CAB402 Assignment – Rust

Tyron Roberts – n10208003

Introduction

Rust is a low-level statically typed multi-paradigm programming language which has a focus on safety and performance. Rust mixes the performance of languages like C++ with simpler syntax, an emphasis on memory safety and a set of tools which combines to make development faster and easier. The richly typed system and unique ownership model in Rust guarantees memory-safety and thread-safety. Allowing developers to forget about different types of compile-time errors. Rust's compiler provides helpful feedback which aims to inform the developer of exactly where the problem is and why it is a problem. This helps make debugging easier compared to the painful experience often found when debugging errors in languages like C and C++. Rust is gaining traction amongst developers with parts of the Mozilla Firefox web browser written in Rust. Additionally, both Microsoft and Facebook have begun using it and contributing to the project. The annual Stack Overflow Developer Survey, which surveyed 65,000 programmers in 2020, ranked Rust as the "most loved" programming language for the fifth year running *(Stack Overflow Developer Survey 2020, 2021).* Additionally, GitHub reported that Rust was the second-fastest-growing language on GitHub in 2019, up 235% compared to the last year *(Nature, 2021).* This highlights the need for a language like Rust which can compete with C and C++ while offering the programmer a more pleasant experience.

A History of Rust

Rust was first created in 2006 by Graydon Hoare as a side project while he was working at browser-developer Mozilla *(Rust (programming language) - Wikipedia, 2021).* Hoare worked as a language engineer which meant he worked on compilers and tools for languages he did not design. This work led him to start sketching out a prototype for a new language that incorporated ideas which were not in widely used systems languages. After showing the prototype to his manager, Mozilla decided to set up a team to develop the language. This project was part of a larger project aimed at rebuilding their browser stack around safer, more current, and easier technologies than C++. Hoare's goal was to make the language safer and less likely to crash when compared to languages like C and C++ by handling memory differently to those languages. Rust was heavily influenced by Cyclone (a safe dialect of C which is an imperative language). It also takes some of the object-oriented features from C++. Additionally, it includes functional elements from languages such as Haskell and OCaml. This results in Rust being a C-like language that supports multiparadigm programming (imperative, functional, and object-oriented). The figure below illustrates where Rust has taken inspiration from.

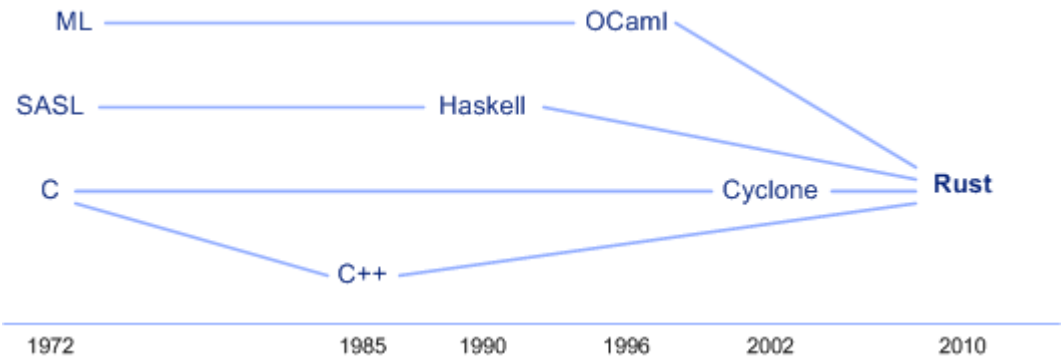

Figure 1: Rust Inspirations (Rust Programming Language, 2021)

