ECE459: Programming for Performance Winter 2020

Lecture 35 — Rust

Patrick Lam & Jeff Zarnett *2019-12-18*

This, Too, Shall Fade and Pass Away

In ECE 459 we've used C and C++ as systems languages. A lot of your previous courses have been in one of those and it's entirely possible that one of those was your first programming language and perhaps even the one you've used the most. The languages themselves have their strengths and weaknesses, of course, but there's no denying that these languages come without some of the niceties found in other languages like clever static type checking and garbage collection.

The nature of the languages make it hard, or even impossible, to write code that is fast, correct, and secure. The focus of this course hasn't been on security. But in many cases, writing insecure fast code isn't the right thing. Is it even possible to write secure C and C++?

Maybe not. The usual arguments are something along the lines of experience. Experience isn't it either, given this quotation from Robert O'Callahan: "I cannot consistently write safe C/C++ code.[1]" (17 July 2017) (Holds a PhD in CS from Carnegie Mellon University; was Distinguished Engineer at Mozilla for 10 years; etc.)

What about use of better tools and best practices? March 2019: disclosure of Chrome use-after-free vulnerability[2]; 0-day attacks observed in the wild. Google implements best practices, and has all the tools and developers that money can buy!

Much of the advice about how to avoid these problems comes down to "try harder", which is... not helpful. If the strategy is just dragging people and saying that they need to pay more attention, or be more careful, or other similar phrase... this is going to constantly be an uphill battle. Expecting people to be perfect and make no mistakes is unrealistic. What we want is to make mistakes impossible.

A lot of the problems we frequently encounter are the kind that can be found by Valgrind, such as memory errors or race conditions. Other tools like code reviews and Coverity (static analysis defect-finding tool) exist. These are good, but not perfect. Valgrind, for example, only reports errors that it actually sees executed, so until and unless every function and every code path is run, it might not report a problem. Static analysis tools try to track down problems at compile-time, and that seems like a lot better of a solution.

I like to solve not just an individual problem, but an entire class of problems all at once. A recent example: if you change the contents of a list in a background thread while it's being rendered, the rendering thread will fail because the list has changed. I can fix the line of code so the list manipulation does not happen during rendering, and that fixes it once, but not forever: in the future another person could write code that calls this function from a background thread. There's no good way (in Java, sadly) to make it so invoking this function incorrectly is a compile-time error, so the best I can do is set a trap in it that throws an error if called inappropriately, so that the responsible developer will find what they did wrong during development and testing. Compile-time error checking is preferable to run-time.

This brings us to Rust. It is an alternative to C/C++. It is a new-school secure systems programming language used by Mozilla's Project Quantum. A design goal of this language is to avoid issues with memory allocation and concurrency. We'll consider both concepts, but we won't dwell too much on the syntax (mostly for time reasons). It's worth reading up on the topic (outside of lecture) if you are curious about the language, though, and it might help you to understand the examples better.

[1] https://robert.ocallahan.org/2017/07/confession-of-cc-programmer.html
[2] https://security.googleblog.com/2019/03/disclosing-vulnerabilities-to-protect.html

Rust

This material is based on *The Rust Programming Language* by Steve Klabnik and Carol Nichols [KN18] and I'll make references as appropriate.

Here's some Rust code.

```
fn main() {
  let x = 42; // NB: Rust infers type "s32" for x.
  println!("x␣is␣{}", x);
}
```

By default, Rust variables are *immutable*.

```
fn main() {
  let x = 42; // NB: Rust infers type "s32" for x.
  x = 17; // compile-time error!
}
```

Let's consider two examples that look similar but have drastically different meanings.

```
let x = 1729;
let x = 88;
println!("shadowed␣x␣is␣{}", x);
```

```
let mut x = 33; // mutable
x = 5;
println!("mutated␣x␣is␣{}", x);
```

