# Rust Course notes

# Razafy Rindra, "Hagamena'

# July 26, 2022

## Contents

	Introduction	3
	Common Programming Concepts in Rust           2.1         Variables and Mutability         2.1.1         Mutable variables         2.1.2         Constants         2.1.3         Shadowing         2.2         2.1.3         Shadowing         2.2         2.2         Data Types         2.2         2.3         Functions         2.4         2.4.1         if statements         2.4.1         if statements         2.4.2         Loops         2.4.2         Loops <td< th=""><th>3 3 3 3 4 4 4 4 4</th></td<>	3 3 3 3 4 4 4 4 4
	Ownership 3.1 Ownership Rules 3.2 Ownership and functions 3.3 References 3.4 Slices	5 5 6 7 8
		10 10 11
6		14 14 15 16 17 18 20
		20 21 21 22
8		23 23 24 25

9	Generics 9.1 A first example	27 27
10	Traits  10.1 Implementations	31 31 32 33
	Lifetimes  11.1 Genreric Lifetime annotations	34 34 35 36
12	Testing 12.1 Writing tests	37 37 40 40 41
	First Rust Project: Building a Command Line Program minigrep  13.1 Creating the program	42 43 44 45 46 47 48 49 50
	Closures 14.1 Capturing environment with closures	<b>51</b> 53
15	Iterators 15.1 Iterator trait	
16	Publishing a Crate  16.1 Release Profiles	60 60 60 61 62
	Cargo Workspace 17.1 External dependencies 17.2 Testing	62 63 63 63

# 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.

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 and associated functions

# 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

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.

fn x(&self) -> &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 p1 = Point {x: 5.0, y:10.0}; // both x() and y() are available.
}
```

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

```
enum Option<T>{
          Some(T),
          None,

None,

fn main() {
          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

# 10 Traits

**Definition 10.1.** Traits allow us to define a set methods that are shared across different types.

# 10.1 Implementations

```
This code outputs

Tweet summary: @user: Hello world

Article summary: (Read more from The Author...)
```

#### 10.2 Trait Bounds

```
%impl syntax and trait bound
    // Put in previous Implementations of Tweet, NewsArticle, Summary are to be put here

pub fn notify(item: &impl Summary){    // This function takes in a reference to something that implements summary
    println!("Breaking news! {}", item.summarize());
}

fn main(){
    \\ Put in the previous defintions of tweet and article
    notify(&article);
}
```

```
This code outputs

Breaking news! (Read more from The Author...)
```

This above & impl syntax is syntax sugar for what is called a "trait bound" which looks like this:

```
Trait Bound

// Put in previous Implementations of Tweet, NewsArticle, Summary are to be put here
pub fn notify<T:Summary>(item: &T){ // Generic that is limited to something that interprets a Summary
trait.

println!("Breaking news! {}", item.summarize());
} // Does the same as the &impl syntax
```

This syntax is useful if we want our function to take in multiple inputs of the same type:

```
Multiple inputs pub fn notify(item1: &impl Summary, item2: &impl Summary){ // Here item1 and item2 can be anything that implements Summary, but they can also be different from eachother.

//...

pub fn notify<T:Summary>(item1: &T, item2: &T){ // item1 and item2 are both of the same type &T which can be anything that implements Summary.

//...

// ...
```

```
Trait bounds to conditionally implement methods

struct Pair<T>{
    x: T,
    y: T,

impl<T> Pair<T>{ //This implementation block is for any pair struct.
    fn new(x: T, y: T) -> Self{ // Every pair struct will have this method
        Self {x,y}
    }

impl<T: Display + PartialOrd> Pair<T>{ // We use trait bounds to say that T has to implement display
    and partial order
    fn cmp_display(&self) { // only the struct that, where T implement display and partial order will
    have this function
    if self.x >= self.y{
        println!("The largest member is x = {}", self.x);
    } else{
        println!("The largest member is y = {}", self.y);
    }
}

}

}
```

```
Blanket implementation

// We can implement a trait on a type that implements another strait

impl<T: Display> ToString for T{ // We implement the ToString trait on any type that implement Display
//
} // We will talk about this later.
```

# 10.3 Return types

Now let us talk about return types

```
This code outputs horse_ebooks: of course, as you probably already know, people
```

# 11 Lifetimes

**Definition 11.1.** A dangling reference is a reference that points to invalid data. Rust does not like dangling references

r is a dangling reference, since it is referencing x which was invalidated after it went out of scope. Rust doesn't let this code complie. It knows at compile that x doesn't live long enough for us to reference it because of the borrow-checker.

