Rust Course notes

Razafy Rindra, "Hagamena"

August 18, 2022

Contents

	Introduction
	Common Programming Concepts in Rust 2.1 Variables and Mutability 4 2.1.1 Mutable variables 4 2.1.2 Constants 4 2.1.3 Shadowing 4 2.2 Data Types 4 2.3 Functions 5 2.4 Control Flow 5 2.4.1 if statements 5 2.4.2 Loops 5
	Ownership 3.1 Ownership Rules 6 3.2 Ownership and functions 7 3.3 References 8 3.4 Slices 8
	Structs 4.1 Defining and Using Structs
	Enums and Pattern matching 12 5.1 Defining Enums 15 5.2 Option Enum 15 5.3 Pattern Matching 14 5.4 Using if let syntax 14
6	Module 18 6.1 Packages and Crates 18 6.2 Defining Modules 18 6.2.1 Definitions, Paths and Privacy 16 6.3 Privacy rules when it comes to structs 17 6.3.1 Privacy rules when it comes to enums 18 6.3.2 Use keyword 19 6.3.3 Modules in sperate files 21
	Common Collections 21 7.1 Vectors 25 7.2 String 25 7.3 Hash maps 25
8	Error handling 24 8.1 Panic! 24 8.2 Result Enum 25 8.3 Error propagation 26

9	Generics	2 8
10	Traits	32
	10.1 Implementations10.2 Trait Bounds	
	10.2 Halt Bounds	
	10.5 Return types	
	Lifetimes	35
10		
	Testing	38
	12.1 Writing tests	
	12.2.2 Test organisation	
	First Rust Project: Building a Command Line Program minigrep	43
	13.0.2 Standard Error	
14	Closures	52
15	Iterators	55
	15.3 Using Iterators to refactor the CLI program we made	
	15.4 Loops vs iterators	
16	Publishing a Crate	61
	16.1 Release Profiles	
1 17	Course Worksman	
	Cargo Workspace 17.1 External dependencies	63 64
	17.1 External dependencies	64
	17.3 Homework	64
18	Installing Binaries from crates.io with cargo install	64
1.0		
19	Box smart Pointer	65
	19.1 Box	
	19.1.1 Box pointer and List enum	
		66 66
	19.2.1 Dereference operator 19.2.2 Deref Coercion	67
	19.3 Drop Trait	68
		69
	19.4 Reference Counting	0.9

	19.5.3 Combining Rc with RefCell to get multiple owners of mutable data	
20	Fearless Concurrency	75
	20.5 Building concurrency features	
0.1		
	Object Oriented Programming features in Rust	81
22	Trait objects	82
	22.1 Static vs Dynamic dispatch	
23	Implementing the state object oriented pattern	84
24	Patterns ————————————————————————————————————	87
	Advanced topic	89

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 immutable 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.

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;
}

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. Immutable reference don't expect the underlying data to change. But we can have many immutable 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 immutable 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 immutable 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.

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

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

s    }

s    s   }

fn main() {}
```

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[..]
}

&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

```
fn main(){
    /* If we have a value that can possibly be null, we wrap it in the options enum.
    enum Option<T>{
        Some(T), //Some store any value
        None, // None store no value
    }
    This forces the type system to enforce that we handle the None case and guarentees that a value exists in the Some case.*/

    let some_number = Some(5);
    let some_string = Some("a string");

let absent_number: Option<i32> = None; // We need to annotate the type since no value is passed in so Rust can't infer the type.

let x: i8 = 5;
    let y: Option<i8> = Some(5);

// let sum = x+y; This code gives an error, since we can't add i8 to an Option<i8>

// In general we need to treat cases if Option is Some or None.

let sum =x+y.unwrap_or(0); // If y is Some it adds, if y is None it treats it as if it was 0.
```

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:
//
// We can do this:
// We can do this:
// Use std::io::{self, Write};
// 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();
}
```

```
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.

n 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:

```
Code with error

fn main() {
    a();
    a();
}

fn a() {
    b();
}

fn b() {
    c(22);

in fn c(num: i32) {
    if num == 22 {
        panic!("Do not pass in 22!");
    }
}
```

```
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::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 create 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| {

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

})

} else {

panie!("Problem opening 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**

A first attempt at dealing with errors with error propagation | #![allow(unused)] | use std::fs::File; | use std::io::{self, Read}; | fn read_username_from_file() -> Result<String, io::Error> { | let f = File::open("hello.txt"); | let mut f = match f { | Ok(file) => file, //If opening the file succeds we take that file and store it in f | Err(e) => return Err(e), // If opening the file fails we take the error and return it | }; | let mut s = String::new(); | match f.read_to_string(&mut s) { // Read the contents of the file and store it in the string | | Ok() => Ok(s), | | Err(e) => Err(e), | | } //read_to_string returns a result type, if we have success we return the string in error we return error. | Proceedings | Proceedings | Proceedings | Procedure | Procedu

```
#![allow(unused)]
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut f = File::open("hello.txt")?; //The ? operator says, if we succed in opening the file is returned and stored in f, if we fail it reurns error
    let mut s = String::new();

f.read_to_string(&mut s)?; // If this call fails, the function will return error
    Ok(s) //If we get to this line, read_to_string succeeded so we return the string
}
fn main() {}
```

This code gives us the same output as in the previous screen.

Reducing our codes to three lines

```
#![allow(unused)]
use std::fs::File;
use std::io::{self, Read};

fn read_username_from_file() -> Result<String, io::Error> {
    let mut s = String::new();
    File::open("hello.txt")?.read_to_string(&mut s)?;
    Ok(s)
}
fn main() {}
```

Can our code get simpler? Yes, we can reduce it to one line by bringing fs into scope

Ultimate implematation

What happens if we try to use our? operator in the main function? Since our main function doesn't return anything we get an error:

```
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(())

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:

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
| 4 |
| 5 |
| 6 |
| 7 |
| 8 | let p1 = Point {x:5, y:10}; // Can pass in two i32 |
| 9 | let p2 = Point {x:5.0, y:10.0}; // Can pass in two f64 |
| 10 | let p3 = Point {x:5, y:10.0}; // Or can pass in one i32 and one f64 (where i32-> T and f64->U)
| 11 |
| 12 | |
```

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.
    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 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:

```
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

```
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

1 }

}

}
```

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 | Continue | Contin
```

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,
    };
}
```

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:

```
ı 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 specifying 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 } }
```

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 funcitons 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.

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;
| 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;
use 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);

fn run(config: Config){
    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);

capacity

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

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.

```
Error handling

use std::env;

use std::fs;

use std::process;

use std::error::Error; // Im,port the Error type

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) = 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

```
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

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![]}
}
```

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.

```
privionment variables
pub in run(config: Config) -> Result<(), Box>dyn Error>>(
let contents = is:read_to_string(config.filename)?;

let results = if config.case_sensitive{
    search&config.query, &contents)
} else {
    search_case_Insensitive(&config.query, &contents)
};

for line in results {
    println!("{}", line);
}

Ok(())

pub struct Config {
    pub query: String,
    pub flename: String,
    pub flename: String,
    pub flename: String;
}

pub fineware: String,
pub case_sensitive: bool,
}

let query = args | 1, clone();
return Err("not enough arguments");
}

let query = args | 1, clone();
let case_sensitive = env::var("CASE_INSENSTIVE").is_err(); // Takes in a key to environment and returns Result, if the key exists and is set the Result will be Ok containing set value, otherwise the Result will be an error

// .is_err() returns a boolean. If Result was Ok it returns true, otherwise it returns false.

Ok(Config {query, filename, case_sensitive})
}
```

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

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 correspond to the three ways a function take parameters

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

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.

