Moving to the Rust programming language

Jaideep Ganguly, ScD

Contents

List of Tables

List of Listings

Preface

Rust is an open-source, community-developed systems programming language that runs blazing fast, prevents segfaults, and guarantees thread safety.

The motivation to write this book came from the desire to develop a concise and yet a comprehensive for a rigorous exposure to Rust. The goal is to help proficient developers in well known languages such as Java, C#, Kotlin, Scala, Go, etc., to quickly migrate to Rust.

The official Rust book is excellent but at nearly 400 pages it is rather verbose. It can be followed even by developers who have less exposure to programming. Consequently, the authors have gone to a great length to write a great book lucid with examples.

For developers proficient in some other language, this book will help them migrate to Rust in just a couple of days. The content has been deliberately limited to just about a hundred pages.

The examples used in the book can be found in github at:
https://github.com/jganguly/rustfun

Jaideep Ganguly received his degrees of Doctor of Science and Master of Science from the Massachusetts Institute of Technology. He received his undergraduate degree from the Indian Institute of Technology, Kharagpur. He started his career at Oracle and has worked with Microsoft, Amazon and is now heading the India R&D center of compass.com.

CHAPTER 1

WHY RUST?

The Rust programming language has been Stack Overflow's most loved language for five years in a row, clearly establishing the fact that a significant section of the developer population love it. Rust solves many pain points present in current popular languages and has a limited number of downsides.

It's quite difficult to write secure code for large programs. It's particularly difficult to manage memory correctly in C and C++. As a result we see a regular procession of security breaches starting from the Morris worm of 1988. Furthermore, It's even more difficult to write multi threaded code, which is the only way to exploit the abilities of modern multi CPU machines. Concurrency can introduce broad new classes of bugs and make ordinary bugs much harder to reproduce.

Rust has been designed to be a safe, concurrent language and yet match or better the performance of C/C++. Rust is not really an object-oriented language, although it has some characteristics of it. It is also not a functional language, but does adhere to many the many tenets of functional programming.

although it does tend to make the influences on a computation's result more explicit, as functional languages do.

1.1 STATICALLY TYPED

Rust is a statically and strongly typed systems programming language. Statically means that all types are known at compile-time, strongly means that these types are designed to make it harder to write incorrect programs. A successful compilation means you have a much better guarantee of correctness than with language such as C or Java, generating the best possible machine code with full control of memory use.

The debate between dynamic versus static typed is long standing and will continue. However, dealing with dynamic typing in large code bases is difficult. Statically typed languages allow the compiler to check for constraints on the data and its behavior and thereby significantly reducing cognitive overheads on the developer to produce correct code. Statically typed languages differ from each other. An important aspect is how they deal with the concept of NULL, which means the value may be something or nothing. Like Haskell and some other modern programming languages, Rust encodes this possibility using an optional type and the compiler requires you to handle the None case.

1.2 TYPE SAFETY

The basis for computers and computer programs is the Von Neumann model which is over seventy years old. This architecture has one memory that holds both the instructions as well as program data. The commonly used programming languages C and C++ offer no support

for automatic memory management. The programmer has to deal with memory management making the program error prone. A common source of error is the well known *buffer overflow*. Buffer overruns are the cause of many computer problems, such as crashing and vulnerability in terms of attacks. The past years have revealed multiple exploits: vulnerabilities that are the result of these types of errors and which could have significant consequences.

Recently, a serious vulnerability known as *Heartbleed* was discovered. The bug was in *OpenSSL* which is commonly used in network routers and web servers. This vulnerability allowed encryption keys to be read remotely from these systems and thereby compromised their security. This vulnerability would have been avoided Rust was used to develop OpenSSL.

Rust is safe by default. All memory accesses are checked and It is not possible to corrupt memory by accident. With direct access to hardware and memory, Rust is an ideal language for embedded and bare-metal development. One can write extremely low-level code, such as operating system kernels or micro-controller applications. However, it is also a very pleasant language to write application code as well. Rust's core types and functions as well as reusable library code stand out in these especially challenging environments. However, unlike many existing systems programming languages, Rust does not require developers to spend their time mired in nitty-gritty details.

Rust's strong type system and emphasis on memory safety, all enforced at compile time, mean that it is extremely common to get errors when compiling your code. This can be a frustrating feeling for programmers not used to such an opinionated programming language. However, the Rust developers have spent a large amount of time working to improve the error messages to ensure that they are clear and actionable. One must not gloss over Rust compile time error messages.

