# Rust Course notes

# Razafy Rindra, "Hagamena"

## July 18, 2022

## Contents

	Introduction
	Common Programming Concepts in Rust           2.1 Variables and Mutability         2.1.1 Mutable variables           2.1.2 Constants         2.1.2 Shadowing           2.2 Data Types         2.3 Functions           2.4 Control Flow         2.4.1 if statements           2.4.2 Loops         2.4.2 Loops
	Ownership 3.1 Ownership Rules
	Structs         1           4.1 Defining and Using Structs         1           4.2 Methods          1
	Enums and Pattern matching       1         5.1 Defining Enums       1         5.2 Option Enum       1         5.3 Pattern Matching       1         5.4 Using if let syntax       1
6	Module         1           6.1 Packages and Crates         1           6.2 Defining Modules         1           6.2.1 Definintions, Paths and Privacy         1           6.3 Privacy rules when it comes to structs         1           6.3.1 Privacy rules when it comes to enums         1           6.3.2 Use keyword         1           6.3.3 Modules in sperate files         2
	Common Collections           7.1 Vectors         2           7.2 String         2           7.3 Hash maps         2
8	Error handling         2           8.1 Panic!

Generics	7
9.1 A first example	

## 1 Introduction

Some quick notes from the Rust book, and the "Let's Get Rusty" online course

# 2 Common Programming Concepts in Rust

## 2.1 Variables and Mutability

#### 2.1.1 Mutable variables

Variables in Rust are immmutable by default. In order to create a mutable variable, we need to add in 'mut' in front of the name like so:

```
fn main() {
    let mut x = 5;
    println!("The value of x is: {x}");
    x = 6;
    println!("The value of x is: {x}");
}
```

#### 2.1.2 Constants

We can also declare constants in Rust like so:

```
const my_number: u32 = 100_000;
```

Constants unlike variables can't be mutable, need to be type annotated and they can't be assigned to return value of a function or any value computed at run time.

## 2.1.3 Shadowing

Shadowing allows us to create a new variable with an existing name, for example:

```
let x = 5;
println!("The value of x is: {x}");
let x = "six";
println!("The value of x is: {x}");
```

## 2.2 Data Types

**Definition 2.1. Scalar data types** represent a single value, **Compound data types** represent a goup of values

Types of scalar data types

- 1. Integers, they can be 8,16,32,64,128 bit signed or unsigned integers.
- 2. Floating-point numbers
- 3. Booleans
- 4. Character, they represent unicode character

Type of compound data types

1. Tuples; a fixed size array of data that can be of different values.

```
let tup = ("Let's Get Rusty!",100_000);
```

2. Arrays, in Rust they are fixed length.

## 2.3 Functions

Functions are defined with an fn keyword like so:

```
fn my_function(x: i32,y: i32){
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
}
```

A piece in code in Rust are either a statement or an expression. Statements perform some action but do not return any value, whilst expressions returns values.

```
fn my_function(x: i32,y: i32) -> i32{
    println!("The value of x is: {}", x);
    println!("The value of y is: {}", y);
    x+y
}
```

In this function, the println are statements since they don't return anything while 'x+y' is an expression, it is returned by the function (note the last expression of a function is implicitly returned).

## 2.4 Control Flow

## 2.4.1 if statements

As in other programming languages

```
fn main() {
    let number = 3;

if number < 5 {
        println!("first condition was true");
} else if number < 22{
        println!("second condition was true.")
} else {
        println!("condition was false");
}
</pre>
```

Note, the condition must be a boolean. We can also set a "if-else" statement in a let statement.

```
fn main() {
   let condition = true;
   let number = if condition { 5 } else { 6 };

println!("The value of number is: {number}");
}
```

## 2.4.2 Loops

We can create loops with the 'loop' keyword, which will execute the code in it until we call break.

```
let mut counter = 0;
let result = loop{
    counter += 1;
    if counter == 10{
        break counter;
}
};
```

We can also use the while statement:

```
let mut number = 3;
while number != 0{
    println!("{}!", number);
    number -= 1;
}
```

Finally the third type of loop we can create are 'for loops'

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

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

We can also loop over a range

```
for number in 1..4{
    println!("{}!", number);
}
```

The last number of the range is excluded.

