Ownership and References

Jonathan Dönszelmann & Vivian Roest

Delft University of Technology

2024-11-18

Last week

- Data Types
  - Describing sizes of things in memory
  - Describing the behavior of values
  - Expressing proofs

Today

Four slightly different topics:

1. Ownership and references
2. Mutability
3. Slices
4. A sample of Enum types (more next lecture)

Ownership

Ownership

Over the last week you may have seen this:

```rust
fn sum(y: Vec<i32>) -> i32 {
  // ...
}

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

Question:

Why doesn't this work?

Ownership

- `vec![1, 2, 3]` is a value
- it lives somewhere in memory

Ownership

- `vec![1, 2, 3]` is a value
- it lives somewhere in memory

- `x` is a “binding”.
- `x` binds a value, like `vec![1, 2, 3]`

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

Rust

Ownership

- `vec![1, 2, 3]` is a value
- it lives somewhere in memory

- `x` is a “binding”.
- `x` binds a value, like `vec![1, 2, 3]`

- a binding has a certain scope
- the scope of `x` is the main function’s scope

Rust

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

Ownership

- `vec![1, 2, 3]` is a value
- it lives somewhere in memory

- `x` is a “binding”.
- `x` binds a value, like `vec![1, 2, 3]`

- a binding has a certain scope
- but the scope could be different, like here

```rust
fn main() {
  if true {
    let x = vec![1, 2, 3];
  }
  // ...
}
```

Ownership

The Rules Of Rust:

- **Every** value (like `vec![1, 2, 3]`)
- at a single point in the program
- has a **single** binding (read “variable name”)
- in a **single** scope
- This binding is called the owner

Rust

```rust
fn main() {
  // x owns vec![1, 2, 3] in the scope of `fn main`
  let x = vec![1, 2, 3];
}
```

Ownership

Rust

```rust
fn main() {
    // x owns vec![1, 2, 3] in the scope of `fn main`
    let x = vec![1, 2, 3];
    // the value is moved
    // y now owns vec![1, 2, 3]
    let y = x;
}
```

Ownership can move, `x` no longer is the owner

Ownership

```rust
fn other(y: Vec<i32>) {
    // now y owns the value
}

fn main() {
  // x owns vec![1, 2, 3] in the scope of `fn main`
  let x = vec![1, 2, 3];
  // the value is moved
  other(x);
}
```

Ownership can move, from function to function

Ownership

```rust
fn main() {
    // x owns vec![1, 2, 3] in the scope of `fn main`
    let x = vec![1, 2, 3];

    // x goes out of scope
    // vec![1, 2, 3] is destroyed
}
```

Rust

If the owner goes out of scope, the value is destroyed

Ownership

Rust

```rust
fn other(y: Vec<i32>) {
    // now y owns it!
    // and vec![1, 2, 3] is deleted here
}

fn main() {
    // x owns vec![1, 2, 3] in the scope of `fn main`
    let x = vec![1, 2, 3];
    // the value is moved
    other(x);
}
```

If the owner goes out of scope, the value is destroyed

Ownership

- Every binding must go out of scope *somewhere*
- So every value is deleted somewhere*

Rust

```rust
use std::mem;

fn main() {
  // x owns vec![1, 2, 3] in the scope of `fn main`
  let x = vec![1, 2, 3];
  // x is moved into the forget function
  // but `forget` promises to never delete the value
  mem::forget(x);
}
```

Ownership

But what if we want this?

```rust
fn sum(y: Vec<i32>) -> i32 {
  // ...
}

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

Ownership

But what if we want this?

- `clone` takes a value, and **duplicates** that value

```rust
// x owns vec![1, 2, 3]
let x = vec![1, 2, 3];
// y now owns a new duplicated *different* instance of `vec![1, 2, 3]`
// x also still owns the original instance
let y = x.clone();
```

Ownership

But what if we want this?

Rust

```rust
fn sum(y: Vec<i32>) -> i32 {
  // ...
}

fn main() {
  let x = vec![1, 2, 3];
  // so clone here!
  let s = sum(x.clone());
  println!("sum of {x:?} is {s}");
}
```

Ownership

Disadvantages

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

Ownership

Can't we just, like, not move `x` into the `sum` function?

```rust
fn sum(y: Vec<i32>) -> i32 {
  // ...
}

fn main() {
  let x = vec![1, 2, 3];
  // avoid moving here?
  let s = sum(x);
  println!("sum of {x:?} is {s}");
}
```

Ownership

Sure! use a reference

```rust
// add an `&` here
fn sum(y: &Vec<i32>) -> i32 {
  // ...
}

fn main() {
  let x = vec![1, 2, 3];
  // use an `&` here
  let s = sum(&x);
  println!("sum of {x:?} is {s}");
}
```

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

Ownership

- `y` does own *something* though
- **all** values have an owner
- `y` just owns a value that's a reference, not the real `vec![1, 2, 3]`
-

```rust
fn sum(y: &Vec<i32>) -> i32 {
    // y owns &vec![1, 2, 3]
    // it goes out of scope here, and the *reference* is deleted
    // not the original value
}
```

