Intro to Rust Lang

Ownership Revisited

Ownership Revisited

Today, we'll provide another way of thinking about ownership:

- Not just avoiding compile errors, but uncovering:
  - How the borrow checker works
  - The purpose of the borrow checker
- How do we write safer code by working *with* the borrow checker?

The Rules

At the beginning of this course, we learned two sets of rules…

- Ownership rules
- Borrowing rules

Rules of Ownership

- Each value in Rust has an *owner*
- A value can only have one owner at a time
- When the owner goes out of scope, the value will be *dropped*

Rules of Borrowing

- At any given time, you may have either
  - One mutable (exclusive) reference to data
  - Any number of immutable (shared) references to the same data
- There can never be dangling references

The Catch

Getting code to compile is one thing. Understanding why is another...

- Sometimes we follow the rules blindly
  - When we break the rules, we may not fully understand why it is a problem in the first place
- The compiler can be overly cautious
  - Rejects code that seems safe
  - Makes us prove safety, even when we "know" it's safe

Today's Objective

Today we're going to address these questions:

- How do ownership rules prevent memory safety issues?
- What makes the borrow checker reject seemingly safe code?

Defining Safety

What is safety?

- Safety is the absence of *undefined behavior*

Defining Unsafe

However, undefined behavior / unsafety can mean many things.

list

- Definition in the Rust Reference is much longer…

Defining Unsafe

Simplification for today: **Unsafety ⇒ Invalid Memory Access**

Unsafety ⇒ Invalid Memory Access

Memory access can be unsafe if we access memory that is:

- Deallocated
  - Ownership rules prevent this
- Overwritten by "someone else"
  - Borrowing rules prevent this

Unsafety ⇒ Invalid Memory Access

Memory access can be unsafe if we access **deallocated** or **overwritten** memory.

- Immutable global variables are trivially safe
  - `static` variables are read-only and valid for program's lifetime
- Today, we'll focus on *local* variables

Local Variables

Local variables live in a function's **stack frame**.

The stack frame:

- Contains everything needed for the function to run
- Is allocated on function call
- Is deallocated on function return

The Stack: Local Variables

Here is a representation of `main`'s stack frame.

```rust
fn main() {
    let x = 1;
}
```

The Stack: Local Variables

Now we call `my_function`, constructing its stack frame.

```rust
fn main() {
    let x = 1;
    my_function(x);
}

fn my_function(arg: i32) {
    let y = 2;
    let z = 3;
}
```

The Heap

Big Data

What if instead of an integer on the stack (`x = 1`)...

```
fn main() {
    let x = 1;
    my_function(x);
}
```

Big Data

We have a 15 GB array?

```
fn main() {
    let beef =
        [0xDEADBEEF; HUGE_NUMBER];

    my_function(beef);
}
```

- 15 GB = your Google Drive storage alt text

Big Data

```rust
fn my_function(arg: [u32; HUGE_NUMBER]) {
    <-- snip -->
}
```

When we call `my_function`, we need to:

- Allocate enough space in the stack frame
- Copy all 15 GB of `0xDEADBEEF`'s...

Big Data

Imagine being required to recreate `beef` on every single stack frame.

```
let beef = [0xDEADBEEF; 2_000_000_000];

my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
```

- Unsustainable!

Motivating the Heap

We probably want to keep our `beef` array around for longer than a single function call.

- We can say that it is **long-lived** data
- We want other functions to use this array, instead of just a single stack frame
- How do we persist this data across function calls?

The Stack?

Instead of storing our array buffer on the stack…

The Heap

Instead of storing our array buffer on the stack...

Let's put it on the **heap**!

The Heap

- If the data lives in the heap…
- The **pointer** lives on the stack

`Box<T>`

The simplest form of heap allocation is `Box<T>`.

Moving a value from stack to heap:

```
let val: u8 = 5;
let boxed: Box<u8> = Box::new(val);
```

- Can access value by dereferencing box as `*boxed`
- Value is automatically dropped when `boxed` goes out of scope

`Box<T>`

Let's put our `beef` array in a `Box`:

```
fn main() {
    let beef = Box::new([0xDEADBEEF; HUGE_NUMBER]);
    my_function(beef);
}

fn my_function(arg: Box<[u32]>) {
    <-- snip -->
}
```

- In reality, this allocates `beef` on the stack and then copies it to the heap
  - Use `Vec<T>` and convert to a boxed slice instead!

