# Moving to the Rust programming language

Jaideep Ganguly, ScD

The Rust
Programming
Language

# Contents

Contents i

List of Tables iv

List of Listings v

Preface vii

1 WHY RUST? 1
- 1.1 STATICALLY TYPED 1
- 1.2 TYPE SAFETY 1
- 1.3 RUNTIME 2
- 1.4 PERFORMANCE 3

2 GETTING STARTED 5
- 2.1 INSTALLATION 5
- 2.2 CREATING PROJECT WITH CARGO 5
- 2.3 BUILD & RUN WITH CARGO 6
- 2.4 CREATING A LIBRARY WITH CARGO 7

3 BASIC CONCEPTS 9
- 3.1 VARIABLES & MUTABILITY 9
- 3.2 SHADOWING 9
- 3.3 DATA TYPES 10
- 3.4 SCALAR DATA TYPE 10
  - 3.4.1 INTEGER TYPE 10
  - 3.4.2 FLOATING-POINT TYPE 11
  - 3.4.3 BOOLEAN TYPE 11
  - 3.4.4 CHARACTER TYPE 11
- 3.5 COMPOUND DATA TYPE 12
  - 3.5.1 TUPLE 12
  - 3.5.2 ARRAY 12
- 3.6 FUNCTIONS 12
- 3.7 CONTROL FLOW 13
- 3.8 LOOPS 13

4 OWNERSHIP, BORROWING, REFERENCING & LIFETIME 15
- 4.1 OWNERSHIP 15
- 4.2 REASSIGNMENT 15
- 4.3 COPY 16
- 4.4 CLONE 17
- 4.5 REFERENCING OR BORROWING 17
- 4.6 MUTABLE REFERENCE 18
- 4.7 DANGLING REFERENCES 19
- 4.8 SLICE TYPE 19
- 4.9 LIFETIME 20
  - 4.9.1 STATIC 21

**5 STRUCT** 23
- **5.1 DEFINING A STRUCT** 23
- **5.2 INSTANTIATING STRUCTS** 23
  - 5.2.1 FIELD INIT SHORTHAND 24
  - 5.2.2 STRUCT UPDATE 24
- **5.3 TUPLE STRUCT** 24
- **5.4 UNIT-LIKE STRUCT** 25
- **5.5 OWNERSHIP OF STRUCT DATA** 25
- **5.6 METHODS** 25
- **5.7 ASSOCIATED FUNCTIONS** 26

**6 ENUM & PATTERN MATCHING** 27
- **6.1 DEFINING AN ENUM** 27
- **6.2 STRUCT & ENUM** 27
- **6.3 OPTION ENUM** 28
- **6.4 MATCH STATEMENT** 29
- **6.5 MATCH WITH OPTION<T>** 30
- **6.6 MATCH & ENUM WITH DATA TYPE** 31

**7 COLLECTIONS** 33
- **7.1 VECTOR** 33
- **7.2 HASHMAP** 34
- **7.3 HASHSET** 34

**8 ERROR HANDLING** 37
- **8.1 RECOVERABLE ERRORS & RESULT ENUM** 38

**9 GENERICS & TRAITS** 39
- **9.1 GENERICS** 39
- **9.2 TRAITS** 39
- **9.3 GENERIC FUNCTIONS** 40

**10 CLOSURES & ITERATORS** 41
- **10.1 CLOSURES** 41
- **10.2 STORING CLOSURES WITH Fn TRAIT** 42

**11 SMART POINTERS** 45
- **11.1 Box** 45
- **11.2 ENABLING RECURSIVE TYPES WITH BOXES** 45
  - 11.2.1 CONS LIST 45
- **11.3 SMART POINTERS LIKE REGULAR REFERENCES WITH DEREF TRAIT** 46
- **11.4 RUNNING CODE ON CLEANUP WITH THE DROP TRAIT** 47

**12 CONCURRENCY** 49
- **12.1 THREADS** 49
- **12.2 MESSAGE PASSING TO TRANSFER DATA BETWEEN THREADS** 50
- **12.3 CHANNELS & OWNERSHIP TRANSFERENCE** 50

**Bibliography** 53

**Index** 54

# LIST OF TABLES

3.1 Integer Types in Rust 10
3.2 Integer Literals in Rust 11

# LIST OF LISTINGS

2.1 Installation in MacOS or Linux 5
2.2 Cargo 5
2.3 Project with Cargo 5
2.4 Project with Cargo 6
2.5 Hello World! 6
2.6 Build & Run 6

3.1 Variables & Mutability 9
3.2 Shadowing 10
3.3 Type 10
3.4 Characters 11
3.5 Tuple 12
3.6 Array 12
3.7 Function 13
3.8 Loops 13

4.1 Ownership 15
4.2 Reassignment 15
4.3 Move 16
4.4 Reassignment through return value 16
4.5 clone method 17
4.6 Copy & Clone 17
4.7 Borrow 18
4.8 Mutable reference 18
4.9 Restriction on mutable reference 18
4.10 Restriction on mutable reference 18
4.11 Restriction on mutable reference 18
4.12 Dangling Reference 19
4.13 Solution to dangling Reference 19
4.14 Slice 20
4.15 String literals are slices 20
4.16 Slices of an Array 20
4.17 Compiler unable to infer Lifetime 20
4.18 References with Lifetime 21

5.1 struct 23
5.2 Instance of a struct 23
5.3 Mutable struct 23
5.4 Function using field init shorthand 24
5.5 struct update 24
5.6 Tuple Struct 24
5.7 Lifetime annotations in struct 25
5.8 impl & struct 25
5.9 Associated Function 26

6.1 Defining an enum 27
6.2 Enum with data 27
6.3 Enum with varying data 28

6.4 Message enum 28
6.5 Methods on enum 28
6.6 Option enum 28
6.7 Option examples 29
6.8 Option example 29
6.9 Match Statement 29
6.10 Match with Option 30
6.11 Placeholder in match 30
6.12 if let 30
6.13 Match & Enum with data type 31

