Consuming Rust bite by byte

Bite 3 – Ownership

Jim Fawcett

https://JimFawcett.github.io

Bite #2

Bite #4

Bite #3 – Ownership

- Our goal is to understand:
  - Mutability
  - Single ownership
  - Borrow
  - RwLock model

Rust Ownership

- Ownership rules are, in principle, quite simple:
  - Rust enforces **Read-Write-Locks** on data access at compile-time.
  - Any number of readers may access a value simultaneously.
  - Writers get exclusive access to a value – no other readers or writers.
- What are readers and writers?
  - Any variable bound to a value with no mut qualifier is a reader.
    - Original owner: `let s = String::from("a string");`
    - References to the data: `let r = &s;`
  - Any variable bound to a value with mut qualifier is a writer:
    - Original owner: `let mut s = String::from("another string");`
    - References to the data: `let mut r = &s;`

Examples of References and RwLocking

- Non-mutable Vec and references - all readers:
  - `let v = vec![1,2,3];`
  - `let r1 = &v;  let r2 = &v;  // each has view of v's data`
- `Creation of reference inhibits owner's ability to mutate`
- Mutable Vec, non-mutable references:
  - `let mut v = vec![1,2,3];`
  - `let r1 = &v;  let r2 = &v;  // each has view of v's data`
  - r1 and r2 borrow v's data // v cannot mutate while borrows are active
  - Borrows end when they go out of scope or are dropped, drop(r1);
- Mutable data, mutable reference – writer v's ability to write borrowed
  - `let mut v = vec![1,2,3];`
  - `let mut r = &v;  // r has borrowed v's ability to mutate`
  - v cannot mutate until borrow ends

Rust won’t allow mutation with an active reference

File Edit Selection View Go Run Terminal Help — main.rs - type_safety - Visual Studio Code

EXPLORER

OPEN EDITORS
- main.rs src 1, U

TYPE_SAFETY
- src
  - main.rs 1, U
- target
- .gitignore U
- Cargo.lock U
- Cargo.toml U

OUTLINE

main.rs

src > main.rs > ...

```rust
fn main() {
    let mut v = Vec::<i32>::with_capacity(3);
    v.push(1);
    v.push(2);
    v.push(3);
    print!("\n  v capacity = {}", v.capacity());

    let r1 = &v[1];
    print!("\n  address of v[1] = {:?}", &v[1] as *const i32);
    print!("\n  address of r1   = {:?}", r1 as *const i32);

    v.push(4);  // fails to compile, can't mutate while borrowed
    print!("\n  address of v[1] = {:?}", &v[1] as *const i32);
    print!("\n  address of r1   = {:?}", r1 as *const i32);

    println!("\n\n  Hello, Ownership!\n");
}
```

TERMINAL — 1: cmd

```
   |                - immutable borrow occurs here

...
12 |     v.push(4);  // fails to compile, can't m
utate while borrowed
   |     ^^^^^^^^^ mutable borrow occurs here
13 |     print!("\n  address of v[1] = {:?}", &v[
1] as *const i32);
14 |     print!("\n  address of r1   = {:?}", r1
as *const i32);
   |                                          --
immutable borrow later used here

error: aborting due to previous error

For more information about this error, try `rustc
 --explain E0502`.
error: could not compile `type_safety`.

To learn more, run the command again with --verbo
se.
C:\temp\type_safety>
```

master* ⊗ 1 ⚠ 0 RLS — Ln 10, Col 5 Spaces: 4 UTF-8 LF Rust

Rust allows mutation if we don’t use the reference

File Edit Selection View Go Run Terminal Help — main.rs - type_safety - Visual Studio Code

EXPLORER

OPEN EDITORS
- main.rs src U

TYPE_SAFETY
- src
  - main.rs U
- target
- .gitignore U
- Cargo.lock U
- Cargo.toml U