The Heap

When we call `my_function`, we can copy the *pointer* into `arg`!

```rust
let beef =
    Box::new([0xDEADBEEF; HUGE_NUMBER]);

my_function(beef);
```

The Heap

Before: 15 GB per array

After: 15 GB + 8 bytes per pointer

```rust
let beef =
    Box::new([0xDEADBEEF; HUGE_NUMBER]);

my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
my_function(beef);
```

Recap: Stack vs. Heap

Variable placement:

- **Stack-allocated:** Data lives on the stack
- **Heap-allocated:** Data lives on the heap, the **pointer** on stack

Recap: Stack vs. Heap

Comparison	Stack	Heap
Manages	Scalar data + Pointers	Dynamically-sized / long-lived data
Allocated on	Function call	Programmer request
Deallocated on	Function return	???

Motivating Ownership

Recall that accessing **deallocated** memory is unsafe.

Motivating Ownership

When is memory deallocated?

- Stack: deallocated when the function returns
    - Valid, unless dangling pointer ✅
        - We'll discuss more in the upcoming Lifetimes lecture...
- Heap: deallocated when ???
    - ⚠️
    - How can we be confident that heap memory is deallocated safely?

Motivating Ownership

Recall the behavior of local variables on the stack:

- Local variable lives in the function's **stack frame**
- Allocated on function call
- Deallocated on function return
  - Sound familiar?

Motivating Ownership

What if we say that data is **"owned"** by the stack frame?

- One stack frame (owner) per variable
- Data is dropped on function return
- Very similar to our previous model of ownership!

Rules of Ownership

Under this alternate ownership model, owners are stack frames:

- Each value in Rust has an *owner*
  - Owner is the *stack frame*
- A value can have only one owner at a time
  - Variables can only be in one stack frame
- When the owner goes out of scope, the value will be *dropped*
  - When the function returns, the stack frame is cleaned up!

Ownership of Closures

Let's re-examine **closures** again using this new ownership model:

- What does the `move` keyword really do?
- Why are captured values dropped when they are?

Ownership of Closures

When is `my_str` dropped? Who has ownership of it?

```
let my_str = String::from("x");

let take_and_give_back = move || { my_str };

assert_eq!(take_and_give_back(), "x");
```

- In `take_and_give_back`, `my_str` is dropped the first time the closure is called
  - Can only be called once

Closure Internals

When we create a closure...

```rust
let do_nothing_closure = |_x| {};
```

Think of it as a struct with an associated function:

```rust
struct Closure;

impl Closure {
    //          vvvvv Notice the immutable reference to `self`!
    pub fn call(&self, _x: &str) -> () {}
}
```

Closure Internals

The `move` keyword tells the closure to take ownership of values from its environment.

```
let take_and_give_back = move || { my_str };

// The `move` keyword tells compiler to put `my_str` in this `Closure`.
struct Closure {
    my_str: String,
}

impl Closure {
    //          vvvv Notice the owned `self` type!
    pub fn call(self) -> String {
        return self.my_str;
    }
}
```

Closure Internals

Then our code is equivalent to

```rust
let my_str = String::from("x");

let take_and_give_back = Closure { my_str };
//                                 ^^^^^^ `my_str` is moved!

let my_returned_str: String = take_and_give_back.call();
```

- What happens when we `call` on the `Closure`?
  - *Think about the stack frames*

Closure Example

First, `my_str` is moved into our `Closure`.

```
let my_str = String::from("x");

let take_and_give_back =
    Closure { my_str };
```

Closure Example

Next, we call our closure, which gives ownership of `my_str` to `Closure::call`'s stack frame.

```
let my_str = String::from("x");

let take_and_give_back =
    Closure { my_str };

let my_returned_str =
    take_and_give_back.call();
```

Closure Example

`Closure::call` gives ownership back to `main`'s stack frame...

```
pub fn call(self) -> String {
    // Gives ownership back!
    return self.my_str;
}
```

Closure Example

- `Closure`'s `my_str` is invalidated
- `my_str` is moved out of `Closure`'s "body"
    - This is why it can only be called once!

Recap: Closure Internals

- New way of thinking about ownership
  - Owners are stack frames
- A closure is really a function that stores ("captures") values in a struct
  - Ways to capture values:
    - **Taking ownership:** Store the value in the struct, via `move` keyword
    - **Borrowing:** Store a reference to the value in the struct

Motivating Borrowing Rules

Recall that accessing **overwritten** memory is unsafe.

Vector Pop

Suppose we want to write this code.

```rust
let mut v = vec![1, 2, 3, 4];
let x = &v[3]; // Get a reference to the last element
v.pop(); // Remove the last element in `v`
println!("{}", x); // What is `x`?
```

- What do you think the compiler will say?

