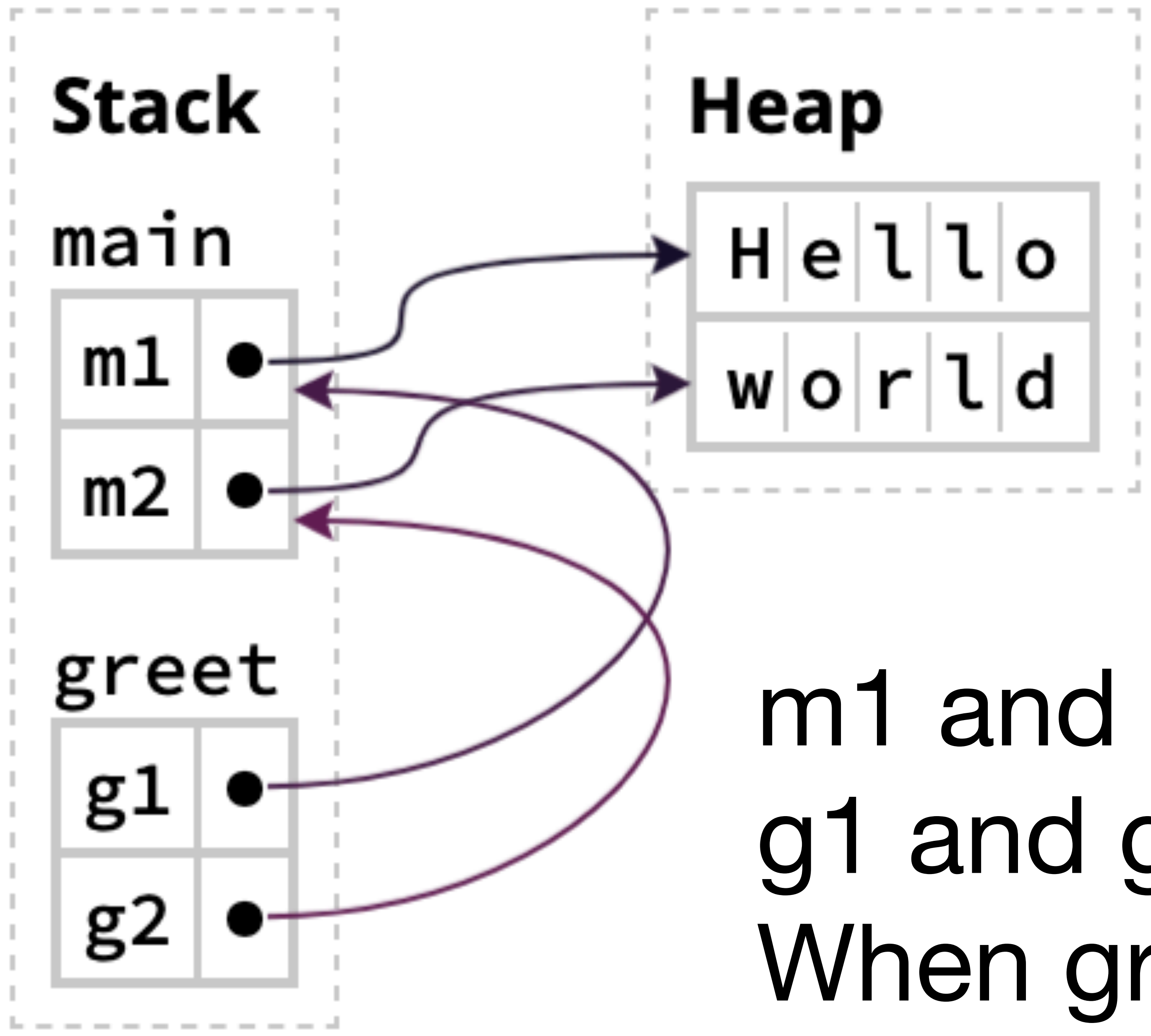


```rust
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world"); L1
    greet(&m1, &m2); L3  // note the ampersands
    let s = format!("{} {}", m1, m2);
}

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

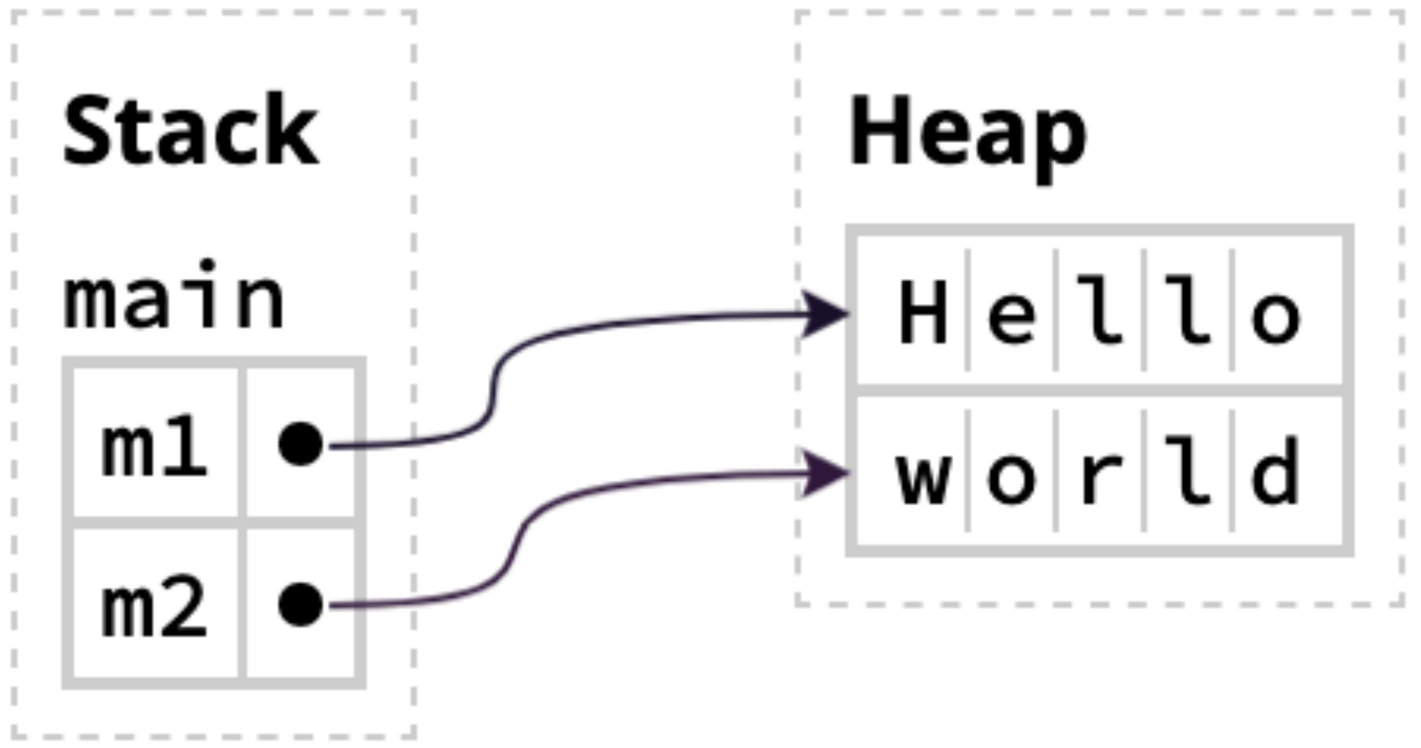
L1



L2



L3



m1 and m2 own their heap data
g1 and g2 do not own anything
