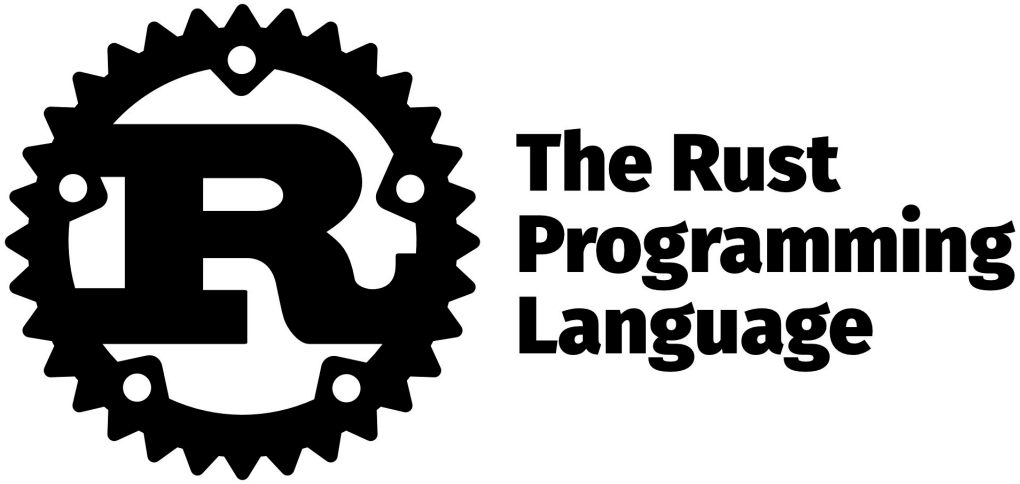
Moving to the Rust programming language

Jaideep Ganguly

Version 1.1.0

Last updated on Saturday 5th September, 2020

Feedback



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 official Rust book is excellent but at nearly 400 pages it is rather daunting to read them in a limited available time. It can be followed even by developers who have less exposure to programming.

The motivation to write this book came from the desire to develop a concise, and yet a comprehensive content, that provides 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. As such, I will not go into explanations of basic terms such as stack, heap, TCP, HTTP, etc., and expect the reader to be familar with them or read them from other books.

For developers proficient in some other language, this book will help them migrate to Rust in just a couple of days. The examples used in the book can be found in github at:
https://github.com/jganguly/rustfun

The ability to write maintainable and correct code is the desire of all developers. My learnings has been shaped through decades of developing large and complex software at Oracle, Microsoft, Amazon and compass.com. Most of the software that I have contributed to are in C/C++, Java, Scala, Kotlin, Python, Go and earlier LISP at MIT. With the advent of Rust 2018 edition, I have moved to Rust completely.

Rust has some fundamental differences with other languages and it is important to understand its **ownership** model thoroughly to understand the concept of borrow chekers and lifetimes. Rust's **closure** is somewhat hard. Since there is an intimate relationship between threads and closures, it is important to master these so as to be able to write concurrent programs fearlessly.

I would like the express my sincere thanks to Himanshi Nagpal for her meticulous review of the book and help make the corrections.

About the author - 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 is Vice President and Head of Compass India R&D Center, and earlier held software engineering director level positions at Amazon and Microsoft.

Revision History

Revision	Date	Author(s)	Description
1.0.1	Sep 03, 2020	Himanshi Nagpal	Listing 3.4 added
1.1.0	Sep 04, 2020	Jaideep Ganguly	Added section on Async/Wait in Concurrency

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. `Cargo` provides automations for your Rust package, i.e., building your package including retrieving "dependencies". 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]
rand = "0.5.5"
```

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**. For example, in listing 2.4, there is a dependency on the external crate `rand`. In the code, there needs to be a declaration to access the contents of the `rand` crate.

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

```

use rand::Rng;

fn main() {
    // we will understand the the let keyword and the :: syntax later
    let number = rand::thread_rng().gen_range(1, 101);
    println!("Hello World {}", number);
}
```

Listing 2.5: Hello World!

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.

In listing 2.5, `println!` is used to print results to the screen. It 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. Using a `!` means that a macro is being called instead of a normal function.

A logical group of code is called a Module. Modules are similar to namespaces in other programming languages. For example, the network module contains networking related functions. Multiple modules are compiled into a unit called `crate`. Rust programs may contain a binary crate or a library crate. A **binary crate** is an executable project that has a `main()` method. **A library crate** is a group of components that can be reused in other projects. Unlike a binary crate, a library crate does not have an entry point main() method.

2.3 BUILD & RUN WITH CARGO

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

```
1 cargo build # build only
2 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`.

2.5 PUBLISHING ON CRATES.IO

Third-party crates can be downloaded using cargo from `crates.io`. You can also publish to `crates.io` for which you'll need an account on `crates.io` to acquire an API token. To do so, visit the home page of `crates.io` and log in via a GitHub account. Next run the command `cargo login Your-API-Token`. Your API token will now be stored in `~/.cargo/credentials.toml`.

Take care when publishing a crate, because a publish is permanent. The version can never be overwritten, and the code cannot be deleted. There is no limit to the number of versions which can be published. You can check which files are included with the following command.

```
cargo package --list
```

and then publish.

```
cargo publish
```

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, control flow and how to organize code 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 have 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 some semantic sense of that value to future maintainers of the code.

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

Example code, throughout this book, are written in separate functions and invoked through the `main` function as shown below. In subsequent examples, the main function will not be stated explicitly, unless it is essential to understanding, as it simply involves calling the specific functions.

```
1 mod basic;
2 fn main() {
3     basic::concept();
4 }
```

Listing 3.1: main.rs for functions explaining basic concepts

```
1 pub fn concept() {
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);
```

```
}
```

Listing 3.2: Variables & Mutability

Output:

```
x=5 y=7 MAX_POINTS=10000000
```

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.

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

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

    println!("{} {}", x,spaces);
}
```

Listing 3.3: Shadowing

Output:

```
12 3
```

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:

```
let input: i32 = input.trim().parse().expect("Not a number);
```

Listing 3.4: Type Annotation

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

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

In this case `input` is read from the console and the type is unknown and so we will have to annotate the type by specifying `i32`.

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 are on a 64-bit architecture and 32 bits if you are on a 32-bit architecture.

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 would 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 are 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 are 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 is 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.

```
20 pub fn data_types() {
21     let x = 3; // Type inference will assign x to be of type i32
22     let num: i32 = "142".parse().expect("Not a number!");
23     let y: f64 = 300.43;
24     let c = 'A';
25     let s = "hello";
26     let t = true;
27
28     println!("{} {} {} {} {} {}", x, num, y, c, s, t);
29 }
```

Listing 3.5: Data types

Output:

```
1 3 142 300.43 A hello true
```

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 access the individual values in a tuple, we can use pattern matching to destructure a tuple value. The `_` is a placeholder, as the variable is not required.

```
31 pub fn tuple_example() {
32     let tup: (i32, f64, u8) = (500, 6.4, 1);
33     let (_, y, _) = tup;
34     println!("The value of y is: {}", y);
35 }
```

Listing 3.6: Data types

Output:

```
1 The value of y is: 6.4
```

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.