Vector Pop

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
4 |     let x = &v[3]; // Get a reference to the last element
  |              - immutable borrow occurs here
5 |
6 |     v.pop(); // Remove the last element in `v`
  |     ^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}", x); // What is `x`?
  |                    - immutable borrow later used here
```

- `x` is invalid! `&v[3]` could now be any value ⇒ undefined behavior
- We cannot mutate a value that someone else is borrowing

Vector Push

What if instead of removing the last element, we *add* an element to the end?

```rust
let mut v = vec![1, 2, 3, 4];
let x = &v[3]; // Get a reference to the last element
v.push(5); // Instead of popping, let's push!
println!("{}", x); // What is `x`?
```

- Surely nothing will happen to `x` this time?

Vector Push

Why isn't this okay???

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
4 |     let x = &v[3]; // Get a reference to the last element
  |              - immutable borrow occurs here
5 |
6 |     v.push(5); // Instead of popping, let's push!
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}", x); // What is `x`?
  |                    - immutable borrow later used here
```

Vector Layout

Recall that vectors are *dynamic arrays*.

- *This is technically a* `String`*, but recall that* `String` *is implemented with a* `Vec<u8>`

Mutating Vectors

- What if pushing `5` onto `v` triggers a resize?
- Resizing means:
  - Allocate new memory for `v`
  - Copy data to new location
  - **Deallocate old memory location**
- `x` would no longer point to valid memory!

Mutating Vectors

```
error[E0502]: cannot borrow `v` as mutable because it is also borrowed as immutable
 --> src/main.rs:6:5
  |
4 |     let x = &v[3]; // Get a reference to the last element
  |              - immutable borrow occurs here
5 |
6 |     v.push(5); // Instead of popping, let's push!
  |     ^^^^^^^^^ mutable borrow occurs here
7 |
8 |     println!("{}", x); // What is `x`?
  |                    - immutable borrow later used here
```

Even though this error may seem unreasonable, the borrow checker is actually preventing you from making a hard-to-find mistake!

We Know Better...

What if you knew that this `push` doesn't resize?

- You as the programmer have vetted the implementation of `push`
- You can *guarantee* that resizing only happens with more elements

Sneak Peek: `unsafe`

You can use an `unsafe` block to tell the compiler that you know better.

```rust
let mut v = vec![1, 2, 3, 4];
let x = unsafe { v.as_ptr().add(3) }; // Get a raw pointer to the last element
v.push(5);
let x = unsafe { *x }; // Dereference the raw pointer
println!("{}", x);
```

- Think of `unsafe` blocks as "trust me bro" blocks
- We will talk about `unsafe` in a few weeks!

`unsafe` vs. Borrow Checking

You should *never* resort to using `unsafe` just to get past the borrow checker.

- `unsafe` should be used sparingly and intentionally
- You must understand *why* you need to bypass the borrow checker
- Therefore, you *must* understand the borrow checker!

The Borrow Checker

The Borrow Checker prevents you from writing unsafe code.

This leads to some questions:

- How do I know if my program is unsafe?
  - How did the borrow checker conclude this?
- How can I tell if my program is safe even if the borrow checker rejects it?

Unveiling the Borrow Checker

> "His blade works so smoothly that the ox does not feel it." - The Dextrous Butcher

- Understand the borrow checker, and you'll speak its language fluently:
  - "My program is unsafe, and here's how I'll fix it"
  - And occasionally...
    - "My program is actually safe, let me tell you why"

Permissions

Permissions of Places

Denote the left side of assignments as **places**.

```
let x = &v[3];
//  ^ place
```

- The borrow checker checks the permissions of **places**

Permissions of Places

Places include:

- Variables, like `a`
- Dereferences of places, like `*a`
- Array accesses of places, like `a[0]`
- Fields of places, like `a.0`, `a.field`
- Any combination of the above, like `*a.x[i].y`

Permissions of Variables

When declared, a variable has the permissions:

- Read: can be copied
- Write: can be mutated (if declared with `mut`)
- Own: can be moved or dropped

References and Permissions

- Variables have the permissions Read (**R**), Write (**W**), and Own (**O**)
- Using references can *temporarily remove these permissions*

Example: Immutable References

Let's revisit our vector pop example.

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

Place	R	W	O
v	?	?	?

Example: Immutable References

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

Place	R	W	O
v	?	?	?

We declare `v`, giving it:

- ?
- ?

Example: Immutable References

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

Place	R	W	O
v	+R	?	+O

We declare `v`, giving it:

- R, O due to variable declaration
- ?