7.1 Vector 33
7.2 Iterate in a Vector 33
7.3 Functions associated with vec 33
7.4 Create a HashMap 34
7.5 Commonly used functions of a HashMap 34
7.6 Commonly used functions of a HashSet 34

8.1 panic! 37
8.2 Another example of panic! 37
8.3 enum Result 38
8.4 enum Result usage example 38
8.5 Example of expect() 38

9.1 Generic example 39
9.2 Example of a trait 39
9.3 Implement a trait 40
9.4 Example of a generic function 40

10.1 Closure example 41
10.2 Closure example 42
10.3 Implementing Cacher 42
10.4 Using Cacher in the generate_force function to abstract away the caching logic 43

11.1 Storing an i32 value on the heap using a box 45
11.2 Definition of List that uses Box<T> in order to have a known size 46
11.3 Using the dereference operator on a Box<i32> 47

12.1 Message passing in concurrent program 51

# 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.

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

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

```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:

```rust
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.

```
fn main() {
    let x = ’A’;
    println!("{}",x);
}
```

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.

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

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.

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

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:

```rust
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.

```rust
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.

```rust
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:

```rust
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.

```rust
#[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.**

```rust
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:

```rust
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:

```rust
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.

```rust
// 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.

```rust
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.

## 4.9.1 STATIC

Rust has a few reserved lifetime names. One of those is `'static`. You might encounter it in two situations:

```rust
// A reference with 'static lifetime:
let s: &'static str = "hello world";

// 'static as part of a trait bound:
fn generic<T>(x: T) where T: 'static {}
```

Both are related but subtly different and this is a common source for confusion when learning Rust.

### 4.9.1.1 REFERENCE LIFETIME

As a reference lifetime `'static` indicates that the data pointed to by the reference lives for the entire lifetime of the running program. It can still be coerced to a shorter lifetime. There are two ways to make a variable with `'static` lifetime, and both are stored in the read-only memory of the binary.

- Make a constant with the `static` declaration.
- Make a `string` literal which has type: `&'static str`.

### 4.9.1.2 TRAIT BOUND

As a trait bound, it means the type does not contain any non-static references. Eg. the receiver can hold on to the type for as long as they want and it will never become invalid until they drop it.

It's important to understand this means that any owned data always passes a 'static lifetime bound, but a reference to that owned data generally does not:

# CHAPTER 5

# STRUCT

## 5.1 DEFINING A STRUCT

A `struct` is a custom data type that lets you name and package together multiple related values that make up a meaningful group. Structs are similar to tuples. Like tuples, the pieces of a struct can be different types. Unlike with tuples, you'll name each piece of data so it's clear what the values mean.

```
struct User {
    username: String,
    email: String,
    sign_in_count: u64,
    active: bool,
}
```

Listing 5.1: struct

## 5.2 INSTANTIATING STRUCTS

To use a struct after we've defined it, we create an instance of that struct by specifying concrete values for each of the fields. We create an instance by stating the name of the struct and then add curly brackets containing key: value pairs, where the keys are the names of the fields and the values are the data we want to store in those fields. We don't have to specify the fields in the same order in which we declared them in the struct.

```
fn main() {
    let user1 = User {
        email: String::from("someone@example.com"),
        username: String::from("someusername123"),
        active: true,
        sign_in_count: 1,
    };
}
```

Listing 5.2: Instance of a struct

To get a specific value from a struct, we can the use dot notation. If we wanted just this user's email address, we could use `user1.email` wherever we wanted to use this value.

If the instance is mutable, we can change a value by using the dot notation and assigning into a particular field.

```
let mut user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("someusername123"),
    active: true,
    sign_in_count: 1,
};

```

```rust
user1.email = String::from("anotheremail@example.com");
```

Listing 5.3: Mutable struct

**Note that the entire instance must be mutable; Rust doesn't allow us to mark only certain fields as mutable.**

### 5.2.1 FIELD INIT SHORTHAND

As with any expression, we can construct a new instance of the struct as the last expression in the function body to implicitly return that new instance.

```rust
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Listing 5.4: Function using field init shorthand

### 5.2.2 STRUCT UPDATE

It's often useful to create a new instance of a `struct` that uses most of an old instance's values but changes some. You can do with the struct update syntax. The syntax `..` specifies that the remaining fields not explicitly set should have the same value as the fields in the given instance.

```rust
let user2 = User {
    email: String::from("another@example.com"),
    username: String::from("anotherusername567"),
    ..user1
};
```

Listing 5.5: struct update

## 5.3 TUPLE STRUCT

You can also define structs that look similar to tuples, called **tuple structs**. Tuple structs have the added meaning the struct name provides but don't have names associated with their fields; rather, they just have the types of the fields. Tuple structs are useful when you want to give the whole tuple a name and make the tuple be a different type from other tuples, and naming each field as in a regular struct would be verbose or redundant.

To define a tuple struct, start with the `struct` keyword and the struct name followed by the types in the tuple.

```rust
struct Color(i32, i32, i32);
let black = Color(0, 0, 0);
let white = Color(255,255,255);
```

Listing 5.6: Tuple Struct

Tuple struct instances behave like tuples: you can destructure them into their individual pieces, you can use a `.` followed by the index to access an individual value, and so on.

## 5.4 UNIT-LIKE STRUCT

You can also define structs that don’t have any fields! These are called unit-like structs because they behave similarly to (), the unit type. Unit-like structs can be useful in situations in which you need to implement a trait on some type but don’t have any data that you want to store in the type itself. We will discuss traits in a later chapter.

## 5.5 OWNERSHIP OF STRUCT DATA

It’s possible for structs to store references to data owned by something else, but to do so requires the use of lifetimes. Lifetimes ensure that the data referenced by a struct is valid for as long as the struct is.