# 3 Ownership

What is ownership? The ownership model is a way to manage memory. How do we manage memory today?

1. Garbage collection. Used in high level languages like java or C#, the Garbage collection manages memory for you. It's pros is that is error free, you don't have to manage memory yourself so you won't introduce memory bugs. It also faster write time.

It's cons is that you have no control over memory, it is slower and has unpredictable runtime performance and larger program size since you got to include a garbage collector.

2. Manual memory management

It's pros are that you have higher control over memory, faster runtime and smaller program size. But it is error prone and has slower writing time.

Notice that these two have opposite trade-offs.

3. Ownership model

This is the way Rust manages memory, it's pros are control over memory, faster runtime and smaller program size and is error free (though it does allow for you to opt-out of memory safetey). It's cons is that you have a slower write time slower than with Manual memory management, Rust has a strict set of rules around memory management.

Stack and Heap During runetime program has access of stack and heap, stack is fixed size and cannot grow or shrink during runetime and it creates stack-frames for each function it executes. Each stack frames stores the local variables of the function they execute, their size are calculated during compile time and variables in stack frames only live as long as the stack lives.

The heap grows and shrinks during runtime and the data stored can be dynamic in size and we control the lifetime of the

Pushing to the stack is faster than allocating on the heap since the heap has to spend time looking for a place to store the data, also accessing data on the stack is faster than accessing data on the heap, since on the heap we must follow the pointer.

```
let x = "hello";
```

This is a string litteral and has fixed size and is stored in the stack-frame.

```
let x = String::from(world");
```

x is of type String which can be dynamic in size so it can't be stored on the stack, we ask the heap to allocate memory for it and it passes back a pointer, which is what is stored on the stack.

# 3.1 Ownership Rules

- 1. Each value in Rust has a variable called its owner.
- 2. There can only be one owner at a time.
- 3. When the owner goes out of scope, the value will be dropped

## Example 3.0.1.

```
fn main() {
    let x = 5;
    let y = x; // Copy

let s1 = String::from("hello");
    let s2 = s1; // move (not a shallow copy)
```

```
s          println!("{}, world", s1);
9      }
10
```

This returns an error since when we created s2 we made it point to the same "hello" on the heap that s1 points to, but in order to ensure memory safetey, Rust invalidates s1.

What if we do want to clone the string? Use the clone() method:

```
fn main() {
    let x = 5;
    let y = x; // Copy

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

println!("{}, world", s1);
}
```

## 3.2 Ownership and functions

## Example 3.0.2.

```
fn main() {
    let s = String::from("hello);
    takes_ownership(s);
    println!("{}", s);
}

fn takes_ownership(some_string: String) {
    println!("{}", some_string);
}
```

This gives us an error since after we pass a parameter in a function it is the same as if we assign the parameter to another variable.

So we move s into some string and after takes ownership scope is done some string is dropped.

## Example 3.0.3.

```
fn main() {
    let s1 = gives_ownership();
    println!("s1 = {}", s1);
}

fn gives_ownership() {
    let some_string = String::from("hello");
    some_string // returning the string moves ownership to s1
}
```

## Example 3.0.4.

```
// function

// function

// function

// function

// function

// function

// scope

// This function takes a String and returns one
fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
// scope

// scope

a_string // a_string is returned and moves out to the calling function
}

// function
```

Moving ownership and giving back is tedious, what if we just want to use variable without taking ownership? Use references.

## 3.3 References

Let us see how references fix the following situation

```
fn main() {
    let s1 = String::from("hello");

let (s2, len) = calculate_length(s1);

println!("The length of '{}' is {}.", s2, len);

fn calculate_length(s: String) -> (String, usize) {
    let length = s.len(); // len() returns the length of a String

(s, length)
}
```

Here in order to calculate the length of the string without taking ownershipe, we need to return a tuple that returns both the string and the length.

```
fn main() {
    let s1 = String::from("hello");

    let len = calculate_length(&s1);

    println!("The length of '{}' is {}.", s1, len);

}

fn calculate_length(s: &String) -> usize {
    let length = s.len(); // len() returns the length of a String length
}
```

Here s is a reference of a string and takes no ownership of the string, it points to s1 which points to the string. So when the function finishes executing s is dropped without affecting s1.