When we print r here, r is referencing x whose lifetime is valid at that point, so we don't get any error,

The bottow-checker is able to figure all this out without help. Now we will talk about situations where we do need to help the compliler.

### 11.1 Genreric Lifetime annotations

longest returns a reference, but we don't know what the lifetime is. First of all x and y can have different lifetimes, secondly we don't know what are their lifetimes.

In order to fix this we need to use generic lifetime annotations

**Definition 11.2. Generic lifetime annotations** describe the relationship between lifetimes of different references and how they relate to eachother.

We call these generic lifetime annotations, "lifetimes".

```
Using Lifetimes

fn main() {

let string1=String::from("abcd");

let string2 = String::from("xyz");

let result = longest(string1.as_str(), string2.as_str());

println!("The longest string is {}", result);

fn longest<'a>(x:&'a str, y:&'a str) -> &'a str{ // What wer are saying is that the lifetime of the return reference is the same as the smallest lifetime of the argument.

if x.len()> y.len(){

x
}else{

y

}else{

y

}

}

}

}
```

How does the borrow-checker know that result is a valid reference? We just told the borrow checker that 'result' has the lifetime equal to the smallest lifetime passed in. So the borrow checker just needs to check that if result is called the smallest lifetime is still valid.

```
| Description with Children Ch
```

This code compliles, since the borrow checker just makes sure that result has the same lifetime as string!

#### 11.1.1 Structs with lifetime annotations

```
struct ImportantExcerpt < 'a>{
    part: & 'a str ,
}
fn main() {
```

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

| part: first_sentence,
| part: first_se
```

Variable i is only valid as long as first\_sentence is in scope, we will get an error if we try to use it after first\_sentence has left scope.

### 11.2 Lifetime elision rules

There are some times when the compiler can deterministically infer the lifetimes annotations by checking the **three lifetime** elision rules

**Definition 11.3.** Input lifetimes are the lifetimes of the argument that are passed in

**Definition 11.4. Output lifetime** are the lifetimes fo the arguments returned

#### Definition 11.5. Three rules:

- 1. Each parameter that is a reference gets its own lifetime parameter
- 2. If there is exactly one input lifetime parameter, that lifetime is assigned to all output lifetime parameters
- 3. If there are multiple input lifetime parameters, but one of them is &self or &mut self the lifetime of self is assigned to all output lifetime parameters.

THe compiler will try to follow these three rules, if it can't determine the lifetimes at the end of these three rules, we will have to manually specify them.

Output

- 1 Announcement! Grand finale
- 2 These past sections we did generics, traits and lifetimes, how fun!!!!

# 12 Testing

# 12.1 Writing tests

Why do we want to write tests? Rust checks that our code works correctly with borrow checker and checking types etc.. But it doesn't check that our functions are working in the way we intend them to. Which is why we need to make tests.

In Rust functions are tests if they have the #[test] attribute. In order to run our tests, we type in cargo test in the terminal. Our test will fail when something inside the test function panics.

In this case our two tests pass, if we changed a '>' into a '<' in the implematation of can\_hold, out larger\_can\_hold\_smaller test would fail.

Both paramaters passed in assert eq! and assert ne! must implement the PartialEq and Debug traits.

```
s use super::*;

#[test]
fn greeting_contains_name(){
    let result = greeting("Carol");
    assert!(
        result.contains("Carol"),
        "Greeting did not contain name, value was {}", result // Custom failure message
    );
}

// Custom failure message
```

```
Failure message - thread 'tests::greeting_contains_name' panicked at 'Greeting did not contain name, value was Hello'
```

Since we have the #|should panic| our tests pass.

Here our test passes only if when our failure message is what we have expected. If we change Guess::new(200) to Guess::new(-2) our test fails.

# 12.2 Running tests and organising into unit tests and integration test

#### 12.2.1 Configuring tests

By default all test are run in parallel in a seperated thread and all output is capture and not printed to screen.

There are two sets of command line options, one sets goes to the cargo test command and the other to the resulting test binary.

If we want to figure out which option we could pass to the cargo test command we type:

```
cargo test —help
```

And to figure out which command we can pass to our resulting test binary we type:

```
cargo test — — help
```

#### **Example 12.0.1.** If we want to run tests serially we can set the option like so:

