CMSC 330: Organization of Programming Languages

Reference Counting
and Interior Mutability
Rust Ownership and Mutation

• Recall Rust ownership rules
  – Each value in Rust has a variable that’s called its owner; there can be only one
  – When the owner goes out of scope, the value will be dropped

• Recall Rust mutability rules
  – Mutation can occur only through mutable variables (e.g., the owner) or references
  – Rust permits only one borrowed mutable reference (and no immutable ones at the same time)
But: Mutation and Sharing is Useful

- Example: a simple spreadsheet

```rust
struct CellStyle { fontSize: f64 }
struct Cell { style: CellStyle }
struct Table { cells: [Cell; 128] }

- So: a Table owns its Cells

- But: a format inspector needs to read and write the cell data
  - Ensuring only one borrowed mutable reference would be awkward
  - Easier if the inspector has its own reference
Another Example

• Suppose you have a multiplayer chess game
  – Local data structures record the board state
  – Maybe the board is owned by the window that contains it

• What happens when a new move comes in from the network?
  – That’s handled by a different software component, not the window

• Simplest design is to have multiple (mutable) references to the board
  – But Rust doesn't allow that
Relaxing Rust's Restrictions

• Architecturally, designating one owner that all accesses must go through can be awkward
  – We might end up wanting shared mutable access to the owner!

• Rust provides APIs by which you can get around the compiler-enforced restrictions against multiple mutable references
  – Use reference counting to manage lifetimes safely
  – Track borrows at run-time to overcome limited compiler analysis
  – Discipline is called interior mutability
  – But: extra checks at space and time overhead; some previous compile-time failures now occur at run-time
Multiple Pointers to a Value

• What’s wrong with this code?

```rust
enum List {
    Nil,
    Cons(i32, Box<List>),
}

fn main() {
    let a = Cons(5, Cons(10, Box::new(Nil)));
    let b = Cons(3, Box::new(a));
    let c = Cons(4, Box::new(a)); // fails
}
```

- `Box::new` takes ownership of its argument, so the second `Box::new(a)` call fails since `a` is no longer the owner

• How to allow something like this code?
  – Problem: Managing lifetime
Managing Lifetimes Dynamically

- Benefit of ownership: compiler knows when to free memory
  
  ```
  let nil_box = Box::new(List::Nil);
  // free memory HERE (nil_box is going out of scope)
  ```

- Suppose `Box` didn't own its data:

  ```
  let nil_box = Box::new(List::Nil);
  let one_list = List::Cons(1, nil_box);
  
  { let two_list = List::Cons(2, nil_box);
    // two_list is going out of scope; free nil_box too?
  }
  ```

- `(Box` does own its data so the above pattern is not allowed.)

```rust
enum List {
    Nil,
    Cons(i32, Box<List>)
}
```
Rc<T>: Multiple Owners, Dynamically

- This is a *smart pointer* that associates a *counter* with the underlying reference
- Calling **clone** copies the pointer, not the pointed-to data, and *bumps* the counter by one
  - By convention, call `Rc::clone(&a)` rather than `a.clone()`, as a visual marker for future performance debugging
    - In general, calls to `x.clone()` are possible issues
- Calling **drop** reduces the counter by one
- When the counter hits **zero**, the data is **freed**
Rc::clone “Shares” Ownership

- Rc associates a refCount with the value

```
let x = Rc::new(42);
let y = Rc::clone(x);
let z = Rc::clone(x);
```

- stack (for example)
- heap

- does heap allocation, like Box::new, but uses reference counting
- clone() increments reference count
enum List {
    Nil,
    Cons(i32, Rc<List>)
}

use List::{Cons, Nil};

fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a)); // ok
}

Nb. Rc::strong_count returns the current ref count
Reference Counting: Summary

- To *create*: `let r = Rc::new(...);`
- To *copy* a pointer: `let s = Rc::clone(&r);`
  - Increments the reference count
- To *move* a reference: `let t = s;`
  - Does *not* increment reference count; `s` no longer the owner
- To *free* is automatic: `drop` is called when variables go out of scope, reducing the count; freed when 0

- See docs: