

Ownership and References

Jonathan Dönszelmann & Vivian Roest

Delft University of Technology

2024-11-26

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:

```
1 fn sum(y: Vec<i32>) -> i32 {  
2     // ...  
3 }  
4  
5 fn main() {  
6     let x = vec![1, 2, 3];  
7     let s = sum(x);  
8     println!("sum of {x:?} is {s}");  
9 }
```

Rust

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]`

```
1 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

```
1 fn main() {  
2     let x = vec![1, 2, 3];  
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
- but the scope could be different, like here

```
1 fn main() {  
2     if true {  
3         let x = vec![1, 2, 3];  
4     }  
5     // ...  
6 }
```

Rust

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

```
1 fn main() {  
2     // x owns vec![1, 2, 3] in the scope of `fn main`  
3     let x = vec![1, 2, 3];  
4 }
```

Rust

Ownership

```
1 fn main() {  
2     // x owns vec![1, 2, 3] in the scope of `fn main`  
3     let x = vec![1, 2, 3];  
4     // the value is moved  
5     // y now owns vec![1, 2, 3]  
6     let y = x;  
7 }
```

Rust

Ownership can move, x no longer is the owner

Ownership

```
1 fn other(y: Vec<i32>) {  
2     // now y owns the value  
3 }  
4  
5 fn main() {  
6     // x owns vec![1, 2, 3] in the scope of `fn main`  
7     let x = vec![1, 2, 3];  
8     // the value is moved  
9     other(x);  
10 }
```

Rust

Ownership can move, from function to function

Ownership

```
1 fn main() {  
2     // x owns vec![1, 2, 3] in the scope of `fn main`  
3     let x = vec![1, 2, 3];  
4  
5     // x goes out of scope  
6     // vec![1, 2, 3] is destroyed  
7 }
```

Rust

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

Ownership

```
1  fn other(y: Vec<i32>) {  
2      // now y owns it!  
3      // and vec![1, 2, 3] is deleted here  
4  }  
5  
6  fn main() {  
7      // x owns vec![1, 2, 3] in the scope of `fn main`  
8      let x = vec![1, 2, 3];  
9      // the value is moved  
10     other(x);  
11 }
```

Rust

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*

```
1 use std::mem;
2
3 fn main() {
4     // x owns vec![1, 2, 3] in the scope of `fn main`
5     let x = vec![1, 2, 3];
6     // x is moved into the forget function
7     // but `forget` promises to never delete the value
8     mem::forget(x);
9 }
```

Rust

Ownership

But what if we want this?

```
1 fn sum(y: Vec<i32>) -> i32 {  
2     // ...  
3 }  
4  
5 fn main() {  
6     let x = vec![1, 2, 3];  
7     let s = sum(x);  
8     println!("sum of {x:?} is {s}");  
9 }
```

Rust

Ownership

But what if we want this?

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

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

Rust

Ownership

But what if we want this?

```
1  fn sum(y: Vec<i32>) -> i32 {  
2      // ...  
3  }  
4  
5  fn main() {  
6      let x = vec![1, 2, 3];  
7      // so clone here!  
8      let s = sum(x.clone());  
9      println!("sum of {x:?} is {s}");  
10 }
```

Rust

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?

```
1  fn sum(y: Vec<i32>) -> i32 {  
2      // ...  
3  }  
4  
5  fn main() {  
6      let x = vec![1, 2, 3];  
7      // avoid moving here?  
8      let s = sum(x);  
9      println!("sum of {x:?} is {s}");  
10 }
```

Rust

Ownership

Sure! use a reference

```
1 // add an `&` here
2 fn sum(y: &Vec<i32>) -> i32 {
3     // ...
4 }
5
6 fn main() {
7     let x = vec![1, 2, 3];
8     // use an `&` here
9     let s = sum(&x);
10    println!("sum of {x:?} is {s}");
11 }
```

Rust

- 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]`
-

```
1 fn sum(y: &Vec<i32>) -> i32 {  
2     // y owns &vec![1, 2, 3]  
3     // it goes out of scope here, and the *reference* is deleted  
4     // not the original value  
5 }
```

Rust

Ownership

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

```
1 fn sum(y: &Vec<i32>) -> i32 {  
2     // ...  
3 }  
4  
5 fn main() {  
6     let x = vec![1, 2, 3];  
7     let s = sum(&x);  
8     println!("sum of {x:?} is {s}");  
9 }
```

Rust

- 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.

```
1 fn example() -> &Vec<i32> {  
2     let a = vec![1, 2, 3];  
3     return &a;  
4     // a goes out of scope here  
5 }  
6  
7 fn main() {  
8     // what are we pointing to?  
9     let ref_to_a = example();  
10 }
```