```
cargo test — —test-threads=1
```

By default standard output is captured for passing test, so will only see printed output for failing tests. We can change that by using this command:

```
Showing output cargo test — —show-output
```

```
fn prints_and_returns_10(a: i32) -> i32{
    println!("I got the value {} {} {} {}^{"}, a);
    10

4 }

#[cfg(test)]
mod tests {
    use super::*;

#[test]
fn this_test_will_pass() {
    let value = prints_and_returns_10(4); // This code will print out if we use above command but
    not by default
    assert_eq!(10,value);
}

#[test]
fn this_test_will_fail() {
    let value = prints_and_returns_10(8); // This code will always print since the test fails
    assert_eq!(5, value);
}
```

In this example we can:

• Only run a specific test by specifiving it's name:

```
cargo test one_hundred
```

• Run tests with a common part of their name:

```
cargo test add
```

This will run tests for add\_two\_and\_two and add\_three\_and\_two

• Run tests by specifying the module:

```
cargo test tests::
```

Which will run all three tests in the module.

To run ignored test

```
cargo test — —ignored
```

#### 12.2.2 Test organisation

**Definition 12.1.** The Rust community puts tests in two main categories

- Unit tests, small, focused, tests one module in isolation and can tests private interfaces
- Integration tests, external to library so tests the public interface of library.

Up until now we have been writing unit tests, they live in the same file as our product code. Integration tests live in a folder called tests at the root of our project.

If we want to run just our integration test we can type:

```
cargo test — test integration_test
```

Because every file in test directory is treated like a seperate crate, it can cause unexpected behaviour.

Assume we have multiple integration test files and want to share some code between them. If we try to create a new file called common.rs where we will store the common code between the integration tests. Rust will treat it like another integration test file, which is not what we want.

To get the behaviour we want let us create a folder in our tests directory, called common and create a file in it called mod.rs and put in the shared code in this file.

This words since files in subdirectories of our test folder do not get compiled as crates. Furthremore, our shared code is now in a module that can be used by the other integration test files.

```
Using common code for integrated tests

use adder;

mod common; //Module declaration it will look content of module in either a file called common.rs or a folder called common with a file called mod.rs

#[test]
fn it_adds_two() {
   common::setup(); //Using the module here
   assert_eq!(4, adder::add_two(2));
```

```
mod.rs file
// Path /adder/tests/common/mod.rs
pub fn setup(){
// set up code
}

// set up code
```

Note we cannot directly test binary crates with integration tests, which is why it is common to see a binary crate that is a thin wrapper around a library crate, so that we can test the library crate with integration tests.

# 13 First Rust Project: Building a Command Line Program minigrep

We will use what we have learned so far to create a command line program which will is a simple verson of the tool grep(which allows us to search for a string in a file). We will call this program "minigrep".

# 13.1 Creating the program

First using args method to read arguments passed in command line use std::env; fn main() {
let args: Vec<String> = env::args().collect();
println!("{:?}", args);
}

The args() function gives us an iterator over the argumens passed by our progam, and collect() turns this iterator into a collection we can use, in this case a vector of strings.

By default we always get the path to our binary passed. In this case, though we don't care about the binary path and only the query and the file name.

Next step is to read in file.

We can read a file, now let us improve our program.

# 13.2 First problem: Extracting out our code

main.rs has too many responsibilities. What we should do is create a library crate and have our binary crate call functions from the library crate.

# Extracting out the argument parsing logic use std::fs; use std::env; the main() { let args: Vec<String> = env::args().collect(); let (query, filename) = parse\_config(&args); println!("Searching for {}", query); println!("In file {}", filename); let contents = fs::read\_to\_string(filename) .expect("Something went wrong reading the file"); println!("With text:\n{}", contents); fn parse\_config(args: &[String]) -> (&str,&str){ let query = &args[1]; let filename = &args[2]; (query, filename)

Now our tuple, doesn't tell us how are two values query and filename are related. So let us create a struct to make this relation more explicit.

