Memory Safety

Andreea Costea

Delft University of Technology

2025-09-09

TU Delft

Today

The Pillars of Memory Safety in Rust

1. Ownership
2. References
3. Mutability
4. Lifetimes
5. Slices
6. A sample of Enum types (more next lecture)

Git Demo

Ownership

```rust
let x = vec![1, 2, 3];                    // L2
let y = x;                                // L3
println!("Here's your vector {x:?}");     // L4
```

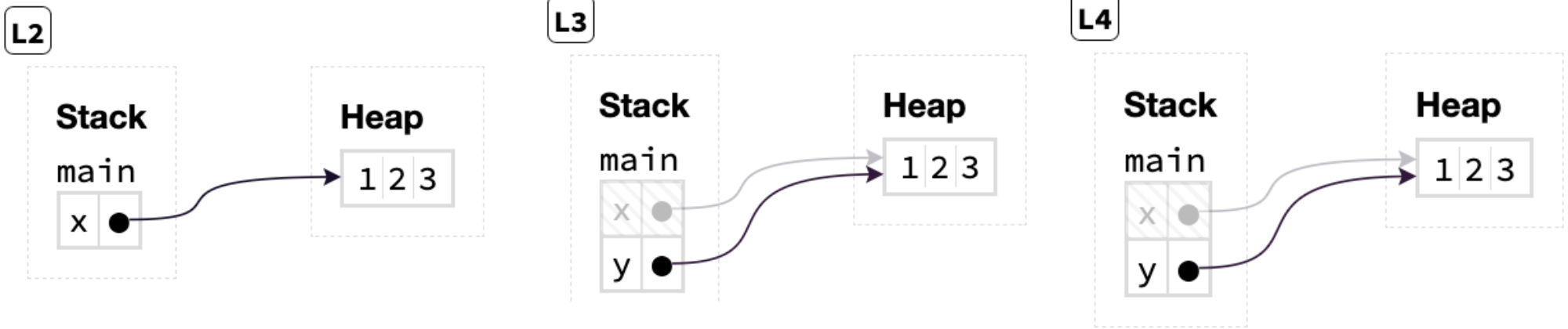
- x is a stack-allocated variable representing the pointer to the buffer stored in the heap, plus length and capacity metadata.

```rust
let x = vec![1, 2, 3];                    // L2
let y = x;                                // L3
println!("Here's your vector {x:?}");     // L4
```

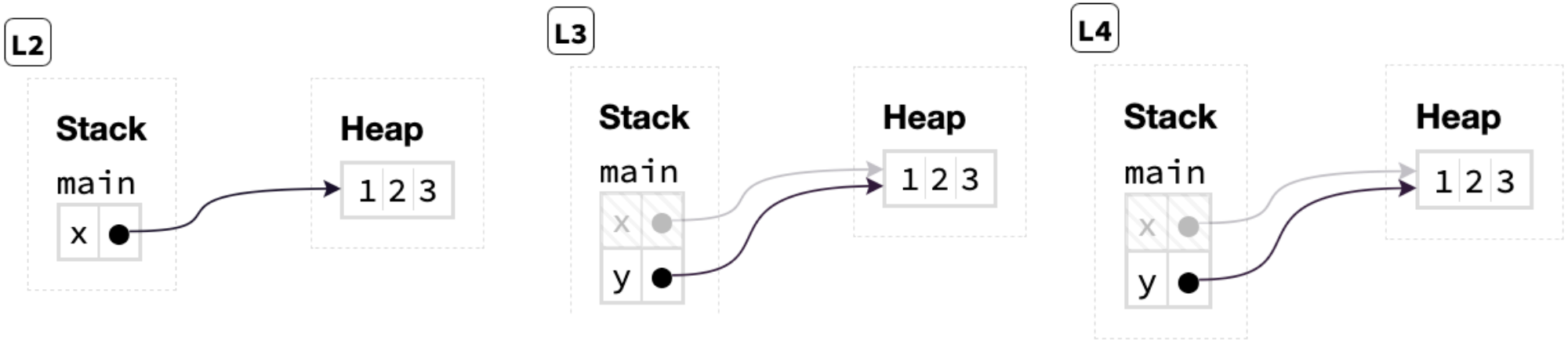
- x is a stack-allocated variable representing the pointer to the buffer stored in the heap, ~~plus length and capacity metadata~~ (for brevity of this presentation).

```rust
let x = vec![1, 2, 3];                    // L2
let y = x;                                // L3
println!("Here's your vector {x:?}");     // L4
```

```rust
let x = vec![1, 2, 3];                    // L2
let y = x;                                // L3
println!("Here's your vector {x:?}");     // L4
```