Example: Immutable References

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

Place	R	W	O
v	+R	+W	+O

We declare `v`, giving it:

- R, O due to variable declaration
- W because `mut`

Example: Immutable References

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

Place	R	W	O
v	R	W	O
x	?	-	?

When we create a reference `x` to `v`, we:

- ?

Example: Immutable References

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

Place	R	W	O
v	R	W	O
x	**+R**	-	**+O**

When we create a reference `x` to `v`, we:

- Give `x` R, O due to variable declaration
  - `x`'s permissions are for the *reference*, not the data it is referring to

Example: Immutable References

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

Place	R	W	O
v	R	W?	O?
x	R	-	O

This move changes `v`:

- ?
- ?

Example: Immutable References

```
let mut v = vec![1, 2, 3, 4];
let x = &v[3]; // v loses W
```

Place	R	W	O
v	R	W?	-
x	R	-	O

This move changes `v`:

- Removes W
- ?

Example: Immutable References

```
let mut v = vec![1, 2, 3, 4];
let x = &v[3]; // <- v loses W, O
```

Place	R	W	O
v	R	-	-
x	R	-	O

This move changes `v`:

- Removes W
- Removes O

Example: Immutable References

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

println!("{}", *x);
```

Place	R	W	O
v	R	-	-
x	R	-	O
*x	R	-	-

We can access our reference `x` by dereferencing it as `*x`.

- `*x`'s permissions are different from `x`'s!
- Can only dereference if `*x` has R permissions
- `*x` can only have R if `v` has R

Example: Immutable References

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

Place	R	W	O
v	R	-	-
x	R	-	O
*x	R	-	-

We can no longer mutate `v`, since we created a reference `x` to it.

When does `v` regain W, O?

- Case 1: All references become unused
- Case 2: Mutate `v` before *any* reference is used
  - Revokes permissions of references

Example: Immutable References

```
let mut v = vec![1, 2, 3, 4];
let x = &v[3]; // <- v loses W, O

v.pop(); // <- v regains W, O
```

Place	R	W	O
v	R	+W	+O
x	R	-	-
*x	-	-	-

This `v.pop()` is safe (Case 1).

- `v` requests W while all references are unused (Case 1)
  - `v` regains W, O, *revokes permissions of all references*
  - `x` loses O, `*x` loses R

Example: Immutable References

However, we cannot access `*x` anymore, as its permissions have been revoked.

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

v.pop(); // Revokes permissions!

// THIS DOES NOT COMPILE!!!
println!("{}", *x); // Requires R on *x
```

Place	R	W	O
v	R	W	O
x	R	-	-
*x	-	-	-

- This code causes an error because we don't have permissions on `*x`

Recap: Immutable References

- Declaring a variable `v` gives it R, O permissions, W if `mut`
- Creating an immutable reference `x` to `v`
  - Gives `x` R, O permissions
  - Removes `v`'s W, O permissions
- Permissions are restored when either:
  - References become **unused**
  - We mutate `v` *before* any reference is used

Mutable References

- **`x` and `*x` have different permissions**
  - Notice how revoking permissions removes R from `*x`, but *keeps* R on `x`
  - We can create as many references as we want...
    - We just can't *access* them without the correct permissions
- Mutable references further illustrate this

Example: Mutable References

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

Place	R	W	O
v	R	-	-
x	R	-	O
*x	**R**	-	-

Recall that when we create an immutable reference `x = &v[3]`:

- `v` loses W and O permissions
- `*x` only has R permissions

Example: Mutable References

```
let mut v = vec![1, 2, 3, 4];
let x = &mut v[3];
//      ^^^^
```

Place	R	W	O
v	R?	-	-
x	R	-	O
*x	R	-?	-

However, when `x` is a mutable reference:

- ?
- ?

Example: Mutable References

```
let mut v = vec![1, 2, 3, 4];
let x = &mut v[3];
//      ^^^^^
```

Place	R	W	O
v	-	-	-
x	R	-	O
*x	R	-?	-

However, when `x` is a mutable reference:

- `v` loses *all* permissions, including R
- ?