When greet() returns, nothing is freed

Dereferencing a pointer

We use * to **dereference** a pointer

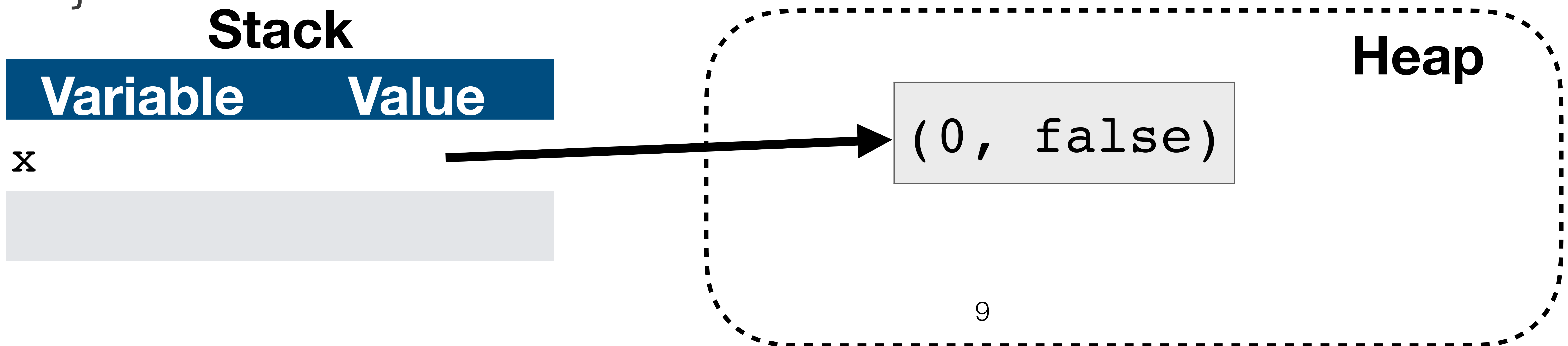
- Dereferencing means to get or assign the value pointed at by the reference

Dereferencing a pointer

We use * to **dereference** a pointer

- Dereferencing means to get or assign the value pointed at by the reference

```
fn main() {
    let mut x: Box<(i32, bool)> = Box::new((0, false));









}
```



Dereferencing a pointer

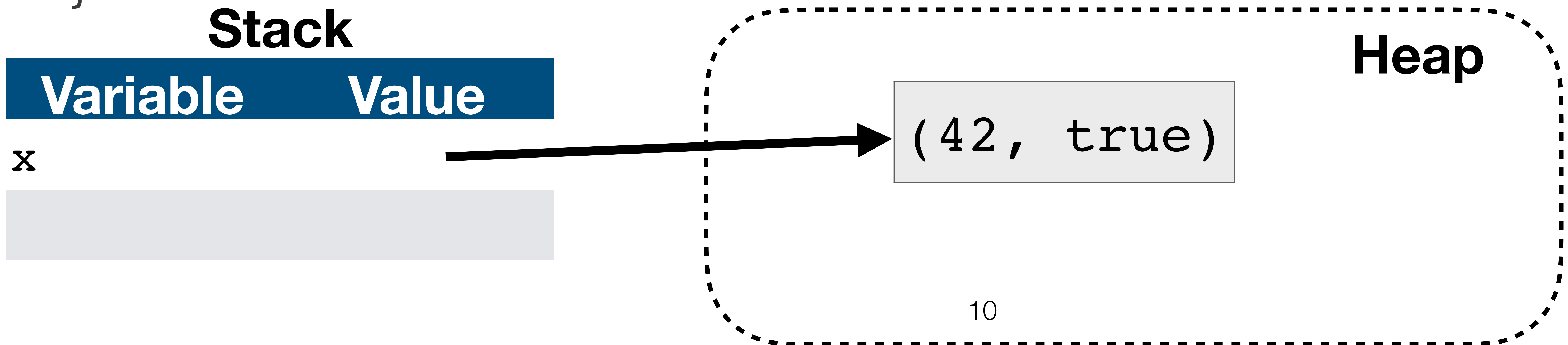
We use * to **dereference** a pointer

- Dereferencing means to get or assign the value pointed at by the reference

```
fn main() {
    let mut x: Box<(i32, bool)> = Box::new((0, false));
    *x = (42, true); // Dereferences and assigns a new value
    println!("{x:?}");



}
```

Output:

```
(42, true)
```



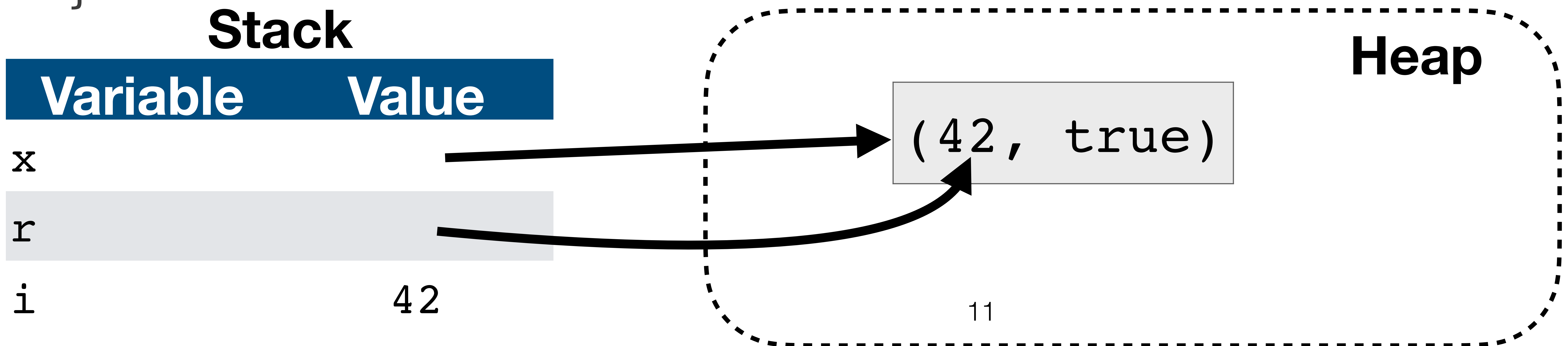
Dereferencing a pointer

We use * to **dereference** a pointer

- Dereferencing means to get or assign the value pointed at by the reference

```rust
fn main() {
    let mut x: Box<(i32, bool)> = Box::new((0, false));
    *x = (42, true); // Dereferences and assigns a new value
    println!("{x:?}");

