LECTURE 17

Theory and Design of PL (CS 538)

March 25, 2020

REFERENCES

WHAT IS A REFERENCE?

- An indirect name for some data
  - Think: a pointer to some data
- Making a new reference in Rust

```
let my_str = String::from("foo"); // variable holding string
let ref_to_str = &my_str;         // reference to my_str
```

WHY USE REFERENCES?

- Reference does not own the data
  - Can have only one owner, but many references
- Reference going out of scope does not drop data
  - Can “borrow” reference to function
- Function can take (mutable) reference and modify caller’s data directly
  - Useful for mutable datastructures

(IM)MUTABLE REFERENCES

- By default, references are *immutable*
  - Can't change underlying data through reference
  - Reference type: `&T`
- Can declare *mutable* references
  - Target must be mutable as well
  - Reference type: `&mut T`

```
let mut my_str = String::from("foo"); // mutable var
let ref_to_str = &mut my_str;         // mutable ref to my_str
```

DE-REFERENCING

- Use * notation to get thing reference is pointing at
- Often not needed due to “auto-deref” (magic)

```rust
let vr: &Vec<i32> = ...;

println!("First element: {}", (*vr)[0]); // Explicit deref

println!("First element: {}", vr[0]);    // Implicit deref

println!("First element: {}", vr.first); // Implicit deref
```

SOMEWHAT CONFUSINGLY

- Reference itself can be mutable

```
// Can't change ref or thing it's pointing at
let immut_ref_to_immut = &my_string;

// Can't change ref, can change thing it's pointing at
let immut_ref_to_mut = &mut my_string;

// Can change ref, can't change thing it's pointing at
let mut mut_ref_to_immut = &my_string;
mut_ref_to_immut = &my_other_string;

// Can change ref and thing it's pointing at
let mut mut_ref_to_mut = &mut my_string;
mut_ref_to_mut = &mut my_other_string;
*mut_ref_to_mut = String::from("???");
```

WHAT'S GOING ON?

- Mutability is not a property of the data!
  - NOT: these bits are mutable or immutable
- Mutability is property of *variable* or *reference*
  - YES: I can mutate data **through** this variable
  - YES: I cannot mutate data **through** that reference

ALIASING

MULTIPLE REFERENCES

- Rust works hard to ensure *one owner* for each data
- Multiple references to same data is problematic
  - Also known as *aliasing*
- References need to follow certain rules for safety

THE GOLDEN RULES

- In any scope, there can be either:
  1. *Any number* of immutable references
  2. *At most one* mutable reference
- … referring to the same variable

> *One or the other: not both!*

MULTIPLE IMMUTABLE

- Can have any number of *immutable* refs to variable
- Safe: none of the refs can change the underlying

```
let my_str  = String::from("foo");
let ref_one = &my_str;
let ref_two = &my_str;

println!("Both refs: {} {}", ref_one, ref_two); // OK
```

AT MOST ONE MUTABLE

- Can only change underlying through single reference
  - Also important in concurrent setting
  - Also enables more optimizations

EXAMPLE

- Can’t make two mutable references to same thing

```
let mut mut_str = String::from("foo");

let ref_one = &mut mut_str; // OK

let ref_two = &mut mut_str; // Not OK
```

CAN'T HAVE BOTH MUTABLE AND IMMUTABLE REFERENCES

```
let mut mut_str = String::from("foo");

let ref_one   = &mut_str;        // OK: Immutable ref

let ref_two   = &mut_str;        // OK: Immutable ref

let ref_three = &mut mut_str;    // Not OK: Mutable ref
```

USE SCOPES TO MANAGE REFS

- Rules only apply to references *currently in scope*

```
let mut mut_str = String::from("foo");
let mut_ref = &mut mut_str;
mut_ref.push("bar");  // OK
mut_str.push("baz");  // Not OK: can't access mut_str
                      // because mut_ref in scope

// Use scopes!
let mut mut_str_2 = String::from("foo");
{
  let mut_ref_2 = &mut mut_str_2;
  mut_ref_2.push("bar");  // OK
}                         // scope ends, mut_ref_2 gone
mut_str_2.push("baz");    // Now OK: no more mut_ref_2
```

ALTERNATIVE READING

- Immutable reference: *shared* reference
  - Shared access to some data
  - Sharing: can't change the data
- Mutable reference: *unique* reference
  - Exclusive access to some data
  - Can modify it: no one else has access

Can't mix shared and unique!

ISN'T A REFERENCE JUST A POINTER?

- In machine code: a reference is just a pointer
- In Rust: a ref. also gives **permissions** to do things
- With an immutable reference, code can:
  - Dereference/read location (obviously)
  - Point to/read **anything reachable** from ref.
- With an mutable reference, code can:
  - Dereference/read/write location (obviously)
  - Point to/read/write **anything reachable** from ref.