**Definition 3.1.** Passing in references as function parameters is called **borrowing**. Since we are not taking ownership of the parameters.

Note that references are immutable by default, so how to we modify value without taking ownership? Mutable references:

```
fn main() {
    let mut s1 = String::from("hello");

change(&mut s1);

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

Now change can mutate the value of s1 without taking ownership. Mutable reference have a restriction, we can only have one mutable reference to a particular piece of data in a particular scope.

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

let r1 = &mut s;
    let r2 = &mut s; // Returns an error since can not borrow twice.

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

This prevents "data races" where two pointers point at the same data and one pointer reads the data and another one tries to write to the data.

What if we try to mix mutable and immutable references? We get another error, we can't have mutable reference if an immutable reference already exists. Immuatble reference don't expect the underlying data to change. But we can have many Immuatble references, since we don't expect the underlying value to change.

Note the scope of a reference starts when it introduced and ends when it's used for the last time, so we can define a mutable reference when all the immuable references are out of scope(so after we use them for the last time).

#### Rules of Refences

- 1. At any given time, we can either have one mutable reference or any number of immuatble reference.
- 2. References must always be valid.

## 3.4 Slices

**Definition 3.2.** Slices let you a contiguous sequence of elements in a collection instead of the whole collection, without taking ownership.

There are two problems with this, the return value of first\_word is not tied to the string. If we change the string, the value of the length of the first word doesn't change.

If we wanted to return the second word, we must retrurn a tuple with the index at the start of the word and the index at the end of the word. We have more values we must keep in sync.

Let us use the string slice

```
fn main() {
    let mut s = String::from("hello world");

    let hello = &s[..5]; //String Slices, tells us we want the value of the string from index 0 to 4
    let world = &s[6..]; //String Slices, tells us we want the value of the string from index 6 to 10

let s2 = "hello word"; // String litteral are string slices!

let word = first_word(s2);

fn first_word(s: &str) -> &str {
    let bytes = s.as_bytes();

for (i, &item) in bytes.iter().enumerate() {
    if item = b' ' {
        return &s[0..i];
    }
}

& &s[..]

}

& &s[..]
}
```

We can also create slice of array

```
fn main() { let a = [1,2,3,4,5]; let slice = &a[0..2]; // This is of type &[i32] }
```

## 4 Structs

## 4.1 Defining and Using Structs

```
let name = user1.username; // Get values from struct with the . operator user1.username = String::from("user234"); // modify specific values with . operator
```

## 4.2 Methods

# 5 Enums and Pattern matching

Structs and enums are the building blocks for creating new types in Rust. In Rust, enums are most similar to the ones from functional programming.

## 5.1 Defining Enums

Enums allow us to enumerate a list of variants. When is it appropriate to use enums over structs?

**Example 5.0.1.** In this example we use enums to enumerate IP addresses, an IP addresse can be one of only two types, version 4 and version 6. So it is natural to use enums if we want to express IP addresses in code.

## 5.2 Option Enum

Many languages have null values, a value can either exists or it is null (there is no value). But the type system cannot guarentee that if you use a value it is not null.

This is not a problem in Rust, since Rust has no null values. We use the option enum

## 5.3 Pattern Matching

Recall, match allows us to compare a value to a set of values. This is very useful for enums.

This program will output:

State quarter from Alaska!

## 5.4 Using if let syntax

```
//Using if let syntax

fn main(){
    let some_value = Some(3);

// Instead of using the match like this, when we only have on case we care about.

match some_value{
    Some(3) => println!("three"),
    _=> (),
}

// We can use the if-let synyax

if let Some(3) = some_value{ // Says if some_value matches with Some(3) execute the bellow code println!("three");
}

