ECE459: Programming for Performance Winter 2021

Lecture 2 — Rust Basics

Jeff Zarnett *2020-09-28*

Getting Started with Rust

Rather than just tell you to go off and learn all of Rust on your own, we will spend some time on the subject and tell you about important features and why and how they work towards the goal of programming for performance.

With that said, reading or watching material about a programming language is not a super effective way of learning it. There is really no substitute for actually writing code in the language. For this reason, some optional practice exercises/material is linked in the course resources. They might be trivial, but you'll gain a much better understanding of the subject by being hands-on. You (probably) can't learn to swim from watching videos...

What's *not* here? This isn't intended to cover how to declare a function, create a structure, create an enumeration, talk about if/else blocks, loops, any of that. The official docs explain the concepts pretty well and you'll get used to the constructs when you use them. We need to focus on the main objective of the course without getting sidetracked in how to print to the console.

This material is mostly based off the official Rust documentation [KNC20] combined with some personal experiences (both the good and bad kind).

Change is painful. Variables in Rust are, by default, immutable (maybe it's strange to call them "variables" if they don't change?). That is, when a value has been assigned to this name, you cannot change the value anymore.

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

For performance, immutability by default is a good thing because it helps the compiler to reason about whether or not a race condition may exist. Recall from previous courses that a data race occurs when you have multiple concurrent accesses to the same data, where at least one of those accesses is a write. No writes means no races!

If you don't believe me, here's an example in C of where this could go wrong:

```
if ( my_pointer != NULL ) {
    int size = my_pointer->length; // Segmentation fault occurs!
    /* ... */
}
```

What happened? We checked if `my_pointer` was null? And most of the time we would be fine. But if something (another thread, an interrupt/signal, etc) changed global variable `my_pointer` out from under us we would have a segmentation fault at this line. And it would be difficult to guard against, because the usual mechanism of checking if it is `NULL`... does not work. This kind of thing has really happened to me in production Java code. Put all the if-not-null blocks you want, but if the thing you're looking at can change out from under you, this is a risk.

Immutable in Rust is forever. The compiler will not let you make changes to something via trickery. You can ignore a `const` declaration in C by taking a pointer to the thing, casting the pointer, and changing through the pointer. Rust does not permit such dark magicks. It kinda defeats the point of Rust.

Of course, if you want for a variable's value to be changeable you certainly can, but you have to explicitly declare it as *mutable* by adding `mut` to the definition, like `let mut x = 42;`. Then later you can change it with `x = 0;`. My

general advice is that you want to minimize the number of times you use this. Still, there are some valid scenarios for using mutation. One is that it might be a lot clearer to write your code such that a variable is mutated; another is that for a sufficiently large/complicated object, it's faster to change the one you have than make an altered copy and have the copy replace the original.

Then there are constants. Constants are both immutable and immortal: they can never change and they are valid for the whole scope they are declared in. This is how you set program-wide constants that are always available and never change, like `const SPEED_OF_LIGHT_M_S: u32 = 299_792_458;`. This is how you would declare a truly global constant, but of course, things like that is somewhat discouraged since global variables are easily abused.

