Smart Pointers

`Box`, `Rc`, `RefCell`, and `Weak`

CS 320

author nlahn@radford.edu; modified by
ibarland@radford.edu.

Quick Recap

- What are Rust's ownership rules?

- Each chunk of data has one and only one owner, who is in charge of deallocating the memory.
  Owner can be a variable, *or* some other chunk of data (one which contains a pointer).

- Each chunk of data can be either:
  - Borrowed mutably once (`&mut`) OR
  - Borrowed immutably multiple times (&)
  - But not both!

Common tactics

- Pass *borrowed* data to functions
  - Borrowing often implemented via a pointer, or some sort of "fat pointer" (a small struct including a pointer).
  - Efficient to pass, though indirection to access.
  - `&mut` involves aliasing; only-one-alias-allowed makes this easier.

- Implement `Copy`
  Can `#[derive(Copy)]` of structs whose elements are `Copy`able.

- Can't copy heap-references (causes

- Drawback (and, the point): A function can't stash borrowed data on the heap.

Why does Rust have ownership rules?

- Rust's ownership rules usually enforce at compile time that there are no memory safety issues, such as:
  - Buffer overflow / index out-of-range
  - Dangling references
    (references that point to data that has been deallocated)
  - Double frees (same data is deallocated multiple times)
  - Memory leaks (data that was never deallocated)

- It is able to do this at compile time because each owner simply deallocates its data when the owner is deallocated,
  and only one owner.

author nlahn@radford.edu; modified by
ibarland@radford.edu.

Downside of ownership?

- The compiler is not perfect
- Sometimes, it rejects code that *we* know is perfectly fine. (Its ownership-reasoning-system is *sound*, but not *complete*.)
- (The compiler can be a bit overzealous at times… a bit too overprotective)

Bending (not breaking) the rules

- Sometimes we need to bend the rules a bit to be more flexible
- Ways of bending the rules include:
  - Allowing for multiple owners
  - Delaying enforcement of memory safety to runtime, instead of compile time
- We will do this using advanced features known as “smart pointers”
- Box is an example of a smart pointer, but it has limitations
- The other smart pointers can be used, but at a cost

author nlahn@radford.edu; modified by ibarland@radford.edu.

Review : Box<T>

- Box<T> is the simplest type of smart pointer available in Rust
- The type T represents the “type of data owned by the Box”
- The Box owns the data it points to
- When the Box gets deallocated, it also allocates the data it points to

- Limitations:
  - A Box’s data can only have one owner
  - A Box’s data can only have one `&mut` or multiple & but not both!

Multiple Owners, via the Rc<T> type

- Rc stands for “ReferenceCounter”
- Rc<T> is just like Box<T>, except….
  - The same T can be owned by **multiple** Rc simultaneously!

- Wait, doesn’t that violated Rust’s ownership rules?!?!
  - It can be seen as “bending”, not “breaking”

author nlahn@radford.edu; modified by
ibarland@radford.edu.

How Rc<T> works....

- Create a new Rc<T> by using:
  - `let my_rc : Rc<String> = Rc::new(value.to_string())`
  - Compare to `Box::new(value)`

So far it looks just like a Box, but there is a new feature:

- `let another_rc : Rc<String> = Rc::clone(&my_rc)`

- This makes a new Rc that *shares ownership* of the same String!

