CMSC 330: Organization of Programming Languages

Smart Pointers
in Rust
Smart Pointers

- A smart pointer is a reference plus metadata, to provide additional capabilities
  - Originated in C++
  - Examples seen so far: String, Vec<T>
- Usually implemented as structs
  - Which must implement the Deref and Drop traits
- New ones we will see: Box<T>, Rc<T>
  - There are several others, such as Ref<T>
  - And you can make your own; see the book!
**Box<T> Smart Pointers**

- **Box<T>** values point to heap-allocated data
  - The **Box<T>** value (the pointer) is on the stack, while its pointed-to **T** value is allocated on the heap
  - Has **Deref** trait – can be treated like a reference
    - More later
  - Has **Drop** trait – will drop its data when it dies

- **Uses?**
  - Reduce copying (via an ownership move)
  - Create dynamically sized objects
    - Particularly useful for recursive types
A Box<T> value points to heap-allocated data. Therefore, it cannot be dropped when the owner goes out of scope.

A. True
B. False
Quiz 1

A Box<T> value points to heap-allocated data. Therefore, it cannot be dropped when the owner goes out of scope.

A. True
B. False
Example: Linked List

- Naïve attempt doesn’t work
  - Compiler complains that it can’t know the size of List
  - The Cons case is “inlined” into the enum
    - Since a List is recursive, it could be basically any size
- Use a Box to add an indirection
  - Now the size is fixed
    - i32 + size of pointer
      - Nil tag smaller

```rust
enum List { 
    Nil, 
    Cons(i32, List) 
}
```
Creating a LinkedList

```rust
type List = Option<Box<[List]>>;

fn main() {
    let list = Cons(1, Cons(2, Cons(3, None)));
    ... // data dropped at end of scope
}
```
Deref Trait

• If x is an int then &x is a &{int}
  – Can use * operator to dereference it, extracting the underlying value
    • *(&x) == x

• Can use * on Box<T> types
  – Deref trait requires deref(&self) -> &T method
  – So that *x translates to *(x.deref())

• deref returns type &T and not T so as not to relinquish ownership from inside the Box type
Deref Coercion

• The Rust compiler automatically inserts one or more calls to `x.deref()` to get the right type
  – When `&T` required but value `x: U` provided, where `U` implements `Deref` trait
  – In particular, at function and method calls

• Also a `DerefMut` trait
  – Deref coercion works with this too (see Rust book)
Example

```rust
fn hello(x: &str) {
    println!("hello {}", x);
}
fn main() {
    let m = Box::new(String::from("Rust"));
    hello(&m); // same as hello(&(*m)[..]);
}
```

- `&m` should have type `&str` to pass it to `hello`
- So, compiler calls `m.deref()` to get `&String`, and then `deref()` again to get `&str`
Drop Trait

• Provides the method `fn drop(&mut self)`
  – Called when the value implementing the trait dies
  – Should be used to free the underlying resources, e.g., heap memory

• May not call drop method manually
  – Would lead to a double free when Rust calls the method again at the end of a scope
  – Can call `std::mem::drop` function in some circumstances
Multiple Pointers to a Value

• What’s wrong with this code?

```rust
fn main() {
    let a = Cons(5,
                 Box::new(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` not owned

• How to allow something like this code?
Rc<T> to the Rescue

• 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
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
Rc::clone produces a new pointer to the same value in the heap. Because it shares the reference, programmer has to destroy the pointed-to value.

A. True  
B. False
Quiz 2

Rc::clone produces a new pointer to the same value in the heap. Because it shares the reference, programmer has to destroy the pointed-to value.

A. True
B. False
More

• See the Rust book for
  – How to get more flexible borrowing rules using `Ref<T>` and `RefCell<T>` types
    • Allows for mutability
  – How to use such pointers to make useful tree-based data structures
    • With lifetimes that may extend beyond the creating scope
  – How you can end up with reference cycles leading to a memory leak
    • And how you can use `Weak<T>` types to prevent them

• Check out *The Rustonomicon* for how to implement your own smart pointers!