In the first case, old "x" still exists but is inaccessible under the name "x". In the second case, the storage cell for "x" used to contain 33 and then contains 5. The difference matters, for instance, when there are references to "x". This example is a bit silly though; the real usage for shadowing is perhaps something more familiar, specifically parsing (or other transformation):

```
let mut guess = String::new();

io::stdin().read_line(&mut guess)
    .expect("Failed␣to␣read␣line");

let guess: u32 = guess.trim().parse()
     .expect("Please␣type␣a␣number!");
```

In this example, the data is read in as a string and then turned into an unsigned integer. We like this because we can re-use the variable name without having things like `guess` and `guess_parsed` or other "what do I call this now" problems.

Rust immutability. By default, a variable in Rust is immutable. You can make it mutable if you choose, explicitly by declaring it as mutable. Lots of concurrency issues involve the internal state of objects that are accessed by different threads. Structs or tuples are either all mutable or all immutable.(Although interior mutability is a thing in Rust. We're not talking about it.)

Rust obviously has compile-time constants and they are truly unmodifiable. These have to be known at compile time, and are truly a fixed value. This is different from an immutable type which is determined at runtime but cannot be changed once it has been assigned.

In C, you can cast away `const`-ness; not so in Rust. If something is not mutable in Rust, you can't cast it into mutability.

Perf implications. We mentioned immutability in Lecture 7. The best way to avoid having to use locks (even read/write locks still require writes to acquire the read lock): have no writes. However, there's a tradeoff. If your data structure is immutable but you want to update it (as we often do with data structures), you need to copy the data structure, at least partially. That can be slow.

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++);
- designed with ergonomics in mind;
- type system obviates need for either garbage collection or manual memory management (which you will get wrong)[3];
- 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?]

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.

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.)

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

```
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)

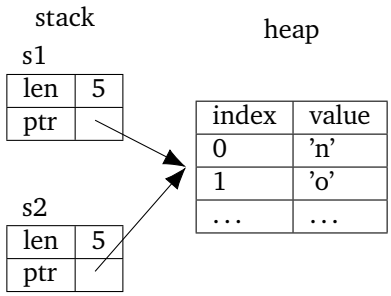
Let's look at an example.

```
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`

Functions and Return Values

Move semantics also applies to function calls and return values, e.g.

```
fn main() {
  let s = String::from("moving␣to␣callee");
  callee(s); // afterwards, s is invalid
}
fn callee(param:String) {
  println!("got␣{},␣param");
} // param goes out of scope, object dropped
```

If you return something, then the ownership passes back to the caller. Tuples help pass ownership of multiple objects, but this is still quite high-overhead for developers.

```
fn main() {
  let s = String::from("459");
  let len = calculate_length(s);
  println!("string␣{}␣has␣length␣{}", s /* we still have it! */,
           len);
}
fn calculate_length(s:&String) -> usize { // note the & for borrow
  s.len() // last expr is return value
} // s is ref so nothing goes out of scope
```

Borrowing and mutation

"Can I please borrow your object?" Like other variables, references are immutable by default. We can have mutable references, though.

```
fn change(s:&mut String) {
  s.push_str("more");
}
fn main() {
  let mut main_str = String::from("some␣");
  change(&mut main_str); // create mutable ref to main_str
}
```

There can be only one (mutable). The following code won't compile:

```
let mut s = String::from("one");
let r1 = &mut s;
let r2 = &mut s; // rustc complains!
```

In fact, while `r1` is in scope, you can't do anything with the original `s`. The only way to access the string is through `r1`. Once `r1` goes out of scope, you can create `r2`.

Since there is only one way to access `r1`, then there will be no race conditions.

This is OK:

```
let mut s = String::from("one");
let r1 = &s;
let r2 = &s; // no problem!
```

But you can't then do `let r3 = &mut s;`.

How many? You can have as many outstanding immutable refs as you want. If there are any immutable refs, you can't have *any* mutable refs. The immutable refs must go out of scope first.