```
struct ImportantExcerpt<’a> {
    part: &’a str,
}

fn main() {
    let novel = String::from("Call me Ishmael. Some years ago...");
    let first_sentence = novel.split(’.’).next().expect("Could not find a ’.’");
    let i = ImportantExcerpt {
        part: first_sentence,
    };
}
```

Listing 5.7: Lifetime annotations in struct

## 5.6 METHODS

Methods are similar to functions. they are declared with the `fn` keyword and their name, they can have parameters and a return value. However, methods, unlike functions, are defined within the context of a struct (or an enum or a trait object, which we cover later), and their first parameter is always `self` which represents the instance of the struct the method is being called on. To define the function within the context of `struct`, we start an `impl` (implementation) block. Consider the following code:

```
// The ‘derive‘ attribute automatically creates the implementation
// required to make this ‘enum‘ printable with ‘fmt::Debug‘.
#[derive(Debug)]
struct Rectangle {
    width: u32,
    height: u32,
}

impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
    }
}

fn main() {
    let rect1 = Rectangle {
        width: 30,
        height: 50,
    };

    println!( "The area of the rectangle is {} square pixels.",rect1.area() );
}
```

Listing 5.8: impl & struct

The main benefit of using methods instead of functions, in addition to using method syntax and not having to repeat the type of `self` in every method's signature, is for organization. We've put all the things we can do with an instance of a type in one `impl` block.

Note that Rust doesn't have an equivalent to the `->` operator; instead, Rust has a feature called automatic referencing and dereferencing. Calling methods is one of the few places in Rust that has this behavior. When you call a method with `object.something()`, Rust automatically adds in `&`, `&mut`, or `*` so object matches the signature of the method. In other words, the following are the same:

```
p1.distance(&p2);
(&p1).distance(&p2);
```

The first one looks much cleaner. This automatic referencing behavior works because methods have a clear receiver—the type of `self`. Given the receiver and name of a method, Rust can figure out definitively whether the method is reading `(&self)`, mutating `(&mut self)`, or consuming `(self)`. **The fact that Rust makes borrowing implicit for method receivers is a big part of making ownership ergonomic in practice.**

Methods can take multiple parameters that we add to the signature after the self parameter, and those parameters work just like parameters in functions.

## 5.7 ASSOCIATED FUNCTIONS

Another useful feature of `impl` blocks is that we're allowed to define functions within `impl` blocks that don't take self as a parameter. These are called associated functions because they're associated with the struct. They're still functions, not methods, because they don't have an instance of the struct to work with. You've already used the `String::from` associated function.

Associated functions are often used for constructors that will return a new instance of the struct. For example, we could provide an associated function that would have one dimension parameter and use that as both width and height, thus making it easier to create a square Rectangle rather than having to specify the same value twice.

```
impl Rectangle {
    fn square(size: u32) -> Rectangle {
        Rectangle {
            width: size,
            height: size,
        }
    }
}
```

Listing 5.9: Associated Function

## CHAPTER 6

# ENUM & PATTERN MATCHING

Enumerations, also referred to as enums, allow you to define a type by enumerating its possible variants. When we have to select a value from a list of possible variants, we use enumeration data types. An enumerated type is declared using the `enum` keyword. A particularly useful `enum`, called `Option`, which expresses that a value can be either something or nothing. Then we'll look at how pattern matching in the `match` expression makes it easy to run different code for different values of an enum. Finally, we'll cover how the `if let` construct is another convenient and concise idiom available to you to handle enums in your code. Rust's enums are most similar to algebraic data types in functional languages, such as F#, OCaml, and Haskell.

## 6.1 DEFINING AN ENUM

```rust
// The ‘derive‘ attribute automatically creates the implementation
// required to make this ‘enum‘ printable with ‘fmt::Debug‘.
#[derive(Debug)]
enum GenderCategory {
    Male,
    Female
}

fn main() {
    let male = GenderCategory::Male;
    let female = GenderCategory::Female;

    println!("{:?}",male);
    println!("{:?}",female);
}
```

Listing 6.1: Defining an enum

The output is:

```
Male
Female
```

## 6.2 STRUCT & ENUM

We can put data directly into each `enum` variant.

```rust
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("127.0.0.1"));
let loopback = IpAddr::V6(String::from("::1"));
```

Listing 6.2: Enum with data

Each variant can have different types - strings, numeric types, or structs - and amounts of associated data as shown below:

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}

let home = IpAddr::V4(127, 0, 0, 1);
let loopback = IpAddr::V6(String::from("::1"));
```

Listing 6.3: Enum with varying data

Consider the `Message` enum as defined below:

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String),
    ChangeColor(i32, i32, i32),
}
```

Listing 6.4: Message enum

Just as we're able to define methods on structs using `impl`, we're also able to define methods on enums. Here's a method named call that we could define on our Message enum:

```
fn main() {
    enum Message {
        Quit,
        Move { x: i32, y: i32 },
        Write(String),
        ChangeColor(i32, i32, i32),
    }

    impl Message {
        fn call(&self) {
            // method body would be defined here
        }
    }

    let m = Message::Write(String::from("hello"));
    m.call();
}
```

Listing 6.5: Methods on enum

## 6.3 OPTION ENUM

Option is a predefined `enum` in the Rust standard library. This `enum` has two values - `Some(data)` and `None`.