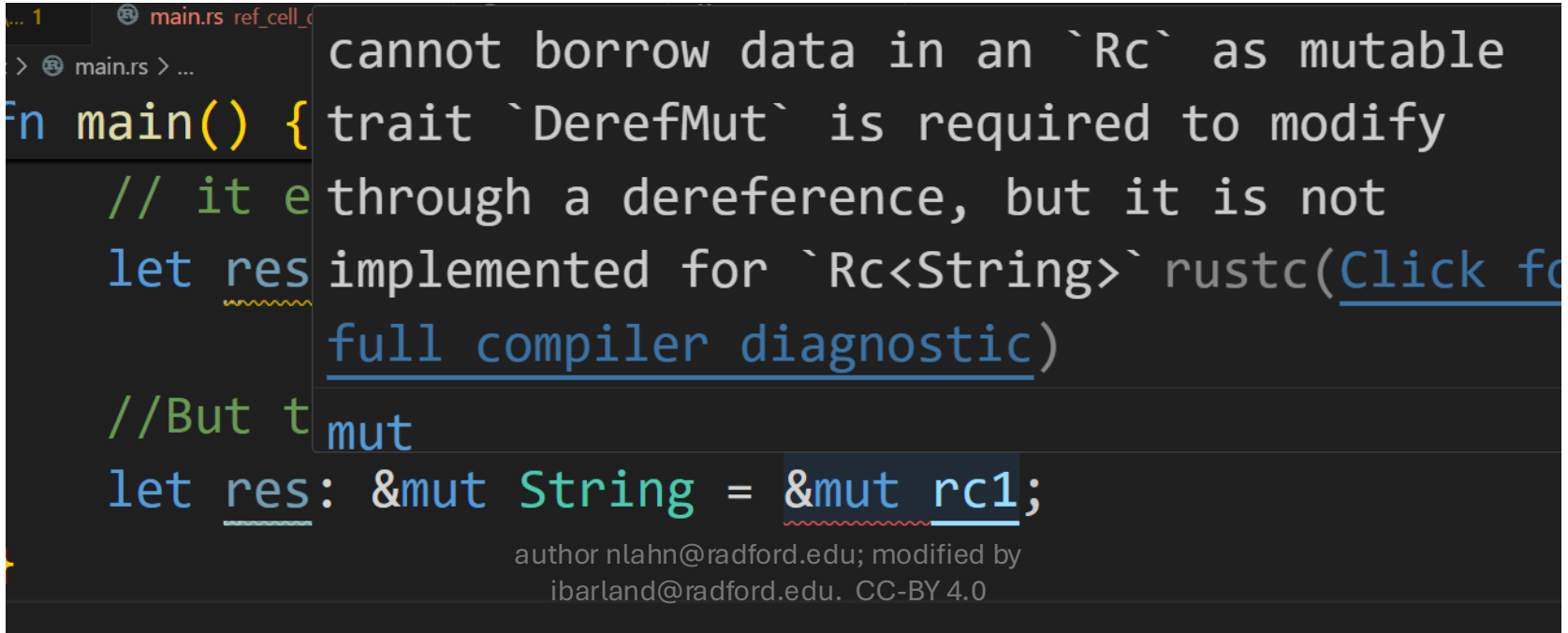
How Rc works

- Each Rc shares access to a counter variable, which keeps track of the number of Rc's sharing ownership
- When you create an Rc using `Rc::new`, the "owners" counter starts at 1
- When you clone an Rc using `Rc::clone`, then the "owners" counter increases by 1
- When an Rc gets deallocated, it calls its Drop trait implementation, which decreases the "owners" counter by 1
- When the owners counter reaches 0, then the memory is cleaned up
- See : See rc_demo.rs

author nlahn@radford.edu; modified by ibarland@radford.edu.

Limitations of Rc

- Rc allows for multiple owners of data, but…
- You can only borrow the data **immutably** &
- You can’t borrow data mutably &mut

