Henry Oehlrich

Mrs. Schmitz

H. English 4

15th Mar 2023

Rust: A Richly Typed, Memory-safe, Algebraic Systems Language for Writing Perfect Code

Rust approaches memory management with a new paradigm. It uses neither a garbage collector nor forces memory management on the programmer; instead it achieves memory safety using a borrow checker. The borrow checker has two rules: data has one owner and data may have multiple readers or one writer. When a piece of non-global data is instantiated, a sized portion of data from the stack, or a chunk of data in the heap, is allocated to it. The variable that the data is assigned to is its owner. Passing data by value (IE not through a pointer) moves the data to the function it was passed into. This data is gone, it can neither be accessed nor modified by the previous owner. In order to share data without moving, you pass a reference (pointer) to the data. This is called borrowing.

In order to understand how the borrow checker functions, one must understand variable lifetimes and scope. When the owner of a value goes out of scope, the value is dropped. A variable's scope is defined by the narrowest set of curly braces (this is often called a block). A variable comes into scope when it is declared and exits scope when program execution exits its block. A variables lifetime is the length of time it is in scope. The lifetime of the owner of a value must exceed the lifetime of all references to it. This makes null and dangling pointers impossible.

```
fn main() {
    // 's' comes into scope as it is declared
    let s: String = String::from("This is a string");

    // Pass 's' by reference
    // 's' still owns the String
    let len: usize = get_len(&s);

    println!("The length of '{}' is {}", s, len);
}   // This scope is now over and 's' is no longer valid
```

```
fn get_len(rs: &String) -> usize {  // 'rs' is declared here
    rs.len();
} // 'rs' goes out of scope here and is dropped
```

A mutable reference to a value must be the only reference to that value. This is to prevent race conditions for reading or modifying data. If another part of the program tries to read or modify a piece of data that is also being modified by a mutable reference, a race condition could occur and inconsistent results would be produced. Rusts imperviousness to race conditions is coined as "Fearless Concurrency".

Rust is a very strongly typed language. This means that data must have a defined or inferred type and must conform to stricter compiler rules and safeties. Rust's type system prevents programmers from making common and redundant mistakes at compile time (or more realistically linted in the editor). Every Rust builtin type has several guarantees if it compiles. For example, the *String* type guarantees that it will always be a valid UTF-8 encoded (Unicode) vector of bytes that can be displayed. While this seems trivial, nearly all systems languages and quite a few higher level languages will allow the programmer to do unsafe or unrepresentable things with strings.

C-style structs allow Rust programmers to create custom data types that improve readability, code structure, and safety. Structs can be thought of as a custom blueprint for a semantically useful datatype. They are type-checked by the compiler and follow a C-style key-value syntax. Structs are private by default and cannot be accessed by other files unless defined with the *pub* keyword.

```
use chrono::{Local, DateTime};
use sha256::digest;

// Define the User struct with several fields
struct User {
    username: String,
    email: String,
    password_hash: String,
    last_logon: DateTime<Local>,
```

```
    google_uuid: Option<String>,
}

fn main() {
    // Construct a user object
    let user = User {
        username: String::from("hoehlrich"),
        email: String::from("henry@oehlrich.xyz"),
        password_hash: digest("password123"),
        last_logon: Local::now(),
        google_uuid: None,
    };

    println!("User '{}' with email '{}' last logged on '{}'.",
        user.username,
        user.email,
        user.last_logon.format("[%Y-%m-%d][%H:%M:%S]")
    );

    // output:  User 'hoehlrich' with email 'henry@oehlrich.xyz'
    //          last logged on '[2023-03-21][09:25:09]'.

}
```

Gjengset (2021)

Rustteam (2023b)

Rustteam (2023a)

Kernighan and Ritchie (1972)

Oatman (2022)

References

Gjengset, Jon. *Rust for Rustaceans: Idiomatic Programming for Experienced Developers*. No Starch, 2021. In this book, the author goes beyond the basics by exploring exploring common advanced programming patterns and concepts. Instead of defining a trait, he shows meaningful, stateful, applications of traits in specific problems.

Kernighan, Bryan and Dennis Ritchie. *The C Programming Language*. Prentice Hall, 1972. In this book, the authors, having created the C programming language, give a deep dive into it. This book teachs the reader C, but more importantly, it teaches how computer registers, memory, and data work and are stored at the most basic level. Although it follows a different paradigm from Rust, it is impossible to cohesively cover a programming language without paying homage to C.

Oatman, Tris. 2022 “https://www.youtube.com/noboilerplate.”. This Youtube channel is an excellent secondary/tertiary source for concise, complete analysis of the Rust language. While other sources merely gives the facts, this channel shows real-world applications as well as fun independent use cases.

Rustteam. 2023 “https://doc.rust-lang.org/stable/rust-by-example/.”. Rust by example is a website that shows you many of the features of rust through thorough code examples and descriptions. It is a great way to get concise, correct code, in order to better explain complex ideas.

Rustteam. 2023 “Rust Programming Language.”. This is the official Rust Language website. It is created and maintained by the Rust Team (Rust maintainers) as well as many contributions form the community. It mostly serves as a launching point for learning and participating in Rust. As well has maintaining links, this site has a blog for Rust news for update and feature announcements.