Ownership

`y` doesn't use as much memory as `vec![1, 2, 3]`

```rust
fn sum(y: &Vec<i32>) -> i32 {
  // ...
}

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

- it doesn't store the whole value
- it just stores where we can *find* the real value, in the stack of `main`
- this is called a pointer

Ownership

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();
}
```

So this does not compile!
https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=cb10ad88b0a86480772ee143322156cb

Ownership

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

```rust
fn main() {
    let x;

    {
        let y = vec![1, 2, 3];
        x = &y;
    }

    println!("{x}")
}
```

"y does not live long enough"
https://play.rust-lang.org/?version=stable&mode=debug&edition=2021&gist=5391df9eeaf4fadd71d0beb0052f868b

Ownership

References mostly act like owned values

Rust

```rust
let x = 10;

assert_eq!(x, x);
// does not compare locations, compares values
assert_eq!(&x, &x);

// we can just print a reference
// just like a value
println!("{}", &x);

// calling methods on values
x.ilog10()
// is the same as on references
(&x).ilog10()
```

Ownership

Last we saw types with “methods”, associated functions.

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();
// x still available
```

Ownership

We can also make a method take self “by value”

```rust
struct A;
impl A {
  // takes ownership of Self
  fn do_something_with_a(self) {}
}


let x = A;
x.do_something_with_a();
// x no longer available
```

Often useful when converting values
an operation like “turn A into B” destroys the old A, and we gives a new B
https://doc.rust-lang.org/stable/std/collections/struct.BinaryHeap.html#method.into_vec

Ownership

- I've been using `Vec` as an example everywhere
- I couldn't have used numbers
- because numbers are `Copy`.

```rust
let a = 3;
let b = a;

// a and b are still valid!
```

https://doc.rust-lang.org/stable/std/marker/trait.Copy.html

Ownership

Types that are `Copy`able are

- Simple to destroy
- Cheap to create more instances of
- Often very simple, like numbers or booleans

Ownership

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!

Ownership

Summary:

- Every value, at a point in the program, has a single binding that owns it
- This makes sure we know precisely when to deallocate memory
- `clone` duplicates a value explicitly
- Types that are `Copy` don't need cloning
- A reference can "borrow" a value, avoiding "move"ing it

Mutability

Mutability

A binding is either mutable, or not

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

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

Question:

Why do we have to mark mutability?

Mutability

- Lots of languages have this distinction (var vs const for example)
- Mutability is sometimes seen as a bit of an antipattern

When a variable is mutable, it could be changed *anywhere*

Rust

```rust
let mut res = 0;
while res < 10 {
  if x > 4 { res = 2; }
  if y < 2 && res < 4 {
    res = 8; x = 8;
  } else {
    res += 1;
  }
}
```

Hard to know with what values `x` and `y` this code even terminates

Mutability

- Lots of languages have this distinction (var vs const for example)
- Mutability is sometimes seen as a bit of an antipattern
- You don’t need mutable variables *that* often

Rust

```rust
fn even_sum(numbers: &Vec<i32>) -> i32 {
  let mut result = 0;
  for i in numbers {
    if i % 2 == 0 {result += i};
  }
  result
}
// vs
fn even_sum(numbers: &Vec<i32>) -> i32 {
  numbers.iter().filter(|i| i%2==0).sum()
}
```

Mutability

Mutability applies to a single binding

```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?

Mutability

A borrow cannot mutate

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

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

Question:

Why not?

Mutability

What if we want to change a value in a function?
we could use moving:

Rust

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

fn main() {
  let mut x = vec![1, 2, 3];
  // x must now be mutable for us to update it here
  x = add_four(x);
}
```

Mutability

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

Or we use a *mutable reference*

Rust
```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

```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

(which is why copying is not allowed)

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

```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

(which is why copying is not allowed)

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?

Mutability

Example 1: growing vectors

push takes `&mut self`: https://doc.rust-lang.org/stable/std/vec/struct.Vec.html#method.push

Rust

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

// first reference, to an element
let first_elem = &x[0];
// second reference, mutable this time
// pushing might mean growing the vector, which might
// change the location of the elements
x.push(4);

// the vector's data might have changed location!
// no clue if this reference is still valid
println!("{}", first_elem);
```

Mutability

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; }
}

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

Question:

Are these functions the same?

Mutability

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);
```

Rust

Mutability

Things get even worse when multiple threads are involved
Can they both mutate the same value? $\rightarrow$ Data races

In fact, some people start with explaining that this rule exists because of threading.

Read more on this:

- https://smallcultfollowing.com/babysteps/blog/2013/06/11/on-the-connection-between-memory-management-and-data-race-freedom/
- https://manishearth.github.io/blog/2015/05/17/the-problem-with-shared-mutability/

quote in that blogpost from kmc:

“My intuition is that code far away from my code might as well be in another thread, for all I can reason about what it will do to shared mutable state.”

Ownership

Summary:

- Bindings are mutable or not
- References are mutable or not
- Whenever something is mutably references, no other reference can exist

Mutability

Want to practice with this?

Weblab: Assignments - Week 1 - Types - All about Vecs

We’ll discuss in the lab tomorrow

Slices

Slices

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

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

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

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

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

Slices

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);
```

Slices

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

```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

```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

```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