What Can Be Built in Rust

Rust is being used in many different applications due to its speed, efficiency, and useful features. Command line tools can be quickly and easily developed in Rust. Programs can be compiled down to a single binary for easy distribution, which means users do not need additional libraries to run it. Rust also automatically generates manual pages which saves the programmer time. Rust is being used in WebAssembly due to its predictable performance which is critical in web apps. Additionally, Rust produces a small code size which means web pages load faster. Rust generated web assembly files does not contain anything extra like a garbage collector and optimization remove any unnecessary code. Rust also works well with JavaScript and allows programmers to use a mix of both in their projects. Rust is used in networking applications for several reasons. Rust has a low footprint which reduces memory and CPU usage. Rust is concurrent at scale which allows you to use any mixture of concurrency approach. The compiler will ensure you do not have any problems in your code. Lastly Rust is used in embedded systems. Rust has a flexible approach to memory that allows the programmer to design things their way. Rust can be integrated into existing C codebases which is important given embedded systems commonly rely on C. Rust code is also potable, once a driver or library is written, it can be used with a variety of different systems. With all these different applications, Rust is proving to be a growing popular language to learn.

Hello World

The following section explains and demonstrates the fundamentals of the syntax and compile process used in Rust. This will be explained through a Hello World programming exercise. The code below will print a few different messages to the console.

```rust
fn main() {
    println!("Hello, world!");
    println!("I am about to call a function,");
    print_message(402);
    let value = five();
    println!("The number is {}", value)
}

fn print_message(unit: i32){
    println!("This program is for my CAB{} assignment.", unit);
}

fn five() -> i32 {
    5
}
```

Figure 2: Hello World Code

Like in many languages, a Rust program will begin executing from the main function. You will notice that the function declaration begins with fn, which is how Rust denotes that it is a function. The first line of the main function uses the println! function. This will print the message “Hello, world!” to the console. A semicolon is used to mark the end of the statement. This program also contains two other functions, print_message and five. The print_message function takes a 32bit integer as its argument. This argument is then passed to the println! function where it is inserted into that string. The function five simply returns the number 5. Rust is different to languages like C and C# because it does not use the return keyword to specify the return value. To return a value, we write a statement but exclude the semicolon. The value returned from five is saved in the variable value, using the let keyword. Rust uses type inferencing, so the type of new variables does not need to be specified.

When looking at the syntax of Rust, you will likely see similarities to several different languages. Rust is similar to C in some ways because statements are followed by a semicolon, functions require the types for each argument and the program starts at the main method. Additionally, the program needs to be compiled before it can be run. Though there are key differences such as specifying the return type of a function after the function name, not using the return key word in functions and not requiring the void return type to be specified if nothing is returned. Rust also has some similarities to F#, variables can be created using the let keyword instead of specifying the type, variables are immutable be default and values are returned from a function by writing the value as the last line. Something interesting to notice is that while type inferencing is used when creating a variable, functions are still required to specify the type of their arguments unlike F#.

When the hello world program is compiled and run, the output is as followed. This is what would be expected given the print statements.

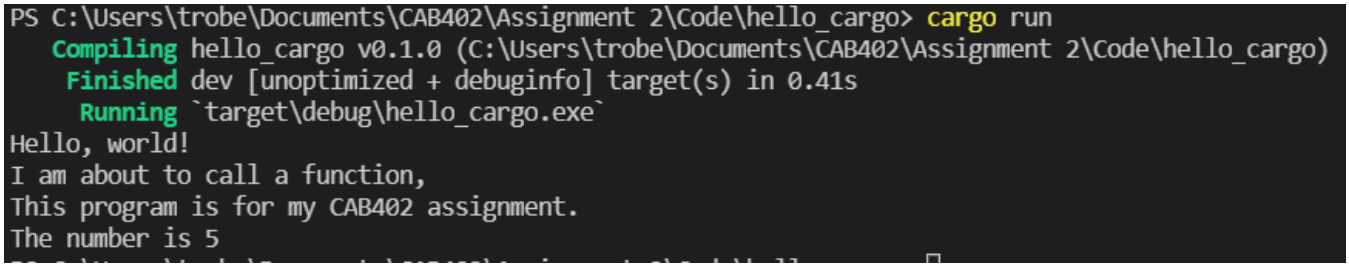

Figure 3: Compiled and Run Hello World

Programs in Rust can be compiled through the 'cargo build' command. They can also be built and run in a single line through the 'cargo run' command. During compilation, the compiler will provide warnings and errors which tell the programmer exactly what the problem is which is very helpful. Coming from other compiler languages like C and C++ the process of compiling and running a program in Rust is incredibly simple and easy. Compiling a C or C++ program requires some knowledge of the process e.g., the order of command line arguments and the different flags you can provide. An example of compiling a C++ program is as followed.

```
g++ -o helloWorld helloWorld.cpp
./helloWorld
```

While this is not a difficult or complicated process, the time spent understanding and writing this is time the programmer could have spent coding. Similar compile and run in one line behavior can be achieved in C and C++ but again, it takes work from the programmer to implement. Another useful part of the compiler is that there is no need to create make files or figure out what files depend on one another to run properly. Features like this make programming in Rust simpler and more efficient compared to C and C++.

Ownership

Ownership is one of Rust's most unique features, and it is what enables Rust to make memory safety guarantees without needing a garbage collector. Some languages implement a garbage collector that looks for unused memory as the program runs. In other languages, it is the job of the programmer to allocate and free memory when needed. Rust's solution to this problem is through the idea of ownership. Ownership is a system where all values have a unique owner, the variable they were assigned to. When a variable is passed to a function, it gives the value owned by that variable to the function. This passes the ownership of that value from the variable to the function. After ownership has been passed, the original variable or old owner can no longer access that value it held. This system can be summarised through the following rules.

- Each value in Rust has a variable that is called its *owner*.
- There can only be one owner at a time.
- When the owner goes out of scope, the value will be dropped.

This implementation of ownership in Rust guarantees memory safety without the need for a garbage collector. When, for example String::from, is called, a mutable string variable is created. To support a mutable, changing piece of text, memory on the heap must be allocated to hold the contents. This also means there must be a way of freeing the memory when we are done with the string. Rust does this by automatically returning the memory once the variable, owner, goes out of scope. Once the variable is out of scope, there is no way of using it anymore so we can safely free that memory.

A way to visualise the idea of ownership in the real world is to think of selling a used car. A car has a single owner who can do what they want with it. If the owner decides to sell the car, they will pass it on to a new owner. Once the car has been sold and passed to the new owner, the old owner does not have access to it anymore and cannot use it. Additionally, we do not care happens to the old owner after it has been sold as they have nothing to do with the car anymore. This is like in Rust where once ownership has been transferred, the old owner cannot use the value and the old owner can go out of scope without impacting the new owner.

While ownership is a great feature of Rust, we do not always want to transfer ownership of the value. Ownership can be avoided by passing references to functions. References allow you to refer to some value without taking ownership of it. Thinking back to the used car, using a reference is like letting some borrow your car. They are allowed to use it, but you are still the owner.

```
takes_ownership(&s); // Pass reference, can use value afterwards

takes_ownership(s);  // Transfer ownership to function, can't use afterwards
```

Figure 4: Passing References

The following code demonstrates how ownership works in Rust. This code will run when compiled and print a message.

```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
```

```
                                    // use x afterward
} // Here, x goes out of scope, then s.
fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.

fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope.
```

Figure 5: Ownership Code

The following code will fail to compile due to the println! Function that is called after takes_ownership.

```
fn main() {
    let s = String::from("hello");  // s comes into scope

    takes_ownership(s);             // s's value moves into the function...
                                    // ... and so is no longer valid here
   println!("{}", s);      // This line will cause an error

    let x = 5;                      // x comes into scope

    makes_copy(x);                  // x would move into the function,
                                    // but i32 is Copy, so it's okay to still
                                    // use x afterward
} // Here, x goes out of scope, then s.

fn takes_ownership(some_string: String) { // some_string comes into scope
    println!("{}", some_string);
} // Here, some_string goes out of scope and `drop` is called. The backing
  // memory is freed.
fn makes_copy(some_integer: i32) { // some_integer comes into scope
    println!("{}", some_integer);
} // Here, some_integer goes out of scope.
```