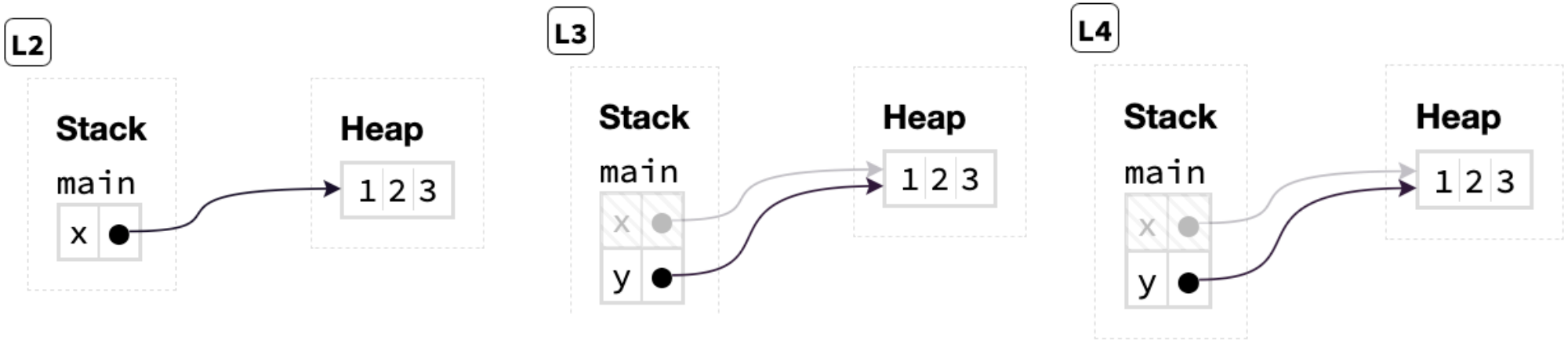


```rust
let x = vec![1, 2, 3];                    // L2
let y = x;                                // L3
println!("Here's your vector {x:?}");     // L4
```



L2: x binds the value `vec![1, 2, 3]` – x is the **owner** of that value

```rust
let x = vec![1, 2, 3];                    // L2
let y = x;                                // L3
println!("Here's your vector {x:?}");     // L4
```



L2: x binds the value `vec![1, 2, 3]` – x is the **owner** of that value

L3: *ownerships moves* from x to y (**MOVE SEMANTICS**)

Why is Rust doing this?

- **Accountability**: someone—the owner—has to be responsible for freeing the memory.

Why is Rust doing this?

- **Accountability**: someone—the owner—has to be responsible for freeing the memory.
  - every value has an owner (for simplicity here, owner is a variable).
  - every variable has a scope.
  - when the owner goes out of scope, the value will be dropped (memory is freed).

Why is Rust doing this?

- **Accountability**: someone—the owner—has to be responsible for freeing the memory.
  - every value has an owner (for simplicity here, owner is a variable).
  - every variable has a **scope**.
  - when the owner goes out of scope, the value will be dropped (memory is freed).

```rust
fn main(){
    let x = vec![1, 2, 3];
    let y = x;
    println!("Here's your vector {y:?}");
}
```

- Scope of `x`: line 2 to end of `main`
- Scope of `y`: line 3 to end of `main`

```rust
fn length(v: Vec<i32>) -> usize {
  v.len()
}
fn main(){
  let x = vec![1, 2, 3];
  let y = length(x);
  println!("The length of the vector is {y:?}");
}
```

- Scope of `x`: line 5 to end of `main`
- Scope of `y`: line 6 to end of `main`
- Scope of `v`: line 1 to end of `foo`

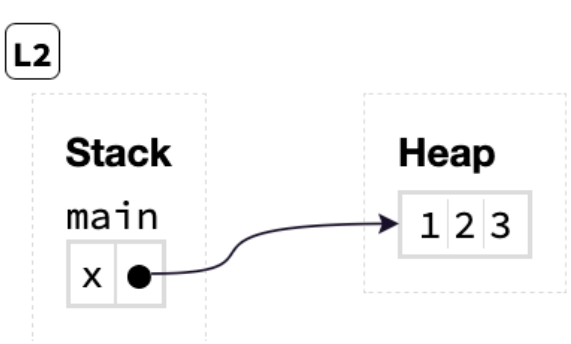
Why is Rust doing this?