In summary, if a program has been written so that no possible execution can exhibit undefined behavior, we say that program is well defined. If a language's safety checks ensure that every program is well defined, we say that language is type safe.

1.3 RUNTIME

Rust strives to have as many zero-cost abstractions as possible, abstractions that are as equally performant as corresponding hand-written code. Zero-cost abstraction means that there's no extra runtime overhead that you pay for certain powerful abstractions or safety features that you do have to pay a runtime cost for other languages. However, be aware that not every abstraction or every safety feature in Rust is truly *zero-cost*.

A programming language runtime, is all the machinery provided by the language itself and which is injected into and supports the execution environment. This can include things as small as some minimal routines for laying out and freeing memory, as in C, to entire virtual machines, interpreters, and standard libraries, as in Java, Python or Ruby. Think of it as the minimal machinery that is both part of the language and must be present, either in the executable itself or installed on the computer, for any given program written in that language to run.

Rust strives to have a very fast run time. It does this in part by compiling to an executable and injecting only a very minimal language runtime and *does not provide a memory manager, i.e., a garbage collector* that operates during the executable's runtime.

1.4 PERFORMANCE

Rust gives you the choice of storing data on the stack or on the heap and determines at compile time when memory is no longer needed and can be cleaned up. This allows efficient usage of memory as well as more performant memory access. Tilde, an early production user of Rust in their Skylight product, found they were able to reduce their memory usage from 5GB to 50MB by rewriting certain Java HTTP endpoints in idiomatic Rust. Savings like this quickly add up when cloud service providers charge premium prices for increased memory or additional machines.

Without the need to have a garbage collector continuously running, Rust projects are well-suited to be used as libraries by other programming languages via foreign-function interfaces. This allows existing projects to replace performance critical pieces with speedy Rust code without the memory safety risks inherent with other systems programming languages. Some projects are being incrementally rewritten in Rust using these techniques.

The fundamental principles of Rust are:

1. *ownership* and safe *borrowing* of data
2. *functions*, *methods* and *closures* to operate on data
3. *tuples*, *structs* and *enums* to aggregate data
4. *matching* pattern to select and destructure data
5. *traits* to define behavior on data

CHAPTER 2

GETTING STARTED

2.1 INSTALLATION

In MacOS or Linux, Rust is installed using the following command in the terminal.

```
# install rust
curl --proto '=https' --tlsv1.2 https://sh.rustup.rs -sSf | sh

# update rust
rustup update

# uninstall rust
rustup self uninstall

# rust compiler
rustc --version
```

Listing 2.1: Installation in MacOS or Linux

2.2 CREATING PROJECT WITH CARGO

`Cargo` is Rust's build system and package manager. For simple projects, you can use the `rustc` compiler but for complex projects you have to use `Cargo`. It is used to manage Rust projects because `Cargo` handles a lot of tasks such as building the code, downloading the libraries the code depends on and building those libraries. `Cargo` comes installed with Rust and you can check whether it is installed by typing the following in a terminal.

```
cargo --version
```

Listing 2.2: Cargo

Let us create a directory **my_rust_project** to store the Rust code in your home directory.

```
cargo new my_rust_project
cd my_rust_project
```

Listing 2.3: Project with Cargo

The first command creates a new directory called **my_rust_project** in your home directory. We've named our project **my_rust_project**, and Cargo creates its files in a directory of the same name. Go into the **my_rust_project** directory and list the files. You'll see that Cargo has generated two files and one directory for us: a Cargo.toml file and a src directory with a main.rs file inside.

It has also initialized a new Git repository along with a .gitignore file. Git files won't be generated if you run cargo new within an existing Git repository; you can override this behavior by using `cargo new --vcs=git`.

The contents of Cargo.toml look as follows:

```
[package]
name = "tpl"
version = "0.1.0"
authors = ["Jaideep Ganguly <ganguly.jaideep@gmail.com>"]
edition = "2018"

# See more keys and their definitions at https://doc.rust-lang.org/
   cargo/reference/manifest.html

[dependencies]
```

Listing 2.4: Project with Cargo

This file is as per the TOML (Tom's Obvious, Minimal Language) format, which is Cargo's configuration format. The first line, `[package]`, is a section heading that indicates that the following statements are configuring a package. The next four lines set the configuration information Cargo needs to compile your program: the name, the version, who wrote it, and the edition of Rust to use. The last line, `[dependencies]`, is the start of a section for you to list any of your project's dependencies. **Packages of code are referred to as crates**.