Figure 6: Ownership code with error

The following is the error received when attempting to compile:

```
   Compiling ownership v0.1.0 (C:\Users\trobe\Documents\CAB402\Assignment 2\Code\ownership)
error[E0382]: borrow of moved value: `s`
 --> src\main.rs:7:20
  |
2 |     let s = String::from("hello");  // s comes into scope
  |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
3 |
4 |     takes_ownership(s);             // s's value moves into the function...
  |                     - value moved here
...
7 |     println!("{}", s);
  |                    ^ value borrowed here after move

error: aborting due to previous error
```

Figure 7: Ownership Error

This produces an error because ownership of the string “hello”, which was held by the variable s, was passed to the function takes_ownership. After this statement, the variable s does not have access to that value anymore. This means that when the println! function tries to print the contents of s, an error occurs because s does not contain a value. We see that this does not happen with the variable x because variables of type i32 pass a copy, they do not transfer ownership. This example shows the usefulness of ownership and how it prevents memory safety problems that would be faced when programming in C and C++. While it may cause more work for the programmer in some cases, ownership makes it easier to track where variables are going and preventing memory runtime errors makes any extra work worth it.

The Compiler

The compiler and its helpful error messages are one of Rust’s most popular features. While most compilers will give the programmer enough information to understand the error, the messages can often be confusing or contain unnecessary information. The error messages in Rust feel like you are talking to another programmer rather then a computer. Of all the different languages I have programmed in, the error messages provided in Rust are the most helpful and easy to understand. The helpful compiler messages are one of the key advantages Rust has over languages like C and C++. Multiple error examples are shown below to demonstrate how useful it is.

Incorrect Argument for Built in Function

In this code example the println! function is being called with a number, instead of a string. The error message provided is very simple “format argument must be a string literal”, which makes it easy to understand the problem. What makes this so useful though is that the compiler understands you were probably trying to print a numebr and shows you how it is done in Rust. There is no need to search online or through documentation to see how the println! function prints a number.

```
PS C:\Users\trobe\Documents\CAB402\Assignment 2\Code\hello_cargo> cargo run
   Compiling hello_cargo v0.1.0 (C:\Users\trobe\Documents\CAB402\Assignment 2\Code\hello_cargo)
error: format argument must be a string literal
 --> src\main.rs:3:14
  |
3 |     println!(6);
  |              ^
  |
help: you might be missing a string literal to format with
  |
3 |     println!("{}", 6);
  |              ^^^^

error: aborting due to previous error
```

Figure 8: Incorrect Argument Error

The equivalent error message in C looks like this.

```
main.c: In function ‘main’:
main.c:5:12: warning: passing argument 1 of ‘printf’ makes pointer from integer without a cast [-Wint-conversion]
     printf(6);
            ^
In file included from main.c:1:0:
/usr/include/stdio.h:365:12: note: expected ‘const char * restrict’ but argument is of type ‘int’
 extern int printf (const char *__restrict __format, ...);
            ^~~~~~
```

Figure 9: C Programming Equivalent

While the messages mean similar things, the C version is much more cluttered and contains a lot of information that is not needed. The programmer does not want to spend time understanding the meaning of this message when it could be reduced to a single sentence like “format argument must be a string literal”.

Ownership Error

This code example uses the ownership error discussed in the ownership section of the report. This error it shows the three different sections where there were problems. This error is also useful because programmers new to rust would not be familiar with ownership so this will help them see what was wrong. In a language like C you would likely get runtime errors instead and struggle to see where the problematic code was.

```
   Compiling ownership v0.1.0 (C:\Users\trobe\Documents\CAB402\Assignment 2\Code\ownership)
error[E0382]: borrow of moved value: `s`
 --> src\main.rs:7:20
  |
2 |     let s = String::from("hello");  // s comes into scope
  |         - move occurs because `s` has type `String`, which does not implement the `Copy` trait
3 |
4 |     takes_ownership(s);             // s's value moves into the function...
  |                     - value moved here
...
7 |     println!("{}", s);
  |                    ^ value borrowed here after move

error: aborting due to previous error
```