```
enum Option<T> {
   Some(T),      //used to return a value
   None          // used to return null, as Rust doesn't support the null keyword
}
```

Listing 6.6: Option enum

Here, the type `T` represents value of any type. **Rust does not support the null keyword.** The value `None`, in the enumOption, can be used by a function to return a null value. If there is data to return, the function can return `Some(data)`.

So, how do you get the `T` value out of a Some variant when you have a value of type `Option<T>` so you can use that value? The `Option<T>` enum has a large number of methods that are useful in a variety of situations. Below are some examples:

```rust
let x: Option<u32> = Some(2);
assert_eq!(x.is_some(), true);

let x: Option<u32> = None;
assert_eq!(x.is_some(), false);

let x: Option<u32> = Some(2);
assert_eq!(x.contains(&2), true);
```

Listing 6.7: Option examples

```rust
fn main() {
   let result = is_even(3);
   println!("{:?}",result);
   println!("{:?}",is_even(30));
}

fn is_even(no:i32) -> Option<bool> {
   if no % 2 == 0 {
      Some(true)
   } else {
      None
   }
}
```

Listing 6.8: Option example

Output:

```
None
Some(true)
```

## 6.4 MATCH STATEMENT

The match statement can be used to compare values stored in an enum. The following example defines a function, `print_size`, which takes `CarType` enum as parameter. The function compares the parameter values with a pre-defined set of constants and displays the appropriate message.

```rust
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

Listing 6.9: Match Statement

## 6.5 MATCH WITH OPTION<T>

The example of `is_even` function, which returns Option type, can also be implemented with match statement as shown below.

```rust
fn main() {
   match is_even(5) {
      Some(data) => {
         if data == true {
            println!("Even no");
         }
      },
      None => {
         println!("not even");
      }
   }
}

fn is_even(no:i32) -> Option<bool> {
   if no%2 == 0 {
      Some(true)
   } else {
      None
   }
}
```

Listing 6.10: Match with Option

Matches in Rust are exhaustive. We must exhaust every last possibility in order for the code to be valid. Especially in the case of `Option<T>`, when Rust prevents us from forgetting to explicitly handle the `None` case, it protects us from assuming that we have a value when we might have null.

Rust also has a pattern we can use when we don't want to list all possible values. If we only care about the values 1, 3, 5, and 7, we can use the special pattern `_` instead to handle the rest. The `()` is just the unit value, so nothing will happen in the `_` case.

```rust
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

Listing 6.11: Placeholder in match

However, the match expression can be a bit wordy in a situation in which we care about only one of the cases. For this situation, Rust provides `if let`. So, for example, if we are only interested in the value of 3, we can write:

```rust
fn main() {
    let some_u8_value = Some(0u8);
    if let Some(3) = some_u8_value {
        println!("three");
    }
}
```

Listing 6.12: if let

## 6.6 MATCH & ENUM WITH DATA TYPE

It is possible to add data type to each variant of an enum. In the following example, Name and `Usr_ID` variants of the enum are of String and integer types respectively. The following example shows the use of match statement with an enum having a data type.

```
// The ‘derive‘ attribute automatically creates the implementation
// required to make this ‘enum‘ printable with ‘fmt::Debug‘.
#[derive(Debug)]
enum GenderCategory {
   Name(String),
   Usr_ID(i32)
}
fn main() {
   let p1 = GenderCategory::Name(String::from("Tim"));
   let p2 = GenderCategory::Usr_ID(100);
   println!("{:?}",p1);
   println!("{:?}",p2);

   match p1 {
      GenderCategory::Name(val)=> {
         println!("{}",val);
      }
      GenderCategory::Usr_ID(val)=> {
         println!("{}",val);
      }
   }
}
```

Listing 6.13: Match & Enum with data type

Output:

```
Name("Tim")
Usr_ID(100)
Tim
```

## CHAPTER 7

# COLLECTIONS

Rust's standard library includes a number of very useful data structures called collections. Most other data types represent one specific value, but collections can contain multiple values. **Unlike the built-in array and tuple types, the data these collections point to is stored on the heap**. This means the amount of data does not need to be known at compile time and can grow or shrink as the program runs. Below are three commonly used collections.

- A vector allows you to store a variable number of values next to each other.
- A hash map allows you to associate a value with a particular key. It's a particular implementation of the more general data structure called a map.

## 7.1 VECTOR

A Vector is a resizable array. It stores values in contiguous memory blocks. The predefined structure `Vec` can be used to create vectors.

- A Vector can grow or shrink at runtime.
- A Vector is a homogeneous collection.
- A Vector stores data as sequence of elements in a particular order. Every element in a Vector is assigned a unique index number starting with zero.
- A Vector will only append values to (or near) the end. In other words, a Vector can be used to implement a stack.
- Memory for a Vector is allocated in the heap.

You create a `Vec` as follows

```rust
let mut instance_name = Vec::new();

# or with a macro
let v = vec![val1,val2,val3];
```

Listing 7.1: Vector

Values in the vector are accessed using the index, e.g., `v[1]`. You can iterate as follows:

```rust
fn main() {
    let v = vec![100, 32, 57];
    for i in &v {
        println!("{}", i);
    }
}
```

Listing 7.2: Iterate in a Vector

Following are some examples of commonly used functions associated with `Vec`.

```rust
let mut v = Vec::new();
v.push(20);
```

```
v.remove(20);
v.contains(&20);    # returns true is the slice contains the element
v.len();
```

Listing 7.3: Functions associated with vec

## 7.2 HASHMAP

A map is a collection of key-value pairs (called entries). No two entries in a map can have the same key. In short, a map is a lookup table. A HashMap stores the keys and values in a hash table. The entries are stored in an arbitrary order. The key is used to search for values in the HashMap. The HashMap structure is defined in the `std::collections` module. This module should be explicitly imported to access the HashMap structure.

You can create a HashMap as follows:

```
let mut instance_name = HashMap::new();
```

Listing 7.4: Create a HashMap

Following are some examples of commonly used functions associated with `HashMap`.