Shadowing. Something that isn't really "changing" the variable but looks a lot like it is, is *shadowing*, which is intended to address the problem of "What do I name this?" In another language you might have a variable `transcript` which you then parse and the returned value is stored in another variable `transcript_parsed`. You can skip that with shadowing, which lets you reuse the original name. An alternative example from the docs:

```
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. Conceptually, there are two variables, one of type `String` and the other of type `u32`. They just happen to have the same name.

0wn3d

The most important thing to tell you about Rust is the concept of *ownership*. This strongly distinguishes Rust from other programming languages. Ownership is Rust's strategy for memory management.

In languages like C, memory management is manual: you allocate and deallocate memory using explicit calls. In other languages like Java, it's partly manual—you explicitly allocate memory but deallocation takes place through garbage collection. Rust opts for neither of those, and uses ownership. That is, the compiler determines (at compile-time, of course) when allocated memory can be cleaned up. This imposes certain restrictions on how you write your code and will inevitably cause at least one moment where you angrily curse at the compiler for its refusal to let you do what you want. The compiler is only trying to help. Promise.

You might be thinking: what's wrong with garbage collection for this purpose? It is well-understood and lots of languages use it. Actually, the real answer is the magic word: performance. A language that is garbage-collected has to deal with two things: a runtime, and the actual costs of collecting the garbage.

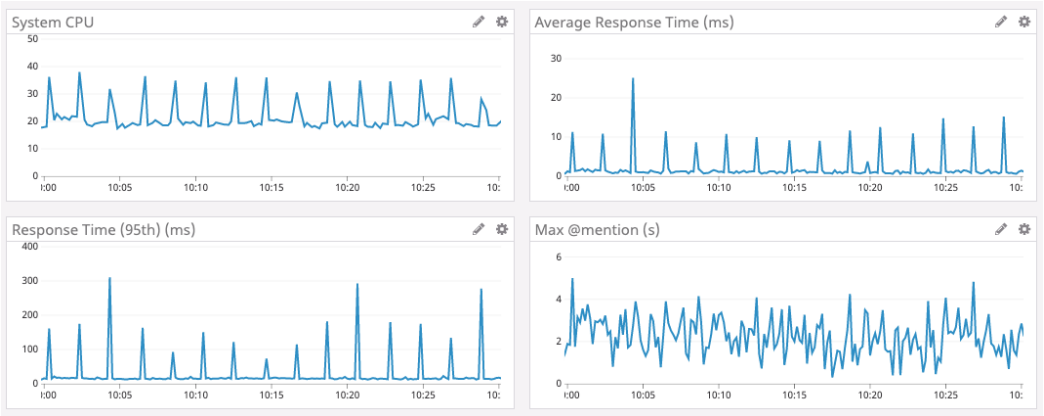
The runtime thing will seem familiar to you if you've programmed in Java or similar; it is the system that, at run-time, must keep track of what memory has been allocated and when to run the garbage collector and such. There's much more to what the Java runtime (JRE) does, but whatever it does comes with some performance penalty (no matter how small) because its functionality does not come for free.

The other part is that the garbage collection process can be expensive. In Java there are two kinds: the first is a simple cleanup of some trash which does not have much impact; the second is the stop-the-world behaviour where all execution is paused and the garbage collector decides what to keep and what to dispose of, and maybe reorganize memory. Also, the Garbage Collector can do this (1) whenever it wants, and (2) take as long as it feels like taking. Neither of which is great for performance, or for predictable performance.

Think you're macho and can avoid garbage collection by using C/C++? As we discussed last time, your C/C++ code is probably wrong. Well, mine is anyway, I don't know about yours. Aside from that, heap allocation and particularly deallocation is still not free even in C/C++. You don't pay garbage collection costs, but you do pay to manage

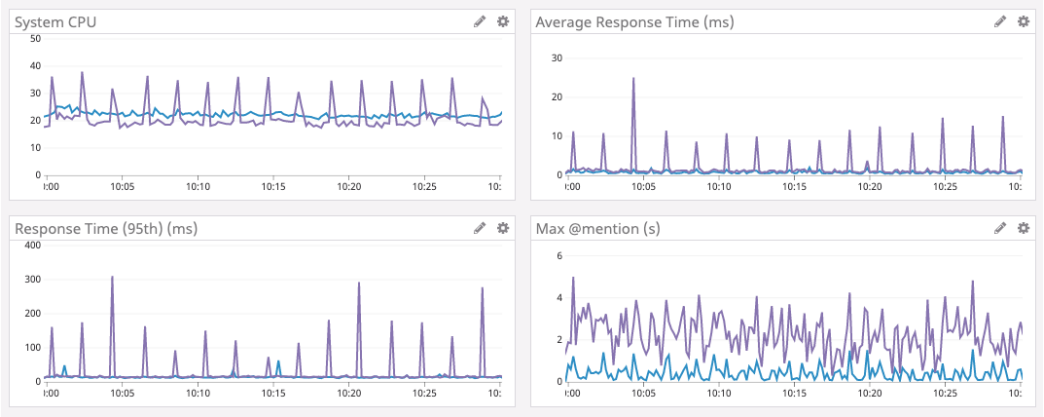
memory-related data structures. Rust makes it easier to allocate some things on the stack rather than the heap, which can in principle improve performance. But, sure, in general C++ and Rust's memory management overhead should be comparable. It's just that Rust is safe.

Real-World Example: Discord. If you want a real-world example of this, consider this graph from an article about a service at Discord [How20]:



To give a quick recap of the article; the garbage collector does its work and it adds a big latency spike. Rust would not have those spikes, because of ownership: when memory is no longer needed, it is trashed immediately and there's no waiting for the garbage collector to come by and decide if it can be cleaned up. The article also adds that even with basic optimization, the Rust version performed better than the Go version. Not only in terms of there being no spikes, but in many dimensions: latency, CPU, and memory usage.

See the following graphs that compare Rust (blue) to Go (purple):



I do recommend reading the article because it goes into some more details and may answer some questions that you have.

The Rules. That long introduction to the concept of ownership didn't explain very much about how it actually works; it just went into the *why* and how it relates to the objectives of this course. But the rules are pretty simple—deceptively so—and they are as follows:

1. Every value has a variable that is its owner.

2. There can be only one owner at a time.

3. When the owner goes out of scope, the value is dropped.

These rules draw a distinction between the value itself and the variable that owns it. So in a statement of `let x = 42;` there is memory associated with the value "42". That memory is the "value" in rule 1, and its owner is the variable `x`.

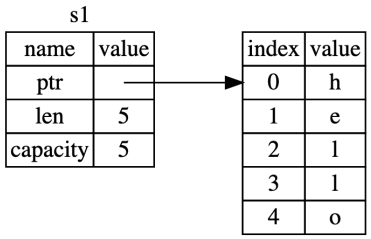
When `x` goes out of scope, then the memory will be deallocated ("dropped"). (This resembles the RAII (Resource Acquisition Is Initialization) pattern in languages like C++). Variable scope rules look like scope rules in other C-like languages. We won't belabour the point by talking too much about scope rules. But keep in mind that they are rigidly enforced by the compiler. See a brief example:

```
fn foo() {
  println!("start");
  { // s does not exist
    let s = "Hello␣World!";
    println!("{}", s);
  } // s goes out of scope and is dropped
}
```

The same principle applies in terms of heap allocated memory (yes, in Rust you cannot just pretend there's no difference between stack and heap, but ownership helps reduce the amount of mental energy you need to devote to this). Let's learn how to work with those! The example we will use is `String` which is the heap allocated type and not a string literal. We create it using the

```
fn main() {
  let s1 = String::from("hello");
  println!("s1␣=␣{}", s1);
}
```

A string has a stack part (left) and a heap part (right) that look like [KNC20]:



This makes it a bit clearer about what is meant when the rules say that when the owner (the stack part) goes out of scope, the value (the heap part) is deallocated.

That covers rules one and three…But that second rule is interesting, because of the "at a time" at the end: it means that there exists the concept of transfer of ownership.

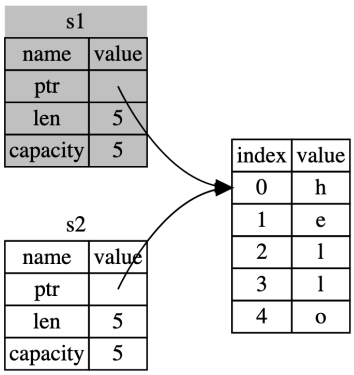
What's yours is mine. Move semantics have to do with assigning ownership from one variable to another. Ownership is really about heap memory, so stack allocated simple types (see the docs for a list—stuff like integers and booleans and floating point types) don't follow move semantics; they follow copy semantics. So the following code creates two integers and they both have the same value (5).