}
```

## 6 Module

In previous courses we have just been writting our code in one file, but now we are going to learn to be more organised, we will learn rust's Module system.

## 6.1 Packages and Crates

#### Definition 6.1. Crates:

When we type "Cargo new", Rust creates a new package containing crates, which contain modules

- Binary crate: Code vou can execute
- Library crate: Code that can be used by other programs.

Convention: If we have "main.rs" file in the src directory then a binary crate with the same name as package will be automatically created and main.rs will be the crate root.

## Definition 6.2. Crate root

Is the source file that rust compiler starts at when building our crate.

If we have "lib.rs" file in the src directory then a library crate with the same name as package will be automatically created and lib.rs will be the crate root.

## Rules around crates

- A package must have at least one crate
- A package can have either 0 or 1 library crate
- A package can have any number binary crate.

If want to make more binary crates, make a folder called bin, each file in that folder will represent another binary crate

## 6.2 Defining Modules

## 6.2.1 Definitions, Paths and Privacy

```
//Back of the house is where food is being made, dishes are clean and where manager is.
// If you want to reference an item in module tree (like a funciton), need to specifyt a path to
```

## 6.3 Privacy rules when it comes to structs

## 6.3.1 Privacy rules when it comes to enums

```
mod back_of_house {
    pub enum Appetizer {
        Soup, // By default if an enum is public so are it's variants.
        Salad,
    }
}

pub fn eat_at_restaurant() {
    let order1 = back_of_house:: Appetizer:: Soup;
    let order2 = back_of_house:: Appetizer:: Salad;
}
```

## 6.3.2 Use keyword

```
Using external packages

//Use keyword

/*Instead of this:
use rand::Rng;
use rand::CryptoRng;
use rand::ErrorKind::Transient;
We can do this:
*/

use rand::{Rng, CryptoRng, ErrorKind::Transient}; // Nested paths

/*
Instead of this:
use std::io
use std::io::Write

/*
We can do this:

use std::io::write

// We can do this:
//

// Glob operator:
// use std::io::* this means all public items underneath io are in scope.
```

## 6.3.3 Modules in sperate files

In order to make our code cleaner we can move our module definitions in different files:

```
// In src/lib.rs
mod front_of_house; // This tells Rust, define our module here but get the contents from a different file
with the same name as our module.

pub use crate::front_of_house::hosting;

pub fn eat_at_restaurant() {
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}
```

```
1 //In src/front_of_house.rs
2 pub mod hosting;
```

```
// In src/front_of_house/hosting.rs

pub fn add_to_waitlist() {} // Needs to live in a directory with the same name as parent module.
```

## 7 Common Collections

**Definition 7.1.** Collection These are some useful data structures included in the standard library.

Unlike arrays and tuples, collections are allocated on the heap so their size can grow or shrink as needed.

In this section we will talk about vectors, strings and hashmaps.

## 7.1 Vectors

```
let v2 = vec![1,2,3]; // Can create a vector with initial values like this.
}// When our scope ends, v2 and all elements in it are dropped.
```

## 7.2 String

In higher programming languages, the complexity of strings is abstracted away from the programmer. In lower programming languages, like in Rust, we have to deal with that complexity.

**Definition 7.2.** Strings are stored as a collection of UTF-8 encoded bytes

What is UTF-8 encoding? We first need to understand ASCII, it is an string encoding. It defines how to take 10's and turn them into strings and vice-versa.

Each ASCII character is stored as a byte, and only 7-bits of that byte is used to represent a character. So only 128 unique characters. It only represents the english alphabet, some special characters and commands.

So other countries came up with their own standards to encode characters. So how does a program know what standard to use when interpratating a collection of bytes.

So unicode was created was used to solve this problem. Unicode represent characters from a lot of languages, and other characters like emojis. It is also backwards compatible with ASCII.

**Definition 7.3.** UTF-8 is a "variable-width" character encoding. Each character in UFT-8 can be represented by 1 byte, 2 bytes, 2 bytes or 4 bytes.

## Three relevant ways a string is represented in unicode

- bytes
- Scalar values, can think about these as building blocks of characters (this is what the char type refers to). They can be characters or parts of characters.
- Grapheme clusters, what we usually mean when we say characters, the glyphs that build up a word.

The problem with indexing into a string Rust doesn't know what value we want to recieve. Bytes, scalaing values or grapheme clusters.

Look at the /collection-strings folder to see the code.

## 7.3 Hash maps

**Definition 7.4.** Hash maps allow us to store key-value pairs and uses a hashing function to determine how to place the keys and values.