Rust

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.

```
1  fn main() {  
2      let x;  
3  
4      {  
5          let y = vec![1, 2, 3];  
6          x = &y;  
7      }  
8  
9      println!("{x}")  
10 }
```

Rust

“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

```
1  let x = 10;
2
3  assert_eq!(x, x);
4  // does not compare locations, compares values
5  assert_eq!(&x, &x);
6
7  // we can just print a reference
8  // just like a value
9  println!("{}", &x);
10
11 // calling methods on values
12 x.ilog10()
13 // is the same as on references
14 (&x).ilog10()
```

Rust

Ownership

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

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

```
1 struct A;  
2 impl A {  
3     // takes a reference to Self  
4     fn do_something_with_a(&self) {}  
5 }  
6  
7  
8 let x = A;  
9 x.do_something_with_a();  
10 // x still available
```

Rust

Ownership

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

```
1  struct A;
2  impl A {
3      // takes ownership of Self
4      fn do_something_with_a(self) {}
5  }
6
7
8  let x = A;
9  x.do_something_with_a();
10 // x no longer available
```

Rust

Often useful when converting values

an operation like “turn A into B” destroys the old A, and we give 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`.

```
1 let a = 3;  
2 let b = a;  
3  
4 // a and b are still valid!
```

Rust

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

Ownership

Types that are Copyable are

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

Ownership

References are Copy:

```
1 let x = vec![1, 2, 3]
2
3 let a = &x;
4 let b = a;
5
6 // all fine!
7 println!("{:?}", a);
8 println!("{:?}", b);
9 println!("{:?}", x);
```

Rust

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

```
1 let x = 3;  
2 let mut y = 3;  
3  
4 x = 4; // illegal  
5 y = 5; // ok!
```

Rust

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*

```
1 let mut res = 0;
2 while res < 10 {
3     if x > 4 { res = 2; }
4     if y < 2 && res < 4 {
5         res = 8; x = 8;
6     } else {
7         res += 1;
8     }
9 }
```

Rust

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

```
1 fn even_sum(numbers: &Vec<i32>) -> i32 {  
2     let mut result = 0;  
3     for i in numbers {  
4         if i % 2 == 0 {result += i};  
5     }  
6     result  
7 }  
8 // vs  
9 fn even_sum(numbers: &Vec<i32>) -> i32 {  
10     numbers.iter().filter(|i| i%2==0).sum()  
11 }
```

Rust

Mutability

Mutability applies to a single binding

```
1 let x = vec![1, 2, 3];  
2 // x.push(4) doesn't work  
3  
4 // move to a mutable binding  
5 let mut y = x;  
6 // works just fine  
7 y.push(4)
```

Rust

Question:

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

Mutability

A borrow cannot mutate

```
1 fn add_four(y: &Vec<i32>) {  
2     // error!  
3     y.push(4);  
4 }  
5  
6 fn main() {  
7     let x = vec![1, 2, 3];  
8     add_four(&x);  
9 }
```

Rust

Question:

Why not?

Mutability

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

```
1  // move the vector to this function
2  fn add_four(mut y: Vec<i32>) -> Vec<i32> {
3      y.push(4);
4      // and move back again
5      y
6  }
7
8  fn main() {
9      let mut x = vec![1, 2, 3];
10     // x must now be mutable for us to update it here
11     x = add_four(x);
12 }
```

Rust

Mutability

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

Or we use a *mutable reference*

```
1 fn add_four(y: &mut Vec<i32>) {  
2     y.push(4);  
3 }  
4  
5 fn main() {  
6     let mut x = vec![1, 2, 3];  
7     // &mut x only possible if x is mutable  
8     add_four(&mut x);  
9 }
```

Rust

Mutability

Mutable references aren't like normal references

- You can't copy them:

```
1 let mut x = vec![1, 2, 3];  
2  
3 let a = &mut x;  
4 let b = a; // a moved into b, not copied  
5  
6 // so a is not valid anymore here  
7 a.push(4);  
8 b.push(5);
```

Rust

Mutability

Mutable references aren't like normal references

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

```
1 let mut x = vec![1, 2, 3];  
2  
3 let a = &mut x;  
4 let b = &mut x; // second reference to x  
5  
6 a.push(4);  
7 b.push(5);
```

Rust

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!

```
1 let mut x = vec![1, 2, 3];  
2  
3 let a = &mut x;  
4 let b = &x; // *immutable* reference to x  
5  
6 a.push(4);  
7 println!("{:?}", b);
```