```
37 pub fn array_example() {
38     let a = [1, 2, 3, 4, 5];
39     let b = [3; 5];                 // 5 is the size of the array
40     let c: [i32; 5] = [1,2,3,4,5]; // type is stated with semicolon
41     println!("a={:?}\nb={:?}\nc={:?}", a,b,c);
42 }
```

Listing 3.7: Data types

Output:

```
1 a=[1, 2, 3, 4, 5]
2 b=[3, 3, 3, 3, 3]
3 c=[1, 2, 3, 4, 5]
```

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.

`{...}` surrounds all formatting directives. `:` separates the name or ordinal of the thing being formatted (which in this case is omitted, and thus means "the next thing") from the formatting

options. The `?` is a formatting option that triggers the use of the `std::fmt::Debug` implementation of the thing being formatted, as opposed to the default Display trait (we will discuss trait in a later chapter), or one of the other traits (like UpperHex or Octal). Thus, `{:?}` formats the "next" value passed to a formatting macro, and supports anything that implements Debug.

3.6 FUNCTIONS

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

```
44 pub fn function_example(a: i32, b:i32) -> i32 {
45     let c = a + b;
46     c
47 }
```

Listing 3.8: Data types

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 will need to include `//` on each line. Alternately, you can use block comments using the familiar /* */ style.

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.

```
49 pub fn ctrlflow_example() {
50     let a = 12;
51
52     if a < 10 {
53         println!("low number");
54     }
55     else if (a > 10) && (a < 20) {
56         println!("moderate number");
57     }
58     else {
59         println!("high number");
60     }
```

```
}
```

Listing 3.9: Control Flow

Rust has the `match` keyword which is similar to `switch` in other languages. We will study that in the context of `enum`.

3.8 LOOPS

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

```
pub fn loop_example() {
    let arr = [10, 20, 30, 40, 50];

    for elem in arr.iter() {
        println!("Element is: {}", elem);
    }

    let mut i = 0;
    loop {
        println!("{} {}", i, arr[i]);
        i = i + 1;
        if (i == arr.len()) {
            break;
        }
    }
}
```

Listing 3.10: Control Flow

Output:

```
Element is: 10
Element is: 20
Element is: 30
Element is: 40
Element is: 50
0 10
1 20
2 30
3 40
4 50
```

3.9 GENERICS

To eliminate duplication of code, we sometimes need to resort to what is referred to as **generic programming**, a style of programming in which algorithms are written in terms of types that

are specified later. These are instantiated later when specific types are provided as parameters. Rust supports 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 3.11: Generic example

3.10 MODULES

To keep the code manageable, you will need to organize the code in separate files or directories. Following is an example of how to achieve that with `mod` and `use` keywords.

```
// mod_org is the name of the folder; it has a file named mod.rs
mod mod_org;
// mod_example.rs file is in "src" directory; contains mod named test
mod mod_example;
use mod_example::mod_test;

fn main() {
   // mod example;  organize code into modules
    mod_org::f1();
    mod_example::mod_test::f2();
}
```

Listing 3.12: Using mod

```
// pub key word is required or else
// f1 is private and not accessible outside the mod
pub fn f1() {
    println!("printing from mod_org::f1");
}
```

Listing 3.13: Function defined in a file in a subfolder

```rust
pub mod mod_test {
    pub fn f2() {
        println!("printing from mod_test::f2");
    }
}
```

Listing 3.14: Functon in a file in "src" directory

3.11 RUNNING TESTS

We often hear about languages with strict compilers, such as Haskell and Rust that "if the code compiles, it works." But this saying is not universally true. A project may compile but may not do anything. While building a complete project, we will need to write unit tests to check that the code compiles and has the behavior we want.

```rust
#[cfg(test)] // Code is not compiled unless running cargo test
mod my_tests {
    #[test]
    fn basic_test() {
        assert_eq!(2==2, true);
    }
}
```

Listing 3.15: Test code

Output:

```
    Finished test [unoptimized + debuginfo] target(s) in 0.35s
    Running target/debug/deps/rust-12e404aeef2cf895

running 1 test
test my_tests::basic_test ... ok

test result: ok. 1 passed; 0 failed; 0 ignored; 0 measured; 0 filtered
   out
```

Tests are run simply by running `cargo test`.

3.12 DOCUMENTATION

Rust has the tool `rustdoc` to generate rust documentation. Comments with `///` will be picked up in the documentation generated. Documentation is generated as follows:

```
rustdoc src/somefile.rs
cargo doc --open
```

`rustdoc` will generate Rust documentation only for public entities.

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 is 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.

```
pub fn example1() {
    {
        let x = 1;
        println!("x: {}", x);
    }

    // println!("x: {}", x); // ERROR
}
```

Listing 4.1: Ownership ends with scope

Compile Error:

```
error[E0425]: cannot find value 'x' in this scope
--> src/fn_04_own.rs:7:23
  |
7 |     println!("x: {}", x); // ERROR
  |                       ^ not found in this scope
```

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.

```
10 pub fn example2() {
11     let a = vec![1, 2, 3];  // a growable array literal
12     let b = a;              // a can no longer be used beyond this line
13     println!("b: {:?}", b);
14     // println!("a: {:?}", a); // ERROR
15 }
```

Listing 4.2: Ownership ends with move

Compile Error:

```
1  error[E0382]: borrow of moved value: 'a'
2  --> src/fn_04_own.rs:14:22
3     |
4  11 |     let a = vec![1, 2, 3];  // a growable array literal
5     |         - move occurs because 'a' has type 'std::vec::Vec<i32>',
6     |           which does not implement the 'Copy' trait
7  12 |     let b = a;              // a can no longer be used
8     |             - value moved here
9  13 |     println!("a: {:?}", b);
10 14 |     println!("a: {:?}", a); // ERROR
11    |                          ^ value borrowed here after move
```

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

```
17 pub fn example3() {
18     let v = vec![1,2,3];
19     let s = sum(v);
20     // println!("sum of {:?}: {}", v, s); // ERROR
21 }
22
23 fn sum(vector: Vec<i32>) -> i32 {
24     let mut sum = 0;
25
26     for item in vector {
27         sum = sum + item;
28     }
29
30     sum
31 }
```

Listing 4.3: Ownership ends with move

Compile error:

```
error[E0382]: borrow of moved value: 'v'
--> src/fn_04_own.rs:20:30
   |
18 |     let v = vec![1,2,3];
   |         - move occurs because 'v' has type 'std::vec::Vec<i32>',
   |           which does not implement the 'Copy' trait
19 |     let s = sum(v);
   |                 - value moved here
20 |     println!("sum of {:?}: {}", v, s); // ERROR
   |                                  ^ value borrowed here after move
```

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

```
pub fn example4(x: i32) -> Vec<i32> {
    let result = vec![x, x+1, x+2, x+3, x+4]; // allocated on heap
    result
}
```

Listing 4.4: Ownership ends with move

```
let x = fn_04_own::example4(10);
println!("series: {:?}",x);
```

Output:

```
series: [10, 11, 12, 13, 14]
```

Let us review some more examples, in particular note the comments within the code. First we list the functions called from `main()` followed by the listings of the functions.

```
    fn_04_own::example5(&mut x);  // x is passed by reference
    println!("{}\n",x);

    let mut x: i32 = 42;
    let y = fn_04_own::example6(&mut x);
    println!("Returned value {:?}\n",y);

    let mut s = "Hello";
    let s = fn_04_own::example7(&mut s);
    println!("Returned slice {:?}\n",s);
```

```
pub fn example5(x: &mut i32){
    *x = *x + 1; // '*' is dereference operator to get value (like C)
}
```

Listing 4.5: example5

```
pub fn example6(x: &mut i32) -> i32 {
    println!("Inside function {}",x);
    *x + 1
```

```
45 }
```

Listing 4.6: example6

```
49 pub fn example7(s: &str) -> &str{
50     println!("Inside function {}",s);
51     &s[1..3]
52 }
```

Listing 4.7: example7

Output:

```
1 43
2
3 Inside function 42
4 Returned value 43
5
6 Inside function Hello
7 Returned slice "el"
```

The macros `print!`, `println!`, and `format!` are special cases and implicitly take a reference to any arguments to be formatted. These macros do not behave as normal functions for reasons of convenience, the fact that **they take references silently** is part of that difference. So, `println!("{}n",x);` produces the same output as `println!("{}n",&x);`

4.3 COPY

During reassignment, for variables in the stack, instead of moving the values owned by the variables, their values are copied. While the code in listing 4.2 did not work, the following code will work correctly.

