Lecture 2 — Rust Basics

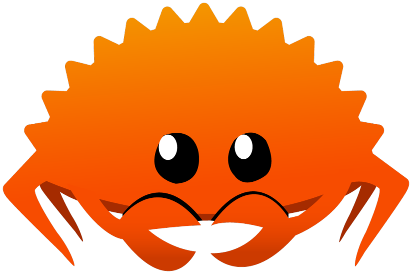
Jeff Zarnett
jzarnett@uwaterloo.ca

Department of Electrical and Computer Engineering
University of Waterloo

November 15, 2020

Rust

We won't tell you to just go learn Rust on your own...



Focus: important features: why & how they support the goal of performance.

Learning to Swim

Reading or watching about a programming language isn’t super effective.

There’s no substitute for writing code!

Suggestion: do practice exercises to become familiar with the language.

Goal-Setting

Some things aren’t here. We’re not covering the very basics of Rust.

The official docs are good and you will get used to the syntax as we use it.

Semicolon

Previously: C/C++/Java, where all statements end with semicolons.

In Rust that is not so: semicolons separate expressions.

The last expression in a function is its return value.

You can use `return` to get C-like behaviour, but you don't have to.

```rust
fn return_a_number() -> u32 {
  let x = 42;
  x+17
}

fn also_return() -> u32 {
  let x = 42;
  return x+17;
}
```

Change is Painful

Variables in Rust are, by default, immutable.

```
fn main() {
  let x = 42; // NB: Rust infers type "i32" for x.
  x = 17;     // compile-time error!
}
```

Immutability is good for performance.

The compiler can reason about the possibility of race conditions.

No writes? No race condition!

This has happened...

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

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

What happened? We checked if `my_pointer` was null?

Diamonds are Forever

Immutability in Rust is forever (ish).

The compiler will not let you make changes to something via trickery.

Rust grudgingly permits such dark magicks, but you you have to brand your code with the `unsafe` keyword and are subject to undefined behaviour.

Change is Necessary

If you want for a variable's value to be changeable you certainly can, but you have to explicitly declare it as *mutable*

Add `mut` to the definition, like `let mut x = 42;`

Generally, minimize the number of times you use this.

Rust forces you to make mutability explicit & has the compiler check your work.

The Tao is Eternal

There are constants, which are different from global variables.

Constants are both immutable and immortal.

`const SPEED_OF_LIGHT_M_S: u32 = 299_792_458;`.

They don't really exist at runtime and have no address.

Rust also has global variables, defined using `static`.

Shadowing

Shadowing is intended to address the problem of “What do I name this?”
An 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!");
```

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.

C++ supports memory management via RAII, and Rust does the same.

Rust does so at compile-time with guarantees, through ownership, which we'll discuss below.

Garbage Collection

You might be thinking: what’s wrong with garbage collection?

The real answer is the magic word: performance!

Runtime and actual costs of collecting the garbage.

JRE & Collection

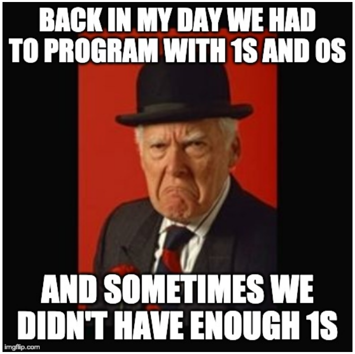
The JRE does stuff at runtime – its functionality is not free.

The Garbage Collector can run whenever it wants and take as long as it likes.

Image Credit: Daria Shevtsova

Klingon Programs Do Not Coddle the Weak

Garbage collection? Luxury! Klingons manage memory manually!


BACK IN MY DAY WE HAD TO PROGRAM WITH 1S AND 0S
AND SOMETIMES WE DIDN'T HAVE ENOUGH 1S
imgflip.com

It's hard to get C/C++ right and all the extra work done to verify that code takes effort that could go elsewhere.

0wn3d

Rust has the concept of *ownership*.

Rust uses ownership as a default memory management strategy.

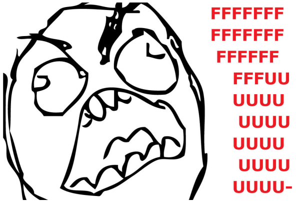
The compiler determines when allocated memory can be cleaned up.

Memory can be cleaned up when when nobody needs it anymore.

f7u12

Ownership imposes certain restrictions on how you write your code.

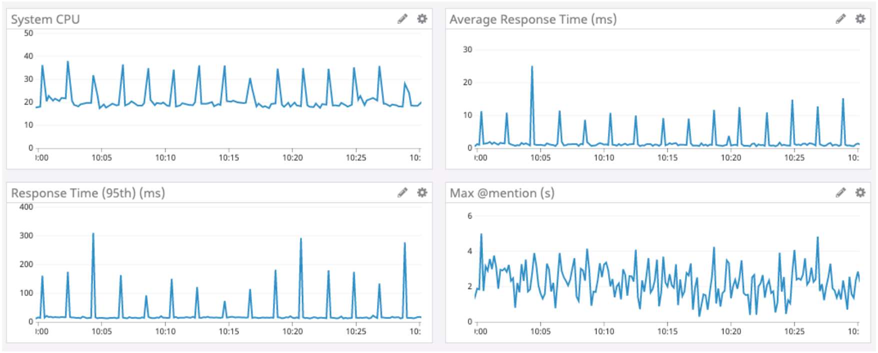
There will inevitably moments where you curse at the compiler for not letting you do what you want.



The compiler is only trying to help. Promise.

Real-World Example: Discord

Let's take a look at a real-life example of Go vs Rust:

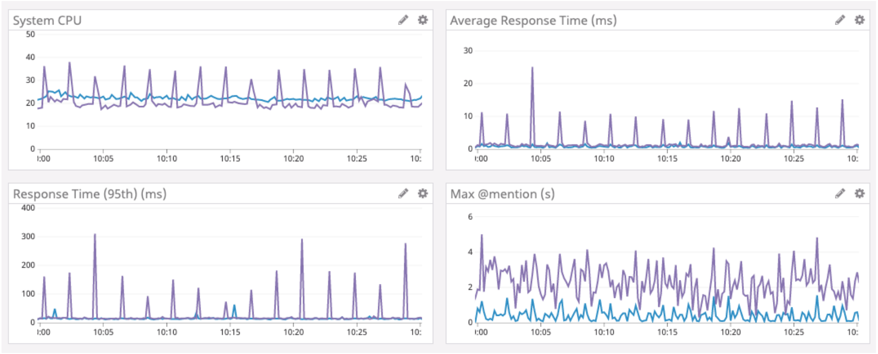


Go garbage collector does its work and it adds a big latency spike.

Real-World Example: Discord

Rust version performed better than the Go version on latency, CPU, and memory usage.

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



Rules

We talked about the why of ownership, but not the what.

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.

Note a distinction between the value itself and the variable that owns it.