OUTLINE

main.rs

src > main.rs > ...

```rust
fn main() {
    let mut v = Vec::<i32>::with_capacity(3);
    v.push(1);
    v.push(2);
    v.push(3);
    print!("\n  v capacity = {}", v.capacity());

    let r1 = &v[1];
    print!("\n  address of v[1] = {:?}", &v[1] as *const i32);
    print!("\n  address of r1   = {:?}", r1 as *const i32);

    v.push(4);  // fails to compile, can't mutate while borrowed
    print!("\n  address of v[1] = {:?}", &v[1] as *const i32);
    //print!("\n  address of r1   = {:?}", r1 as *const i32);

    println!("\n\n  Hello, Ownership!\n");
}
```

TERMINAL — 1: cmd

```
C:\temp\type_safety>
cargo -q run

  v capacity = 3
  address of v[1] = 0x8df574
  address of r1   = 0x8df574
  address of v[1] = 0x8dba84

  Hello, Ownership!

C:\temp\type_safety>
```

master* ⊗ 0 ⚠ 0 RLS — Ln 14, Col 7 Spaces: 4 UTF-8 LF Rust

Hello Ownership!

- Rust's ownership policies:
  - Every value has one and only one owner
    - Owner disposes resource when it goes out of scope or is dropped
  - Ownership can be transferred with a move
  - Ownership can be borrowed with a reference
    - References hold a view into value
    - Original value's owner can't mutate value while borrowed
    - Immutable references can be shared. Mutable references are exclusive
    - Borrowing ends when reference goes out of scope or is dropped
    - This fits very well with pass by reference for function arguments
  - Values are, by default, immutable, but can be made mutable
    - `let x = 3;  // x is immutable`
    - `let mut y = 3;  // y is mutable`

Immutable References

- Any number of immutable references may be declared for a value:
  - `let mut s = String::from(“a string”);`
  - `let r1 = &s;`
  - `let r2 = &s;`
- The original owner can not mutate until all active references are dropped or go out of scope:
  - `fn show(s:&String) { … }`
  - `let mut t = String::from(“another string”);`
  - `show(&t);`
  - `t.push_str(“ with more stuff”);`
    `// mutation ok, left &t’s scope, e.g. show function exit`

Mutable References

- Only one mutable reference may be declared for a value:
  - `let mut s = String::from("a string");`
  - `let mut r1: &String = &s;`
  - `// let mut r2: &String = &s;  // won't compile`
  - `// let r3 = &s;  // won't compile`
- The original owner can not mutate until active reference is dropped or goes out of scope (same as before):
  - `fn show(s:&String) { … }`
  - `let mut t = String::from("another string");`
  - `show(&t);  // copies reference to show stack frame, e.g., a borrow`
  - `t.push_str(" with more stuff");`
    `// mutation ok, &mut t went out of scope`

Copies, Moves

- Copy
  - Data resides in one contiguous block of memory (blittable)
  - `let x = 3.5;`
  - `let y = x;`
  - y gets copy of x's value ==> two separate locations holding the same value.
  - **Copy binding creates new owner of new data.**
- Move
  - Data resides in two or more blocks, usually one in stack, one in heap.
  - `let s = String::from("a string");`
  - `let t = s;`
  - s value moved to t, s becomes invalid
  - **Move binding transfers ownership**

Ownership summary

- These simple rules provide memory safety:
  - `let x = y` ==> copy if blittable, otherwise move ==> transfer of ownership
    - Can't use y if moved from
  - `let r1 = &x; let r2 = &x;`
    ==> may have any number of immutable references
    - x may not be mutated while there are active references
  - `let mut z = …`
  - `Let mut r3 = &z;` ==> may only have one mutable reference
    - Z may not be mutated while there is an active reference
- References become inactive when they go out of scope or are dropped:
  - `drop(r3);`

Exercises

1. Create a function that accepts a mutable reference. Attempt to display the reference with print!. Why does this fail to compile?
    - How can you display the function's argument?
2. What do Copy and Move operations have to do with ownership?
    - How can you avoid transfer of ownership when binding to a non-blittable value?
3. Why is an owner not allowed to mutate when there are active references to its value?

References

Link	Description
ConsumingRustBite2 - Undefined Behavior	Lack of memory safety
ConsumingRustBite4 - Interior mutability	Checking ownership rules at run-time
Rust Models	Expanded discussion in Rust Models presentation

That’s all until Bite #4

Bite #4 discusses interior mutability. The defers ownership rule checking to run-time.