Figure 10: Ownership Error

Incorrect Argument for Custom Function

The following example passes a value of the wrong type to the print_message function from the Hello World section. The message clearly points out that the argument was incorrect then states what the expected type was.

```
PS C:\Users\trobe\Documents\CAB402\Assignment 2\Code\hello_cargo> cargo run
   Compiling hello_cargo v0.1.0 (C:\Users\trobe\Documents\CAB402\Assignment 2\Code\hello_cargo)
error[E0308]: mismatched types
 --> src\main.rs:4:19
  |
4 |     print_message("402");
  |                   ^^^^^ expected `i32`, found `&str`

error: aborting due to previous error

For more information about this error, try `rustc --explain E0308`.
```

Figure 11: Incorrect Argument Error

An interesting feature of some Rust errors is that you can ask for a more detailed explanation from the compiler through 'rustc –explain E0308', where the error number is specific for that case. The following image is the result of that command.

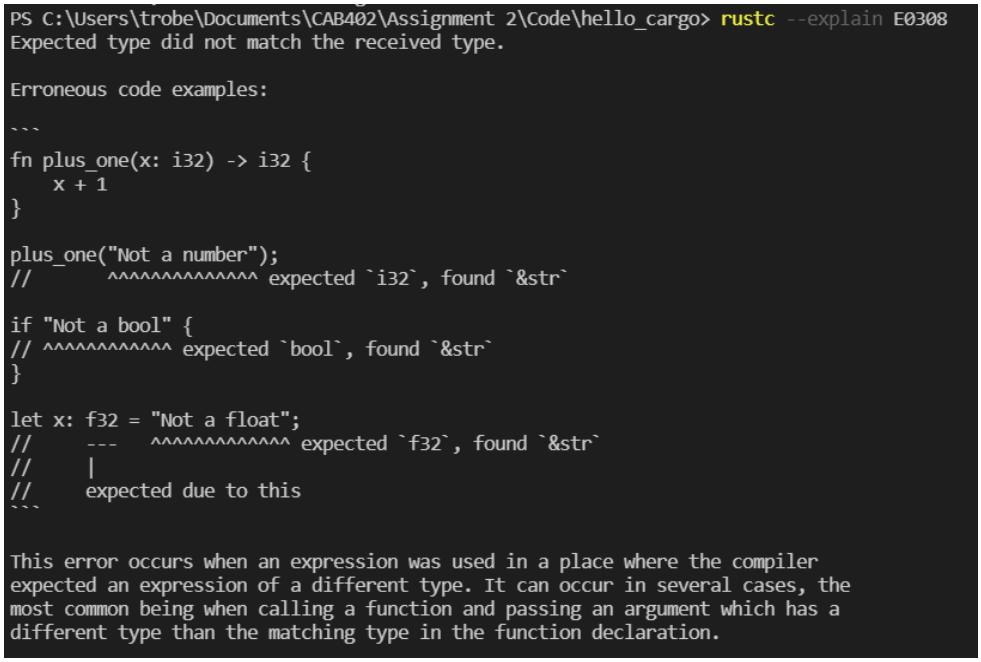

Figure 12: More Details On Error

This tool is very helpful because it gives a few simple, common examples of how this error occurs and follows it up with a small explanation on the error.

Overwriting Immutable Variable

This final example has a variable called value being created and then it is reassigned to a new value on the next line. Variables in Rust are immutable by default, so this causes an error. This message is helpful and useful but it is even better due to the help Rust offers. The compiler understands you would like a mutable variable so it informs you of the mut keyword. If you were new to Rust this would save time but telling you how to create a mutable variable.