The file `src/main.rs` has been generated by Cargo.

```
fn main() {
    println!("Hello World");
}
```

Listing 2.5: Hello World!

A handy way to print results to a screen is `println!` which calls a Rust macro. If it called a function instead, it would be entered as println (without the !). The set of curly brackets, {}, is a placeholder. For now, you just need to know that using a ! means that you're calling a macro instead of a normal function.

Rust files end with the `.rs` extension and `fn` is the keyword to denote a function. Cargo placed the code in the `src` directory and we have a Cargo.toml configuration file in the top directory. Cargo expects your source files to live inside the `src` directory. The top level project directory is just for README files, license information, configuration files, etc. Using Cargo helps you organize your projects.

2.3 BUILD & RUN WITH CARGO

To build and run, you need to `cd` to `my_rust_project`.

```
cargo build # build only
cargo run   # build and run
```

Listing 2.6: Build & Run

`cargo build` command creates an executable file in `target/debug/tpl`. We can run it with `./target/debug/hello_world`. Running cargo build for the first time also causes Cargo to create a new file at the top level Cargo.lock. This file keeps track of the exact versions of dependencies in your project. We can also use cargo run to compile the code and then run the resulting executable all in one command. Cargo also provides a command called `cargo check`. that checks your code to make sure it compiles but doesn't produce an executable. Cargo check is much faster than cargo build, because it skips the step of producing an executable. If you're continually checking your work while writing the code, using cargo check will speed up the process. When your project is finally ready for release, you can use `cargo build --release` to compile it with optimizations. This command will create an executable in target/release instead of target/debug. The optimizations make your Rust code run faster, but turning them on lengthens the time it takes for your program to compile.

2.4 CREATING A LIBRARY WITH CARGO

Cargo is used to create a library named `tpllib` using the command:
`cargo new --lib my_rust_lib` to create the library `my_rust_lib`.

We will learn more about how to structure your code in separate modules and libraries.

CHAPTER 3

BASIC CONCEPTS

In this chapter, we will discuss basic concepts in programming in the context of Rust. These include variables, basic types, functions, comments, and control flow which are part of every Rust program.

3.1 VARIABLES & MUTABILITY

In Rust, variables are immutable by default. This restriction contributes safety and easy concurrency that Rust offers. You have the option to make your variables mutable. While Rust encourages you to favor immutability but sometimes you may want to opt out. In Rust variables are assigned using the `let` keyword and `mut` makes them mutable.

Constants are declared using the `const` keyword instead of the `let` keyword and the **type** of the value must be annotated. We will discuss **type** shortly. Constants are **immutable** and are **evaluated at compile time**. This means that they can be set only to a **constant** expression but not to the result of a function call or any other value that will be computed at runtime.

Constants can be declared in any scope, including the global scope, which makes them useful for values that many parts of code need to know about. Naming hard coded values used throughout your program as constants is useful in conveying the meaning of that value to future maintainers of the code.

In contrast, a `let` binding is about a run-time computed value.

```
1 fn main() {
2     let x = 5;
3     let mut y = 6;
4     y = y + 1;
5     const MAX_POINTS: u32 = 100_000*100;
6     println!("x={} y={} MAX_POINTS={}",x,y,MAX_POINTS);
7 }
```

Listing 3.1: Variables & Mutability

3.2 SHADOWING

One can declare a new variable with the same name as a previous variable, and the new variable `shadows` the previous variable. Shadowing is different from marking a variable as `mut`, because we will get a compile-time error if we accidentally try to reassign to this variable without using the `let` keyword. By using `let`, we can perform a few transformations on a value but have the variable will be immutable after those transformations have been completed.

The other difference between `mut` and shadowing is that because we're effectively creating a new variable when we use the let keyword again, we can change the type of the value but reuse the same name.