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 adaptor method, map for main() { for 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 4 }
```

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 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

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 will 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.
- 3. 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 order to run our adder package we write

```
ı 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. IoResult 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
```

19 Box smart Pointer

Definition 19.1. A **pointer** is a general concept for a variable that stores a memory addresse that referes or "points" to some data in memory. The most common type of pointer in Rust is a **reference** they borrow the values they point to, so they don't take ownership. References don't have any special capabilities so they don't have any overhead. Unlike **Smart Pointers** which is a data structure that acts with a structure with metadata and extra capabilities tacked on. In many cases Smart Pointers own the data they point to. Example of smart pointers are *String* and *Vectors*. Smart pointers are implemented using structs with the deref and drop trait. The **deref** trait lets instances of smart pointer struct to be treated as a reference. The **drop** trait allows you to customise the code that is run when an instance of our smart pointer goes out of scope.

We will only talk about the most common smart pointers in Rust.

19.1 Box

Definition 19.2. Box is a smart pointer that allows you to allocate values on the heap.

```
Example for main() {
let b=Box::new(5); // Storing 5 on the heap and on the stack we are storing memory adress to value of 5 on the heap.
println!("b = {}", b);
}
```

When our Box goes out of scope it will be de-allocated, the box smart pointer on the stack will be de-allocated but also the data on the heap will be de-allocated.

Box's don't have any overhead except storing the data on the heap, they also don't have many other capabilities.

Typical uses of Box

- When we have a type whose exact size can't be known at compile time but what to use value in context that needs to know exact size
- When we have large amount of data and what to transfer ownership but not copy the data.
- When we own a value and only care that value implements a specific trait rather then it being a specific type. This is known as a **Trait object**.

Now we will look at a practical example of when we want to use the Box pointer.

19.1.1 Box pointer and List enum

Definition 19.3. The **Cons list** is a data structure that comes from Lisp. Where each element in the list holds the value of the list and points to the next value until we reach the end of the list and point to Nil.

```
List enum | enum List{ //Error recursive type has infinite size | Cons(i32, List), | Nil, | Nil, |
```

Rust needs to know how much space this will take up at compile time but Rust doesn't know in this case, since our list can be "arbitrarily" large. This is a problem the Cons list shares with all other recursive data types.

To understand how to fix this with the Box pointer, we must first understand how computes the size of non-recursive enums.

```
non-recursive enum | enum Message {
    Quit ,
    Move {x: i32 , y: i32 } ,
    Write (String) ,
    Change Color (i32 , i32 , i32 ) ,
    }
```

Rust will figure out the size needed to store an instance of Message, it will go through each variant and see at how much space each variant needs. Rust will figure out which variant uses the most amount of space this is the most space Message will take up (since we can only use one variant at time per instance).

But now for our list Enum we notice that the Cons variant stores a tuple type consisiing of an integer and a list enum, so rust must know how much space the list enum takes!

How do we fix this?

Why did this fix our error? Well if we look at each variant to see how much space List takes up we see that "Nil" takes no space and "Cons" stores an integer and a Box Smart Pointer, which is a fixed size pointer, while it points to some arbitrarily amount of data on the heap on the stack it is of fixed size.

```
Implementation
| enum List{
| Cons(i32, Box<List>),
| Nil,
| 4 |
| 5 |
| 6 | use List::{Cons, Nil};
| 7 |
| 8 | fn main() {
| let list = Cons(1,Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
| 0 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
| 1 |
|
```

Our code compiles without errors.

19.2 Deref Triat

19.2.1 Dereference operator '*'

Definition 19.4. For references the **deference operator** follows memory address stoered in reference to the actual value.

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

assert_eq!(5,x); //True
    assert_eq!(5,*y); //True
    assert_eq!(5,y); //Error can't compare `{integer}` with `&{integer}`
}
```

We can modify this example to use smart pointers, specifically the smart pointer Box.

```
Now with the Box Smart Pointer fn main () { let x = 5; let y = Box::new(x); assert_eq!(5,x); assert_eq!(5,*y); // Derefence operator works the same since Box implements the deref trait } }
```

Let us build our own Box pointer to understand how the deref trait works in more detail.