```
Config strut
    use std::env;
    use std::fs;

# main() {
    let config= parse_config(&args);

# println!("Searching for {}", config.query);
    println!("In file {}", config.filename);

# let contents = fs::read_to_string(config.filename)
    .expect("Something went wrong reading the file");

# println!("With text:\n{}", contents);

# struct Config{
    query: String,
    filename: String,

# filename: String,

# let query = args[1].clone(); // Use the clone method since we don't want to take ownership

# let filename = args[2].clone();

# Config {query, filename}

# Config {query, filename}
```

Note this is not the most elegant way of doing this, since we are cloning our string. But better way of doing this would require us to use lifetimes, so while this is not efficient it is simpler. We will look at how to handle this more efficiently later.

There is more room for improvement, the parse\_config is very closely related to our Config struct, so let us make this relationship more explicit by turning by putting it in an impl block.

```
Constructor function
    use std::env;
    use std::fs;

a fn main() {
    let args: Vec<String> = env::args().collect();
```

```
let config= Config::new(&args);

println!("Searching for {}", config.query);
println!("In file {}", config.filename);

let contents = fs::read_to_string(config.filename)
.expect("Something went wrong reading the file");

println!("With text:\n{}", contents);

println!("With text:\n{}", contents);

struct Config{
   query: String,
   filename: String,
}

impl Config{
   fn new(args: &[String]) -> Config{ // Calling this function new is convention for constructor functions.
   let query = args[1].clone();
   let filename = args[2].clone();

Config {query, filename}
}
}
```

# 13.3 Second problem: Error handling

If we call our program and don't pass in enough arguments we don't get a useful error message.

```
thread 'main' panicked at 'index out of bounds: the len is 1 but the index is 1', src/main.rs:25:21
```

Let make this error message less confusing, by changing our Config::new function

```
fn new(args: &[String]) -> Config{
    if args.len() < 3{
        panic!("not enough arguments");
}

let query = args[1].clone();

let filename = args[2].clone();

Config {query, filename}

}</pre>
```

Now the error message is

```
thread 'main' panicked at 'not enough arguments', src/main.rs:26:13
```

Now this error message also has a lot of noise, since we are using panic! Let us fix this with the Result type.

```
Using the Resut type for error handling
```

```
use std::env;
use std::fs;
suse std::process; // Let's us exit program without panicking.

fn main() {
    let args: Vec<String> = env::args().collect();

    let config= Config::new(&args).unwrap_or_else(|err|{
        println!("Problem parsing arguments {}", err);
        process::exit(1);
});

println!("Searching for {}", config.query);
println!("In file {}", config.filename);

let contents = fs::read_to_string(config.filename)
.expect("Something went wrong reading the file");

println!("With text:\n{}", contents);
}
```

```
struct Config {
    query: String,
    filename: String,

    filename: String,

    impl Config {
        fn new(args: &[String]) -> Result < Config, &str > {
            if args.len() < 3 {
                return Err("not enough arguments");
            }
            let query = args[1].clone();
            let filename = args[2].clone();
            Ok(Config {query, filename})
            }
        }
}</pre>
```

Note the unwrap\_or\_else() is a function that takes in a closure, In the Ok case it returns the value stored in Ok, and in the Err case it will execute the closure passing it Err. We will learn more about closures later.

Now the error message is

```
Problem parsing arguments not enough arguments
```

# 13.4 Extracting some more logic from main.rs

```
use std::env;
use std::fs;
suse std::process;

fn main() {
    let args: Vec<String> = env::args().collect();

    ret config= Config::new(&args).unwrap or else(|err|{
        println!("Problem parsing arguments {}", err);
        process::exit(1);
});

println!("Searching for {}", config.query);
println!("In file {}", config.filename);

run(config);

fn run(config: Config){
    let contents = fs::read_to_string(config.filename)
    .expect("Something went wrong reading the file");

println!("With text:\n{}", contents);

// ...Config struct and impl here...
```

Now let us treat error handling in our run function so that we don't have to call expect() and panic if we get an error.

```
if let Err(e) = run(config){
    println!("Application error: {}", e); //If run returns error then we execture this code block.

process::exit(1);
}

fn run(config: Config) -> Result<(), Box<dyn Error>>{
    let contents = fs::read_to_string(config.filename)?; //If read_to_sting returns an Error type, that
    Error type will be automatically returned from run().

println!("With text:\n{}", contents);

Ok(())

// Put config struct and impl block here.
```

# 13.5 Extracting functions into a library crate

Our main.rs code is started to get bloated so let us extract run function and Confic struct with impl block into library crate

```
main.rs
    use std::env;
use minigrep::Config;

fn main() {
    let args: Vec<String> = env::args().collect();

    let config= Config::new(&args).unwrap_or_else(|err|{
        println!("Problem parsing arguments {}", err);
        process::exit(1);
});
```

