**Rust Prime Number Generation Post-mortem**

By: Andrew Murwin

In the realm of honors contracts, this one was relatively straightforward. Learn Rust and use it to write a picture maker. The specifications were straightforward and helpful, and outlined everything I needed to do. Compared my last coding honors contract in CSE205, which was programming custom GUIs, this one was a lot less painful, and a lot “shorter”. Compared to 1000+ lines of code for JavaFX, 360 lines post-post commenting seems easy by comparison. In the end though, Rust is a much different programming language than Java.

**What Went Right**

1. Learning Rust

Like all programming languages, Rust takes time to adjust to. On the surface, it holds a syntax similar to that of C/C++, which made adjusting to the syntax easy, since that was the majority of CSE240. However, there were other aspects that were annoying to deal with. One was variable declaration. When creating a variable, you don’t have to declare its type, which makes it like python and easy to adjust to. At the same time though, you can declare the variable type, and sometimes you’re forced to. Rust variables get their data type by reading the code for the first assign statement, but if there isn’t a guaranteed one, the programmer must designate a type. For the most part, Rust felt like python with semicolons.

2. BMP File Formatting

I went in expecting BMP formatting to be painful, especially after hearing that Professor Selgrad couldn’t figure it out in Java, a language I consider to be relatively similar to C/C++ and Rust. Luckily, Rust file I/O is significantly simpler than most languages requiring just one command to write anything to a file. With file writing out of the way, that only left the actually difficult part, creating a .bmp file. Unlike with multithreading, I had to write my own code for creating the file. I attempted to write my code based purely on what I could find for bitmap file headers, but all of the different formats made it too confusing to test. Eventually, I found a bitmap writer written in C++. While not particularly translatable, it showed me what each set of bytes was supposed to represent and where to get the data for it. Based on this, and some trial and error, I was able to write the bitmap write. There are two interesting things to note about bitmaps, which may cause issues if not accounted for. The first is that, for some reason, mathematicians were allowed to have input in the original formatting, so rather than the origin being in the top left like every other program, bitmap origins are in the bottom left, which can cause the row order to be reversed. Second, RGB values are stored as BGR (Blue, Green, Red) order specific values, which may cause off-coloring issues.

3. Debugging

Considering that I was working in the fabulous Visual Studio Code (aka the fancy text editor) for my entire project, the majority of my “debugging” consisted of print statements spontaneously added and removed as progress continued. Due to the safe nature of Rust, more advanced debugging was almost never necessary, which allowed my to progress quickly through most parts of the project. One part that was troublesome was debugging the file writing. If a file is written incorrectly, computers don’t actually tell you what’s wrong with the file, and viewing the bytes of the file can be troublesome. Luckily, CSE240 required an assignment that revolved around file I/O. Adding a print statement to working code seemed like a much better use of time than learning file reading in Rust, and I turned out to be correct. Overall, Rust actually makes debugging easy by adding a literal “debug” option to printing, which can print much more complicated values.

**What Went Wrong**

1. Getting Started

In all honesty, the most difficult part of completing the honors contract was the setup work required to have a project. Between the day I originally reached out to do a contract and the day I had a project outline to work on, a month and a half had passed, and even then, the project outline technically wasn’t “official.” Since I never technically confirmed the validity of the project until around a week before completion, there was always this nagging that it wouldn’t meet the specification. Everything worked out in the end, but my lack of communication took no part in achieving that result.

2. Coronavirus

This one seems self-explanatory. Nothing goes as expected when the world is falling apart. Would I have taken on this project had I known a global pandemic was going to keep me trapped inside for a month? Probably, but the reality is I shouldn’t have. Trying to adjust to practically living in my laundry room was difficult at best, and with distractions hitting some of my grades a lot harder than I expected, it was a struggle to find time for everything which fixing my grades. Luckily, I have the mindset where I can sit down and work on a program for 12+ hours at a time, otherwise this project wouldn’t have been completed.

3. Only Learning Enough Rust

Given the time constraint of about a month while also balancing classes and finals, I took the learning approach of wide and shallow, which gave me enough knowledge, but not necessarily all the knowledge to know how to use it efficiently. As a safe programming language, Rust does everything it can to make sure that memory is handled correctly and can’t be corrupted. This goes all the way down to variable declaration. Within Rust, there are effectively two overarching types of variables: mutable and immutable. Mutable variables are standard variables that can be changed, and immutable variable function as constants. The issue is Rust is very strict with where the two types can be used, especially in function calls. For this reason, copying data between variables can be difficult and caused lots of hassle.

The bigger issue though was scoping. Within Rust, all variables have an “owner” who their lifespan is tied to. The issue is the ownership is transferred whenever the variable is used in a function. This means that a variable can only be used in one function, unless it is passed by reference. Even functions like print would take over ownership, which made debugging difficult. To make it worse Rust also struggles when dealing with mutable pass by reference, causing the same issues all over again. Trying to understand how mutability works and is prioritized wasted a couple days since everything I did seemed to be needed as mutable in one place and immutable in another. Eventually, the program just ended up being a chain of functions so the Vectors could be passed though.

4. Multithreading

Multithreading is an interesting problem to say the least. Theoretically, it’s a great idea, as it allows for accelerated runtime through simultaneous calculations. Getting it to work, on the other hand, is a feat in of itself. In Rust, threads prevent corruption through “locking down” data, which means they are the only thread that can access it, which is a smart idea. Except when it comes to Vectors. After about six hours of testing, I got the multithreading to calculate, but I couldn’t store the data since multiple threads couldn’t access the same Vector at the same time, and I was unable to get the data lockdown to work properly. It did end up turning out that in the design specifications there was a library for easy multithreading, which would have saved me a lot of hassle from the beginning if I had noticed it. It was an unfortunate incident of getting so engulfed I forgot what the project was supposed to test.