ECE459: Programming for Performance Winter 2019

Lecture 37 — Ownership in Rust

Patrick Lam *2019-04-05*

Runtime safety. We said that Rust is safe. One way in which it is safe is for arrays. Rust has tuples and structs. Hard to go out of bounds on those. Rust also has arrays. Like with any language, one can imagine going beyond the ends of an array. Rust defines the behaviour of going beyond the end of an array: it is a runtime exception ("panic"), unlike C/C++, where it is undefined behaviour.

```
let a = [1,2,3,4,5];
let index = 10;
println!("error!␣{}", a[index]); // panics here.
```

What's special about Rust?

Let's step back and do some Rust propaganda.

- harder to write unsafe code: compiler + runtime ensure safety. No arrays-out-of-bounds accesses, null pointers (at all), wild pointers;
- yet can still write low-level code;
- supports zero-cost abstractions (like C++ as discussed in L18);
- designed with ergonomics in mind;
- type system obviates need for either garbage collection or manual memory management (which you will get wrong)[1];
- type system prevents race conditions;
- dependency management using crates.

As far as I know, Firefox's rendering engine, Project Servo, is the largest deployed Rust codebase. [size?]

Rust lectures outline. Here's what else we are going to talk about. Chapter references from *The Rust Programming Language*.

Ch4: what's special about Rust: *ownership*;
Ch15: smart pointers (briefly);
Ch16: fearless concurrency.

Also relevant to this course, but here we are in week 12, so we're not going to talk about it.

Ch10: lifetimes;
Ch19: unsafe Rust.

Ownership in Rust [Chapter 4.1]

Also known as "how to fight the borrow checker and win".

Rust uses ownership types to manage its heap. Ownership types were not invented by the Rust community, but Rust is the first production-scale language to deploy it. The alternatives are `malloc/free` in C, or `new`/GC in Java. Rust does still have a stack, but we'll see when things go on the stack vs when they go on the heap.

[1] Well, mostly. Sometimes you need to use ref-counted data, and we'll see that.

The Rules

1. Each value in Rust has a variable that *owns* it.
2. This variable is *unique*.
3. When the owner goes out of scope, the value will be dropped (aka freed).

Variable scopes are fairly standard.

```
fn main() {
  println!("start");
  { // no s
    let s = "I␣am␣s";
    println!("s␣is␣{}", s);
  } // s now out of scope
}
```

OK, let’s put something on the heap. We’ll be using Rust `String` objects rather than string literals. String literals are compile-time constants. String objects contain a heap component, which may be allocated and freed.

(What can go wrong with heap allocation? You might not free/free too late; free too early; double free. GC manages this through an approximation: if you have no more pointers to it, then it doesn’t spark joy, and you don’t need it anymore. Rust goes a different way.)

```
fn main() {
  let s = String::from("hello"); // immutable String
  let mut s2 = String::from("459␣assignments"); // mutable String
  s2.push_str(",␣maybe?");
  println!("got␣string␣{}", s);
}
```

Rust uses rule #3: if something goes out of scope, then drop (free) it. This is quite like C++ RAII (Resource Acquisition is Initialization).

Still, we need a solution for objects that live beyond their original scope, e.g. return values.

```
fn return_a_string() -> String {
  let s = String::from("longevity");
  return s; // transfers ownership (moves) to caller
}

fn main() {
  let returned_string = return_a_string();
  println!("string␣{}", returned_string);
}
```

So, Rust frees owned values when variables go out of scope. Also, Rust calls “drop” (akin to a destructor) on objects that go out of scope. Note that going out of scope, not the drop call, is what actually causes the free.

Transferring Ownership (aka move semantics)

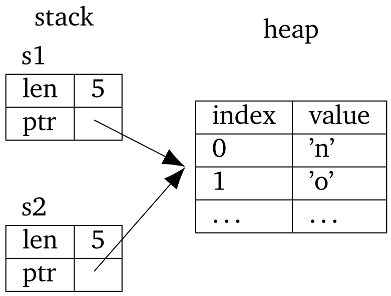
Let’s look at an example.

```
let s1 = String::from("no␣surprise");
println!("can␣print␣{}", s1);
let s2 = s1;
println!("can␣still␣print␣{}", s2);
println!("this␣line␣won’t␣compile!␣{}", s1); // no longer owns
```

(Note that string literals, or ints, or anything only on the stack, doesn’t have this behaviour—they are copied, or technically, they have the “Copy” trait.)

OK, so what’s going on? Let’s take a step back.

Rust strings are a hybrid, containing both a stack part and a heap part.



The assignment `let s2 = s1` carries out a shallow copy of the stack part. Rust does not automatically copy the heap part.

Now, recall that Rust has automatic memory reclamation. How can that work? What gets freed? Here, `s1` and `s2` are in the same scope. When they go out of scope, what should get freed? We don’t want to double free.

Key idea. For automatic memory reclamation to work, we give `let s2 = s1` *move semantics* (as in C++). After the move, `s1` is no longer valid. Ownership of the heap part is moved from `s1` to `s2` by the assignment.

That is, `s1` no longer owns the heap object and is not responsible for freeing the heap part when it goes out of scope. Only when `s2` goes out of scope do we free the heap object. And because the heap object only has one owner, it is only freed once.

Note: deep copy is possible with “clone”, but we have to trigger that explicitly.

Want to know more about ownership? Here’s a blog post:

`http://squidarth.com/rc/rust/2018/05/31/rust-borrowing-and-ownership.html`

References