```
println!("Searching for {}", config.query);
println!("In file {}", config.filename);

if let Err(e) = minigrep::run(config){
    println!("Application error: {}", e); //If run returns error then we execture this code block.

process::exit(1);
}

20
}
```

# 13.6 Test Driven Development

Want we want to do is write a test that fails, then implement the logic that would make the test pass and then if neccesary refactor our code and make sure our test still passes.

Currently our program just prints out the contents of our file, but we want our program to only print out lines in our file that include our query. To do this we want to create a function called search. But before creating that function, let us write a failing test.

Now let us first define the search funtion, returning an empty vector:

```
pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str>{// Recall we need to put in
lifetimes
vec![]
}
```

As expected our test will fail, since we returned an empty string. Now let us modify the search function to make this test pass.

Now let us modify our run function to make use of the search function.

```
new run function

pub fn run(config: Config) -> Result <(), Box<dyn Error>>{
    let contents = fs::read_to_string(config.filename)?;

for line in search(&config.query,&contents){
    println!("{}", line);
}

Ok(())

9 }
```

```
Output of our program with query: frog Farst Searching for frog In file poem.txt
3 How public, like a frog
```

```
Output of our program with query: dog

Searching for dog

In file poem.txt
```

#### 13.6.1 Improving our program with environment variables

Now our program is done but we can improve it. Currently our search logic is search sensitive, so let us give it the option to do case insensitive searching with environment variables.

First we add in a failing test

```
Creating our failling function pub fn search_case_insensitive<'a>(
query: &str,
contents: &'a str,
) -> Vec<&'a str>{
vec![]}
}
```

```
Making this test pass

pub fn search_case_insensitive<'a>(
query: &str,
contents: &'a str,

> > Vec<&'a str>{
let query = query.to_lowercase(); // Note to_lowercase() returns a new string, so we are not modifying strings.

let mut results = Vec::new();
for line in contents.lines(){
    if line.to_lowercase().contains(&query){
        results.push(line);
    }

results.push(line);
}

results

results
```

Now we will modify our struct to tell us when we want our search to be case sensitive or not, and we will use environment variables for this.

Now if we run our code normally:

```
cargo run to poem.txt

Searching for to

In file poem.txt
Are you nobody, too?

How dreary to be somebody!
```

Now set CASE INSENSITIVE to true:

```
export CASE_INSENSITIVE=true
```

```
cargo run to poem.txt

Searching for to

In file poem.txt

Are you nobody, too?

How dreary to be somebody!

To tell your name the livelong day

To an admiring bog!
```

#### 13.6.2 Standard Error

Command line programs our expected to send errors to Standard Error stream and not Standard output. Since if a user wants to send output stream to a file they will still send errors to screen,

Since now if we send our output stream into a file this is what happens:

```
cargo run > output.txt  Problem parsing arguments not enough arguments
```

Our output.txt file contains our error message. Let us fix this

Now if we do cargo run > output.txt we get the "Problem parsing arguments not enough arguments" error message in the terminal and the output.txt file is empty since there is no output.

```
cargo run to poem.txt > output.txt

Searching for to

In file poem.txt
Are you nobody, too?
How dreary to be somebody!
```

This is our output.txt file when we run our program without errors.

# 14 Closures

**Definition 14.1. Closures** are like functions but they don't have names. They can be stored in variables or passed in as parameters of a function.

We don't have to annotate the type of input and output values, the compiler is able to determine the types. As closures are usually short and used in a narrow context. But closures can only have one type inferred by the compiler.

In order to define structs, enums or function parameters that use closures we need to use generics and trait bounds.

```
Struct containing closure

struct Cacher<T>
where
T: Fn(u32)->u32, // Any function that takes in an u32 and returns a u32

calculation: T,
value: Option<u32>,

}
```

```
Closures can't have many types infered for each parameters |
| let example_closure = |x| x;
| let s = example_closure(String::from("hello")); // Compiler infers that the example closure takes in a string and outputs a string.
| let n = example_closure(5); //Error expected a string but got an integer.
```

**Example 14.1.1.** In this example, let us say that we are working on the backend of a fitness app, which makes workout plans based on specification by the user. We will simulate a function that makes an expensive calculation, and we will simulate the code that will create the workout.

This code works, but can use some refactoring. Since we are calling our expensive function in multiple places, if we change how this function is called we will have to change all the callsites. Furthermore we call our function multiple times when we don't need to, for example in the first "if block" we call it twice when we only need to call it once and pass in the return value in the print statements.

But notice we still call our expensive function when we may not need it (when intensity > 25 and the random number is 3). Let us fix this with closures.

#### Refactoring our function with Cacher