```
52 pub fn copy_trait_example() {
53     let a = 42;
54     let b = 94;
55     let c = a + b;
56     println!("The sum of {} and {} is {}", a, b, c); // NO ERROR
57 }
```

Listing 4.8: Copy trait

Output:

```
1 The sum of 42 and 94 is 136
```

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 is 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 will not 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 will 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 a tuple `(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:

```
1 let s1 = String::from("hello");
2 let s2 = s1.clone();
3
4 println!("s1 = {}, s2 = {}", s1, s2);
```

Listing 4.9: 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.

```
60 pub fn struct_copy_example() {
61     #[derive(Debug,Clone,Copy)]
62     struct Person {
63         age: i8
64     }
65
66     let alice = Person { age: 42 };
67     let bob = alice;
68
69     println!("alice: {:?}\nbob: {:?}", alice, bob);
70 }
```

Listing 4.10: Struct and the Copy trait

Output:

```
alice: Person { age: 42 }
bob: Person { age: 42 }
```

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.

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

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

Listing 4.11: Struct and the Copy trait

Invoking the function will result in:

Output:

```
The length of 'hello' is 5.
```

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.

```
pub fn mut_ref_example() {
    let mut s = String::from("Hello");
    change(&mut s);
    println!("{}",s);
}

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

Listing 4.12: Mutable reference

Output:

```
Hello world!
```

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

```
pub fn mut_ref_restrict() {
    let mut s = String::from("hello");

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

    // ERROR: will not compile
    // cannot borrow 's' as mutable more than once at a time
    println!("{}, {}", r1, r2);
}
```

Listing 4.13: Mutable reference

Compile error:

```
error[E0499]: cannot borrow 's' as mutable more than once at a time
--> src/fn_04_own.rs:88:14
   |
98 |     let r1 = &mut s;
   |              ------ first mutable borrow occurs here
99 |     let r2 = &mut s;
   |              ^^^^^^ second mutable borrow occurs here
```

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

```
pub fn mut_ref_restrict2() {
    let mut s = String::from("hello");

    // ERROR: will not compile
    // cannot borrow 's' as mutable because it is also borrowed as
   immutable.
    let r1 = &s;     // no problem
    let r2 = &mut s; // problem

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

Listing 4.14: Mutable reference

Compile error:

```
error[E0502]: cannot borrow 's' as mutable because it is also borrowed
   as immutable
--> src/fn_04_own.rs:101:14
    |
```

```
111 |     let r1 = &s;     // no problem
    |              -- immutable borrow occurs here
112 |     let r2 = &mut s; // problem
    |              ^^^^^^ mutable borrow occurs here
```

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

```
108
109     // ERROR: will not compile
110     // cannot borrow 's' as mutable because it is also borrowed as
       immutable.
```

Listing 4.15: 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 is 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.

```
124 fn dangle() -> &String { // ERROR: will not compile
125     let s = String::from("hello");
126     &s
127 }
```

Listing 4.16: 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 will generate a comile time error.

The solution here is to simply return the String directly.

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

```
126     s
127 }
```

Listing 4.17: main

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.**

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

Listing 4.18: 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.

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

Listing 4.19: String literals are slices

The type of `s` here is `&str`. It is 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:

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

Listing 4.20: 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 will want a function to return a borrowed value. Consider the following code which will fail to compile.

```
136 pub fn lifetime_example(x: &str, y: &str) -> &str { // Error
137     if x.bytes().len() > y.bytes().len() {
138         x
139     } else {
140         y
```

```
    }
}
```

Listing 4.21: Lifetime

Compile error:

```
error[E0106]: missing lifetime specifier
--> src/fn_04_own.rs:125:46
    |
136 | pub fn lifetime_example(x: &str, y: &str) -> &str { // Error
    |                            ----     ----   ^ expected named
    |                                              lifetime parameter
```

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.

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

Listing 4.22: Lifetime

Output:

```
Alice
```

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:

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

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`.

CHAPTER 5

STRUCT

A struct is a user-defined type that we can use to store one or more variables of different types. A struct allows us to group related code together and model our application after entities in the real world. There are three types of structures (struct) that can be created using the `struct` keyword:

- The classic structs as in the C language.
- Tuple structs, which are, basically, named tuples.
- Unit structs or field-less that are useful for generics.

5.1 DEFINING A STRUCT

Since `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. But unlike with tuples, you have to name each piece of data so that it is clear what the values mean.

```
1 struct User {
2     username: String,
3     email: String,
4     sign_in_count: u64,
5     active: bool
6 }
```

Listing 5.1: struct

5.2 INSTANTIATING STRUCTS

To use a struct after we have 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 do not have to specify the fields in the same order in which we declared them in the struct.

```
1 fn main() {
2     let user = User {
3         email: String::from("john@company.com"),
4         username: String::from("john"),
5         active: true,
6         login_count: 5
```

```
    };
}
```

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 `user.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.

```
user.email = String::from("doe@company.com");
```

Listing 5.3: Mutable struct

Note that the entire instance must be mutable as Rust does not 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.

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

Listing 5.4: Function using field init shorthand

If the function parameters are same sa the struct fields, the function parameter values are mapped to the struct attribute values and this is a convenient shorthand.

5.2.2 STRUCT UPDATE

It is 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.

```
let user_another = User {
    email: String::from("mary@company.com"),
    username: String::from("mary"),
    ..user
};
```

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 do not 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.

```
struct Color(u8, u8, u8);
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 STRUCT

You can also define structs that do not 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 do not have any data that you want to store in the type itself and is particularly relevant for generics.** We will discuss traits in the following chapter chapter.

```
pub struct Function;    // unit struct, has no data

impl Function {
    pub fn say_hello(&self) {
        println!("{:?}", "Hello" );
    }
}
```

Listing 5.7: Unit struct

```
let f = st::Function{};
f.say_hello();
```

Listing 5.8: Invoking unit struct function

5.5 METHODS

Methods are similar to functions. they are declared with the `fn` keyword and their name, they can have parameters and a return value. **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 struct printable with 'fmt::Debug'.
#[derive(Debug)]
struct Rect {
    width: u32,
    height: u32,
}

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

fn main() {
    let rect = Rect {
        width: 30,
        height: 50,
    };

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

Listing 5.9: Implementing a method

5.6 OWNERSHIP OF STRUCT DATA

It is 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.

Notice the usage of lifetime `'a` in the following example. Also, note the usage of the `pub` keyword before `struct` and their attributes. These are required as the struct resides in a separate `mod st`. Without `pub`, you will get a compile error.

Methods, of course can take additional parameters as shown in the example below.

```
pub struct Rect<'a> {
    pub id: &'a str,
    pub width: i32,
    pub length: i32
}

impl<'a> Rect<'a> {
    pub fn area(&self) -> i32 {
        self.width * self.length
```

```
18      }
19
20      pub fn volume(&self, height: i32) -> i32 {
21          self.area()*height
22      }
23  }
```

Listing 5.10: Lifetime in struct

Invoking:

```
1  let r = st::Rect {id:"abc",width:10, length:20};
2  println!("Area = {}",r.area());
3  println!("Volume = {}",r.volume(10));
```

Output:

```
1  Area = 200
2  Volume = 2000
```

Using methods helps organize the code, all things we can do with the instance of a type will be collated with the `struct`. Passing the parameter `&self` enables the method to access all the properties of the structure inside the method.

Rust does not 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:

```
1  p1.distance(&p2);
2  (&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 do not take self as a parameter. These are called associated functions because they are associated with the struct. They are functions, not methods, because they do not have an instance of the struct to work with. An example is 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 5.11: Associated Function

CHAPTER 6

TRAIT

In Rust, a `trait` can be thought of as an equivalent of a Java interface that is used to achieve abstraction. A struct can implement a trait using the `impl` keyword and specify its own definition of the trait's methods.

6.1 INTRODUCTION TO TRAITS

The best way to get introduced to `trait` is to study the following example and you will readily understand how abstractions are introduced and implemented.

```
pub trait Animal {
    fn eat(&self) {
        println!("I eat grass");
    }
}

pub struct Herbivore;

impl Animal for Herbivore{
    fn eat(&self) {
        println!("I eat plants");
    }
}

pub struct Carnivore;

impl Animal for Carnivore {
    fn eat(&self) {
        println!("I eat meat");
    }
}
```

Listing 6.1: Trait implementations

Invoking:

```
use tra::Animal;

let h = tra::Herbivore;
h.eat();

```

```
let c = tra::Carnivore;
c.eat();
```

Listing 6.2: Invoking trait definitions

Output:

```
I eat plants
I eat meat
```

Notice the default implementation of `eat` in `Animal`. If `Animal` is not implemented for `Herbivore`, then the default implementation of the `trait` is invoked and the output will be `I eat grass`. The structs implement the same trait but exhibit different behaviors because of their respective unique implementations.

6.2 TRAIT BOUND

Consider the following listing.

```
pub trait Activity {
    fn fly(&self);
}

#[derive(Debug)]
pub struct Eagle;

impl Activity for Eagle {
    fn fly(&self) {
        println!("{:?} is flying",&self);
    }
}

pub fn activity<T: Activity + std::fmt::Debug>(bird: T) {
    println!("I fly as an {:?}",bird);
}

pub struct Hen;
```

Listing 6.3: Trait implementations

Invoking:

```
use tra::Activity;
let eagle = tra::Eagle;
eagle.fly();
tra::activity(eagle);
```

Listing 6.4: Invoking trait definitions

Output:

```
Eagle is flying
I fly as an Eagle
```

But adding the following line will result in a compile error. This is because the `struct Hen` does not implement `trait Activity`.

```
tra::activity(hen);
```

Listing 6.5: Invoking trait definitions

Thus while the function `activity` takes a generic `T` as an argument, the generic `T` must implement `trait Activity`. **Trait bounds** allow a a function to only **accept types** that **implement** a certain **trait**.

Any invocation of the function with an instance of a struct that does not implement the trait will result in a compile error. Sun a function is said to be **trait bound**.

6.3 TRAIT OBJECT

Trait objects behave more like traditional objects, they contain both **data** and **behavior**. In trait objects, the data is referenced through a pointer to the data that is actually stored in the heap. So, even if the size of the data in the heap changes, the size of the pointer in the trait object remains the same and this makes it much more predictable and manageable while dealing with memory. This automatically implies that you cannot add data to a trait object. This is key to understanding trait objects, we are pointing to data at one specific point in time. The behavior of a trait object comes from a traditional trait that we studied earlier.

Let's understand trait object through an example. Note the use of `Box` operator to create objects on the heap. The size of a trait is not known at compile-time. Therefore, traits have to be wrapped inside a Box when creating a vector trait object. We will learn more about the `Box` operator in a later chapter. A trait object is an object that can contain objects of different types at the same time (e.g., a vector). The `dyn` keyword is used when declaring a trait object. So,

- `Box<Trait>` becomes `Box<dyn Trait>`
- `&Trait` and `&mut` Trait become `&dyn Trait` and `&mut dyn Trait`

```
#[derive(Debug)]
pub struct Horse;

#[derive(Debug)]
pub struct Deer;

#[derive(Debug)]
pub struct Tiger;

#[derive(Debug)]
pub struct Duck;

```

```
pub trait Sound {
    fn sound(&self);
}

impl Sound for Horse {
    fn sound(&self) {
        println!("{:?} neighs",&self)
    }
}

impl Sound for Deer {
    fn sound(&self) {
        println!("{:?} barks",&self)
    }
}

impl Sound for Tiger {
    fn sound(&self) {
        println!("{:?} roars",&self)
    }
}

impl Sound for Duck {
    fn sound(&self) {
        println!("{:?} quacks",&self)
    }
}

pub struct SoundBook {
    pub sounds: Vec<Box<dyn Sound>>
}

impl SoundBook {

    pub fn run(&self) {
        for s in self.sounds.iter() {
            s.sound();
        }
    }
}
```

Listing 6.6: Trait object

Invoking:

```
    let sound_book = tra::SoundBook {
        sounds: vec! [
            Box::new(tra::Horse{}),
```

```
4             Box::new(tra::Deer{}),
5             Box::new(tra::Tiger{}),
6             Box::new(tra::Duck{})
7         ]
8 };
9
10 sound_book.run();
```

Listing 6.7: Invoking trait object

Output:

```
1 Horse neighs
2 Deer barks
3 Tiger roars
4 Duck quacks
```

CHAPTER 7

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**. We will next look at how pattern matching in the `match` expression makes it easy to run different code for different values of an enum. Finally, we will study how the `if let` construct, a convenient and concise idiom, that is available to us to handle enums in our code. Rust's enums are most similar to algebraic data types in functional languages, such as F#, OCaml, and Haskell.

7.1 DEFINING AN ENUM

The following is an example of a basic enum.

```
#[derive(Debug)]
pub enum Gender {
    Male,
    Female
}
```

Listing 7.1: Enum

Invoking:

```
use enu::Gender;
let male = enu::Gender::Male;
let female = enu::Gender::Female;
println!("{:?}",male);
println!("{:?}",female);
```

Listing 7.2: Invoking enum

Output:

```
Male
Female
```

7.2 STRUCT & ENUM

Rust enums can contain a context, it can be a different one for each variant of the enum. We can put data directly into each `enum` variant.

```
#[derive(Debug)]
pub struct MyBlack {
    pub name: String,
    pub rgb: (u8,u8,u8)
}

#[derive(Debug)]
pub enum Color {
    Black(MyBlack),
    White(u8,u8,u8)
}
```

Listing 7.3: Enum

Invoking:

```
let my_black = enu::MyBlack {
    name: String::from("my black"),
    rgb: (10,10,10)
};
let black = enu::Color::Black(my_black);
let white = enu::Color::White(255,255,255);
println!("{:?}",black);
println!("{:?}",white);
```

Listing 7.4: Enum with context

Output:

```
Black(MyBlack { name: "my black", rgb: (10, 10, 10) })
White(255, 255, 255)
```

As in the case of struct, we can also define methods on an enum.

7.3 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 indicate null, Rust does not support null
}
```

Listing 7.5: Option enum

Here, the type `T` represents value of any type. **Rust does not support the null keyword.** The value `None`, in the enum `Option`, can be used by a function to return a null value. If there is data to return, the function can return `Some(data)`. To get the `T` value out of `Option<T>`, this 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 y = x.unwrap();  // unwraps and gets the value
```

Listing 7.6: Option examples

7.4 MATCH STATEMENT

Match does what switch does in other languages such as C and Java. The `match` statement can be used to compare values stored in an `enum`.