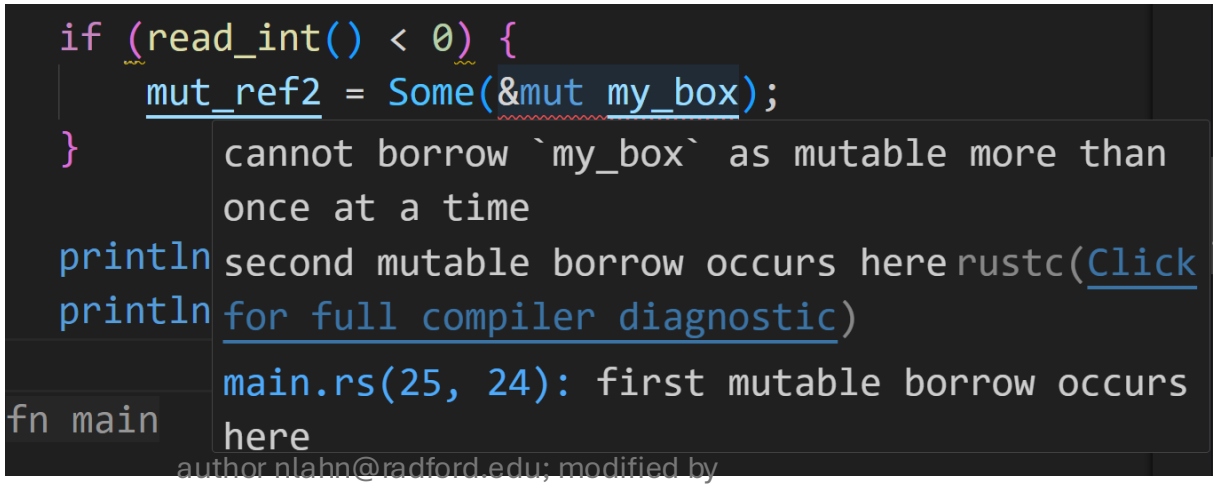


Why can't Rc be mutable?

- Box can easily be mutable!
- Why can't Rc?
  - Answer :
    - Rc can have multiple owners
    - Rust doesn't allow for multiple mutable references
    - Rust can't allow multiple Rc's to return mutable references because it can't guarantee counts at compile time!
- But sometimes Rust can be over zealous…

Why can’t we just use Box?

- Sometimes, the Rust compiler doesn’t know something is safe, even if we know it will be
- If Rust **can’t prove** that multiple &mut on the same box is **impossible**, it won’t allow it.
- See box_not_multiple_mut.rs



Solution : `RefCell<T>`

- `RefCell` is a special smart pointer that checks for multiple `&mut` at runtime instead of compile time!
- It's the same as Box, except for it delays the multiple `&mut` checking until **runtime**
- It does this by keeping a count of all the & and &mut references to its data
  - If it finds that a `&mut` and another reference are used simultaneously, it will panic!

- Important : `RefCell` still enforces Rust's ownership, but it does it at **runtime** instead of **compile time**

RefCell example

- See ref_cell_demo.r
- Compare this with the box_not_multiple_mut.rs code from earlier.
- This one works, but it could panic at runtime!

Limitations of RefCell

- `RefCell` is enforced at runtime
  - If Rust can enforce the constraints at compile time, use Box instead
- `RefCell`, like Box, only allows for one owner…
  - What if we want the multiple owners **and** runtime enforcement of reference counts?

author nlahn@radford.edu; modified by ibarland@radford.edu.

Interior Mutability Design Pattern

- What if we want the multiple owners **and** runtime enforcement of reference counts?
- Solution:
  - Use the type Rc<RefCell<T>>
  - Rc allows for multiple owners
  - RefCell is a single chunk of memory shared by multiple Rc owners
  - The single RefCell keeps track of the number of & and &mut attached to it so as to enforce Rust's ownership rules at **runtime**

Beware of Memory Leaks

- Most of the time, memory leaks in Rust are impossible
- However, if you use things like `Rc<RefCell<T>>`, then it's possible to introduce a memory leak
  - Each chunk of data can have multiple owners
  - A cycle is never deallocated…
  - This can be resolved using ***yet another*** special smart pointer known as `Weak<T>`, but this is beyond our current scope…
  - You'll encounter this if you try to make a doubly linked list or a graph data structure (anything with cycles)

author nlahn@radford.edu; modified by ibarland@radford.edu.

Summary

Rust has a whole lot of “smart pointers”

- `Box<T>` : Most basic; one owner, enforces ownership rules at compile time!
- `Rc<T>` : Allows for multiple owners, avoids memory safety issues at compile time (still very safe), but deallocation handled at runtime (slightly less efficient than `Box<T>`)
- `RefCell<T>` : Like Box, there is just one owner, but it doesn’t not enforce ownership rules involving `&mut` and `&` counts at compile time. Instead, enforces these at runtime
- `Rc<RefCell<T>>` : Allows for a combination of multiple owners (`Rc`) and runtime enforcement of ownership (`RefCell`)
- `Weak<T>` : Variant of `Rc<T>` that will not prevent data from being deallocated; can be used to avoid memory leaks from cycles with `Rc<T>`