    let r: &i32 = &x.0; // Creates a reference to x.0
    let i: i32 = *r; // Dereferences the reference
    println!("i = {i}");
    println!("r = {r}"); // Automatic dereference!
}
```

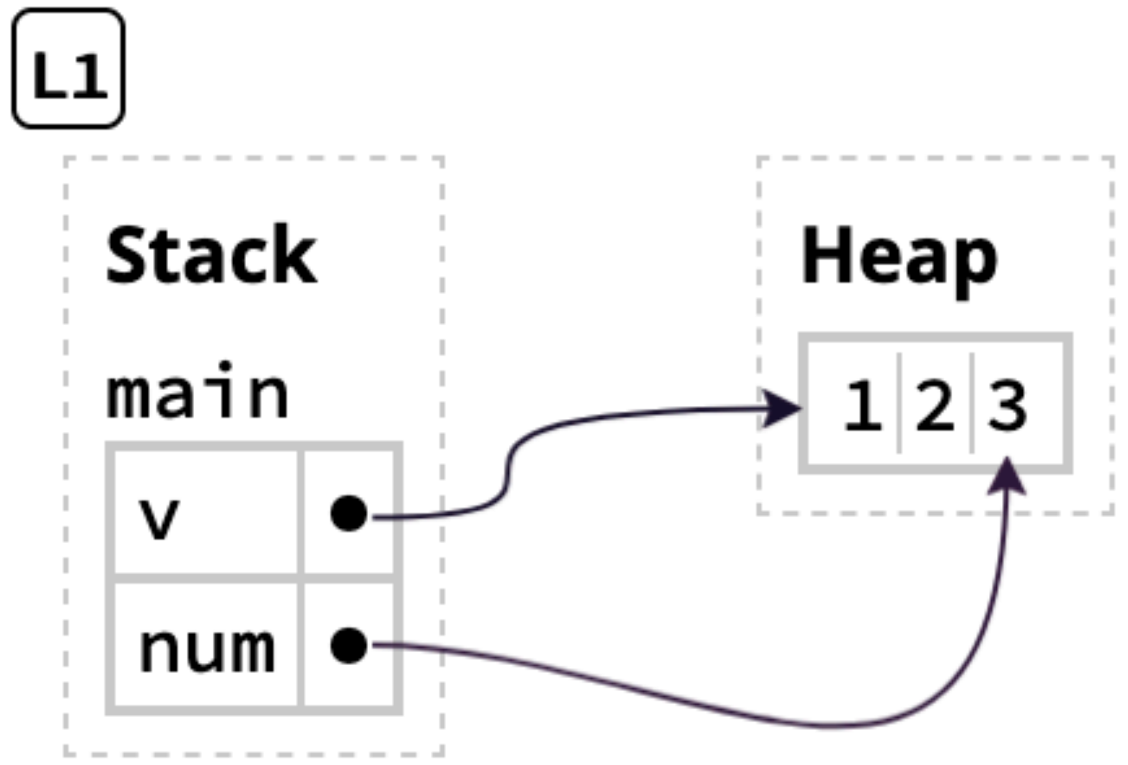
Output:
```
(42, true)
i = 42
r = 42
```



References + modification = sadness

num points to the third element of v

```
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
```

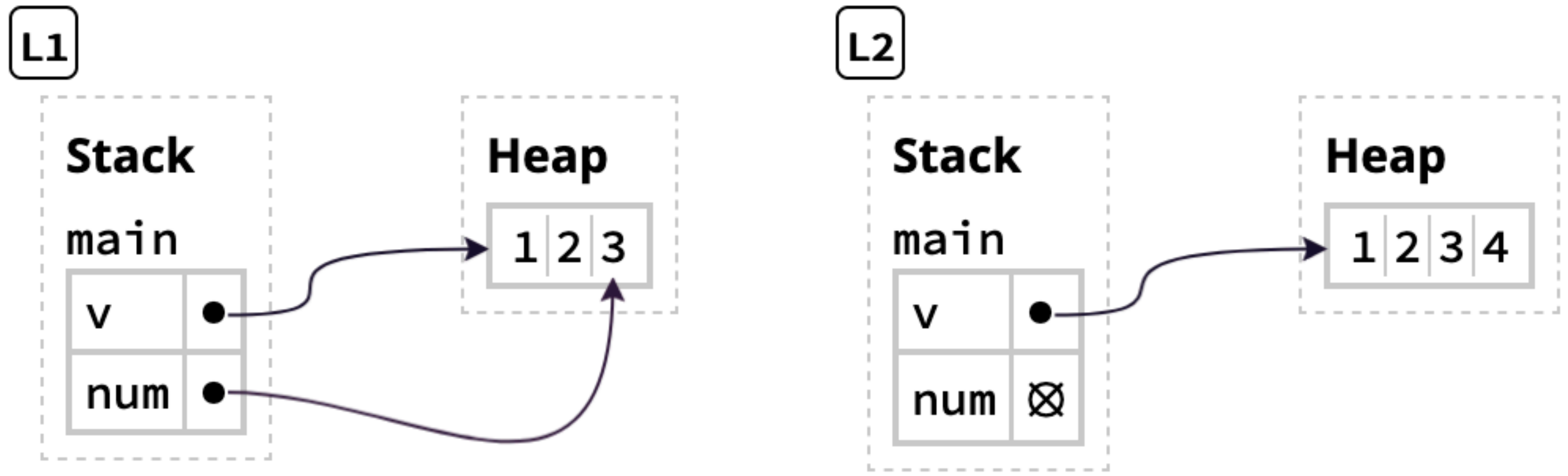


References + modification = sadness

num points to the third element of v

Pushing a new element might cause v’s heap memory to be reallocated

```
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
```



References + modification = sadness

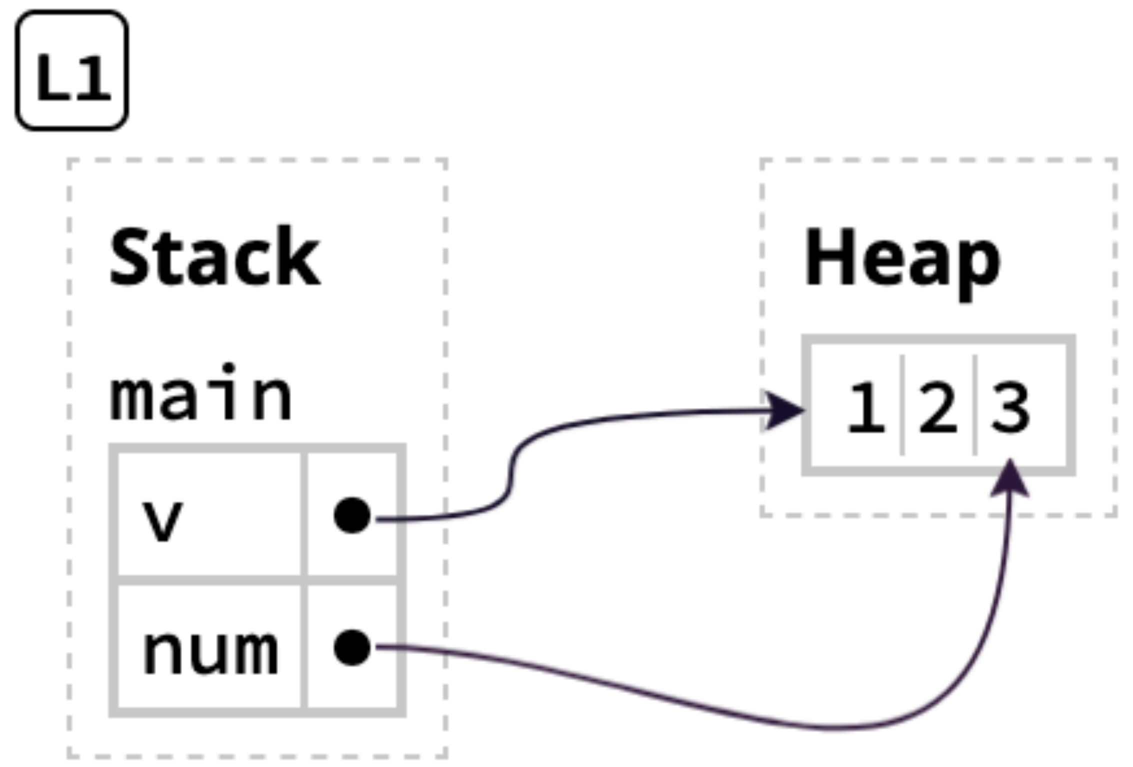
num points to the third element of v

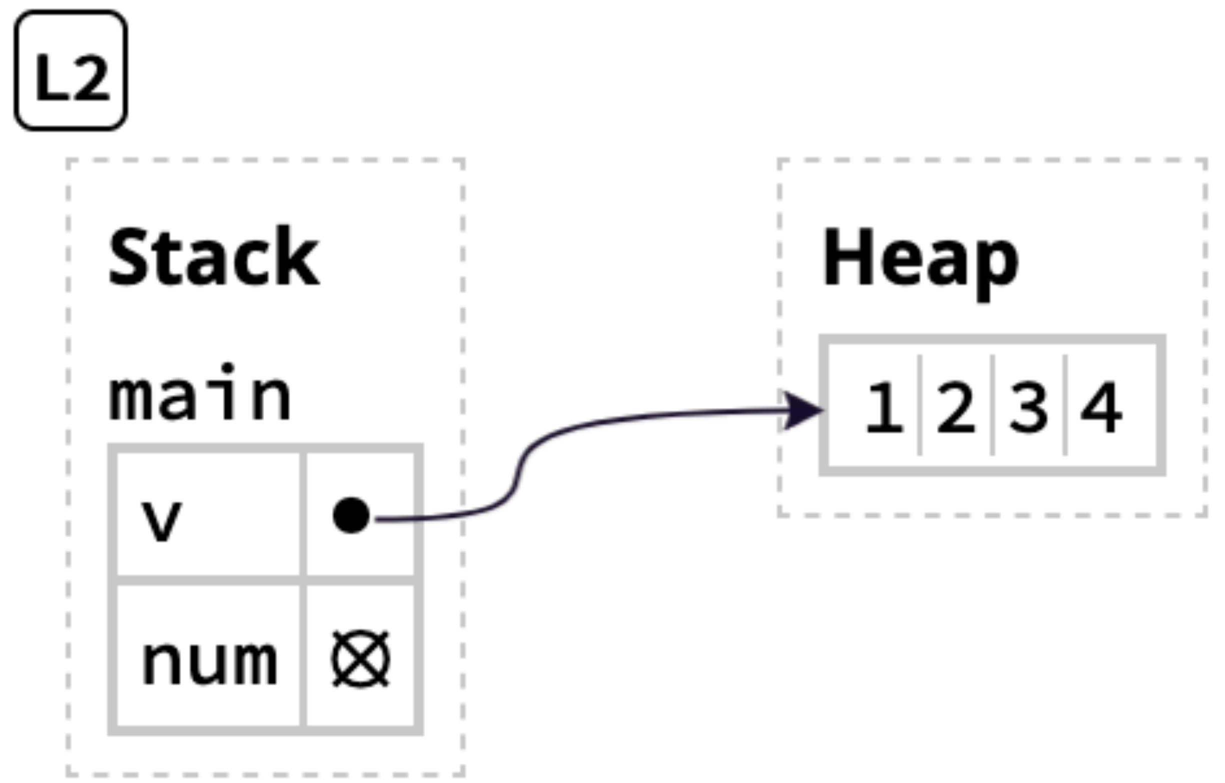