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u 32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

Table 3.1: Integer Types in Rust

```
fn main() {
    let x = 5;
    let x = x + 1;
    let x = x * 2;

    let spaces = "   ";
    let spaces = spaces.len();
}
```

Listing 3.2: Shadowing

3.3 DATA TYPES

Every value in Rust is of a certain data type. There are two data type subsets - *scalar* and *compound*. Rust is a statically typed language, which means that the types of all variables must be declared at compile time. The compiler can usually infer what type we want to use based on the value and usage. But in cases when many types are possible, such as when we convert a `String` to a numeric type using `parse`, we must add a type annotation as below:

```
fn main() {
    let x = 3; // Type inference will automatically assign x to be of type i32
    let num: i32 = "42".parse().expect("Not a number!");
    println!("{}",num);
}
```

Listing 3.3: Type

Note that `.expect("Not a number!")` is necessary as otherwise the code will not compile. We will discuss this later. In the above code, `i32`, i.e, a 32 bit integer, is the `type` of `num`.

3.4 SCALAR DATA TYPE

A scalar data type represents a single value. Rust has four primary scalar data types: ***integers, floating-point numbers, booleans, characters***.

3.4.1 INTEGER TYPE

An integer is a number without a fractional component.

The `isize` and `usize` types depend on the computer the program is being executed; 64 bits if you're on a 64-bit architecture and 32 bits if you're on a 32-bit architecture.

Number Literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

Table 3.2: Integer Literals in Rust

You can write integer literals in any of the forms shown in Table 3-2. A literal is a notation for representing a fixed value. Note that all number literals except the byte literal allow a type suffix, such as 57u8, and _ as a visual separator, such as 1_000 for ease of reading.

Rust's defaults are generally good choices, and **integer types default to i32**. This type is generally the fastest, even on 64-bit systems. The primary situation in which you'd use `isize` or `usize` is when indexing some sort of collection.

Let's say you have a variable of type u8 that can hold values between 0 and 255. If you try to change the variable to a value outside of that range, such as 256, integer overflow will occur. Rust has some interesting rules involving this behavior. When you're compiling in debug mode, Rust includes checks for integer overflow that cause your program to panic at runtime if this behavior occurs. Rust uses the term panicking when a program exits with an error. When you're compiling in release mode with the `-- release` flag, Rust does not include checks for integer overflow that cause panics. Instead, if overflow occurs, Rust performs two's complement wrapping. 2's complement of a binary number is 1 added to the 1's complement of the binary number. And 1's complement of a binary number is another binary number obtained by toggling all bits in it, i.e., transforming the 0 bit to 1 and the 1 bit to 0.

3.4.2 FLOATING-POINT TYPE

Rust also has two primitive types for floating-point numbers, which are numbers with decimal points. Rust's floating-point types are `f32` and `f64`, which are 32 bits and 64 bits in size, respectively. **The default type is f64 because on modern CPUs it's roughly the same speed as f32 but is capable of more precision.**

3.4.3 BOOLEAN TYPE

As in most other programming languages, a Boolean type in Rust has two possible values: `true` and `false`. Booleans are one byte in size. The Boolean type in Rust is specified using `bool`.

3.4.4 CHARACTER TYPE

Rust's `char` type is the language's most primitive alphabetic type. `char` literals are specified with single quotes, as opposed to `string` literals, which use double quotes. Rust's char type is four bytes in size and represents a Unicode Scalar Value, which means it can represent a lot more than just ASCII. Accented letters; Chinese, Japanese, and Korean characters; emoji; and zero-width spaces are all valid `char` values in Rust.

```
1 fn main() {
2     let x = 'A';
```

```
3     println!("{}",x);
4 }
```

Listing 3.4: Characters

3.5 COMPOUND DATA TYPE

Compound types can group multiple values into one type. Rust has two primitive compound types - **tuples** and **arrays**.

3.5.1 TUPLE

A tuple is a general way of grouping together a number of values with a **variety of types** into one compound type. **Tuples have a fixed length - once declared, they cannot grow or shrink in size.** The variable **tup** binds to the entire tuple, because a tuple is considered a single compound element. To get the individual values out of a tuple, we can use pattern matching to destructure a tuple value. The `_` is a placeholder, as the variable is not required.

```
1 fn main() {
2     let tup: (i32, f64, u8) = (500, 6.4, 1);
3     let (_, y, _) = tup;
4     println!("The value of y is: {}", y);
5 }
```

Listing 3.5: Tuple

3.5.2 ARRAY

Another way to have a collection of multiple values is with an array. Unlike a tuple, every element of an array must have the same type. Arrays in Rust are different from arrays in some other languages because arrays in Rust have a fixed length, like tuples.

```
1 fn main() {
2     let a = [1, 2, 3, 4, 5];
3     let b = [3; 5];                    // 5 is the size of the array
4     let c: [i32; 5] = [1,2,3,4,5];     // type is stated with a semicolon
5 }
```