```
scores.insert(String::from("Blue"), 20); // Overwrites the Blue key with the value 20.
//["hello", "world", "wonderful", "word"]
for word in text.split_whitespace() { //Splits up the string by the whitespace
let count = map.entry(word).or_insert(0); // .entry retirns an enum representing the value at that
*count += 1; //or_insert returns a mutable reference to our value, so we can deincrement it and add 1, even if it doesn't do anything sine the key already exists.
```

# 8 Error handling

## 8.1 Panic!

If program fails in an unrecoverable way we can use panic! macro that quits the program and returns an error message.

```
fn main() {
   panic!("crash and burn");
}
```

This will return

```
thread 'main' panicked at 'crash and burn', src/main.rs:3:5
note: run with `RUST_BACKTRACE=1` environment variable to display a backtrace
```

Note if we have a function that sends out a panic! if given some bad inputs, but we have multiple functions that can call this function. If our program panics!, we don't know from where the error is comming from. To resolve this problem we use backtrace. Like so:

Running it with backtrace to see where we have the bad function call

## 8.2 Result Enum

Let's talk with recoverable errors, that we can handle gracefully without crashing.

We have the "Result" Enum, like the options Enum it contains two variable, but in this case they represent success and failure:

```
enum Result < T, E > {
    Ok(T), // represents success
    Err(E), // represents failure
}
```

```
An example, using the result enum to deal with errors opening files

use std::s::File;

use std::io::ErrorKind; //Let us match on the type of error we get

fn main(){

let f = File::open("hello.txt"); //Returns an Result Enum

let f = match f{ //Need to resolve what we do with the different enum variables

Ok(file) => file, // Assign the file to f

Err(error) => match error.kind(){ // If there is an error match based on the kind of error

ErrorKind::NotFound => match File::create("hello.txt"){ // If file doesn't exists create

the file

Ok(fc) => fc, // If no error we can creat the file

Err(e) => panie!("Problem creating the file: {:?}", e) // If there is error panie!

};

other_error=>{ // If we get an error not of the file not found kind panie!

panie!("Problem opening the file: {:?}", other_error)

}

// Another way to write this code using closures. More about this in chapter 13

let f = File::open("hello.txt").unwrap_or_else(|error| { // This will give us back the file or call this anonymous function, aka closure |error| {...}

if error.kind() = ErrorKind:NotFound {

File::create("hello.txt").unwrap_or_else(|error| { // This will give us back the file or call this anonymous function, aka closure |error| {...}

if error.kind() = ErrorKind:NotFound {

File::create("hello.txt").unwrap_or_else(|error| { // This will give us back the file or call this anonymous function, aka closure |error| { ...}

panie!("Problem creating the file: {:?}", error);

}) else {

panie!("Problem opening the file: {:?}", error);
})

});
```

## 8.3 Error propagation

**Definition 8.1.** When we have a function whose implementation calls something that can fail, we often want to return that error back to the caller instead of handling it in the function.

This is called **error propagation** 

```
fn main(){
    let f = File::open("hello.txt")?; // Error message: the `?` operator can only be used in a function that returns `Result` or `Option`
```

In order to fix this, we can let main() return a result type.