```
1
2 struct Cacher<T>
3 where
```

We might want to use our cacher in different contexts, but there are two problems prevent us from doing this:

- 1. Calling our value method will return the same value even if we input a different arg parameter. So we need to cache a different value for each of argument being passed in. We can fix this implematation by storing our values in a Hashmap. The keys are the arguments passed in and the values are the result of the expensive calculation with that argument pass in.
- 2. Another problem is that we are using hard coded types, we are saying that our program must take in integers and output integers. We can fix this by using generics.

#### 14.1 Capturing environment with closures

Unlike function, closures have access to variables in the scope where the closure is defined. For example:

```
fn main(){
    let x = 4;
```

Now what happens if we change closure to function?

```
fn main() {
    let x = 4;

    fn equal_to_x(z: i32) -> bool {
        z == x //Error: can't capture dynamic environment in a fn item use the `|| { ... }` closure form instead
    }

let y = 4;

assert!(equal_to_x(y));

assert!(equal_to_x(y));
```

Since closure can capture environment they need to use extra memory to store that context. Since function don't capture their environment they don't need that overhead.

Closure capture their environment in three ways, which correpsond to the three ways a function take parameters

- 1. Taking Ownership
- 2. Borrowing mutably
- 3. Borrowing immmutably

These three ways are encoded in the function traits:

- 1. FnOnce, FnOnce takes ownership of the variables inside the closures environment. Since closures can't take ownership of the same variables more than once, these closures can only be called once.
- 2. FnMut, Mutably borrows values.
- 3. Fn, Immutably borrows values.

Rust infers which of these traits to use when we create a closure based on how we use the values inside the closures environment.

But we can, force the closure to take ownership of the values it uses inside it's environment by using the move keyword

```
fn main() {
    let x = vec![1,2,3];

    let equal_to_x = move |z| z == x; // Closure takes ownership of x

    println!("Can't use x here: {:?}", x); // Get error here since we are using a borrowed value after it has been moved.

let y = vec![1,2,3];

assert!(equal_to_x(y));

11
}
```

#### 15 Iterators

**Definition 15.1.** The **Iterator pattern** allows us the iterate over a sequence regardless how the sequence is stored (arrays, vectors, maps, custom data structures). Iterator encapsulates the logic of how to iterate over these structures.

```
Iterator over vector
    fn main() {
        let v1 = vec![1,2,3];
        let v1_iter = v1.iter();

for value in v1_iter{
        println!("Got {}", value);
        }
}
```

```
Output | Got 1 | Got 2 | Got 3 | Got 3
```

#### 15.1 Iterator trait

How do iterators work? All iterators implement the iterator trait

```
pub trait Iterator{
    type Item; // associated type
    fn next(&mut self) -> Option<Self::Item>;

// methods with default implematation elided
}
```

So we only need to implement the next method which returns the next item in the iteration wrapped in Some and when we reach the end it returns None.

```
Test example

#[test]
fn iterator_demonstration(){
    let v1 = vec![1,2,3];

    let mut v1_iter = v1.iter();

    // .iter immutatable reference
    // .iter_mut mutable reference
    // into_iter to own types

assert_eq!(v1_iter.next(), Some(&1));
    assert_eq!(v1_iter.next(), Some(&2));
    assert_eq!(v1_iter.next(), Some(&3));
    assert_eq!(v1_iter.next(), None);
}

assert_eq!(v1_iter.next(), None);
}
```

**Definition 15.2.** There are two types of methods on iterators

- 1. Adaptators they take in an iterator and return another iterator
- 2. Consumer they take in a iterator and returns another type.

```
The consumer method, sum

#[test]

fn iterator_sum(){
    let v1 = vec![1,2,3];

let v1_iter = v1.iter();

let total: i32 = v1_iter.sum();

assert_eq!(total,6);

}
```

```
The adaptor method, map | fn main() { | let v1: Vec<i32> = vec![1,2,3]; | v1.iter().map(|x| x+1); // map takes in a closure and applies it to the elements of iterator | }
```

This returns a warning, in Rust iterators are lazy and do nothing unless consumed. Let us use the consumer method collect which takes our iterator and transformes it into a collection.