Listing 3.6: Array

After the semicolon, the number 5 indicates the array contains five elements. Variable b will contain `[3, 3, 3, 3, 3];` In variable c, `i32` is the type of each element. If you try to access an element of an array that is past the end of the array, the program will compile but Rust will panic and cause a *runtime* error. This is the first example of Rust's safety principles in action. In many low-level languages, this kind of check is not done, and when you provide an incorrect index, invalid memory can be accessed. Rust protects you against this kind of error by immediately exiting instead of allowing the memory access and continuing.

3.6 FUNCTIONS

Functions are defined with the `fn` keyword. Below is an example of a typical function.

```
fn my_function(a: i32, b:i32) -> i32 {
    let c = a + b;
    c
}
```

Listing 3.7: Function

In function signatures, you must declare the type of each parameter as well as the return type. Function bodies are made up of a series of statements optionally ending in an expression. Statements are instructions that perform some action and do not return a value. Expressions evaluate to a resulting value. In Rust, the return value of the function is synonymous with the value of the final expression in the block of the body of a function. You can return early from a function by using the return keyword and specifying a value, but most functions return the last expression implicitly.

In Rust, the idiomatic comment style starts a comment with two slashes, and the comment continues until the end of the line. For comments that extend beyond a single line, you'll need to include `//` on each line.

3.7 CONTROL FLOW

Control flow is achieved through `if(condition)` `else if(condition)` `else` statements. The `condition` must evaluate to `true` or `false` as in other common languages.

3.8 LOOPS

Loops are achieved with `for`, `iter`, `loop` key words. Following are some examples.

```
loop {
    ...
}

let a = [10, 20, 30, 40, 50];

for element in a.iter() {
    println!("the value is: {}", element);
}
```

Listing 3.8: Loops

CHAPTER 4

OWNERSHIP, BORROWING, REFERENCING & LIFETIME

Rust, as we stated earlier, is a ***type safe language***, i.e., the compiler ensures that every program has well-defined behavior. Rust is able to do so without a garbage collector, runtime, or manual memory management. This is possible through Rust's concept of ***ownership***, This is Rust's most unique feature and it enables Rust to make memory safety guarantees without needing a garbage collector. It is important to understand how ownership works in Rust as without that you will not be able to compile the code. Rust ownership rules can be stated as follows:

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

4.1 OWNERSHIP

Ownership begins with assignment and ends with scope. When a variable goes out of scope, its associated value, if any, is ***dropped***. A dropped value can never be used again because the resources it uses are immediately freed. However, a value can be dropped before the end of a scope if the compiler determines that the owner is no longer used within the scope.

```
fn main() {
    {
        let x = 1;
        println!("x: {}", x);
    }

    println!("x: {}", x); // ERROR: x not in scope
}
```

Listing 4.1: Ownership

While most languages would not allow you to use x outside of its local scope, in Rust, when the anonymous scope ends, the value owned by `x`, which is 1, is dropped.

4.2 REASSIGNMENT

Reassignment of ownership (as in `let b = a)` is known as a ***move***. A move causes the former assignee to become uninitialized and therefore not usable in the future.

```
fn main() {
    let a = vec![1, 2, 3];  // a growable array literal
    let b = a;              // move: a can no longer be used
    println!("a: {:?}", b); // error: borrow of moved value: a
}
```

Listing 4.2: Reassignment

Similarly, consider the following code. If we were to use v after this move, the compiler would complain:

```
fn sum(vector: Vec<i32>) -> i32 {
    let mut sum = 0;

    for item in vector {
        sum = sum + item;
    }

    sum
}

fn main() {
    let v = vec![1,2,3];
    let s = sum(v);

    println!("sum of {:?}: {}", v, s); // ERROR: v was MOVED!
}
```

Listing 4.3: Move

Another form of reassignment occurs when returning a value from a function.

```
fn create_series(x: i32) -> Vec<i32> {
    let result = vec![x, x+1, x+2];
    result
}

fn main() {
    let series = create_series(42);
    println!("series: {:?}", series);
}
```

Listing 4.4: Reassignment through return value

This form of reassignment will not cause any problem because when a function exits, its corresponding scope ends. There's no way to later access the old scope or its local variables. We do, however, retain access to return values.