Pushing a new element might cause v’s heap memory to be reallocated

Now num points at invalid memory

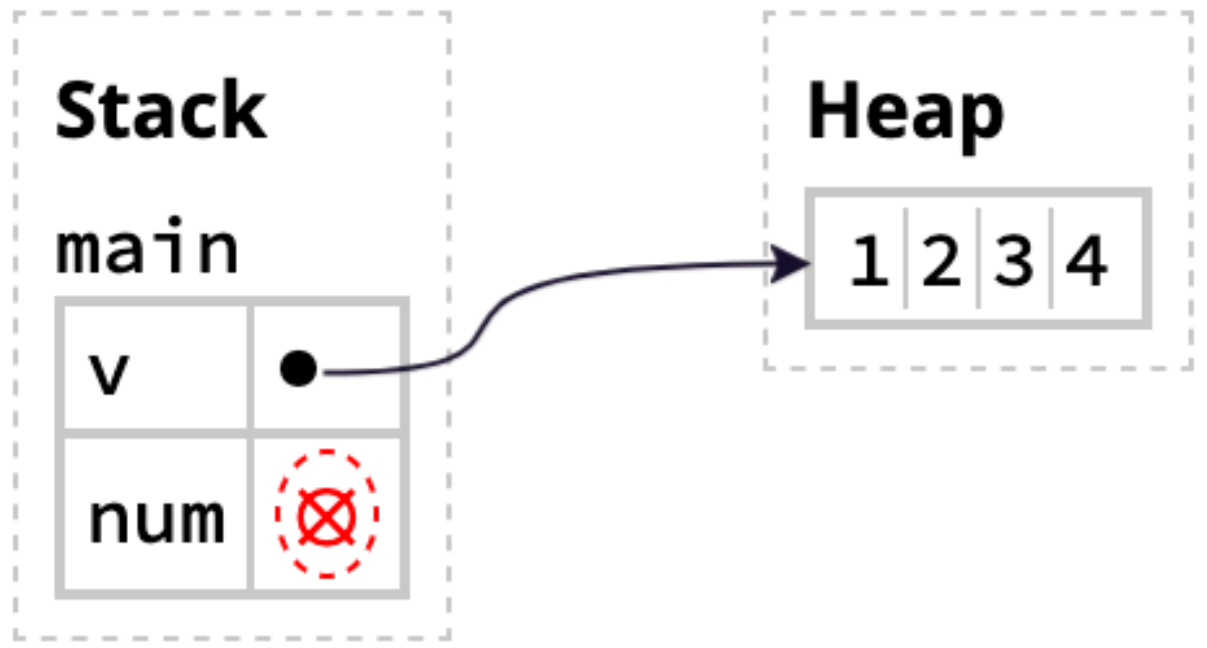
If Rust allowed this, dereferencing the reference would be **undefined behavior**