You also can't commit use-after-free errors: you can't return a ref that outlives its value.

```
fn dangle() -> &String {
  let s = String::from("hello");
  &s // rustc complains: s goes out of scope with active refs
}
```

Rust also has *smart pointers*, which may be reference counted. This is like C++'s smart pointers, specifically `shared_ptr` (but Rust can tell you about some things at compile time which C++ will tell you at runtime). Normal Rust objects are more like `unique_ptr`.

We need reference counted heap objects e.g. to implement graphs. We don't have enough time to talk about smart pointers, but Chapter 15 of the Rust book is good.

Fearless Concurrency

As with many other aspects of Rust, we trade compiler errors for runtime errors; in this case, runtime concurrency errors like race conditions. That is, the type system ensures concurrency safety!

Rust uses a fork/join model like pthreads. It delegates to the operating system's threads support and hence implements 1:1 threads.

```
let handle = thread::spawn(|| { // closure (can put args between ||)
  // thread code goes here
});
// main thread continues here
handle.join.unwrap(); // unwrap: panic in case of error
```

This is not too different from C++.

OK, how do we share data between threads? We can move it from main to thread:

```
let v = vec![1,2,3];
let handle = thread::spawn(move || { // move: everything accessed inside closure is moved
  println!("vector␣{:?}", v);
}); // no longer have access to v in main
handle.join().unwrap();
```

Rust is saving you from being able to concurrently access `v` in main and thread.

But that's only one way! This isn't quite enough.

Message Passing

One way to share data is message passing. We've seen this before (OpenMPI). In this case, each value still only has one owner. We use *channels*. The ownership passes from the sender, through the channel, to the receiver.

```
use std::thread;
use std::sync::mpsc; // multi producer, single consumer

fn main() {
  let (tx, rx) = mpsc::channel(); // tx is cloneable
  thread::spawn(move || { // here, tx goes to closure
    let val = String::from("april");
    tx.send(val).unwrap(); // val moved from sender
  });
  let received = rx.recv().unwrap();
  println!("got: {}", received);
}
```

Note the send/receive pair. There is also `try_recv` to do nonblocking receives.

Shared State

People debated for a long time which was better: shared state (like pthreads) or channels. Rust supports both. Of course, the problem with shared state is race conditions. Like manual memory management, we can manually acquire and release mutexes. What could possibly go wrong? Rust's ownership system will help.

We'll need to talk about multiple ownership. But let's talk about mutexes first.

```
use std::sync::Mutex;
fn main() {
  let m = Mutex::new(5); // mutex guards access to an i32
  {
    let mut num = m.lock().unwrap();
    // unwrap: maybe some other thread panicked while holding lock;
    //         then we panic too.
    *num = 6; // "deref" the mutex (is actually a smart pointer)
  } // release lock when num goes out of scope
  println!("m={:?}", m);
}
```

Well, that's fine, but it's just one thread. We really do need multiple ownership to share data. The shared data needs to be owned by all threads, and a naive solution will get rejected by the borrow checker. Instead, we have to use *reference counted cells*, implemented by `Arc`.

```
use std::sync::{Mutex, Arc};
use std::thread;
fn main() {
  let counter = Arc::new(Mutex::new(0)); // atomic reference cell
  let mut handles = vec![];

  for _ in 0..10 {
    let counter = Arc::clone(&counter); // clone the Arc
    let handle = thread::spawn(move || {
      let mut num = counter.lock().unwrap();
      *num += 1;
    });
    handles.push(handle);
  }
  for handle in handles {
    handle.join().unwrap();
  }
  println!("result: {}", *counter.lock().unwrap());
}
```

Rust guarantees that you have the appropriate lock, using ownership (possibly multiple ownership). Rust does not guarantee lack of deadlocks.

References

[KN18] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018. https://doc.rust-lang.org/book/.