MUTATION CAN INVALIDATE POINTERS

```
struct Triple(i32, i32, i32);
enum MyEnum {
    Small(i32),
    Big(Triple),
}

let mut my_enum = Big(Triple(1, 2, 3));
let mut im_ref: &i32 = &0;   // points at 0
if let Big(b) = my_enum {
    im_ref = &b.2;           // points at last field in Big: 3
}

let m_ref = &mut my_enum;        // mutable ref (not allowed)
*m_ref = Small(42);              // change Big to Small
println!("Uh oh: {}", im ref);   // what does this point to?
```

PASSING ARGUMENTS:

THREE WAYS

“MOVING” ARGUMENTS

- Operationally: arguments passed “by value”
- Ownership of argument passes into the function
  - Caller can’t use arguments after calling!
  - “Arguments moved into function”
- Function can return argument to return ownership

EXAMPLE: MOVE

```
fn take_own(s: String) { ... }

fn main() {
  let my_string = String::from("Hello!");

  take_own(my_string);  // Pass the string to function

  println!("Still there? {}", my_string);  // Not OK: it's gone!
}
```

EXAMPLE: MUTABLE MOVE

- Not super intuitive behavior…

```
fn take_mut_own(s: mut String) { s = String::from("wow");  }

// Pretty much the same as:
fn take_own(s: String) {
    let mut owned_string = s;
    owned_string = String::from("amazing");
}

fn main() {
  let my_string = String::from("Hello!");    // Isn't mutable...
  take_mut_own(my_string);                   // ... but this works?
  println!("Still there? {}", my_string);  // Not OK: it's gone!
}
```

“BORROWING” ARGUMENTS

- Operationally: arguments passed “by reference”
- Ownership of argument *doesn’t* change
  - Original owner (caller, caller-of-caller, …) owns arg.
- “Function borrows arguments” (from the owner)
  - Will give it back to owner when done with it

EXAMPLE: BORROW

```rust
fn take_borrow(s: &String) { ... }          // Can't mutate s

fn main() {
  let my_string = String::from("Hello!");

  take_borrow(&my_string);                  // "Borrow" ref to fn

  println!("Still there? {}", my_string);   // OK: still owner
}
```

EXAMPLE: MUTABLE BORROW

```rust
fn take_mut_borrow(s: &mut String) {
  // Assign new string to s
  *s = String::from("amazing");
}

fn main() {
  let mut my_string = String::from("Hello!");  // Note: need mut!

  take_mut_borrow(&mut my_string);             // "Borrow" ref mut

  println!("Still there? {}", my_string);      // OK: still owner
}
```

MATCHING, MOVING, BORROWING

VARIABLES ARE KEY

- Anywhere there are variables:
  - Think about ownership rules
  - Think about borrowing rules
- So far, we've seen variables from:
  - Let-bindings
  - Function arguments

A PUZZLE

```
let my_str = String::from("Hello world!");
let maybe_str = Some(my_str);

match maybe_str {
  None => println!("Nothing!"),
  Some(s) => println!("Something!"),
}

println!("Still there? {}", maybe_str.is_none());
```

- Is this program accepted, or not?
- What is ownership situation of s?

MATCHING CAN MOVE DATA

- Often: matching on enums with data inside
  - Example: `Option<T>`
- The inner data is *moved* into the match arm
  - Variable from match arm has ownership
- Typical ownership rules apply
  - Data is dropped at the end of the arm

REVISITING

```rust
let my_str = String::from("Hello world!");
let maybe_str = Some(my_str);

match maybe_str {
  None => println!("Nothing!"),
  Some(s) => println!("Something!"),  // String *moved* into s
                                      // s dropped here
}

println!("Still there? {}", maybe_str.is_none());  // Not OK!
```

- Even `maybe_str` is dropped: inner `s` is gone!

MATCHING AND BORROWING

```
let my_str = String::from("Hello world!");
let maybe_str = Some(my_str);
let maybe_ref = &maybe_str;

match maybe_ref {
  None => println!("Nothing!"),
  Some(s) => println!("Something: {}", s),  // what is type of s?
}

println!("Still there? {}", maybe_str.is_none());
```

- Is this program accepted or not?
- What's the ownership status of s?

MATCHING ON A REFERENCE

- Rust will infer how to borrow inner values
  - Matching on `&T` type: arms borrow immutably
  - Matching on `&mut T` type: arms borrow mutably
- Also called "default binding modes"
  - Usually: Just Works
  - Sometimes: inference goes wrong (Doesn't Work)

IMMUTABLE BORROW

```rust
let my_str = String::from("Hello world!");
let maybe_str = Some(my_str);
let maybe_ref = &maybe_str;  // immutable ref

match maybe_ref {  // match on *immutable* ref
  None => println!("Nothing!"),
  Some(s) => println!("Something: {}", s),  // can't mutate s
}

println!("Still there? {}", maybe_str.is_none());
```

MUTABLE BORROW

```
let my_str = String::from("Hello world!");
let mut maybe_str = Some(my_str);
let maybe_ref = &mut maybe_str;  // mutable ref

match maybe_ref {  // match on *mutable* ref
  None => println!("Nothing!"),
  Some(s) => *s = String::from("Good bye!"),  // mutate s
}

println!("What's here? {}", maybe_str.unwrap());
```

- Prints the new string: Good bye!

FORCING A BORROW

- Can force match to borrow on owned data

```
let my_str = String::from("Hello world!");
let mut maybe_str = Some(my_str);

match &maybe_str {  // force immutable borrow
  None => println!("Nothing!"),
  Some(s) => println!("Something: {}", s),  // can't mutate s
}

match &mut maybe_str {  // force mutable borrow
  None => println!("Nothing!"),
  Some(s) => *s = String::from("Good bye!"),  // mutate s
}
```

OLD-STYLE SYNTAX

```rust
let my_str = String::from("Hello world!");
let mut maybe_str = Some(my_str);

match maybe_str {  // force immutable borrow
  None => println!("Nothing!"),
  Some(ref s) => println!("Something: {}", s),  // can't mutate s
}

match maybe_str {  // force mutable borrow
  None => println!("Nothing!"),
  Some(ref mut s) => *s = String::from("Good bye!"),  // mutate s
}
```

- “Deprecated”, but try it if you have bizarre errors