Example: Mutable References

```
let mut v = vec![1, 2, 3, 4];
let x = &mut v[3];
//      ^^^^
```

Place	R	W	O
v	-	-	-
x	R	-	O
*x	R	W	-

However, when `x` is a mutable reference:

- `v` loses *all* permissions, including R
- `*x` (*not* `x`) gains W permissions

Example: Mutable References

```
let mut v = vec![1, 2, 3, 4];
let x = &mut v[3];
//      ^^^^
```

Place	R	W	O
v	-	-	-
x	R	-	O
*x	R	W	-

- `v` loses *all* permissions, including R
  - Prevents creation of other references (both mutable and immutable)
  - Avoids simultaneous **aliasing** and mutation
- `*x` (*not* `x`) gains W permission
  - We can write to the data, but we can't reassign `x`

Recap: Mutable References

An immutable reference `x` of `v`:

- Removes W and O permissions for `v`
- `*x` can only take R if `v` has R

A mutable reference `x` to `v`:

- Removes *all* permissions for `v` (including R)
  - Prevents creation of any other references
- `*x` (not `x`) gains W permission

Soundness vs. Completeness

Fixing a Safe Program

Suppose we have a vector of numbers:

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

Fixing a Safe Program

We want to take this number `2` …

```
let mut v = vec![1, 2, 3, 4];
//               ^
```

Fixing a Safe Program

We want to take this number `2` ...and add it to the number in the first index.

```
let mut v = vec![1, 2, 3, 4];
//               ^
```

Fixing a Safe Program

Here is a seemingly reasonable solution:

```
let mut v = vec![1, 2, 3, 4];
let slot1 = &mut v[0];
let slot2 = &v[1];
*slot1 += *slot2;
```

Fixing a Safe Program

```
error[E0502]: cannot borrow `v` as immutable because it is also borrowed as mutable
  --> src/main.rs:9:18
   |
8  |     let slot1 = &mut v[0];
   |                 - mutable borrow occurs here
9  |     let slot2 = &v[1];
   |               ^ immutable borrow occurs here
10 |     *slot1 += *slot2;
   |     ---------------- mutable borrow later used here
   |
   = help: use `.split_at_mut(position)` to obtain
           two mutable non-overlapping sub-slices
```

- Let's break down the permissions

Fixing a Safe Program

```rust
let mut v = vec![1, 2, 3, 4];
let slot1 = &mut v[0];
let slot2 = &v[1];
*slot1 += *slot2;
```

Place	R	W	O
v	-	-	-
*slot1	R	W	-

Recall when we create a mutable reference `slot1 = &mut v[0]`:

- `v` loses all permissions
- `*slot1` gains W, R permissions

Fixing a Safe Program

```
let mut v = vec![1, 2, 3, 4];
let slot1 = &mut v[0];
let slot2 = &v[1];
*slot1 += *slot2;
```

Place	R	W	O
v	-	-	-
*slot1	R	W	-
*slot2	-	-	-

Next, let's look at whether our references are accessed safely:

- Mutating `*slot1` requires R, W
  - We have both ✅
- Reading `*slot2` requires R
  - We don't have R ❌

Fixing a Safe Program

```
let mut v = vec![1, 2, 3, 4];
let slot1 = &mut v[0];
let slot2 = &v[1];
*slot1 += *slot2;
```

Place	R	W	O
v	-	-	-
*slot1	R	W	-
*slot2	-	-	-

Reading `*slot2` requires R.

- `*slot2` can only take R if `v` has R
- `v` "gave" R to `*slot1`!

Fixing a Safe Program

```rust
let mut v = vec![1, 2, 3, 4];
let slot1 = &mut v[0];
let slot2 = &v[1];
*slot1 += *slot2;
```

Place	R	W	O
v	-	-	-
*slot1	R	W	-
*slot2	-	-	-

Issue: Single place `v` represents *all* elements in the vector.

- Borrow checker does not see each index as a different place
- Borrow checker can't know that this code is safe
  - But we (as humans) do!

Fixing a Safe Program

Solution 1: No References.

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

v[0] += v[1];

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

- This works because we mutate only `v[0]`
    - Program evaluation is right to left

Fixing a Safe Program

Solution 2: Drop into `unsafe`.

`split_at_mut` uses `unsafe` under the hood:

```
let mut v = [1, 0, 3, 0, 5, 6];

let (left, right) = v.split_at_mut(2);

assert_eq!(left, [1, 0]);
assert_eq!(right, [3, 0, 5, 6]);
```

- Divides a mutable slice into two mutable slices at index `mid`
  - `left` contains `[0, mid)`
  - `right` contains `[mid, len)`
- We will talk about this in a few weeks!

Recap

- Ownership rules prevent access to deallocated memory
  - Think of owners as **stack frames**
- Borrow checker checks **permissions** of **places**
  - References temporarily remove permissions
- Sometimes, borrow checker can't know your program is safe
  - If you conclude it's safe after reasoning about stack frames and RWO permissions...
    - Consider using `unsafe`

Next Lecture: Lifetimes

- Thanks for coming!