Rust Smart Pointers

CPSC 310 - Programming Languages

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
  - Has Drop trait; will drop its data when it dies
- Uses?
  - Reduce copying (via an ownership move)
  - Create dynamically sized objects (useful for recursive types)

Example: Linked List

- Naive attempt does not work

```
enum List {
    Nil,
    Cons(i32, List)
}
```

  - Compiler complains that it cannot know the size of `List`
  - The `Cons` case is "inlined" into the `enum`

- Use a `Box` to add an indirection

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

  - Now the size is fixed; `i32` + size of pointer

Creating a Linked List

- Example

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

use List::{Cons, Nil};

fn main() {
    let list = Cons(1,
        Box::new(Cons(2
            Box::new(Nil))));
    ... // 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
- 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 to not 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, for when an object is mutable
  - `Deref` coercion works with this too

Example

- Example

```
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`
- Compiler calls `m.deref()` to get `&String` and then `deref()` again to get `&str`

`Drop` Trait

- The `Drop` trait provides the method `fn drop(&mut self)`
  - Called when the value implementing the trait goes out of scope
  - Should be used to free the underlying resources
- 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

Summarizable Example

- Recall the Summarizable example

```
pub trait Summarizable {
    fn summary(&self) -> String {
        String::from("none") // default implementation
    }
}
```

- Let us make a general summary printing function
- First attempt: `fn print_summary(s : Summarizable) {...}`
  - This means the caller moves the argument to the function when calling
  - This means the data in the argument needs to be moved
  - How many bytes is the data?

Summarizable Example (continued)

- Second attempt (still incorrect)

```
fn print_summary(s: &Summarizable) {
    print!("{}", s.summary);
}
```

  - Which implementation of `summary` should be used? (We don't know the type of `s`)

Dynamic Dispatch

- Object oriented languages, like Java, have dynamic dispatch which determines the correct method to use at run-time.
- To implement dispatch in Rust, we use trait objects
- A trait object pairs data with runtime type information

Trait Objects

- Use the `dyn Summarizable` wrapped in a `Box`

```
fn print_summary(s: Box<dyn Summarizable>) {
    println!("{}", s.summary());
}
```

- Callers use `Box` to put the data on the heap

```
pub fn main() {
    let b = Box::new(42);
    print_summary(b);
}
```

- `Box<i32>` is a pointer to a heap-allocated `i32`
- `Box<dyn Summarizable>` is a fat pointer to a heap-allocated `Summarizable`

Box: a Kind of Smart Pointer

- A smart pointer is a reference plus metadata to provide additional capabilities
  - Originated in C++
  - Examples seen so far: `String`, `Vec<T>`, `Box<T>`
- Usually implemented as `structs`
  - must implement the `Deref` and `Drop` traits
- Check out The Rustonomicon for how to implement your own smart pointers

Box Summary

- Use `Box<T>` to heap-allocate data, and reduce copying
  - Useful for non-cyclic, immutable data structures
- Use trait objects of type `Box<dyn Trait>` to implement dynamic dispatch
  - For any trait type `Trait`
  - `Box` lets you use fat pointers for `dyn` trait objects to provide runtime type information to enable dynamic dispatch
  - If you try to pass traits without `Box`, you may get errors about `Sized` because the compiler does not know how big things are

Rust Ownership and Mutation

- Recall Rust ownership rules
  - Each value in Rust has a variable that is 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 or references
  - Rust permits only one borrowed mutable reference

Mutation and Sharing is Useful

- Example: Multiplayer board game
  - Local data structures record the board state
- What happens when a new move comes in from the network?
- Simple design is to have multiple (mutable) references to the board
  - Rust does not allow that

Rust's Restrictions

- Rust provides APIs by which you can get around the compiler enforced restrictions against multiple mutable references
  - Use reference counting to manage lifetime safety
  - Track borrows at run-time to overcome limited compiler analysis
  - This is called interior mutability
  - The extra checks cost space and time overhead; some previous compile-time failures now occur at runtime

Multiple Pointers to a Value

- What is wrong with this code?

```
fn main() {
    let a = Cons(1,
        Box::new(Cons(2
            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 then owner

- How do we allow something like this code?
  - Problem: lifetime management

Managing Lifetimes Dynamically

- The benefit of ownership is that the compiler knows when to free memory

```
let nil_box = Box::new(List::Nil);
```

- Suppose that `Box` did not own its data

```
let nil_box = Box::new(List::Nil);
let l1 = List::Cons(1, nil_box); // error
{
  let l2 = List::Cons(2, nil_box);
  // l2 is about to go out of scope; free nil_box?
}
```

`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
- Calling drop reduces the counter by one
- When the counter hits zero, the data is freed

List with Sharing

- Example

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

use List::{Cons, Nil};

fn main() {
    let a Rc::new(Cons(1,
      Rc::new(Cons(2,
        Rc::new(Nil))));
    let b = Cons(3, Rc::clone(&a));
    let c = Cons(4, Rc::clone(&a)); // ok
}
```

Reference Counting Summary

- To create: `let r = Rc::new(...);`
- To copy a pointer: `let s = Rc::clone(&r);`
- 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