Rust

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>

```
1  let mut x = vec![1, 2, 3]
2
3  // first reference, to an element
4  let first_elem = &x[0];
5  // second reference, mutable this time
6  // pushing might mean growing the vector, which might
7  // change the location of the elements
8  x.push(4);
9
10 // the vector's data might have changed location!
11 // no clue if this reference is still valid
12 println!("{}", first_elem);
```

Rust

Mutability

Example 2: copying elements:

```
1 fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {  
2     for i in 0..src.len() { dst[i] = *src; }  
3 }  
4  
5 fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {  
6     let value = *src;  
7     for i in 0..src.len() { dst[i] = value; }  
8 }
```

Rust

Question:

Are these functions the same?

Mutability

Example 2: copying elements:

What if `src` is an element in `dst`?

```
1 fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {  
2     for i in 0..src.len() { dst[i] = *src; }  
3 }  
4  
5 fn fill_vector_with_ref(src: &u32, dst: &mut Vec<u32>) {  
6     let value = *src;  
7     for i in 0..src.len() { dst[i] = value; }  
8 }  
9  
10 let mut x = vec![1, 2, 3];  
11 fill_vector_with_ref(&x[1], &mut x);
```

Rust

Mutability

But Rust would reject this program.

```
1 let mut x = vec![1, 2, 3];  
2 // obviously wrong  
3 // mutable *and* immutable reference at the same time  
4 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? → 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?

Web lab: 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:

```
1  let x = vec![1, 2, 3, 4];
2
3  // index 0, and 1 (excluding 2)
4  let a: &[u32] = &x[0..2]
5  // all elements at indexes starting from 2
6  let b = &x[2..]
7
8  // you can iterate over a slice
9  for i in b {
10     println!("{i}");
11 }
12
13 // or get its length
14 println!("{}", a.len());
```

Rust

Slices

Slices can be mutable:

```
1  let mut x = vec![1, 2, 3, 4];  
2  
3  // index 0, and 1 (excluding 2)  
4  let a: &mut [u32] = &mut x[0..2]  
5  for i in a {  
6      *i += 3;  
7  }  
8  
9  // prints 4, 5, 3, 4  
10 println!("{:?}", x);
```

Rust

Slices

Some things coerce to slices:

```
1 // input is a slice
2 fn sum(res: &[u32]) -> u32 {
3     // ...
4 }
5
6 // but we can call it with a vector!
7 let x = vec![1, 2, 3];
8 sum(&x);
9 // or a bit of a vector
10 sum(&x[1..]);
11 // or an array
12 sum(&[1, 2, 3]);
```

Rust

So writing `sum` like this is more flexible

Slices

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

```
1 fn sum(input: &[u32]) -> u32 {  
2     if input.is_empty() {  
3         0  
4     } else {  
5         // add element 0 to everything after element 0  
6         input[0] + sum(&input[1..])  
7     }  
8 }
```

Rust

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?

```
1 struct X {  
2     a: bool,  
3     b: bool,  
4 }
```

Rust

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.

```
1 // NOTE: only 1-7 are valid
```

Rust

```
2 type WeekDay = u8;
```

```
3
```

```
4
```

```
5 // ???
```

```
6 let x: WeekDay = 8;
```


Enums

Sometimes, you know that not all values are possible.

```
1 // Only has 7 possible values
2 enum WeekDay {
3     Monday,
4     Tuesday,
5     Wednesday,
6     Thursday,
7     Friday,
8     Saturday,
9     Sunday,
10 }
11
12 // we can only choose one of the valid values!
13 let x: WeekDay = WeekDay::Monday;
```

Rust

Enums

```
1 // Only has 7 possible values
2 enum WeekDay {
3     Monday,
4     Tuesday,
5     Wednesday,
6     Thursday,
7     Friday,
8     Saturday,
9     Sunday,
10 }
11
12 // we can only choose one of the valid values!
13 let x: WeekDay = WeekDay::Monday;
```

Rust

Enums

Unlike in many other programming languages, enums can have values

```
1 enum IpAddress {  
2     Ipv4([u8; 4]),  
3     Ipv6([u8; 16]),  
4 }  
5  
6 let x: IpAddress = IpAddress::Ipv4([127, 0, 0, 1]);
```

Rust

Enums

How many possible values?

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

Rust

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

Enums are sometimes called “sum types”

Enums

Another example: `Option<T>`

```
1 enum Option<T> {  
2     Some(T),  
3     None  
4 }  
5  
6 // 257 possible values  
7 // 256 if Some, or one more: None  
8 let x: Option<u8> = Some(3);
```

Rust

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>