The deref trait allows the Rust compiler to take any value that implements deref call the deref method to get a reference which the compiler knows how to dereference.

More clearly at the last line of our above example here is what the compiler calls:

```
assert_eq!(5,*(y.deref()));
```

This allows us to treatregular reference and types that implements the deref trait the same.

Why doesn't deref return the value itself? The ownership system! If Rust returns the value directly, it will move ownership of the value outside of our smart pointer.

19.2.2 Deref Coercion

Definition 19.5. Deref Coercion is a convenience feature in Rust that happens automatically for types that implement the deref trait, deref coercion will convert a reference of one type to a reference of a different type.

```
Example |
// MyBox implementation here
fn main () {
    let m = MyBox::new(String::from("Rust"));
    hello(&m);
}

fn hello(name: &str) {
    println!("Hello, {}!", name);
}
```

This code works even if &m is a reference to my box

Rust sees that the type being passed to hello is different than the type expected by the function, and automatically performs these chained deref calls to get the correct type.

If Bust didn't have automatic deref coercion we would have to write our code like this

```
n hello(&(*m)[...]);
```

Definition 19.6. Deref Mut:

Similar to how we use the deref trait to override deref operator for immutatable references, we can use the DerefMut for mutable references.

Rust does deref coercion in the following cases:

• Immutable ref to immutable ref

- mutable ref to mutable ref
- mutable ref to immutable ref

Rust cannot perform deref coercion when going from immutable ref to mutable ref due to the borrowing rules, we can only have one mutable reference to a specific piece of data in a specific scope.

19.3 Drop Trait

Definition 19.7. The **drop trait** can be implemented on any type and it tells you what to do when the value goes out of scope. It is mostly used with smart pointers, for example in the box pointer it tells the compiler to deallocate the data stored on the heap.

```
Example struct CustomSmartPointer{
    data:String,

}

impl Drop for CustomSmartPointer{ // Drop trait requires that we implement the drop method.
    fn drop(&mut self) {
        println!("Dropping CustomeSmartPointer with data `{}`!", self.data);

}

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

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

println!("CustomSmartPointers created.");

}
```

```
Output CustomSmartPointers created.

1 CustomSmartPointer with data `other stuff`!

2 Dropping CustomeSmartPointer with data `my stuff`!
```

What happens if we want to customise the cleanup behaviour? Like if we want to clean up early, for example if we are managing locks, we might want to drop the smart pointer to release the lock so that other code can use the data.

We are not allowed to use the drop method manually since when our variable goes out of scope, Rust will call the drop method again which may lead to a double free.

To manually cleanup the value early, we call the drop function:

```
Output | CustomSmartPointers created.

Dropping CustomeSmartPointer with data `my stuff `!

Dropped before end of main

Dropping CustomeSmartPointer with data `other stuff `!
```

19.4 Reference Counting

Definition 19.8. A **reference counting smart pointer** allows us to share ownership. While most of the time ownership is clear, there are cases where a single value has mutliple owners, for example in a graph a node is owned by all the edges pointing to it, and should not be cleaned up unless no edges point to it. So we use a reference counting smart pointer which keep track of the number of references to a value and when there are no more reference the value is cleaned up. An analogy is when watching tv, we only turn off the tv when everyone in the room has left, if we turn it off while there are still people in the room, there will be panic.

The referece counting smart pointer is used when we have a value on the heap and multiple parts of our program read that value and we don't know which part of our program is going to use our value last at compile time.