```rust
    };
    let black = fn_07_enu::Color::Black(my_black);
    let white = fn_07_enu::Color::White(255,255,255);
    println!("{:?}",black);
    println!("{:?}",white);
    }

    let some_u8_value = 4u8;
    match some_u8_value {
```

Listing 7.7: Match example

Matches in Rust are exhaustive. We must exhaust every last possibility in order for the code to be valid and compile. 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
        3 => println!("Three"),
        5 => println!("Five"),
        7 => println!("Seven"),
        9 => println!("Nine"),
        _ => (),
    }

    fn_07_enu::match_example(4);
```

Listing 7.8: Placeholder in match

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

```
25 pub fn match_example(number: i32) {
26
27     match is_even(number) {
28         Some(data) => {
29             if data == true {
30                 println!("Even number");
31             }
32         },
33         None => {
34             println!("Not an even number");
35         }
36     }
37 }
38
39 fn is_even(number:i32) -> Option<bool> {
40     if number%2 == 0 {
41         Some(true)
42     } else {
43         None
44     }
45 }
```

Listing 7.9: match with Option

Invoking

```
1 enu::match_example(4);
```

Output:

```
1 Even number
```

7.5 IF LET STATEMENT

The `if let` syntax of Rust makes the code concise when we are interested in only one of the cases.

```
1 fn main() {
2     let some_u8_value = Some(0u8);
3     if let Some(3) = some_u8_value {
4         println!("three");
5     }
6 }
```

Listing 7.10: if let example

CHAPTER 8

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 is a particular implementation of the more general data structure called a map.

8.1 VECTOR

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

- A Vector is a homogeneous collection.
- A Vector can grow or shrink at runtime.
- 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 create a vector using `Vec` as follows:

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

Listing 8.1: Vector

Or with a macro:

```
let v = vec![];
```

Listing 8.2: Create a vector using vec! macro

Values in the vector are accessed using the index, e.g., `v[1]` .

Let us create a vector and print its length.

```
let mut v = vec!["Hello","how","are","you"];
println!("{}",v.len());
```

Listing 8.3: Length of vector

Output:

```
4
```

We will now add another word to the vector.

```
v.push("today");
println!("{:?}",v);
```

Listing 8.4: Push

Output:

```
["Hello", "how", "are", "you", "today"]
```

We can use pop to remove the added word.

```
v.pop();
println!("{:?}",v);
```

Listing 8.5: Pop

Output:

```
["Hello", "how", "are", "you"]
```

A word can also be inserted into a specific index location.

```
v.insert(4, "sir");
println!("{:?}",v);
```

Listing 8.6: Insert

Output:

```
["Hello", "how", "are", "you", "sir"]
```

Remove a value by referring to its index.

```
v.remove(0);
println!("{:?}",v);
```

Listing 8.7: Remove by index

Output:

```
["how", "are", "you", "sir"]
```

It is also possible to remove a specific value.

```
let index = v.iter().position(|x| x == &"sir" ).unwrap();
v.remove(index);
println!("{:?}",v);
```

Listing 8.8: Remove by value

Output:

```
["how", "are", "you"]
```

Iterate over a vector:

```
for i in &v {
    println!("{}", i);
}
```

Listing 8.9: Iterate

Output:

```
how
are
you
```

8.2 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. The hash map will only contain one key/value pair.

In the following example, we create a HashMap and then manipulate its content.

```
use std::collections::HashMap;
let mut hm: HashMap<String,String> = HashMap::new();
hm.insert("MA".to_string(),"Massachusetts".to_string());
hm.insert("NY".to_string(),"New York".to_string());
hm.insert("CA".to_string(),"California".to_string());

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

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

println!("{:?}", hm.len());
```

Listing 8.10: Insert and remove elements in a HashMap

Output:

```
key: NY val: New York
```

```
key: MA val: Massachusetts
key: CA val: California

key: NY val: New York
key: MA val: Massachusetts
2
```

Note that you cannot update a HashMap using `hm["MA"]="MASSACHUSETTS".to_string();` This will generate a compile error. This is because Indexing immutably and indexing mutably are provided by two different traits: `Index` and `IndexMut`, respectively. Currently, `HashMap` does not implement `IndexMut`, while `Vec` does. We can, of course, remove a key and then insert the same key with the updated value. Alternately, we can update as follows:

```
*hm.get_mut("MA").unwrap() = "MASSACHUSETTS".to_string();
```

Listing 8.11: Update a hashmap

8.3 HASHSET

A HashSet's unique feature is that it is guaranteed not to 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. Consider a HashSet to be a HashMap where we just care about the keys (HashSet<T> is, in actuality, just a wrapper around HashMap<T, ()>).

Following is an example code to create a `HashSet`, insert and remove values.

```
use std::collections::HashSet;
let mut hs: HashSet<String> = HashSet::new();
hs.insert("Tennis".to_string());
hs.insert("Tennis".to_string());
hs.insert("Soccer".to_string());
hs.insert("Badminton".to_string());

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

hs.remove("Badminton");
    for val in hs.iter() {
        println!("val: {}", val);
    }
}

println!("{:?}", hs.len());
```

Listing 8.12: HashSet example

```
val: Soccer
val: Badminton
val: Tennis

val: Soccer
val: Tennis
2
```

Note that "Tennis" has been inserted twice and yet the `HashSet` has only one occurrence of "Tennis".

CHAPTER 9

ERROR HANDLING

Errors are fairly routine in software. Luckily, Rust has multiple mechanisms for handling situations when things go 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 will discover errors and handle them appropriately before you have deployed your code to production.

In Rust, errors are 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.

Most languages do not distinguish between these two kinds of errors and handle both in the same way, using mechanisms such as exceptions. 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. Below is an example:

```
let x = 100;
    if (x > 10) {
        panic!("I am panicking, can't proceed any further");
    }
    println!("I won't reach this line!");
}
```

Listing 9.1: An example of panic!

9.1 RECOVERABLE ERRORS

The enum `Result <T,E>` is used to handle recoverable errors. It has two enumerations - `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 9.2: enum Result

```
use std::fs::File;
let f = File::open("mypicture.jpg");   // file does not exist
match f {
    Ok(f)=> {
        println!("file found {:?}",f);
    },
    Err(e)=> {
        println!("file not found \n{:?}",e);   //handled error
    }
}
println!("I will print this");
```

Listing 9.3: Recoverable error

Output:

```
Os { code: 2, kind: NotFound, message: "No such file or directory" }
I will print this
```

`Result<T,E>` implements the `unwrap()` function that returns the actual result when an operation succeeds. It returns a panic with a default error message if an operation fails. The `expect()` function, on the other hand, can return a custom error message in case of a panic.

```
let f = File::open("somefile.txt").expect("File not found!");
```

Listing 9.4: Example of expect

CHAPTER 10

INPUT & OUTPUT

In this chapter we will review how to read from and write to standard input, i.e., the keyboard as well as read from file input and write to file output.

10.1 STANDARD I/O - READ & WRITE

Rust's standard library features for input and output are organized around two traits.