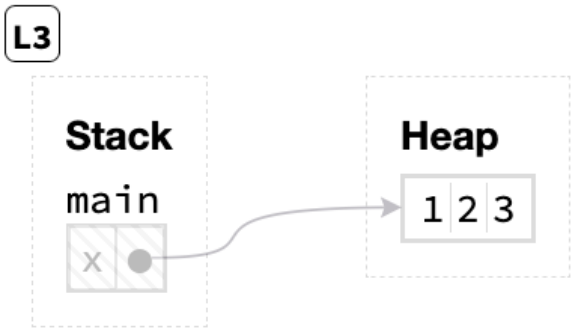
- **Accountability**: someone—the owner—has to be responsible for freeing the memory.
  - every value has an owner (for simplicity here, owner is a variable).
  - every variable has a scope.
  - when the owner goes out of scope, the **value will be dropped** (memory is freed).

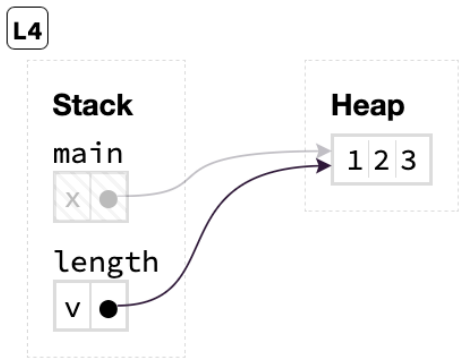
```rust
fn length(v: Vec<i32>) -> usize {      // L4
  v.len()
}                                      //L6
fn main(){
  let x = vec![1, 2, 3];               // L2
  let y = length(x);                   // L3  L7
  println!("The length of the vector is {y:?}");
}                                      // L9
```

```rust
fn length(v: Vec<i32>) -> usize {      // L4
  v.len()
}                                      //L6
fn main(){
  let x = vec![1, 2, 3];               // L2
  let y = length(x);                   // L3  L7
  println!("The length of the vector is {y:?}");
}                                      // L9
```

Rust



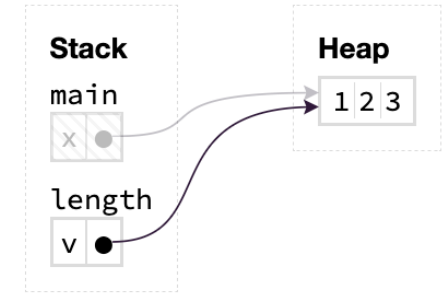




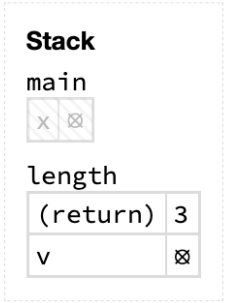
```rust
fn length(v: Vec<i32>) -> usize {      // L4
  v.len()
}                                      //L6
fn main(){
  let x = vec![1, 2, 3];               // L2
  let y = length(x);                   // L3  L7
  println!("The length of the vector is {y:?}");
}                                      // L9
```

Rust

L4



L6



L7

L9

```rust
1 fn length(v: Vec<i32>) -> usize {      // L4
2   v.len()
3 }                                      //L6
4 fn main(){
5   let x = vec![1, 2, 3];               // L2
6   let y = length(x);                   // L3  L7
7   println!("The length of the vector is {y:?}");
8 }                                      // L9
```

- every data structure that allocates memory on the heap has a `drop` method.
- when a variable goes out of scope, its `drop` method is called.
- `Vec`'s `drop` deallocates the heap buffer at line 3

Safety Principle:

Every value in Rust has a variable that’s called its owner. There can only be one owner at a time. When the owner goes out of scope, the value is dropped.

Question:

Where does the value owned by `x` go out of scope?