```
use std::collections::HashMap;
let mut stateCodes = HashMap::new();
stateCodes.insert("MA","Massachusetts");
stateCodes.insert("NY","New York");
stateCodes.len();

for (key, val) in stateCodes.iter() {
  println!("key: {} val: {}", key, val);
}

if stateCodes.contains_key(&"MA") {
  println!("found key");
}

stateCodes.remove(&"MA");
```

Listing 7.5: Commonly used functions of a HashMap

If we insert a key and a value into a hash map and then insert that same key with a different value, the value associated with that key will be replaced. The hash map will only contain one key/value pair.

## 7.3 HASHSET

Consider a HashSet as a HashMap where we just care about the keys ( HashSet<T> is, in actuality, just a wrapper around HashMap<T, ()>). A HashSet's unique feature is that it is guaranteed to not have duplicate elements. HashSet is a set of unique values of type T. Adding and removing values is fast, and it is fast to ask whether a given value is in the set or not. The HashSet structure is defined in the `std::collections` module.

Following are some examples of commonly used functions associated with `HashSet`.

```
use std::collections::HashSet;
let mut names = HashSet::new();
names.insert("Tim");
names.insert("Tom");
names.insert("Tom");
names.len();

for name in names.iter() {
    println!("{}",name);
}

match names.get(&"Tim") {
    Some(value)=>{
        println!("found {}",value);
    }

    None => {
        println!("not found");
    }
}

if names.contains(&"Tom") {
    println!("found name");
}
```

Listing 7.6: Commonly used functions of a HashSet

```
Tom
Tim
found Tim
found name
```

Note that "Tom" has been inserted twice and yet the hashset has only one occurrence of "Tom".

## CHAPTER 8

# ERROR HANDLING

Errors are a fact of life in software. Rust has several features for handling situations when something goes wrong. In many cases, Rust requires you to acknowledge the possibility of an error and take some action so that your code will compile. This requirement makes your program more robust by ensuring that you'll discover errors and handle them appropriately before you've deployed your code to production.

Rust groups run time errors into two major categories: recoverable and unrecoverable errors. For a recoverable error, such as a file not found error, it's reasonable to report the problem to the user and retry the operation. Unrecoverable errors are always symptoms of bugs, like trying to access a location beyond the end of an array.

Most languages don't distinguish between these two kinds of errors and handle both in the same way, using mechanisms such as exceptions. Rust does not have exceptions. Instead, it has the type `Result<T, E>` for recoverable errors and the `panic!` macro that stops execution when the program encounters an unrecoverable error. We first discuss `panic!` first and then about the returning values of `Result<T, E>` .

In Rust, errors can be classified into two major categories **Recoverable** and `UnRecoverable`. A recoverable error is an error that can be corrected. A program can retry the failed operation or specify an alternate course of action when it encounters a recoverable error. Recoverable errors do not cause a program to fail abruptly. An example of a recoverable error is `File Not Found error`. Unrecoverable errors cause a program to fail abruptly. A program cannot revert to its normal state if an unrecoverable error occurs. It cannot retry the failed operation or undo the error. An example of an unrecoverable error is trying to access a location beyond the end of an array.

Unlike other programming languages, Rust does not have exceptions. It returns an enum `Result<T, E>` for recoverable errors. For unrecoverable errors, Rust calls the `panic!` macro that causes the program to exit abruptly and provide feedback to the caller of the program.

```
fn main() {
   panic!("Hello");   // will print: thread ‘main’ panicked at ‘Hello’, main.rs:3
   println!("End of main"); //unreachable statement
}
```

Listing 8.1: panic!

```
fn main() {
   let a = [10,20,30];
   a[10]; // automatically invokes a panic since index 10 cannot be reached
}
```

Listing 8.2: Another example of panic!

## 8.1 RECOVERABLE ERRORS & RESULT ENUM

`enum Result <T,E>` can be used to handle recoverable errors. It has two variants - `OK(T)` and `Err(E)`. `T` and `E` are generic type parameters. `T` represents the type of the value that will be returned in a success case within the `OK variant`, and `E` represents the type of the error that will be returned in a failure case within the `Err variant`.

```
enum Result<T,E> {
   OK(T),
   Err(E)
}
```

Listing 8.3: enum Result

```
use std::fs::File;
fn main() {
   let f = File::open("main.jpg");   // main.jpg doesn't exist
   match f {
      Ok(f)=> {
         println!("file found {:?}",f);
      },
      Err(e)=> {
         println!("file not found \n{:?}",e);   //handled error
      }
   }
   println!("end of main");
}
```

Listing 8.4: enum Result usage example

The standard library contains a couple of helper methods that both enums - `Result<T,E>` and `Option<T>` implement. The `unwrap()` function returns the actual result an operation succeeds. It returns a panic with a default error message if an operation fails. On the other hand, with the `expect()` function, the program can return a custom error message in case of a panic.

```
fn main(){
   let f = File::open("pqr.txt").expect("File not able to open");
   // file does not exist
   println!("end of main");
}
```

Listing 8.5: Example of expect()

CHAPTER 9

# GENERICS & TRAITS

Generics are a facility to write code for multiple contexts with different types. In Rust, generics refer to the parameterization of data types and traits. Generics allows to write more concise and clean code by reducing code duplication and providing type-safety. **The concept of Generics can be applied to methods, functions, structures, enumerations, collections and traits.**

## 9.1 GENERICS

The <T> syntax known as the type parameter, is used to declare a generic construct. T represents any data-type.

```
struct Data<T> {
   value:T,
}

fn main() {
   //generic type of i32
   let t:Data<i32> = Data{value:350};
   println!("value is :{} ",t.value);

   //generic type of String
   let t2:Data<String> = Data{value:"Tom".to_string()};
   println!("value is :{} ",t2.value);
}
```

Listing 9.1: Generic example

## 9.2 TRAITS

Traits can be used to implement a standard set of behaviors (methods) across multiple structures. Traits are like interfaces in Object-oriented Programming.