4.3 COPY

During reassignment, for variables in the stack, instead of moving the values owned by the variables, their values are copied. So, while the code in listing 4.3 did not work, the following code will work correctly.

```
fn sum(left: i32, right: i32) -> i32 {
    left + right
}

fn main() {
    let a = 42;
    let b = a;
    let s = sum(a, b);

    println!("this sum of {} and {} is {}", a, b, s); // no error!
}
```

A copy creates an exact duplicate of a value that implements the `Copy` trait. We will study `Struct` and `Trait` in a later chapter. The example with `Vec<i32>` fails to compile because `Vec<i32>` does not implement the `Copy` trait. The reason is that types such as integers have a known size at compile time and are stored on the stack and so it is quick to make copies

of the values. In other words, there's no difference between deep and shallow copying here, so calling `clone` wouldn't do anything different from the usual shallow copying and we can leave it out.

Rust has a special annotation called the `Copy` trait that we can place on types like integers that are stored on the stack. If a type has the `Copy` trait, an older variable is still usable after assignment. Rust won't let us annotate a type with the `Copy` trait if the type, or any of its parts, has implemented the `Drop` trait. If the type needs something special to happen when the value goes out of scope and we add the `Copy` annotation to that type, we'll get a compile-time error.

As a general rule, any group of simple scalar values can be Copy, and nothing that requires allocation or is some form of resource is Copy.

Examples are: `u32, i64, book, f64, char, (i32, f64)`
but `(i32,String)` is not.

Rust will never automatically create "deep" copies of your data. Therefore, any automatic copying can be assumed to be inexpensive in terms of runtime performance.

4.4 CLONE

If we do want to deep copy the heap data of the String, not just the stack data, we can use a common method called `clone`. For example:

```
let s1 = String::from("hello");
let s2 = s1.clone();

println!("s1 = {}, s2 = {}", s1, s2);
```

Listing 4.5: clone method

Structs do not implement `Copy` by default. Reassignment of a struct variable leads to a move, not a copy. However, it is possible to automatically derive the `Copy` and `Clone` trait as follows.

```
#[derive(Debug,Clone,Copy)]
struct Person {
    age: i8
}

fn main() {
    let alice = Person { age: 42 };
    let bob = alice;

    println!("alice: {:?}\nbob: {:?}", alice, bob);
}
```

Listing 4.6: Copy & Clone

4.5 REFERENCING OR BORROWING

Many resources are too expensive in terms of time or memory be copied for every reassignment. In these cases, Rust offers the option to borrow. To do so, we precede the assignee variable with the ampersand `&` character. Non-copyable value can be passed as an argument to a function if it is borrowed.

```rust
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len); // no error
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

Listing 4.7: Borrow

The ampersands are references, they allow you to refer to some value without taking ownership of it. Note that the reference in the above example is passed by value.

4.6 MUTABLE REFERENCE

If it is necessary to mutate a reference, you will need to annotate the type with `mut` in the caller function and with `&mut` in the function arguments.

```rust
fn main() {
    let mut s = String::from("hello");
    change(&mut s);
}

fn change(some_string: &mut String) {
    some_string.push_str(", world");
}
```

Listing 4.8: Mutable reference

But mutable references have one big restriction. **You can have only one mutable reference to a particular piece of data in a particular scope.**

```rust
let mut s = String::from("hello");

let r1 = &mut s;
let r2 = &mut s;

println!("{}, {}", r1, r2); // ERROR: will not compile
```

Listing 4.9: Restriction on mutable reference

We also cannot have a mutable reference while we have an immutable one.

```rust
let mut s = String::from("hello");

let r1 = &s;     // no problem
let r2 = &mut s; // BIG PROBLEM

println!("{}, {}", r1, r2, r3);
```

Listing 4.10: Restriction on mutable reference

However, the following code will work because the last usage of the immutable references occurs before the mutable reference is introduced.