Question:

Where does the value owned by `x` go out of scope?

```rust
fn main(){
 let x = vec![1, 2, 3];
 println!("x is {x:?}");
}
```

Question:

Where does the value owned by x go out of scope?

```rust
fn main(){
 let y = vec![1, 2, 3];
 if y.len() > 2 {
    let x = y;
    println!("x is {x:?}");
 }
 println!("end of main");
}
```

Question:

Where does the value owned by x go out of scope?

```rust
fn main(){
 let y = 3;
 let x = y;
 println!("y = {y:?}");
}
```

Question:

Where does the value owned by `x` go out of scope?

```rust
fn main(){
 let y = 3;
 let x = y;
 println!("y = {y:?}");
}
```

- the values on the stack are **copied**, not moved

Question:

Where does the value owned by `x` go out of scope?

```rust
fn main(){
 let y = 3;
 let x = y;
 println!("y = {y:?}");
}
```

L2

Stack

`main`

y	3

L3

Stack

`main`

y	3
x	3

L5

Stack

`main`

(empty frame)

Recap

- Every value has a single owner
- When the owner goes out of scope, the value is dropped
- Ownership can be moved between variables
- Some types (scalar types) are `Copy`, and don't move, but copy instead

But what if we want this?

```rust
fn length(v: Vec<i32>) -> usize { v.len() }
fn main(){
  let x = vec![1, 2, 3];
  let y = length(x);
  println!("The length of {x:?} is {y:?}");
}
```

But what if we want this?

```rust
fn length(v: Vec<i32>) -> usize { /*L4*/ v.len() }
fn main(){
  let x = vec![1, 2, 3];
  let y = length(x.clone());     // L7 DEEP COPY
  println!("The length of {x:?} is {y:?}");
}
```

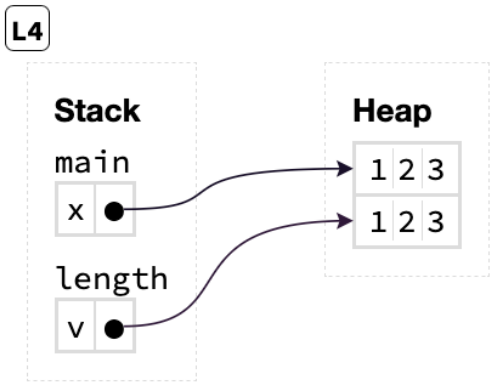
But what if we want this?

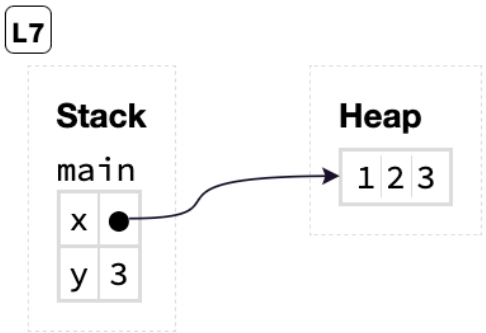
```rust
fn length(v: Vec<i32>) -> usize { /*L4*/ v.len() }
fn main(){
  let x = vec![1, 2, 3];
  let y = length(x.clone());     // L7 DEEP COPY
  println!("The length of {x:?} is {y:?}");
}
```

- `clone` duplicates the value on the heap (deep copy)

But what if we want this?

```rust
fn length(v: Vec<i32>) -> usize { /*L4*/ v.len() }
fn main(){
    let x = vec![1, 2, 3];
    let y = length(x.clone());     // L7 DEEP COPY
    println!("The length of {x:?} is {y:?}");
}
```





Disadvantages of Cloning

- Using clone we double the amount of memory needed
- Cloning takes `O(n)` time for a vector of `n` elements
- Can we do better?