We will need to `use` the following traits in the following functions.

```
use std::io::Write;
use std::io::Read;
use std::fs::OpenOptions;
use std::fs;
```

Listing 10.1: Modules required

The following example reads values from the standard input.

```
pub fn std_inp() {
    let mut line = String::new();
    println!("Please enter your name:");
    let nb = std::io::stdin().read_line(&mut line).unwrap();
    println!("Hi {}", line);
    println!("# of bytes read , {}", nb);
}
```

Listing 10.2: Reading from standard input

The following example writes to the standard output.

```
pub fn std_out() {
    let b1 = std::io::stdout()
        .write("Hi ".as_bytes()).unwrap();
    let b2 = std::io::stdout()
        .write(String::from("There\n").as_bytes()).unwrap();
    std::io::stdout().
        write(format!("#bytes written {}",(b1+b2))
            .as_bytes()).unwrap();
}
```

Listing 10.3: Writing to standard output

10.2 COMMAND LINE ARGS

Command line parameters can be used to pass values to the main() function. The `std::env::args()` returns the command line arguments. The following example writes to the standard output.

```
pub fn cl_arg() {
    let cmd_line = std::env::args();
    println!("# of command line arguments:{}",cmd_line.len());
    for arg in cmd_line {
        println!("{}",arg);
    }
}
```

Listing 10.4: Command Line Arguments

10.3 FILE I/O - READ & WRITE

Following is an example code to read from a file.

```
pub fn file_read(filename: &str){
    let mut file = std::fs::File::open(filename).unwrap();
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();
    print!("{}", contents);
}
```

Listing 10.5: Read from a file

Following is an example code to write to a file.

```
pub fn file_write(filename: &str, s: &str) {
    let mut file = std::fs::File::create(filename)
        .expect("Create failed");
    file.write_all(s.as_bytes())
        .expect("write failed");
    println!("Write completed" );
}
```

Listing 10.6: Write to a file

10.4 APPEND TO A FILE

Following is an example code to append to a file.

```
pub fn file_append(filename: &str, s: &str) {
    let mut file = OpenOptions::new()
        .append(true).open(filename)
        .expect("Failed to open file");
    file.write_all(s.as_bytes()).expect("write failure");
```

```
    println!("Appended file {}",filename);
}
```

Listing 10.7: Append to a file

10.5 COPY A FILE

Following is an example code to copy file.

```
pub fn file_copy(src: &str, des: &str) {
    let mut file_inp = std::fs::File::open(src).unwrap();
    let mut file_out = std::fs::File::create(des).unwrap();
    let mut buffer = [0u8; 4096];
    loop {
        let nbytes = file_inp.read(&mut buffer).unwrap();
        file_out.write(&buffer[..nbytes]).unwrap();
        if nbytes < buffer.len() {
            break;
        }
    }
}
```

Listing 10.8: Copy a file

10.6 DELETE A FILE

Following is an example code to delete a file.

```
pub fn file_delete(filename: &str) {
    fs::remove_file(filename).expect("Unable to delete file");
    println!("Deleted file {}",filename);
```

Listing 10.9: Delete a file

CHAPTER 11

CLOSURES

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

11.1 CLOSURES

Rust's closures are anonymous functions that can be saved in a variable or can be passed 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. There is an important differentiation between closures and functions. **Unlike functions, closures can capture values from the scope in which they are defined**. 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 users of the function agree on what types of values a function uses and returns. But closures are not exposed through interfaces. They are stored in variables and and the variables are used withing its scope.

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.

Below is an example of a closure.

```
let some_closure = |number| {
    println!("calculating ...");
    thread::sleep(Duration::from_secs(3));
    number + 1
};
```

Listing 11.1: 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 is similar to the closure definitions in Smalltalk and Ruby languages. This closure has one parameter named `number`. 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 are using

a closure because we want to define the code to call at one location and call it at a later location. The first time we call `some_closure` with the `u32` 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.

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 required.

```
let some_closure = |number: u32| -> u32 {
    println!("calculating ...");
    thread::sleep(Duration::from_secs(3));
    number + 1
};
```

Listing 11.2: Closure example

Consider the following example:

```
pub fn closure_example1(x: i32, y: i32) {
    let add = |x,y| {
        x + y
    };
    println!("Closure {:?}", add(x,y));
}
```

Listing 11.3: Closure example 1

Invoking:

```
fn_11_clo::closure_example1(3,4);
```

Output:

```
Closure 7
```

In the following example, notice that the variable y is not in the parameter list of `add` and it it is captured in the closure.

```
pub fn closure_example2(x:i32) {
    let y = 3;
    let add = |x| {
        x + y
    };
    println!("Closure {:?}", add(x));
}
```

Listing 11.4: Closure example 2

Invoking:

```
    fn_11_clo::closure_example2(4);
```

Output:

```
    Closure 7
```

In the following example, a closure is passed to another function. Notice that to be able to pass around closures, we need to use one of the `FnOnce`, `FnMut`, or `Fn` traits. These traits each represent more and more restrictive properties about closures/functions, indicated by the signatures of their `call_...` method, and particularly the type of `self`.

- `FnOnce (self)` are functions that can be called once.
- `FnMut (&mut self)` are functions that can be called if they have `&mut` access to their environment.
- `Fn (&self)` are functions that can be called if they only have `&` access to their environment.

A closure `|...| ...` will automatically implement as many of those as it can.

- All closures implement `FnOnce`: a closure that can't be called once doesn't deserve the name. Note that if a closure only implements `FnOnce`, it can be called only once.
- Closures that don't move out of their captures implement `FnMut`, allowing them to be called more than once (if there is unaliased access to the function object).
- Closures that don't need unique/mutable access to their captures implement Fn, allowing them to be called essentially everywhere.

```
pub fn closure_example3(x:i32) {
    let y = 3;
    let add = |x| {
        x + y
    };
    println!("Receives Closure {:?}", add(5));
}

fn receive_closure<F>(closure: F)
where
    F: Fn(i32) -> i32,
{
    let result = closure(4);
    println!("closure => {}", result);
}
```

Listing 11.5: Closure example 3

Invoking:

```
fn_11_clo::closure_example3(5);
```

Output:

```
Receives Closure 8
```

11.2 STORING CLOSURES WITH Fn TRAIT

With `struct` and `closure` we can create what is known as the **memoization or lazy evaluation** pattern. 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 will cache the resulting value automatically.

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, i.e., even if two closures have the same signature, their types are still considered different.** All closures implement at least one of these traits: `Fn` , `FnMut` or `FnOnce` and are provided by the standard library.

In listing 11.7 the `struct Cacher` has a `calculation` field of the generic type `T`. The trait bounds on `T` specify that it is 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.

```
12 struct Cacher<T>
13 where
14     T: Fn(u32) -> u32,  // trait bound
15 {
16     calc: T,            // calc stores the closure that is trait bound
17     value: Option<u32>, // Result of calling the function calc
18 }
```

Listing 11.6: Cacher

```
40 impl<T> Cacher<T>
41 where
42     T: Fn(u32) -> u32 , // trait bound
43 {
44     fn new(calc: T) -> Cacher<T> {
45         Cacher {        // expression returning the function
46             calc,
47             value: None
48         }
49     }
50
51     fn func(&mut self, arg: u32) -> u32 {
52         match self.value {
53             Some(v) => v, // value exists, return v
54             None => {     // value does not exit
55                 let v = (self.calc)(arg);   // invoke calc with arg
```

```
                self.value = Some(v);        // wrap value in Option
                v                            // return v
            }
        }
    }
}
```

Listing 11.7: Cacher Implementation

Below is an example of the usage of `Cacher`.

```
use std::thread;
use std::time::Duration;
pub fn generate_force(hp: u32, random_number: u32) {

    let mut my_closure = Cacher::new(|number| {
        println!("calculating HP ...");
        thread::sleep(Duration::from_secs(1));
        number
    });

    if hp < 25 {
        println!("Low HP drive slow {}", my_closure.func(hp));
        println!("Low HP drive steady {}", my_closure.func(hp));
    } else {

        if random_number == 3 {
            println!("No HP generated");
        } else {
            println!(
                "Sufficient HP {}", my_closure.func(hp)
            );
        }
    }

}
```

Listing 11.8: Using Cacher

Output:

```
calculating HP ...
Low HP drive slow 20
Low HP drive steady 20
```

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 12

SMART POINTERS

Pointers in a language is a variable that stores the address of another variable. This address refers to, or "points at," some other data. We are already familiar with the most common kind of pointer in Rust, the reference. References are indicated by the `&` symbol and borrow the value they point to. But references do not have any special capabilities other than referring to data. However, they also do not have any overhead and are the kind of pointers we use frequently.