```
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



Pointer safety principle

“Data must never be aliased and mutated at the same time”

For Boxes, the pointers are moved rather than copied so only one Box owns any given piece of heap data (no aliases)

References are non-owning pointers: they create temporary aliases

- Rust must disallow mutation while a reference is alive

Borrow Checker

Every variable has three kinds of **permissions**

- Read (R): data can be copied to another location
- Write (W): data can be modified in place
- Own (O): data can be mOved or drOpped

Creating a variable with let makes the variable RO

Creating a variable with let mut makes the variable RWO

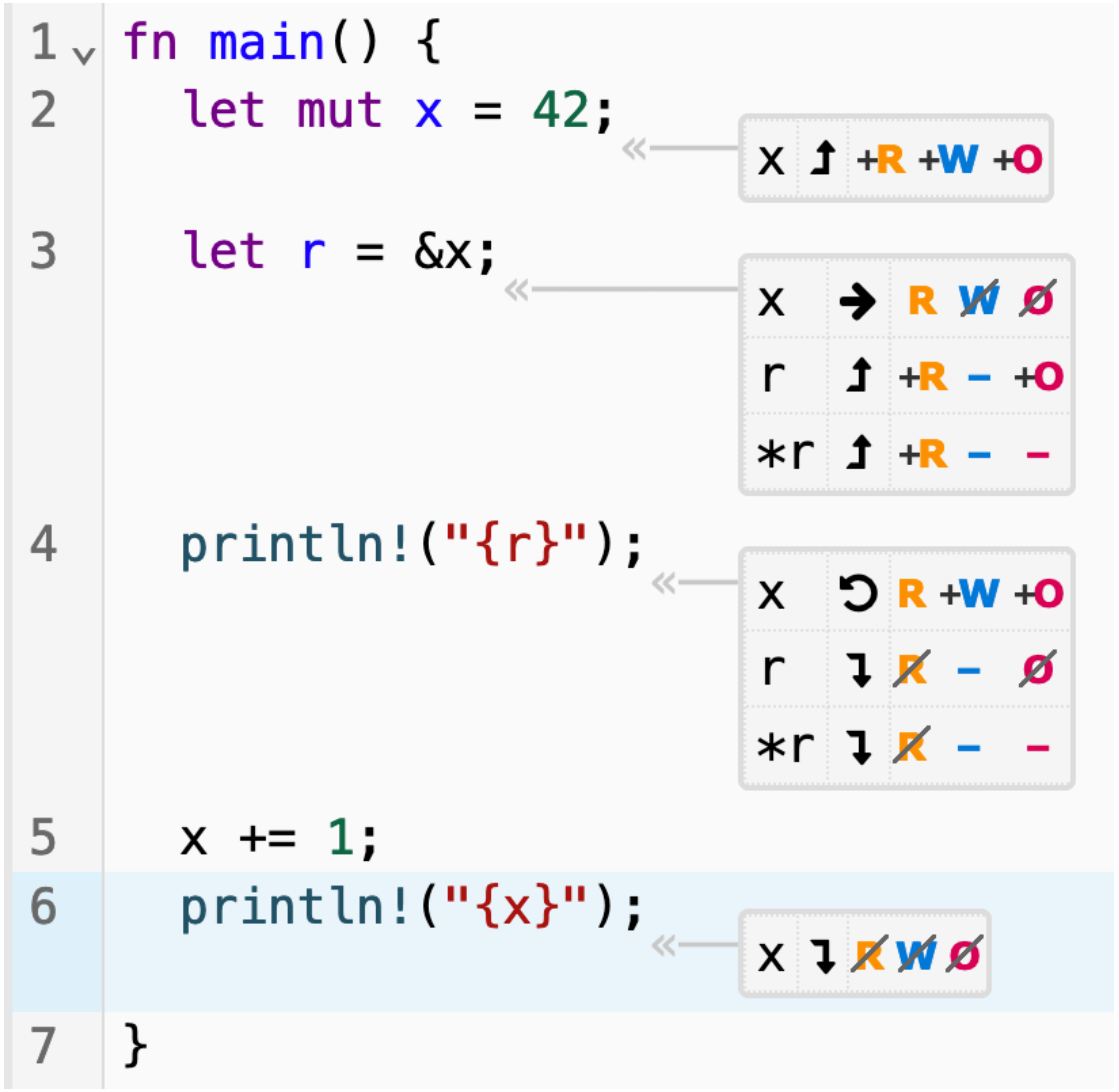
Rust checks that variables have appropriate permissions each place they are used at compile time using the Borrow Checker

References change permissions

Creating a reference to a variable temporarily removes W and O permissions from the variable

This enforces the pointer safety principle, “data must never be aliased and mutated at the same time,” by disallowing mutation while the reference is alive and aliasing the value

After the reference’s final use, the variable’s permissions are regained



```
fn main() {
    let mut x = 42;

    let r = &x;

    println!("{r}");

    x += 1;
    println!("{x}");
}
```

x	↑	+R	+W	+O

x	→	R	~~W~~	~~O~~
r	↑	+R	–	+O
*r	↑	+R	–	–

x	↻	R	+W	+O
r	↓	~~R~~	–	~~O~~
*r	↓	~~R~~	–	–

x	↓	~~R~~	~~W~~	~~O~~

Why does Rust prevent this code from compiling in terms of permissions? (What would happen if it didn’t prevent it?)

```rust
let mut v: Vec<i32> = vec![10, 20, 30];
let r: &i32 = &v[0];
v.push(40); // ERROR on this line
println!("{}", *r);
```

A. v never had W permission

B. v lost W permission when the reference was created

C. v lost O permission when the reference was created

D. *r doesn’t have R permission

E. r doesn’t have R permission

Why do references cause O permission to be temporarily dropped?

A. A limitation of Rust’s analysis

B. Moving or dropping the value pointed to by the reference would cause the reference to point at invalid memory

C. O permission is needed for writing; writing and aliasing is disallowed by the pointer safety principle so O permission is dropped while the reference is alive