Remark. The reference counting smart pointer we will see is only used in single threaded programs, we will later see how to use reference counting in multi-threaded programs.

```
Trying to use Box pointer

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

}

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

fn main() {
    // b and c both point to a
    let a = Cons(5, Box::new(Cons(10, Box::new(Nil))));
    let b = Cons(3,Box::new(a)); // Value of a moved here
    let c = Cons(4,Box::new(a)); // Error, value used after move

| Cons(i32, Box<List>),
    Nil,
    |
    |// b and c both point to a
    |
    |// b and c both point to a
    |/
```

```
Using reference counting

use std::rc::Rc;

enum List{
    Cons(i32, Rc<List>),
    Nil,

}

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

fn main() {
    // b and c both point to a
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));

let b = Cons(3,Rc::clone(&a));

let c = Cons(4,Rc::clone(&a));

}
```

Here we want both b and c to point to the Cons list a. We can't pass in a reference to a in b since we expect an owned type, so we will have mismatched types. And we can't pass in a directly since then we will move ownership of a into b. So the answer is to use the clone method, here the clone method doens't deep copy the value it only increasess the reference count.

```
In more details
fn main() {
    let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
    println!("count after creating a = {}", Rc::strong_count(&a)); // 1
    let b = Cons(3,Rc::clone(&a));
    println!("count after creating b = {}", Rc::strong_count(&a)); // 2
    {
        let c = Cons(4,Rc::clone(&a));
        println!("count after creating c = {}", Rc::strong_count(&a)); // 3
    }
    println!("count after c leaves scope = {}", Rc::strong_count(&a)); // 2
}
```

Remark. The reference counting smart pointer, let's multiple parts of our program to read the same data but not to modify it. If we allowed multiple mutable references we would be violating the borrowing rules.

19.5 Interior mutability

Definition 19.9. Interior mutability is a design pattern that let's us to mutate data even when there are immutable reference to that data. To do so it uses unsafe code inside a data structure to bypass the borrowing rules. We will learn more about unsafe code later.

Even though the borrowing rules are not enforced at compile time they can still be enforced at run time.

19.5.1 RefCell Smart pointer

Definition 19.10. The **RefCell** Smart pointer represents single ownership over the data it holds, like the Box smart pointer But the Box enforces borrowing at compile time while RefCell enforces borrowing rules at run time.

The advantages of checking the borrowing rules at run time is that certain memory safe scenarios are allowed which would be dissallowed at compile time. The RefCell smart pointer is useful when you are sure that you are following the borrowing rules but the compile can't tell that.

Remark. We can only use RefCell in single threaded programs. We will look at what to do for multi-threaded programs later.

Mutating a value inside an immutable value is called the interior mutability pattern.

```
fn main() { let a = 5; let b = &mut a; //Error: a is immutable so cannot borrow as mutable let mut c = 10; let d = &c; *d = 20; //Error: d is an immutable reference so the data it referes to cannot be written }
```

We can solve this with some inderection. We can use a data structure that stores some value, where the value in the data structure is mutable but when we get a reference to the data structure the reference is immutable. So code outside the data structure can't mutatate the value, but we can call methods on the data structure to mutate the data.

This is what the RefCell smart pointer does with one caviate, instead of calling methods to mutate the data we call methods to get a mutable or immutable reference of the data.

19.5.2 Use case for interior mutability pattern

This is a library that tracks a value against maximal value and sends message depending on ratio between value with maximal. Let us implement our tests:

We need an mutable reference but that would break the function signature of send defined in our messenger trait. In this situation we can use the RefCell.

Now recall that RefCell checks the borrowing rules at runtime, so what happens when we have two mutable references a the same time?

```
Two borrows impl Messenger for MockMessenger {
    fn send(&self, message: &str) {
        let mut one_borrow = self.sent_messages.borrow_mut();
        let mut two_borrow = self.sent_messages.borrow_mut();

        one_borrow.push(String::from(message));
        two_borrow.push(String::from(message));

    }

} // Tests fails: thread 'tests::it_sends_an_over_75_percent_warning_message' panicked at 'already borrowed: BorrowMutError', src/lib.rs:60:53
```

So the interior mutability pattern gives us flexability but we must still follow the borrowing rules.