Smart pointers originated in C++. In Rust, the different smart pointers defined in the standard library provide functionality beyond what is provided by references. In Rust, which uses the concept of ownership and borrowing, an additional difference between references and smart pointers is that references are pointers that only borrow data. **Smart pointers, in many cases, own the data they point to.**

Smart pointers are mostly implemented using structs. These structs implement the `Deref` and `Drop` traits.

- The `Deref` trait allows an instance of the smart pointer struct to behave like a reference so that the code works with either references or smart pointers.
- The `Drop` trait allows us to customize the code that is run when an instance of the smart pointer goes out of scope.

12.1 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 do not have any performance overhead, other than storing their data on the heap instead of on the stack. `Box` is useful under these circumstances.

- A type whose size is unknown at compile time and we want to use a value of that type in a context that requires an exact size.
- A large amount of data, we want to transfer ownership but do not want the data to be copied.
- Own a value and its type must implement a certain trait rather being of a particular type.

```
1 let x = Box::new(100);
2 println!("x = {}", x);
```

Listing 12.1: Storing an i32 value on the heap using a box

Output:

```
x = 100
```

We define the variable `x` to have the value of a Box that points to the value 100 which is allocated on the heap. We can access the data in the box in the same way we access data on the stack. Just as in any owned value, when a box goes out of scope it will be deallocated.

At compile time, Rust needs to know how much space a type takes up. One type whose size can not be known at compile time is a recursive type. The nesting of values could theoretically continue infinitely and so Rust will not know how much space a value of a recursive type actually needs. However, boxes have a known size, so by inserting a box in a recursive type definition, you can have recursive types.

12.1.1 CONS LIST

To understand the concept of a Box better, let us review the 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 is not 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.

Since Box<T> is a pointer, Rust knows how much space a Box<T> needs as **a pointer's size does not change based on the amount of data it is pointing to.** Conceptually, we will have lists "holding" other lists. This is now more like placing items next to each other.

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

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

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

Listing 12.2: Definition of List that uses Box<T> in order to have a known size

Boxes provide only the indirection and heap allocation, they do not have any other special capabilities. They also do not have any performance overhead. 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.

12.2 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.

```
let x = 5;
let y = Box::new(x);
assert_eq!(5, x);
assert_eq!(5, *y);
```

Listing 12.3: Using the dereference operator on a Box<i32>

Let us define a custom type that behaves like Box<T> and see why the dereference operator does not work like a reference on our newly defined type. Implementing the `Deref` trait makes it possible for smart pointers to work in ways similar to references. The `type Target = T;` syntax defines an associated type for the `Deref` trait to use.

```
let x = MyBox{a:100};
println!("{}",*(x.deref()));

#[derive(Debug)]
struct MyBox<T> { // same as: struct MyBox<T>(T);
    a: T
}

use std::ops::Deref;
impl<T> Deref for MyBox<T> {
    type Target = T;

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

Output:

```
Dereferenced: 100
```

12.3 DROP TRAIT

In languages such as C/C++, 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.

Rust alleviates this problem by providing the `Drop` trait. This trait is important to smart pointers. This trait enables us to customize what happens when a value is about to go out of scope. In the implementation for the `Drop` trait on any type, you can specify what needs to happen which can include activities such as releasing resources such as files, network connections, database connections, etc. 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.

We specify the code to run when a value goes out of scope by implementing the `Drop` trait. The `Drop` trait requires us to implement one method named `drop` that takes a mutable reference to self. To see when Rust calls drop, let us implement drop in the following example.

```
let x = mysmaptr{ data : String::from("Hello") };
println!("struct mysmaptr with data {}", x.data);

struct mysmaptr {
    data: String
}

impl Drop for mysmaptr {
    fn drop(&mut self) {
        println!("Dropping struct mysmaptr with data {}", self.data);
    }
}
```

Output:

```
struct mysmaptr with data Hello
Dropping struct mysmaptr with data Hello
```

CHAPTER 13

CONCURRENCY

Concurrent programming means different parts of a program execute independently to take advantage of multiple CPUs. The features that run these independent parts are called **threads**. Splitting the computation in your program into multiple threads can improve performance because the program does multiple tasks at the same time, but it also adds complexity. Because threads can run simultaneously, there's no inherent guarantee about the order in which parts of your code on different threads will run. This can lead to problems, such as **race conditions**, **deadlocks**. Historically, concurrent programming has been difficult and error prone. Bugs that occur only in certain situations and are hard to reproduce and fix reliably.

Before we dive into details, let's review some terms:

- **Parallel** computing is the ability to do multiple things simultaneously. This is possible only if you have multiple cores or CPUs. This is obvious as you cannot make one processor do two computations at the same time.
- **Concurrent** computing is the ability to do multiple things but not at once.
- **Task** is a generic term for some computing running in a parallel or concurrent system. A **thread** can be thought of as a task.
- **Asynchronous** computing refers to language features that enable parallelism or concurrency.
- **Cooperative Multitasking** means each task decides when to yield to another task.
- **Preemptive Multitasking** means the system decides when to yield to another task.
- **Native Threads** - Also called 1:1 threading, are tasks provided by the operating system, 1 task per 1 thread. These are part of the system and the operating system schedules them. They are resource heavy and a limited number of them can be created.
- **Green Threads** - Also called N:M threading, where the runtime provides task abstractions and maps N program threads to M system threads. These are not part of the system, the runtime does their scheduling. The stack growth can cause issues as the green thread can run out the small amount of stack (there are ways of handling it) and there is a overhead calling into C because C expects a real stack.

Developers of Rust discovered that the **ownership** and **type systems** are the keys to help manage memory safety and address concurrency problems. By leveraging Rust's unique concept of ownership and type checking, many concurrency errors are reduced to compile-time errors in Rust rather than runtime errors. Rather than spending significant 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 as **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. Rust offers a variety of tools for modeling problems in whatever way is appropriate for your situation and requirements.

13.1 THREADS

Let us first include some modules that we will need for our example codes.