```
let x = 42;
```

Out of Scope

Variable scope rules look like scope rules in other C-like languages.

They are rigidly enforced by the compiler.

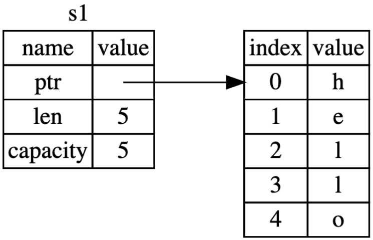
```
fn foo() {
  println!("start");
  { // s does not exist
    let s = "Hello World!";
    println!("{}", s);
  } // s goes out of scope and is dropped
}
```

Stack and Heap

The same principle applies in terms of heap allocated memory.

```
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:



Back to Rule 2

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 transferring ownership from one variable to another.

Some types use copy semantics instead, such as simple types.

Copy semantics are great when copies are cheap and moving would be cumbersome.

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

Move instead

But simple types are the exception and not the rule. Let’s look at what happens with types with a heap component:

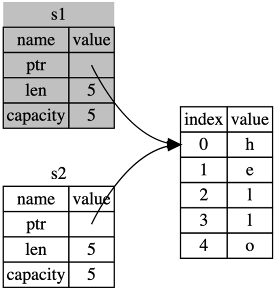
```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1;
}
```

Rust won’t automatically create a copy if you don’t ask explicitly.

(You ask explicitly by calling `clone()`).

Move instead

Here's what actually happens:



If both s1 and s2 were pointing to the same heap memory, it would violate the second rule of ownership!

Not Allowed

An attempt to use s1 will result in a compile-time error.

```
fn main() {
  let x = 5;
  let y = x;
  dbg!(x, y); // Works as you would expect!

  let x = Vec<u32>::new(); // similar to the std::vector type in C++
  let y = x;
  dbg!(x, y); // x has been moved, this is a compiler error!
}
```

This is Helpful

The compiler is even kind enough to tell you what went wrong and why:

```
plam@amqui ~/c/p/l/l/L02> cargo run
   Compiling move v0.1.0 (/home/plam/courses/p4p/lectures/live-coding/L02)
error[E0382]: use of moved value: ‘x‘
 --> src/main.rs:8:10
  |
6 |     let x = Vec::<u32>::new(); // similar to the std::vector type in C++
  |         - move occurs because ‘x‘ has type ‘std::vec::Vec<u32>‘, which does
              not implement the ‘Copy‘ trait
7 |     let y = x;
  |             - value moved here
8 |     dbg!(x, y); // x has been moved, this is a compiler error!
  |          ^ value used here after move

error: aborting due to previous error

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

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

It’s Dangerous to Go Alone, Take This

Move semantics also make sense when returning a value from a function.

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

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

Wait, come back with that...

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 one is easy to fix... clone or return all the thing?

Maybe we can let a function borrow it...

But Do the Rules Work?

Consider this from the perspective of things that can go wrong in a C program.

- Memory leak (fail to deallocate memory)
- Double-free
- Use-after-free
- Accessing uninitialized memory
- Stack values going out of scope when a function ends

(We'll eventually learn to break rules in a future lecture)

A free lunch?

Rust of course doesn’t solve every problem in the world.

It does solve memory management well.

I wouldn’t even say that it requires more of the programmer: it requires more of the programmer at compile-time, not at debug-time.

Downsides:

- Static typing
- Ecosystem
- Compiler