Henry Oehlrich

Mrs. Schmitz

H. English 4

15 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, 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}   // This scope is now over and 's' is no longer valid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]'.
}
```

Unlike other popular languages Rust does not have a higher level of object abstraction above structs. Struct *impl* (implementation) blocks are Rust's form of object oriented methods. Everything in the *impl* block will be associated with the attached struct. Functions defined in implementation blocks that operate on objects are called methods. Methods may take any form or reference of *self* as the first parameter; *self* is the object instance that the method is begin called on.

Enums, or enumerations, allow the programmer to define a type that can be one of a set of values. An enum of colors might include red, blue, yellow, etc; an instance of the colors enum might be red. Enumerations allow a single type to have multiple variants. This is semantically demonstrated with the double colon namespacing after the identifier. Functions that take an enum can take any variant of that type. Enum variants are also able to hold data.

```
// Define the Event enum
enum Event {
    LeftClick(i32, i32),    // LeftClick has x and y pos
    RightClick(i32, i32),   // RightClick has x and y pos
    KeyPress(char),       // Key press stores the key pressed
```

```
    Quit,                // Quit does not need to store data
}

fn main() {
    let k_key = KesPress('k');
}
```

In order to add different functionality to different variants of enums, Rust allows the programmer to match patterns on enums. This is similar to the C or Java switch statement. *Kernighan and Ritchie (1972).* It is much safer than either of these languages implementations, though; in Rust you must handle each and every possible control flow path when matching on enums (even if the handling crashes the program, it must be explicitly stated). Pattern matching can extract and handle the data that may (or may not) be held in an enum instance.

Rust has no nulls. In order to represent the value of none, Rust uses the *Option* enum. The *Option* enum is defined in the standard library. *Option* is either a value *Some(v)*, or *None*. Functions that may or may not return a value should return that value (or lack thereof) as an Option. The calling portion of the function can then match on that enum, storing the data if it is valid or proceeding as necessary if the data doesn't exist.

```
// Print google_uuid if it exists, if not give a message
// Remember from a previous example that google_uuid is an Option<String>
match user.google_uuid {
    Some(v) => println!("google uuid: {}", v),
    None => println!("user does not have a google uuid"),
}
```

Perhaps the most useful and paradigm shifting enum is the standard library's *Result* enum. The *Result* type is Rust's answer to error handling. It can either hold a value (*Ok(v)*), or an error (*Err(e)*). In order to access the value (or error) contained in the Result, it must be explicitly handled with a match statement or a temporary call to unwrap. Errors in results do not interrupt the program control flow as is done with other popular languages; instead, idiomatic Rust programs will propagate any errors up the stack to be dealt with.

A common pattern with *Result* types is to match on the wrapped type assigning the value

to a variable and returning the error if it exists. This pattern is so common that the Rust Team assigned it to the *?* operator. At compile time, any uses of *?* will be automatically expanded to the long-form match statements; this is similar, but not the same, as macro expansions.

```
fn read_data() -> Result<String, Box<dyn Error>> {
    // Expanded version of error handling
    let data = match fs::read_to_string("file.txt") {
        Ok(v) => v,
        Err(e) => return Err(Box::new(e)),
    };

    // This is the same as above
    let data_ = fs::read_to_string("file.txt")?;

    // Return the data if there hasn't been an error
    return Ok(data);
}
```

Generics in Rust are placeholders for concrete types. The previously discussed enums *Option* and *Result* are both generics. Using generics allows functions, methods, structs, and enums to take multiple possible types without code duplication. Each generic type (specified in angle brackets after the function or struct name) is represented by a letter (commonly T and U). Generic identifiers may be reused in order to specify that two parameters/attributes share the same type. Similarly to the *?* operator, generics do not decrease code performance. In the monomorphization step of compilation, the compiler expands all generics into variants required by the code.

```
fn main() {
    // p1 is defined with coords as two i32s
    let p1 = Point { x: 1, y: 2 };
    // p2 is defined with coords as two f32s
    let p2 = Point { x: 0.34, y: 123.21 };
}

// Struct Point is defined with two attributes of generic type T
struct Point<T> {
    x: T,
    y: T,
}
```

Traits define behavior that can be implemented on multiple types (they are similar to interfaces in other popular languages **?**.) A trait includes a set of function signatures that all types that implement the trait must also implement. Traits can include default implementations that are injected into types that implement the trait but do not override the default function by reimplementing it.

```
// Every type that implements Sized must implement the associated methods
trait HasSize {
    fn get_len(&self) -> usize;

    fn allocated_size(&self) -> usize;
}

struct Sentence {
    text: String,
    author: String,
}

impl HasSize for Sentence {
    fn get_len(&self) -> usize {
        return self.text.len();
    }

    fn allocated_size(&self) -> usize {
        return self.text.capacity();
    }
}

fn main() {
    let sentence = Sentence {
        text: String::from("Almost done with the rough draft"),
        author: String::from("Henry Oehlrich"),
    };

    println!(
        "Sentence with len {} and allocaed size {}",
        sentence.get_len(),
        sentence.allocated_size(),
    )
}
```

Trait bounds allow the programmer to specify the limits or bounds to generic types.

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.