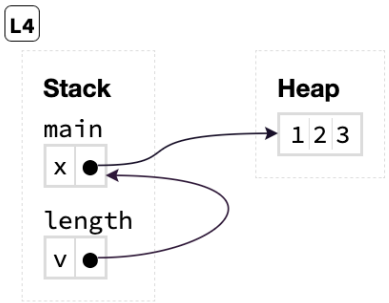
Borrowing

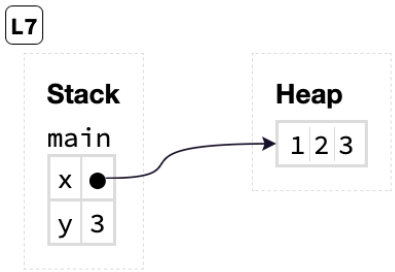
But what if we want this?

```rust
fn length(v: &Vec<i32>) -> usize { /*L4*/ v.len() }
fn main(){
  let x = vec![1, 2, 3];
  let y = length(&x);                  // L7 create a REFERENCE
  println!("The length of {x:?} is {y:?}");
}
```

But what if we want this?

```rust
fn length(v: &Vec<i32>) -> usize { /*L4*/ v.len() }
fn main(){
    let x = vec![1, 2, 3];
    let y = length(&x);                 // L7 create a REFERENCE
    println!("The length of {x:?} is {y:?}");
}
```





But what if we want this?

```rust
fn length(v: &Vec<i32>) -> usize { /*L4*/ v.len() }
fn main(){
  let x = vec![1, 2, 3];
  let y = length(&x);                // L7 create a REFERENCE
  println!("The length of {x:?} is {y:?}");
}
```

- We call this **borrowing**
- v borrows the value `vec![1, 2, 3]`, x still owns it

But what if we want this?

```rust
fn length(v: &Vec<i32>) -> usize { /*L4*/ v.len() }
fn main(){
  let x = vec![1, 2, 3];
  let y = length(&x);                // L7 create a REFERENCE
  println!("The length of {x:?} is {y:?}");
}
```

- We call this **borrowing**
- v borrows the value `vec![1, 2, 3]`, x still owns it
- v does own *something* though
  - **all** values have an owner
  - v just owns a value that's a reference, not the real `vec![1, 2, 3]`

Watch out though! We can only reference something that still exists.

```rust
fn example() -> &Vec<i32> {
    let a = vec![1, 2, 3];
    return &a;
    // a goes out of scope here
}

fn main() {
    // what are we pointing to?
    let ref_to_a = example();
}
```

Borrowing

```rust
fn main() {
    let x;
    {
        let y = vec![1, 2, 3];
        x = &y;
    }
    println!("{x}");
}
```

Borrowing

```rust
fn main() {
    let x;
    {
        let y = vec![1, 2, 3];
        x = &y;
    }
    println!("{x}");
}
```

- y is out of scope before the last use of x.
- y does not live long enough.

Safety Principle:

The lender needs to outlive all of its (alive) references: a reference lives from the place it started borrowing until its last use.

Types with Behaviours

The `&self` means we get a reference to the value when we call the method.

```rust
struct A;
impl A {
  // takes a reference to Self
  fn do_something_with_a(&self) {}
}
...
let x = A;
x.do_something_with_a(); // BORROW
// x still available
```

Types with Behaviours

We can also make a method takes onwership (useful for converting values)

```rust
struct A; struct B;
impl A {
  // takes ownership of Self
  fn do_something_with_a(self) -> B {...}
}
...
let x = A;
let b = x.do_something_with_a();  //MOVE
// x no longer available
```

References are `Copy`

```Rust
let x = vec![1, 2, 3]

let a = &x;
let b = a;

// all fine!
println!("{:?}", a);
println!("{:?}", b);
println!("{:?}", x);
```

Once we have one reference, it doesn't matter how many more we create!

Mutability

Mutability

A binding is either mutable, or not

```rust
let x = 3;
let mut y = 3;

x = 4; // illegal
y = 5; // ok!
```

```rust
let x = vec![1, 2, 3];
// x.push(4) doesn't work

// move to a mutable binding
let mut y = x;
// works just fine
y.push(4)
```

Question:

Why is it ok to add mutability to a value later on?

A borrow cannot mutate

```rust
fn add_four(y: &Vec<i32>) {
  // error!
  y.push(4);
}
fn main() {
  let mut x = vec![1, 2, 3];
  add_four(&x);
}
```

What if we want to change a value in a function?

- we could move owenrship:

```rust
// move the vector to this function
fn add_four(mut y: Vec<i32>) -> Vec<i32> {
    y.push(4);
    y     // and move back again
}

fn main() {
    let mut x = vec![1, 2, 3];
    x = add_four(x);
}
```

What if we want to change a value in a function?