```
fn main() {
      let x = 5;
      let y = x;
}
```

But simple types on the stack are the exception and not the rule. Let's look at what happens with types with a heap component, though:

```
fn main() {
  let s1 = String::from("hello");
  let s2 = s1;
}
```

Here, no copy is created. For performance reasons, Rust won't automatically create a copy if you don't ask explicitly. (You ask explicitly by calling `clone()`). Cloning an object can be very expensive since it involves an arbitrary amount of memory allocation and data copying. This point is a thing that students frequently get wrong in ECE 252 in that that when doing a pointer assignment like `thing* p = (thing*) ptr;` that no new heap memory was allocated and we have `p` and `ptr` pointing to the same thing. But that's not what happens in Rust [KNC20]:



If both `s1` and `s2` were pointing to the same heap memory, it would violate the second rule of ownership: there can be only one! So when the assignment statement happens of `let s2 = s1;` that transfers ownership of the heap memory to `s2` and then `s1` is no longer valid. An attempt to use it will result in a compile-time error.

The compiler is even kind enough to tel you what went wrong and why (and is super helpful in this regard compared to many other compilers) [KNC20]:

```
$ cargo run
   Compiling ownership v0.1.0 (file:///projects/ownership)
error[E0382]: borrow of moved value: ‘s1‘
 --> src/main.rs:5:28
  |
2 |     let s1 = String::from("hello");
  |         -- move occurs because ‘s1‘ has type ‘std::string::String‘, which does not
   implement the ‘Copy‘ trait
3 |     let s2 = s1;
  |              -- value moved here
4 |
5 |     println!("{}, world!", s1);
  |                            ^^ value borrowed here after move

error: aborting due to previous error

For more information about this error, try ‘rustc --explain E0382‘.
error: could not compile ‘ownership‘.

To learn more, run the command again with --verbose.
```

Move semantics also make sense when returning a value from a function. In the example below, the heap memory that's allocated in the `make_string` function still exists after the reference `s` has gone out of scope because ownership is transferred by the `return` statement to the variable `s1` in `main`.

```
fn make_string() -> String {
  let s = String::from("hello");
  return s;
}

fn main() {
  let s1 = make_string();
  println!("{}", s1);
}
```

This works in the other direction, too: passing a variable as an argument to a function results in either a move or a copy (depending on the type). You can have them back when you're done only if the function in question explicitly returns it!

```
fn main() {
  let s1 = String::from("world");
      use_string( s1 ); // Transfers ownership to the function being called
      // Can't use s1 anymore!
}

fn use_string(s: String) {
    println!("{}", s);
    // String is no longer in scope - dropped
}
```

This example is easy to fix because we can just add a return type to the function and then return the value so it goes back to the calling function. Great, but what if the function takes multiple arguments that we want back? We can `clone()` them all...which kind of sucks. We can put them together in a package (structure/class/tuple) and return that. Or, we can let the function borrow it rather than take it...But that's for next time!

Do the Rules Work? With a stronger understanding of the rules and their practicalities, the obvious question is: do they work?! There's no point in having the rules if they don't accomplish the goal. We'll assume for the moment that there are no bugs in the compiler that violate the expected behaviour. And then let's consider this from the perspective of some things that can go wrong in a C program.

(Okay, alright, before we get there—we'll eventually learn to break rules and to use reference counted objects where if we get it wrong we can leak.)

- Memory leak (fail to deallocate memory)—does not happen in Rust because the memory will always be deallocated when its owner goes out of scope.
- Double-free—does not happen in Rust because deallocation happens when the owner goes out of scope and there can only be one owner.
- Use-after-free—does not happen in Rust because a reference that is no longer valid results in a compile time error.
- Accessing uninitialized memory—caught by the compiler.
- Stack values going out of scope when a function ends—the compiler will require this be moved or copied before it goes out of scope if it is still needed.

References

[How20] Jesse Howarth. Why Discord is switching from Go to Rust, 2020. Online; accessed 2020-09-12. URL: `https://blog.discord.com/why-discord-is-switching-from-go-to-rust-a190bbca2b1f`.

[KNC20] Steve Klabnik, Carol Nichols, and Rust Community. The Rust Programming Language, 2020. Online; accessed 2020-09-12. URL: https://doc.rust-lang.org/book/title-page.html.