```
Collect | fn main() { | let v1: Vec<i32> = vec![1,2,3]; | let v2: Vec<_> = v1.iter().map(|x| x+1).collect(); | assert_eq!(v2,vec![2,3,4]); // true | } |
```

```
fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe>{
    shoes.into_iter().filter(|s| s.size == shoe_size).collect()
    // into_iter -> create an iterator that takes ownership
    // filter takes in closure and creates a new iterator, with only the elements that return true when
```

#### 15.2 Creating our own iterators

The zip method takes in two iterators and zips them up into one iterator containing pairs of values. The first iterator is the one in which the method is called on and the parameter. skip(n) is an adaptor method that creates a new iterator that skips the n elements.

### 15.3 Using Iterators to refactor the CLI program we made

Note in our CLI program we created an vector of strings called args using env::args.collect(), but env::args is an iterator. In our implementation block for config we had to use the clone() method to get the query and filename since args is a reference of to a [String] so we can't take ownership. Since we have ownership of iteratros, let us use them to remove this clone operation.

```
Improved Config
impl Config{
    impl Config{
        pub fn new(mut args: env::Args) -> Result < Config, & 'static str > {
            args.next(); // Returns our first command line argument, we are discarding it since it is the path
        to our program which we don't care about.

d

let query = match args.next() { // query is taking ownership of it's string
            Some(arg) => arg,
            None => return Err("Didn't get a query string"),
            };

let filename = match args.next() { // filename is taking ownership of it's string
            Some(arg) => arg,
            None => return Err("Didn't get a filename"),
            };

let case_sensitive = env::var("CASE_INSENSITIVE").is_err();

Ok(Config {query, filename, case_sensitive}) // Ok is taking ownership of the strings.
}
```

# Improved search function with iterator adaptors pub fn search < 'a>(query: &str, contents: & 'a str) -> Vec<& 'a str>{ contents .lines() .filter(|line| line.contains(query)) // filter to lines that contains our query .collect() // Here rust knows what to collect into since it is specified as return type.

```
tune Hall.Is He |
use std::process;
use iterators::Config;
fn main() {
    let config= Config::new(env::args()).unwrap_or_else(|err|{
        eprintln!("Problem parsing arguments {}", err);
        process::exit(1);
}

println!("Searching for {}", config.query);
println!("In file {}", config.filename);
if let Err(e) = iterators::run(config){
        eprintln!("Application error: {}", e);
        process::exit(1);
}

process::exit(1);
}
```

# 15.4 Loops vs Iterators

Is there a performance impact to using loops vs iterators? **NO**, using higher level abstractions like iterators over loops doesn't have an impact on performance. So using iterators and loops is equivalent. In general people prefer to use iteratros since it is a higher level of abstraction and we get access to useful methods.

# 16 Publishing a Crate

#### 16.1 Release Profiles

Definition 16.1. Release Profiles allow you to configure how your code is compiled. There are two profile

- 1. **Dev** defined with good default for development.
- 2. Release define with good defaults for release.

If we run cargo build, it compiles with dev profile if we run cargo build –release it compiles with release profile. We can customise the setting in the cargo.toml file.

```
package | name = "my_crate"
    version = "0.1.0"
    edition = "2021"

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

[dependencies]

[profile.dev]
    opt-level = 0

profile.release | opt-level = 3
```

Here opt-level tells the level of optimization we want, 0 being the least and 3 being the most. We can modify the level as we want. The full list of setting is on cargo documentation.

#### 16.2 Uploading code to crates.io

Before talking about how to upload our code to carates.io, we wiil show how to prepare our code.

#### 16.2.1 Preparing our code

**Documentation Comments** Documentation comments are useful when documenting public api so that others know how to use our code. Documentation code starts with three slashes and uses Markdown for formatting. Rust will turn our documentation into a html format that is easy to read.

In first line we explain what the function does and then we have an example section that gives an example of how to use our function. Note Rust will run our assert statement as a test, if we run cargo test. This forces our documentation to be in-synced with the code.

In order to build the html documentation for our crate we write:

```
cargo doc —open
```

#### Other commonly used section

- 1. Panics: The scenarios in which the function being documented could panic.
- 2. Errors: If the function returns a Result, describing the kinds of errors that might occur and what conditions might cause those errors.
- Safety: Explains why our function is unsafe and conditions when our function panics.

**Documentating item containing the comment** If we use //! we will instead of documenting the item following the comment it documents the item containing the comment in this example in lib.rs:

```
//! My Crate
//! 'my_crate' is a collection of utilities to make perfoming certain
//! calculations more convenient
```

So this is documenting our library crate.

# 16.3 Exporting a public API

We have an example where we create an Art module that let's us mix colours:

Here is how we would access these functions in main.rs

```
use my_crate::kinds::PrimaryColor;
use my_crate::utils::mix;

fn main() {
    let red = PrimaryColor::Red;
    let yellow = PrimaryColor::Yellow;
    mix(red,yellow);
}
```

Let's say that we want users to have access of our enums and mix function at the top level without having to mention their modules.

```
Use the "pub use" keyword pub use self::kinds::PrimaryColor;
pub use self::kinds::SecondaryColor;
pub use self::utils::mix;
```

If we go back to main.rs we can import our items from top level of our library:

```
use my_crate::PrimaryColor;
use my_crate::mix;
```

pub use statement allow us to make the public api different from the internal structure of our program.

# 16.4 Publishing to crate.io

Currently we must have a github account to log into crate.io, we need to go to the account setting to get an API token (which we should not share with anyone) and run

```
cargo login tokenabcsdfefefmakcn
```

Which will log us in, after we are successfuly logged in. Now we can publish but before we do that we must double-check the meta data. When we are developping uniquely our name does not matter but when publishing to cargo.io our name must be unique. So we must go to crates.io and check that our desired name isn't taken. In this case our name is "my\_crate". We also need a description and a licence.

Publishing to crates.io is permanant! If we want to upload a new version we can update the version number in cargo.toml using the sematic versioning rules. While we can't delete or modify our crates on cargo.io we can stop certain version from being used with the

```
cargo yank --vers *version_number*
```

So for anyone with this version in their cargo.lock file they can keep using this version. But those downloading our library for the first time won't be allowed to download this version.

If we want to undo our yank just do:

```
cargo yank ---vers *version_number* ---undo
```

# 17 Cargo Workspace

**Definition 17.1. Cargo Workspaces** help us manage multiple related packages that are developped in tandem. They share common dependencies resolution by having one Cargo.lock file, and they share one output directory and various settings like profiles.

**Definition 17.2.** Packages in our workspace are called **Workspace members** 

```
Setting up workspace [
| [workspace]
| adder", |
| "add-one", |
| one | add-one |
|
```

Note the Cargo.lock and target directory are made at the root of our workspace. If we want our adder directory to use our add-one file we need to put it in the dependencies of the adder Cargo.toml file:

```
In add/adder/Cargo.toml [dependencies] add-one = {path = "../add-one"}
```

And in main.rs we write:

```
In add/adder/src/main.rs

use add_one;

fn main() {
    let num = 10;
    println!(
        "Hello, world! {} plus one is {}!",
        num,
        add_one::add_one(num)
    );

10 }
```

In order to run our adder package we write

```
cargo run —p adder
```

# 17.1 External dependencies

If we add a dependency in adder and add-one package they will resolve to the same version.

So we can add in rand as a dependency in add-one and when we do cargo build it will be added to the Cargo.lock file. Even though rand is a dependency of our add-one package, it doesn't mean we can use rand in every package.

If we try to bring rand into the scope of main.rs in adder folder we get an error, we must add rand in dependencies of adder Cargo.toml file.

# 17.2 Testing

If we want to run tests just for add-one write

```
cargo test -p add-one
```

#### 17.3 Homework

Add another library add-two, which has a another function similar to add-one but adds two.

# 17.4 Publishing

If we want to publish a package withing workspace we must publish them individually. So cd in each directories and run cargo publish in each directory.

# 18 Installing Binaries from crates.io with cargo install

This is a convenient way for Rust developers to use tools built by other rust developers and published to crates. IoResul *Remark*. We can only install packages with binary targets. We need something we can execute.

All binaries installed using cargo install are installed in the installation root's bin directory.

**Example 18.0.1. Installing ripgrep** For example we can install Rust's version of grep ripgrep like this:

```
cargo install ripgrep
```

The penultimate line of the output tells us where our package was installed and the last line tells us the name of of our executable.

One cool thing with these binaries is that we can use them to extend cargo with custom commands. If we have a binary in our path prefixed by cargo, i.e. cargo-something. We can run this as a subcommand of cargo by typing

cargo something