```
trait some_trait {
   // abstract or method which is empty
   fn method1(&self);

   // this is already implemented , this is free
   fn method2(&self){
      //some contents of method2
   }
}
```

Listing 9.2: Example of a trait

**Traits can contain concrete methods (methods with body) or abstract methods (methods without a body). We use a concrete method if the method definition will be shared by all structures implementing the Trait. However, a structure can choose to override a function defined by the trait. We use abstract methods if the method definition varies for the implementing structures.**

Consider the following example:

```
struct Book {
   name: &'static str,
   id:u32
}

trait Printable {
   fn print(&self);
}

impl Printable for Book {
   fn print(&self){
      println!("Printing book with id:{} and name {}",self.id,self.name)
   }
}

fn main(){
   let b = Book {
      id:98304,
      name:"Moving to Rust"
   };
   b.print();
}
```

Listing 9.3: Implement a trait

## 9.3 GENERIC FUNCTIONS

The example defines a generic function that displays a parameter passed to it. The parameter can be of any type. The parameter's type should implement the Display trait so that its value can be printed by the `println!` macro.

```
use std::fmt::Display;

fn main(){
   print_gen(110 as u8);
   print_gen(220 as u16);
   print_gen("Hi There!");
}

fn print_gen<T: Display>(t: T){
   println!("Inside print_gen function:");
   println!("{}",t);
}
```

Listing 9.4: Example of a generic function

CHAPTER 10

# CLOSURES & ITERATORS

In this chapter we study features of Rust that are similar to many features of other languages that are referred to as **functional** languages.

## 10.1 CLOSURES

Rust's closures are anonymous functions that can be saved in a variable or pass as arguments to other functions. One can create the closure in one place and then call the closure to evaluate it in a different context. **Unlike functions, closures can capture values from the scope in which they are defined, this is an important differentiation**.

Unlike functions, closures do not require annotating the types of the parameters or the return values. Type annotations are required on functions because they are part of an explicit interface exposed to its users. Defining this interface rigidly is important for ensuring that everyone agrees on what types of values a function uses and returns. But closures are not used in an exposed interface like this. They are stored in variables and used without naming them and exposing them to users of the library.

Closures are usually short and relevant only within a narrow context rather than in any arbitrary scenario. Within these limited contexts, the compiler is reliably able to infer the types of the parameters and the return type. This is similar to how the compiler is able to infer the types of most variables. Making programmers annotate the types in these small, anonymous functions would be bothersome and largely redundant as the compiler already has that information available with it.

```rust
let some_closure = |num| {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num + 1
};
```

Listing 10.1: Closure example

The closure definition comes after the = to assign it to the variable some_closure. To define a closure, we start with a pair of vertical pipes (|), inside which we specify the parameters to the closure; this syntax was chosen because of its similarity to closure definitions in Smalltalk and Ruby. This closure has one parameter named `num`: if we had more than one parameter, we would separate them with commas, like `|param1, param2|`.

Note that the let statement means `some_closure` contains the definition of an anonymous function, not the resulting value of calling the anonymous function. Recall that we're using a closure because we want to define the code to call at one point, store that code, and call it at a later point;

As with variables, we can add type annotations if we want to increase explicitness and clarity at the cost of being more verbose than is strictly necessary.

```
let some_closure = |num: u32| -> u32 {
    println!("calculating slowly...");
    thread::sleep(Duration::from_secs(2));
    num + 1
};
```

Listing 10.2: Closure example

The first time we call `some_closure` with the String value, the compiler infers the type of `num` and the return type of the closure to be `u32`. Those types are then locked in to the closure in `some_closure`, and we get a type error if we try to use a different type with the same closure.

## 10.2 STORING CLOSURES WITH Fn TRAIT

We can create a struct that will hold the closure and the resulting value of calling the closure. The struct will execute the closure only if we need the resulting value, and it will cache the resulting value so the rest of our code doesn't have to be responsible for saving and reusing the result. This is the pattern known as **memoization or lazy evaluation**.

To make a struct that holds a closure, we need to specify the type of the closure because a struct definition needs to know the types of each of its fields. Each closure instance has its own unique anonymous type: that is, even if two closures have the same signature, their types are still considered different. The Fn traits are provided by the standard library. All closures implement at least one of these traits: `Fn`, `FnMut` or `FnOnce`.

In listing 10.3 the `struct Cacher` has a `calculation` field of the generic type `T`. The trait bounds on `T` specify that it's a closure by using the `Fn` trait. Any closure we want to store in the `calculation` field must have one `u32` parameter (specified within the parentheses after `Fn`) and must return a `u32` (specified after the `->`).

The value field is of type `Option<u32>`. Before we execute the closure, value will be `None`. When code using a Cacher asks for the result of the closure, the Cacher will execute the closure at that time and store the result within a `Some` variant in the value field. Then if the code asks for the result of the closure again, instead of executing the closure again, the Cacher will return the result held in the `Some` variant.

```
struct Cacher<T>
where T: Fn(u32) -> u32 {
    calculation: T,
    value: Option<u32>,
}

impl<T> Cacher<T>
where T: Fn(u32) -> u32 {
    fn new(calculation: T) -> Cacher<T> {
        Cacher {
            calculation,
            value: None,
        }
    }

    fn value(&mut self, arg: u32) -> u32 {
        match self.value {
            Some(v) => v,
            None => {
                let v = (self.calculation)(arg);
                self.value = Some(v);
                v
            }
        }
    }
}
```

Listing 10.3: Implementing Cacher

We now demonstrate the usage of `Cacher`.