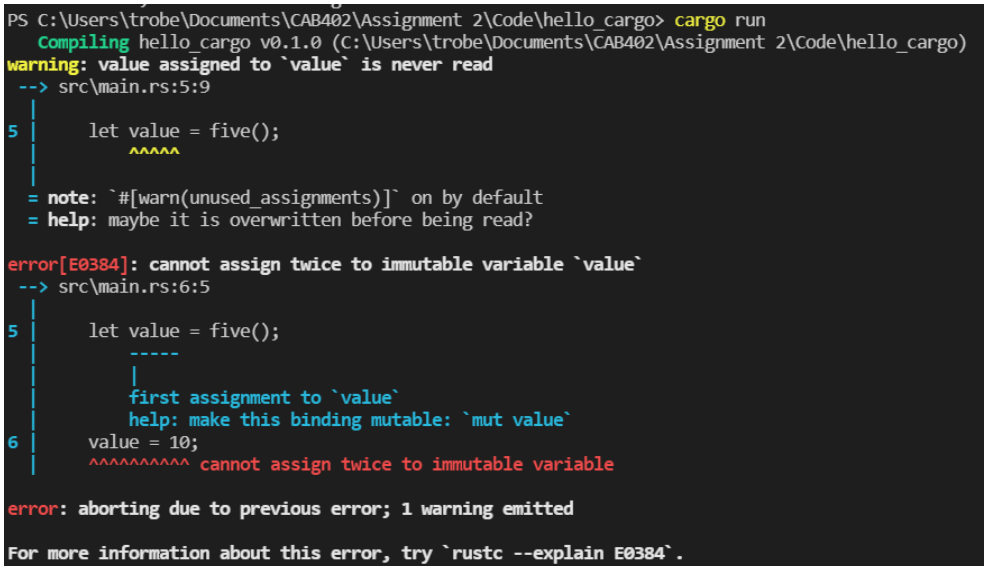

Figure 13: Immutable Variable Error

Conclusion

In the context of paradigms, Rust is a multi-paradigm language. Rust has functional features such as immutable variables by default and higher order functions. Rust has object-oriented features like structs and enums holding data which can have methods applied to them and allowing encapsulation to hide the implementation of an object. Rust also has imperative features such as mutable state and for loops. Having all these paradigms gives the programmer choice in how they wish to write their code. Rust's simpler syntax makes it less taxing to program compared to languages like C and C++. The unique idea ownership makes programming memory safe applications an easier process. The helpful messages from the compiler makes programming more efficient and less stressful and it has shown me how much better error messages could be in other languages. Overall Rust is a low-level statically typed multi-paradigm programming language which has an emphasis on safety and performance. My experience learning about it for this report has made me want to work with it more and learn other less popular languages like Rust to see what else I am missing out on.

References

Doc.rust-lang.org. 2021. *Hello, World! - The Rust Programming Language*. [online] Available at: https://doc.rust-lang.org/book/ch01-02-hello-world.html [Accessed 30 May 2021].

Doc.rust-lang.org. 2021. *Understanding Ownership - The Rust Programming Language*. [online] Available at: https://doc.rust-lang.org/book/ch04-00-understanding-ownership.html#:~:text=Ownership%20is%20Rust's%20most%20unique,how%20ownership%20works%20in%20Rust [Accessed 30 May 2021].

En.wikipedia.org. 2021. *Rust (programming language) - Wikipedia*. [online] Available at: https://en.wikipedia.org/wiki/Rust_(programming_language) [Accessed 30 May 2021].

IBM Developer. 2021. *Rust Programming Language*. [online] Available at: https://developer.ibm.com/technologies/web-development/articles/os-developers-know-rust/ [Accessed 30 May 2021].

InfoQ. 2021. *Interview on Rust, a Systems Programming Language Developed by Mozilla*. [online] Available at: https://www.infoq.com/news/2012/08/Interview-Rust/ [Accessed 30 May 2021].

Nature 2021. *Why scientists are turning to Rust*. [online] Nature.com. Available at: https://www.nature.com/articles/d41586-020-03382-2 [Accessed 30 May 2021].

Serokell Software Development Company. 2021. *Why You Should Use Rust in 2021*. [online] Available at: https://serokell.io/blog/rust-guide [Accessed 30 May 2021].

Stack Overflow. 2021. *Stack Overflow Developer Survey 2020*. [online] Available at: https://insights.stackoverflow.com/survey/2020 [Accessed 30 May 2021].

Stack Overflow Blog. 2021. *What is Rust and why is it so popular? - Stack Overflow Blog*. [online] Available at: https://stackoverflow.blog/2020/01/20/what-is-rust-and-why-is-it-so-popular/ [Accessed 30 May 2021].