```rust
let mut s = String::from("hello");
let r1 = &s; // no problem
let r2 = &s; // no problem
println!("{} and {}", r1, r2);
// r1 and r2 are no longer used after this point

```

```
let r3 = &mut s; // no problem
println!("{}", r3);
```

Listing 4.11: Restriction on mutable reference

The scopes of the immutable references r1 and r2 end after the `println!` where they are last used, which is before the mutable reference `r3` is created. These scopes don't overlap, so this code is allowed.

The benefit of having these restriction is that Rust can prevent data races at compile time. A data race is similar to a race condition and happens when these three behaviors occur:

- Two or more pointers access the same data at the same time.
- At least one of the pointers is being used to write to the data.
- There's no mechanism being used to synchronize access to the data.

Data races cause undefined behavior and can be difficult to diagnose and fix when you're trying to track them down at runtime. Rust prevents this problem from happening because it won't even compile code with data races.

4.7 DANGLING REFERENCES

The Rust compiler guarantees that references will never be dangling references. if you have a reference to some data, the compiler will ensure that the data will not go out of scope before the reference to the data does.

```
fn main() {
    let reference_to_nothing = dangle();
}

fn dangle() -> &String { // ERROR: compile failed
    let s = String::from("hello");

    &s
}
```

Listing 4.12: Dangling Reference

Because s is created inside dangle, when the code of dangle is finished, s will be deallocated. But we tried to return a reference to it. That means this reference would be pointing to an invalid String. Rust won't let us do this.

The solution here is to return the String directly.

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

    s
}
```

Listing 4.13: Solution to dangling Reference

4.8 SLICE TYPE

Slices let you reference a contiguous sequence of elements in a collection rather than the whole collection. **Slices not have ownership.**

```
let s = String::from("hello world");
let hello = &s[0..5];
let world = &s[6..11];
```

Listing 4.14: Slice

We can create slices using a range within brackets by specifying [starting_index..ending_index], where starting_index is the first position in the slice and ending_index is one more than the last position in the slice. If you drop the starting_index then it is taken as 0 and If you drop the ending_index then it is taken as the length which is `s.len` in this case. The type that signifies **string slice** is written as `&str`.

String literals are slices. Consider:

```
let s = "Hello, world!";
```

Listing 4.15: String literals are slices

The type of s here is `&str`. It's a slice pointing to that specific point of the binary. This is also why string literals are immutable. `&str` is an immutable reference.

Similarly, consider the following array:

```
let a = [1, 2, 3, 4, 5];
let slice = &a[1..3];
```

Listing 4.16: Slices of an Array

This slice has the type `&[i32]`. It works the same way as string slices do, by storing a reference to the first element and a length. You'll use this kind of slice for all sorts of other collections.

4.9 LIFETIME

Sometimes we'll want a function to return a borrowed value. Consider the following code which will fail to compile.

```
// ERRORS!
fn longest(x: &str, y: &str) -> &str {
    if x.bytes().len() > y.bytes().len() {
        x
    } else {
        y
    }
}

fn main() {
    let alice = "Alice";
    let bob = "Bob";

    println!("{}", longest(alice, bob));
}
```

Listing 4.17: Compiler unable to infer Lifetime

Notice that `x` and `y` are borrowed but only one of them is returned. A **lifetime** is the scope within which a borrowed reference is valid. The Rust compiler is smart enough to infer lifetimes in many cases, meaning that we don't need to explicitly write them but sometimes, as in this case, we do need to specify them.

Now consider the code below which compiles and runs correctly.

```
fn longest<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.bytes().len() > y.bytes().len() {
        x
    } else {
        y
    }
}

fn main() {
    let alice = "Alice";
    let bob = "Bob";

    println!("{}", longest(alice, bob));
}
```

Listing 4.18: References with Lifetime

The change allows the compiler to determine that the lifetime (valid scope) of the value whose borrowed reference it returns matches the lifetime of the parameters x and y. In other words, there is no way for the longest function to return a reference to a dropped value.

Bibliography

[1] Steve Klabnik and Carol Nichols, with contributions from the Rust Community, *The Rust Programming Language*. https://doc.rust-lang.org/book, 2018.

[2] Donald E. Knuth, *The Art of Computer Programming*. Addison-Wesley, 2011.

[3] Alfred V. Aho, Monica S. Lam, Ravi Shethi , Jeffery D. Ullman, *Compilers: Principles, Techniques, and Tools*. Prentice Hall, 2006.

[4] Kanglin Li, Mengqi Wu, Sybex, *Effective Software Test Automation: Developing an Automated Software Testing Tool*. Sybex , 2004.

[5] Martin Odersky, *The Scala Language Specification Version 2.9* PROGRAMMING METHODS LABORATORY EPFL, SWITZERLAND, 2014.