```
fn generate_force(hp: u32, random_number: u32) {

    let mut some_closure = Cacher::new(|num| {
        println!("calculating slowly...");
        thread::sleep(Duration::from_secs(2));
        num
    });

    if hp < 25 {
        println!("Today, do {} pushups!", some_closure.value(hp));
        println!("Next, do {} situps!", some_closure.value(hp));
    } else {

        if random_number == 3 {
            println!("No force generated");
        } else {
            println!(
                "Power is {} hp!", some_closure.value(hp)
            );
        }
    }
}
```

Listing 10.4: Using Cacher in the generate_force function to abstract away the caching logic

Instead of saving the closure in a variable directly, we save a new instance of `Cacher` that holds the closure. Then, in each place we want the result, we call the `value` method on the `Cacher` instance. We can call the `value` method as many times as we want, or not call it at all, and the expensive calculation will be run a maximum of once.

CHAPTER 11

# SMART POINTERS

## 11.1 Box

The most straightforward smart pointer is a box, whose type is written `Box<T>`. Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data.

Boxes don't have performance overhead, other than storing their data on the heap instead of on the stack. But they don't have many extra capabilities either. You may use them most often when:

- you have a type whose size can't be known at compile time and you want to use a value of that type in a context that requires an exact size
- you have a large amount of data and you want to transfer ownership but ensure the data won't be copied when you do so
- you want to own a value and you care only that it's a type that implements a particular trait rather than being of a specific type

```
fn main() {
    let b = Box::new(5);
    println!("b = {}", b);
}
```

Listing 11.1: Storing an i32 value on the heap using a box

We define the variable b to have the value of a Box that points to the value 5, which is allocated on the heap. This program will print b = 5; in this case, we can access the data in the box similar to how we would if this data were on the stack. Just like any owned value, when a box goes out of scope, as b does at the end of main, it will be deallocated.

## 11.2 ENABLING RECURSIVE TYPES WITH BOXES

At compile time, Rust needs to know how much space a type takes up. One type whose size can't be known at compile time is a recursive type, where a value can have as part of itself another value of the same type. Because this nesting of values could theoretically continue infinitely, Rust will not know how much space a value of a recursive type needs. However, boxes have a known size, so by inserting a box in a recursive type definition, you can have recursive types.

### 11.2.1 CONS LIST

A cons list is a data structure that comes from the Lisp programming language and its dialects. In Lisp, the cons function (short for "construct function") constructs a new pair from its two arguments, which usually are a single value and another pair. These pairs containing pairs form a list. The cons function concept has made its way into more general functional programming jargon: "to cons x onto y" informally means to construct a new container instance by putting the element x at the start of this new container, followed by the container y.

Each item in a cons list contains two elements: the value of the current item and the next item. The last item in the list contains only a value called Nil without a next item. A cons list is produced by recursively calling the cons function. The canonical name to denote the base case of the recursion is Nil.

Although functional programming languages use cons lists frequently, the cons list isn't a commonly used data structure in Rust. Most of the time when you have a list of items in Rust, Vec<T> is a better choice to use. Other, more complex recursive data types are useful in various situations, but by starting with the cons list, we can explore how boxes let us define a recursive data type without much distraction.

Because a Box<T> is a pointer, Rust always knows how much space a Box<T> needs as **a pointer's size doesn't change based on the amount of data it's pointing to.** This means we can put a Box<T> inside the Cons variant instead of another List value directly. The Box<T> will point to the next List value that will be on the heap rather than inside the Cons variant. Conceptually, we still have a list, created with lists "holding" other lists, but this implementation is now more like placing the items next to one another rather than inside one another.

```rust
enum List {
    Cons(i32, Box<List>),
    Nil,
}

use crate::List::{Cons, Nil};

fn main() {
    let list = Cons(1, Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
}
```

Listing 11.2: Definition of List that uses Box<T> in order to have a known size

Boxes provide only the indirection and heap allocation, they don't have any other special capabilities. They also don't have any performance overhead that these special capabilities incur, so they can be useful in cases like the cons list where the indirection is the only feature we need.

The `Box<T>` type is a smart pointer because it implements the `Deref` trait, which allows `Box<T>` values to be treated like references. When a `Box<T>` value goes out of scope, the heap data that the box is pointing to is cleaned up as well because of the `Drop` trait implementation. Let's explore these two traits in more detail.

## 11.3 SMART POINTERS LIKE REGULAR REFERENCES WITH DEREF TRAIT

Implementing the `Deref` trait allows you to customize the behavior of the dereference operator, * (as opposed to the multiplication or glob operator). By implementing `Deref` in such a way that a smart pointer can be treated like a regular reference, you can write code that operates on references and use that code with smart pointers too.

Let's first look at how the dereference operator works with regular references. Then we'll try to define a custom type that behaves like Box<T> and see why the dereference operator doesn't work like a reference on our newly defined type. We'll explore how implementing the Deref trait makes it possible for smart pointers to work in ways similar to references. Then we'll look at Rust's deref coercion feature and how it lets us work with either references or smart pointers.

```rust
fn main() {
    let x = 5;
    let y = Box::new(x);

    assert_eq!(5, x);
    assert_eq!(5, *y);
}
```

Listing 11.3: Using the dereference operator on a Box<i32>

```rust
fn main() {
    let x = 5;
    let y = MyBox::new(x);

    println!("{:?} {:?} {:?}",x,y,*y);
}


#[derive(Debug)]
struct MyBox<T>(T);

impl<T> MyBox<T> {
    fn new(x: T) -> MyBox<T> {
        MyBox(x)
    }
}

use std::ops::Deref;

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &T {
        &self.0
    }
}
```

Output:

```
5 MyBox(5) 5
```

## 11.4 RUNNING CODE ON CLEANUP WITH THE DROP TRAIT

The second trait important to the smart pointer pattern is `Drop`, which lets you customize what happens when a value is about to go out of scope. You can provide an implementation for the `Drop` trait on any type, and the code you specify can be used to release resources like files or network connections. We're introducing Drop in the context of smart pointers because the functionality of the Drop trait is almost always used when implementing a smart pointer. For example, `Box<T>` customizes Drop to deallocate the space on the heap that the box points to.

In some languages, the programmer must call code to free memory or resources every time they finish using an instance of a smart pointer. If they forget, the system might become overloaded and crash. In Rust, you can specify that a particular bit of code be run whenever a value goes out of scope, and the compiler will insert this code automatically. As a result, you don't need to be careful about placing cleanup code everywhere in a program that an instance of a particular type is finished with, the program still won't leak resources.

Specify the code to run when a value goes out of scope by implementing the `Drop` trait. The `Drop` trait requires you to implement one method named drop that takes a mutable reference to self. To see when Rust calls drop, let's implement drop with `println!` statements for now.