```
use std::sync::mpsc;
use std::thread;
use std::time::Duration;
use std::sync::{Arc,Mutex};
```

```

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

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

    handle.join().unwrap();
```

Listing 13.1: Thread example

Invoking function `concur_example()` will result in the following:

Output:

```
Hi # 1 from the main thread!
Hello # 1 from the spawned thread!
Hi # 2 from the main thread!
Hello # 2 from the spawned thread!
Hi # 3 from the main thread!
Hello # 3 from the spawned thread!
Hi # 4 from the main thread!
Hello # 4 from the spawned thread!
Hello # 5 from the spawned thread!
```

```
Hello # 6 from the spawned thread!
Hello # 7 from the spawned thread!
Hello # 8 from the spawned thread!
Hello # 9 from the spawned thread!

```

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. If the call to `handle.join().unwrap();` is moved from line 17 to line 11, then the output will no longer will be interleaved as the main thread will wait for the spawned thread to finish before starting its own `loop`.

13.2 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 has implementation of a `channel` to send and receive messages between concurrent sections of the code. A `channel` has two halves, a transmitter and a receiver. Let's look at the following code that has multiple producers of messages and a single receiver.

```

pub fn concur_example2() {

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

    let tx1 = mpsc::Sender::clone(&tx);
    thread::spawn( move || {
        let vals = vec![
            String::from("Hello"),
            String::from("from"),
            String::from("thread-1"),
        ];

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



    thread::spawn(move || {
        let vals = vec![
            String::from("Hi"),
```

```
            String::from("your"),
            String::from("thread-2"),
        ];

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

    let result = rx.recv_timeout(Duration::from_millis(1));
```

Listing 13.2: Channel example

Invoking function `concur_example2()` will result in the following:

Output:

```
Got: Hello
Got: Hi
Got: your
Got: from
Got: thread-1
Got: thread-2
```

Listing 13.3: Channel

13.2.1 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. **By adding the `move` keyword before the closure, we force the closure to take ownership of the values it's using rather than allowing Rust to infer that it should borrow the values.** Any attempt to access the variable `vals` after it has been sent will generate a compile error. This is how Rust ensures safety in concurrency.

13.3 SHARED STATE CONCURRENCY

Channels in any programming language are similar to single ownership because once a value is transferred down a channel, you can no longer use that value. Shared memory concurrency is like multiple ownership: multiple threads can access the same memory location at the same time. Smart pointers make multiple ownership possible but multiple ownership can add complexity because these different owners need managing. Rust's type system and ownership rules greatly assist in getting this management correct. We will look at `mutex`, a common concurrency primitive, for shared memory. But management of mutexes can be incredibly tricky as one has to get the locking and unlocking correct at all times. However, thanks to Rust's type system and ownership rules, you cannot go wrong on locking and unlocking.

`Mutex<T>` is a smart pointer. More accurately, the call to lock returns a smart pointer called `MutexGuard`, wrapped in a `LockResult` that we handled with the call to `unwrap`. The `MutexGuard` smart pointer implements `Deref` to point at our inner data. The smart pointer also has a `Drop` implementation that releases the lock automatically when a `MutexGuard` goes out of scope, which happens at the end of the inner scope in listing 13.4. As a result, we don't risk forgetting to release the lock and blocking the `mutex` from being used by other threads because the lock release happens automatically.

```
59              println!("{:?}",e);
60              process::exit(0);
61          },
62          _ => ()
63      }
64
65
66      for received in rx {
67          println!("Got: {}", received);
68      }
69  }
70
71
72  pub fn mutex_example() {
73      let counter = Arc::new(Mutex::new(0));
74      let mut handles = vec![];
75
76      for _ in 0..10 {
77          let counter = Arc::clone(&counter);
```

Listing 13.4: Mutex example

`Arc` is an atomically referenced counter type. Basically, it is a primitive type that is safe to be shared across multiple threads. The rest of the code is self explanatory, it uses concepts already discussed earlier.

13.4 ASYNC/AWAIT

`Async/await` are special pieces of Rust syntax that make it possible to yield control of the current thread rather than blocking it allowing other code to make progress while waiting on an operation to complete. The `async/await` syntax lets you write code that feels synchronous but is actually asynchronous. In Rust, deferred computations due to "long" running programs are called **futures**. While most of the concepts are fairly similar with other programming languages, in Rust you need to pick a runtime to actually run your asynchronous code. Recall that a runtime describes software/instructions that are executed while your program is running, especially those instructions that you did not write explicitly, but are necessary for the proper execution of your code. Since Rust targets everything from bare metal, embedded devices to programs running in an advanced OS, it focuses on zero-cost abstractions and you will have to select a runtime which is a library code implementing the specific features.

An async application should pull in at least two crates from Rust's ecosystem:

1. futures, an official Rust crate that lives in the rust-lang repository
2. A runtime of your choosing, such as tokio, async_std, etc.

The de facto standard library providing a runtime system for green threads and asynchronous I/O is `tokio` which we will use. `tokio` is an event-driven, non-blocking I/O platform for writing asynchronous applications with the Rust programming language. `tokio` has zero-cost abstractions and delivers bare-metal performance. It leverages Rust's ownership, type system, and concurrency model to reduce bugs and ensure thread safety. It is scalable because of its minimal footprint, and handles backpressure and cancellation naturally.

We first add the dependencies in `Cargo.toml`.

```
futures = { version = "0.3.*" }
tokio = {version = "0.2.*", features = ["full"] }
```

The "full" feature flag tells `tokio` to enable all public APIs. Irrespective of which runtime we choose to use, we need to do three distinct operations in an async code.

1. Start the runtime.
2. Spawn a Future. Future is an object thats act as a proxy for a result that is initially not known but will be known at a future time.
3. Spawn blocking or CPU intensive tasks which are long running.

The `tokio` macro `#[tokio::main]` lets us pull another trick to further simplify our code and make it as if Rust had an asynchronous runtime built-in. This macro will create the default runtime, enter the runtime and block on it. The macro will expand to:

```
let mut rt = tokio::runtime::Runtime::new().unwrap();
rt.enter(|| {
    println!("in rt.enter()");
    tokio::spawn(future::lazy(|_| println!("in tokio::spawn()")));
});
rt.spawn(future::lazy(|_| println!("in rt.spawn()")));
rt.block_on(future::lazy(|_| println!("in rt.block_on()")));
```

Listing 13.5: tokio macro expansion

In the following example, you will notice that `async` is the syntactic sugar to eliminate the need for implementing `Future`. This will be evident when you compare the two "long running functions". Notice that the first set of tasks runs sequentially and takes 4 seconds to complete while the second set of identical tasks take 3 seconds to complete because they run concurrently.

```
use std::error::Error;
use std::time::{Duration, Instant};
use std::thread;
use futures::future;
use futures::join;
use tokio::macros::support::Future;
```

```rust

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {

    // Sequential execution
    let t1 = Instant::now();
    let mut x1 = 100;
    let r1 = long_running_fn_1(&mut x1).await;
    let r2 = long_running_fn_2().await;
    // let r = join!(long_running_fn_1(&mut x), long_running_fn_2(101));

    let t2 = Instant::now();
    println!("{} {}  {:?}",r1,r2,t2-t1);

    // Concurrent execution
    let tasks = vec![
        tokio::spawn(async move { long_running_fn_1(&mut x1).await }),
        tokio::spawn(async move { long_running_fn_2().await }),
    ];
    // join the tasks
    let t1 = Instant::now();
    let r = futures::future::join_all(tasks).await;
    let t2 = Instant::now();
    println!("{:?} {:?}",r,t2-t1);

    Ok(())
}

fn long_running_fn_1(x: &mut i32) -> impl Future<Output = i32> {
    thread::sleep(Duration::from_secs(1));
    *x = *x + 1;
    future::ready(*x)
}

async fn long_running_fn_2() -> i32 {
    thread::sleep(Duration::from_secs(3));
    42
}
```

Listing 13.6: Async/Await example

Output:

```
101 42  4.00133155s
[Ok(102), Ok(42)] 3.002139593s
```

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.