- Or we use a *mutable reference*

```rust
fn add_four(y: &mut Vec<i32>) {
    y.push(4);
}

fn main() {
    let mut x = vec![1, 2, 3];
    // &mut x only possible if x is mutable
    add_four(&mut x);
}
```

Mutability

Mutable references aren't like normal references

- You can't copy them:

```rust
let mut x = vec![1, 2, 3];

let a = &mut x;
let b = a; // a moved into b, not copied

// so a is not valid anymore here
a.push(4);
b.push(5);
```

Mutability

Mutable references aren't like normal references

- You can't copy them
- You can't have two at the same time at all!

```rust
let mut x = vec![1, 2, 3];
let a = &mut x;
let b = &mut x; // second reference to x

a.push(4);
b.push(5);
```

Error: cannot borrow x as mutable more than once at a time

Mutability

Mutable references aren't like normal references

- You can't copy them
- You can't have two at the same time
- Nor can you have a mutable and normal reference at the same time!

```rust
let mut x = vec![1, 2, 3];
let a = &mut x;
let b = &x; // *immutable* reference to x

a.push(4);
println!("{:?}", b);
```

Error: cannot borrow `x` as immutable because it is also borrowed as mutable.

Mutability

Mutable references aren't like normal references

- You can't copy them
- You can't have any other reference at the same time!

A better name for a "mutable reference" is an "exclusive reference"

Question:

But why?

Example 1: growing vectors

```rust
let mut x = vec![1, 2, 3]

// first reference, to an element
let first_elem = &x[0];
// second reference, mutable this time
x.push(4);      // COMPILATION ERROR

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

Example 2: copying elements:

```rust
fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {
  for i in 0..src.len() { dst[i] = *src + 1; }
}

fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {
  let value = *src + 1;
  for i in 0..src.len() { dst[i] = value; }
}
```

Question:

Are these functions the same?

Example 2: copying elements—What if `src` is an element in `dst`?

```Rust
fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {
  for i in 0..src.len() { dst[i] = *src; }
}

fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {
  let value = *src;
  for i in 0..src.len() { dst[i] = value; }
}

let mut x = vec![1, 2, 3];
fill_vector_with_ref(&x[1], &mut x);
```

Mutability

But Rust would reject this program.

```rust
let mut x = vec![1, 2, 3];
// obviously wrong
// mutable *and* immutable reference at the same time
fill_vector_with_ref(&x[1], &mut x);
```

Mutability

Safety Principle:

A mutable borrow can only be created if its lender has no other borrows living at that time.

Mutability

Safety Principle:

A mutable borrow can only be created if its lender has no other borrows living at that time.

Safety Principle:

The lender cannot be modified as long as one of its (shared) borrowers still lives.

Example 1: growing vectors - **How can we fix this error?**

```rust
let mut x = vec![1, 2, 3]

// first reference, to an element
let first_elem = &x[0];
// second reference, mutable this time
x.push(4);      // COMPILATION ERROR

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

Example 1: growing vectors - **How can we fix this error?**

```rust
let mut x = vec![1, 2, 3]

// SWAP the order of the borrowing creation
x.push(4);
// first reference, to an element
let first_elem = &x[0];

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

Example 1: growing vectors - **How can we fix this error?**

```rust
let mut x = vec![1, 2, 3]

// SWAP the order of the borrowing creation
x.push(4);          // The life of &mut x STARTS and ENDS here
// first reference, to an element
let first_elem = &x[0];       // The life of &x STARTS here

println!("{}", first_elem);  // The life of &x ENDS here
```

- lifetimes are associated with references
- a reference lives until it is last used

Recap

Ownership: Who owns what?

- Every value in Rust has a single owner — usually a variable.
- When the owner goes out of scope, the value is dropped (freed).
- Ownership can be moved (transfer ownership to another variable) or copied (if the type implements Copy).

Recap

Ownership: Who owns what?

- Every value in Rust has a single owner — usually a variable.
- When the owner goes out of scope, the value is dropped (freed).
- Ownership can be moved (transfer ownership to another variable) or copied (if the type implements Copy).

References: Borrowing instead of owning

- You can borrow a value instead of taking ownership using & (shared reference) or &mut (mutable reference).
- Shared borrows (&T): many at once, but read-only.
- Mutable borrows (&mut T): only one at a time, and no shared borrows while it's active.
- These rules ensure no data races or use-after-free.

Recap

The three rules that matter most

- **Ownership rule**: exactly one owner at a time.
- **Borrowing rules**:
  - Many immutable borrows allowed, OR
  - One mutable borrow allowed, but not both.
- **Lifetime rule**: a reference must not outlive its owner.

Slices

Sometimes you want to reference more than one thing at a time:

```Rust
let x = vec![1, 2, 3, 4];

let a: &[u32] = &x[0..2] // index 0, and 1 (excluding 2)
let b = &x[2..]          // elements at indexes starting from 2

for i in b {             // you can iterate over a slice
  println!("{i}");
}

println!("{}", a.len()); // or get its length
```

Slices can be mutable:

```rust
let mut x = vec![1, 2, 3, 4];

// index 0, and 1 (excluding 2)
let a: &mut [u32] = &mut x[0..2]
for i in a {
  *i += 3;
}

// prints 4, 5, 3, 4
println!("{:?}", x);
```

Some things coerce to slices:

```rust
// input is a slice
fn sum(res: &[u32]) -> u32 {  /* ... */ }

// but we can call it with a vector!
let x = vec![1, 2, 3];
sum(&x);
// or a bit of a vector
sum(&x[1..]);
// or an array
sum(&[1, 2, 3]);
```

So writing `sum` like this is more flexible

Slices

This gives us a fun way to write `sum`:

```rust
fn sum(input: &[u32]) -> u32 {
  if input.is_empty() {
    0
  } else {
    // add element 0 to everything after element 0
    input[0] + sum(&input[1..])
  }
}
```

Works for anything that looks like a sequence of u32, like vectors

Enums

Enums

- Last lecture: all about types
- Next lecture: all about enum types

But here are the basics, so you can get started using them

Enums

Question:

How many possible values does a `bool` have?

Enums

Question:

How many possible values does a `u8` have?

Enums

Question:

How many possible values does a u32 have?

Enums

Question:

How many possible values does this type have?

```Rust
struct X {
    a: bool,
    b: bool,
}
```

Enums

- We call a struct a “product type”.
- If type `A` has `n` possible values
- If type `B` has `m` possible values
- Then a struct with `A` and `B` in it has $n \times m$ possible values

Enums

Sometimes, you know that not all values are possible.

```rust
// NOTE: only 1-7 are valid
type WeekDay = u8;


// ???
let x: WeekDay = 8;
```

Enums

Sometimes, you know that not all values are possible.

```rust
// Only has 7 possible values
enum WeekDay {
    Monday,  Tuesday,   Wednesday,
    Thursday,  Friday,  Saturday, Sunday,
}

// we can only choose one of the valid values!
let x: WeekDay = WeekDay::Monday;
```

Enums

```rust
// Only has 7 possible values
enum WeekDay {
    Monday,  Tuesday,   Wednesday,
    Thursday,  Friday,  Saturday, Sunday,
}

// we can only choose one of the valid values!
let x: WeekDay = WeekDay::Monday;
```

Enums

Unlike in many other programming languages, enums can have values

```rust
enum IpAddress {
    Ipv4([u8; 4]),
    Ipv6([u8; 16]),
}

let x: IpAddress = IpAddress::Ipv4([127, 0, 0, 1]);
```

Enums

How many possible values?

```rust
enum IpAddress {
    Ipv4([u8; 4]), // 2^32 ~= 4 billion
    Ipv6([u8; 16]), // 2^128 ~= a lot
}
```

In total: $2^{32} + 2^{128}$

Enums are sometimes called “sum types”

Enums

Another example: `Option<T>`

```rust
enum Option<T> {
    Some(T),
    None
}

// 257 possible values
// 256 if Some, or one more: None
let x: Option<u8> = Some(3);
```

Assignment: 5 minutes

- Create an enum for a `JSON` value called `Value`
- a `JSON` value is either:
  - a floating point number
  - a string
  - true
  - false
  - null
  - a list of other `JSON` values
  - a json object, `std::collections::HashMap<String, Value>`

JSON spec

https://www.json.org/json-en.html