```rust
struct MySmartPointer {
    data: String,
}

impl Drop for MySmartPointer {

    fn drop(&mut self) {
        println!("Dropping CustomSmartPointer with data {} !", self.data);
    }
}

fn main() {

    let c = MySmartPointer {
        data: String::from("my stuff"),
    };

    let d = MyPointer {
        data: String::from("other stuff"),
    };

    println!("MySmartPointers created.");
}
```

# CHAPTER 12

# CONCURRENCY

Concurrent programming means different parts of a program execute independently is very important to take advantage of multiple processors in the current generation of computers. Historically, concurrent programming has been difficult and error prone.

Developers of Rust discovered that the ownership and type systems are a powerful set of tools to help manage memory safety and concurrency problems. By leveraging ownership and type checking, many concurrency errors are compile-time errors in Rust rather than runtime errors. Rather than spending enormous amounts of time trying to reproduce the exact circumstances under which a runtime concurrency bug occurs, in Rust, incorrect code will refuse to compile and present an error explaining the problem. Rust developers have nicknamed this aspect of Rust **fearless concurrency**. Fearless concurrency allows you to write code that is free of subtle bugs and is easy to refactor without introducing new bugs.

Many languages are dogmatic about the solutions they offer for handling concurrent problems. For example, Erlang has elegant functionality for message-passing concurrency but has only obscure ways to share state between threads. Supporting only a subset of possible solutions is a reasonable strategy for higher-level languages, because a higher-level language promises benefits from giving up some control to gain abstractions. However, lower-level languages are expected to provide the solution with the best performance in any given situation and have fewer abstractions over the hardware. Therefore, Rust offers a variety of tools for modeling problems in whatever way is appropriate for your situation and requirements.

## 12.1 THREADS

Many programming languages provide their own special implementation of threads. Programming language provided threads are known as green threads, and languages that use these green threads will execute them in the context of a different number of operating system threads. For this reason, the green-threaded model is called the M:N model: there are M green threads per N operating system threads, where M and N are not necessarily the same number.

```rust
use std::thread;
use std::time::Duration;

fn main() {

    let handle = thread::spawn(|| {
        for i in 1..10 {
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1));
        }
    });

    handle.join().unwrap();

    for i in 1..5 {
        println!("hi number {} from the main thread!", i);
        thread::sleep(Duration::from_millis(1));
```

```
     }
 }

```

Calling `join` on the handle blocks the thread currently running until the thread represented by the handle terminates. Blocking a thread means that thread is prevented from performing work or exiting.

## 12.2 MESSAGE PASSING TO TRANSFER DATA BETWEEN THREADS

An increasingly popular approach to ensuring safe concurrency is message passing, where **threads** or **actors** communicate by sending each other messages containing data. Here's the idea in a slogan from the Go language documentation: ***"Do not communicate by sharing memory; instead, share memory by communicating."***

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx, rx) = mpsc::channel();

    thread::spawn(move || {
         let val = String::from("hi");
         tx.send(val).unwrap();

         // println!("val is {}", val);
         // Uncommenting the above line will result in compile error
         // Once the value has been sent to another thread, that thread could
         // modify or drop it before we try to use the value again.
         // Potentially, the other thread's modifications could
         // cause errors or unexpected results due to inconsistent or nonexistent
         // data. Rust gives an error if we try to compile the code
    });

    let received = rx.recv().unwrap();
    println!("Got: {}", received);
}
```

The receiving end of a channel has two useful methods: `recv` and `try_recv`. We're using `recv`, short for receive, which will block the main thread's execution and wait until a value is sent down the channel. Once a value is sent, `recv` will return it in a `Result<T, E>`. When the sending end of the channel closes, `recv` will return an error to signal that no more values will be coming.

The `try_recv` method doesn't block, but will instead return a `Result<T, E>` immediately: an Ok value holding a message if one is available and an Err value if there aren't any messages this time. Using `try_recv` is useful if this thread has other work to do while waiting for messages: we could write a loop that calls `try_recv` every so often, handles a message if one is available, and otherwise does other work for a little while until checking again.

We've used `recv` in this example for simplicity; we don't have any other work for the main thread to do other than wait for messages, so blocking the main thread is appropriate.

## 12.3 CHANNELS & OWNERSHIP TRANSFERENCE

The ownership rules play a vital role in message sending because they help you write safe, concurrent code. Preventing errors in concurrent programming is the advantage of thinking about ownership throughout your Rust programs.

To demonstrate clearly that the two threads are communicating with each other, see listing 12.1

```rust
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    let (tx, rx) = mpsc::channel();

     thread::spawn(move || {
             let vals = vec![
             String::from("hi"),
             String::from("from"),
             String::from("the"),
             String::from("thread"),
         ];

    for val in vals {
        tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
 }
```

Listing 12.1: Message passing in concurrent program

```rust

use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {

    let (tx, rx) = mpsc::channel();

    let tx1 = mpsc::Sender::clone(&tx);

    thread::spawn(move || {
        let vals = vec![
            String::from("hi"),
            String::from("from"),
            String::from("the"),
            String::from("thread"),
        ];

        for val in vals {
            tx1.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    thread::spawn(move || {
        let vals = vec![
            String::from("more"),
            String::from("messages"),
            String::from("for"),
            String::from("you"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    for received in rx {
        println!("Got: {}", received);
    }
}
```

# 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.