```
Fixing our code by making main() return Result
```

```
#![allow(unused)]
use std::error::Error;
use std::fs::File;

fn main() -> Result <(), Box<dyn Error>>{ //In the error case we return a "Trait" object, which we will
talk about in chap 17
let f = File::open("hello.txt")?;

Ok(())

Ok(())
```

When should we be using the Result enum and the panic! macro? In general we should in default use the Result enum, this prevents the program from crashing, and error propagation, which let's the caller decide what to do with the error.

We should only use the panic! macro in exceptional circumstances, for example circumstances where recovering from the error is impossible or recovering from that state is impossible.

Another appropriate place to allow our code to panic! is in example code. We can use methods like unwrap or expect for brevity, and since there is no context for determinaning with how to deal with errors.

We may also use unwrap or expect in prototype code, in order to get code out quickely in order to test it and then introduce error handling after.

We may also use expect or unwrap in test code.

Lastly, we may use expect or unwrap when we know a call to function will succeed.

## 9 Generics

In the next three sections we will be covering generics, traits and lifetimes. These are all ways to reduce code duplication.

## 9.1 A first example

Say we have a list of numbers and we want to find the largest element, then naively we can write it like this:

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

let mut largest = number_list[0];

for number in number_list {
    if number > largest {
        largest = number;
    }

println!("The largest number is {}", largest); // prints 100
```

The problem with this code is that if we want to find the largest element of another function, we will have to rewrite the for loop again. An easy fix to this is to put the logic used to find the largest number into another function:

```
fn main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let largest = get_largest(number_list); //Returns 100

    println!("The largest number is {}", largest);

let number_list = vec![102, 34, 6000, 89, 54, 2, 43, 8];

let largest = get_largest(number_list); //Returns 6000

println!("The largest number is {}", largest);

fn get_largest(number_list: Vec<i32>) -> i32{
    let mut largest = number_list[0];
    for number in number_list {
        if number > largest {
            largest = number;
        }
    }

largest

largest

largest

largest

largest

largest
```

But what if we want to use the same logic as in our get\_largest function over a slightly different set of arguments. Let say we are looking for the largest character in a vector.

We can use "generics" to modify our get largest function to be able to take in both a Vec<i32> and Vec<char>

```
Using generics to generalise our function

in main() {
    let number_list = vec![34, 50, 25, 100, 65];

    let largest = get_largest(number_list); //Returns 100

println!("The largest number is {}", largest);

let char_list = vec!['y', 'm', 'a', 'q'];

let largest = get_largest(char_list); //Returns 'y'

println!("The largest character is {}", largest);

fn get_largest<T: PartialOrd + Copy>(list: Vec<T>) -> T{ // Generic types are specified in <> after the function name

let mut largest = list [0];

for element in list {
    if element > largest {
        largest = element;
    }

largest = largest {
        largest = element;
    }

largest = largest
```

*Remark.* PartialOrd and Copy are traits. We need to specify them to restrict our function to only accepting types that can be ordered and copied (like ints or char).

We will talk more about traits in next section.

```
Generic Types in Structs

| struct Point<T, U>{
| x: T, |
| y: U, // If we only used one generic, then both x and y would have to be of the same type T

| fn main() {
| let p1 = Point{x:5, y:10}; // Can pass in two i32 |
| let p2 = Point{x:5.0, y:10.0}; // Can pass in two f64 |
| let p3 = Point {x: 5, y:10.0}; // Or can pass in one i32 and one f64 (where i32-> T and f64->U)
```

We can also use generics in enums, recall the Option and Result enum are implemented using generics.

```
fn main(){
    enum Option<T>{ // Use only one generic
        Some(T),
        None,
}

enum Result<T,E>{ // Two generics, T and E
        Ok(T),
        Err(E),
}
```

```
Generics in Method Definitions

struct Point<T>{
    x: T,
    y: T,
    }

impl<U> Point<U> { // Note we don't need to use the same name, implentation uses a generic and calls it U.
    self.x
}

impl Point<f64> { // Here the method is only defined for Points that have a type parameter of f64
    fn y(&self) -> f64 {
        self.y
    }

fn main() {
    let p = Point {x:5, y:10}; // x() is avaliable as a method but not y()
    let pl = Point {x:5, y:10}; // both x() and y() are available.
}
```

Finally let us talk about performance, generics allow us to reduce duplication without incuring a performance hit

```
let integer = Option::Some(5);
let float = Option::Some(5.0);
}
```

At compile time Rust will turn the Option enum into two enums one for i32 and one for f64