Mutable references

We can create mutable references that allow modification using &mut var

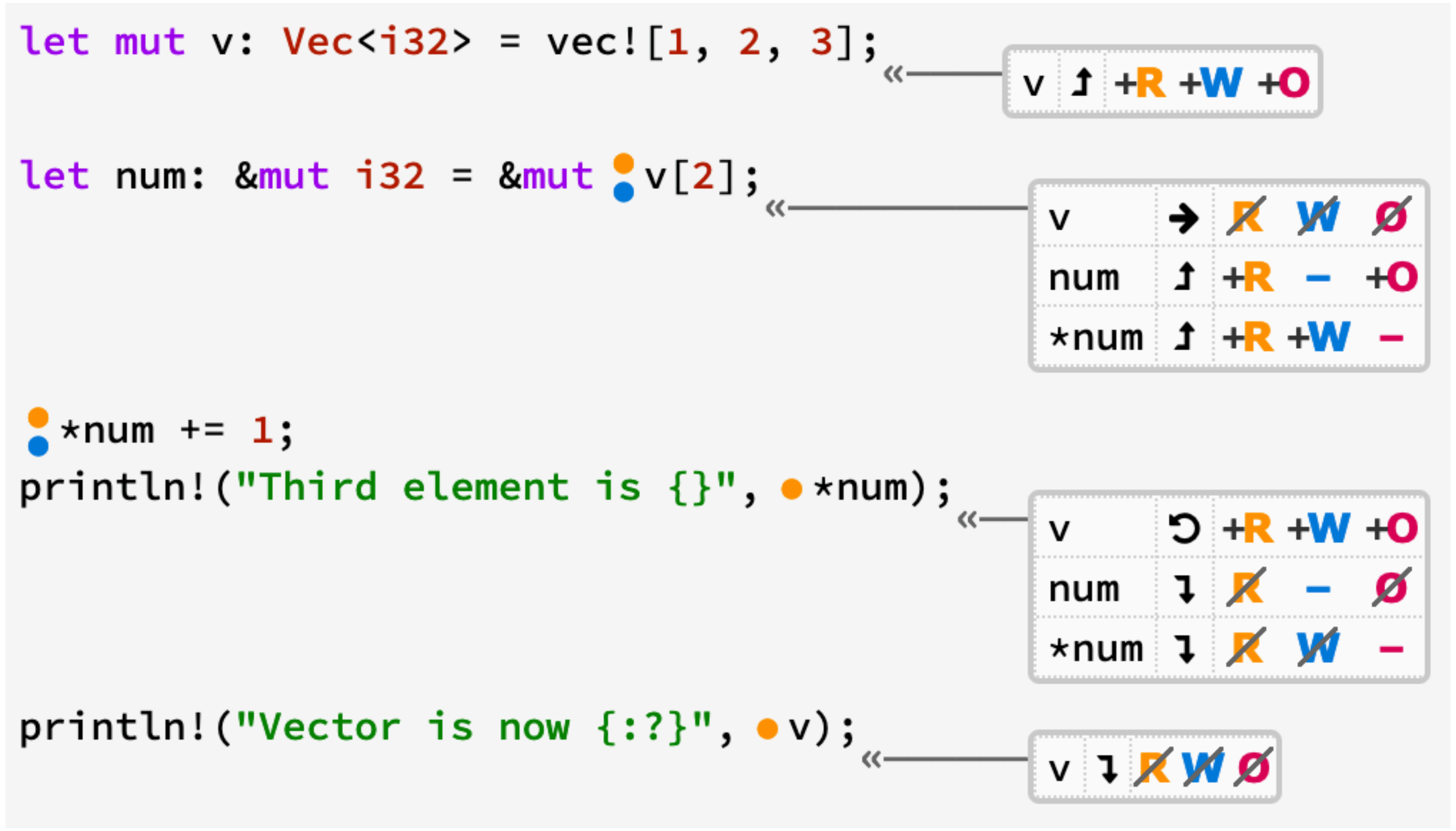
```rust
fn append_bang(s: &mut String) {
    s.push('!');
}

fn main() {
    let mut msg = String::from("References aren't so tricky");
    append_bang(&mut msg);
    println!("{msg}");
}
```

Mutable references remove RWO

When creating a mutable reference, RWO are temporarily dropped on the underlying variable

When num is created, v loses RWO

After the last use of num, v regains RWO



```
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~~

Pointer safety principle with references

PSP: Data must never be aliased and mutated at the same time

Rust allows a single mutable reference (&mut var) to a variable; OR
any number of shared references (&var)

When any reference is alive, the variable does not have WO (because the reference is an alias so the value must not be mutated through the variable)

When a mutable reference is alive, the variable does not have any permissions (because the mutable reference allows mutation so aliases must not be allowed)

Data must outlive references

It’s not possible to return a reference to a stack variable because the variable would be dropped at the end of the function so the reference would point to invalid memory

Rustc tracks an object’s lifetime to ensure all references are dropped before the underlying data is dropped

- This can be one of the most difficult parts of Rust!

I got an error, now what?

When you get a borrow checker error, it means one of two things

1. Your code is trying to do something actually unsafe and the borrow checker just preventing it! This is the most common reason

2. Your code is actually safe, but Rust doesn’t know how to prove it. This is *significantly* less common

The first step is to examine the code closely to determine which it is.

I got an error, now what?

Ask yourself:

Could this code cause undefined behavior by trying to modify/reallocate memory while a reference to it is held? If so, that's a bug!

Does this code try to keep a reference to data that has been dropped? If so, that's a bug!

Is the code trying to mutate an argument through a shared (i.e., not mutable) reference? Does the caller expect the argument to be modified? If so, make the reference mutable. If not, that's a bug!

I got an error, now what?

In some cases, you can clone() the data from the reference and modify that

With arrays, you can work with indices rather than references

Sometimes (e.g., when working with arrays), it really can be a limitation of Rust's analysis.

- Creating a reference to any element in an array precludes modifying any other element of the array while the reference is alive