19.5.3 Combining Rc with RefCell to get multiple owners of mutable data

All tests pass

```
println!("a after = {:?}", a);
println!("b after = {:?}", b);
println!("c after = {:?}", c);
}
```

```
Output | a after = Cons(RefCell { value: 15 }, Nil) | b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil)) | c after = Cons(RefCell { value: 4 }, Cons(RefCell { value: 15 }, Nil))
```

19.5.4 Recap of the smart pointers we have seen

- Rc: enables multiple ownership of the same data; allows only immutable borrows checked at compile time.
- Box: Allow single ownership to piece of data; allows immutable or mutable borrows checked at compile time
- RefCell: Allows single ownership to piece of data; allows immutable or mutable borrows checked at runtime. (So we mutate value inseide RefCell<T> even when RefCell<T> is immutable)

19.6 Reference Cycles

Rust is known for being a memory safe language, so it provides certain guarentees like you can't have data races. But it doesn't guarentee that we will not have memory leaks. We can create a memory leak with the Rc and RefCell smart pointers, with these two smart pointers we can create references where items reference eachother in a cylce. Which will create a memory leak.

```
 \begin{array}{l} \textbf{if let Some(link)} = \texttt{a.tail()} \{ \text{ // udr if let since we only care about Some} \\ *\texttt{link.borrow\_mut()} = \texttt{Rc::clone(\&b): // We get mutable ref to the data and use dereference operator} \\ \end{array}
```

We have created a cycle since list a reference list b and list b references list a. More clearly the tail of list a points to list b but the tail of list b points to list a, so when we try to print it on the last line, to print list a it will print list b, but to print list b it will print list a, etc...

These circular dependencies also cause a memory leak. Indeed at the end of main a and b will get cleaned up, when b get's cleaned up the memory on the stack is not cleaned up since it is still reference in list a. Then a gets cleaned up but the memory on the heap doesn't get cleaned up since it is reference in the tail of list b.

So we have data on the heap but we don't have any stack variables pointing on this list.

19.6.1 Tree structure without referece cycles

Definition 19.11. The **Weak** smart pointer is a version of the Rc smart pointer that holds a non-owning reference

```
use std::cell::RefCell;
use std::re::[Re, Weak];

#[derive(Debug)]
struct Node{
    value: j32.
    parent: RefCell<Weak<Node>>,//Can't use Re since it will cause a reference cycle.
    children: RefCell<Vec<Re<Node>>>,
}

in main() {
    let leaf = Rc::new(Node{
        value: 3.
        parent: RefCell::new(weak::new()).
        children: RefCell::new(vee![]).
};

println'("leaf parent = {:?}". leaf parent borrow().upgrade()): //upgrade attempts to turn Weak into
Re, returns None if value was dropped otherwise it returns Some with Re

let branch = Re::new(Node{
        value: 5.
        parent: RefCell::new(Weak::new()).
        children: RefCell::new(Weak::new()).
        children: RefCell::new(Weak::new()).

* value: 5.

* parent: RefCell::new(Weak::new()).

* children: RefCell::new(Weak::new()).

* value: 5.

* parent: RefCell::new(weak::new()).

* value: 5.

* parent: RefCell::new(weak::new()).

* children: RefCell::new(weak::new()).

* value: 5.

* parent: RefCell::new(weak::new()).

* value: 3.

* value: 3.

* parent: RefCell::new(weak::new()).

* value: 3.

* parent: RefCell::new(weak::new()).

* value: 3.

* value: 4.

* value: 5.

* valu
```

```
Strong and weak count
i fn main() {
    let leaf = Rc::new(Node{
        value: 3,
        parent: RefCell::new(weak::new()),
        children: RefCell::new(vec![]),
};

println!(
    "leaf strong = {}, weak = {}",
        Re::strong_count(&leaf), // 1
        Rc::weak_count(&leaf), // 0

);

| Comparison of the count of th
```

```
Output | leaf strong = 1, weak = 0 | branch strong = 1, weak = 1 | leaf strong = 2, weak = 0 | leaf parent = None | leaf strong = 1, weak = 0
```

20 Fearless Concurrency

Definition 20.1. Concurrent programming is when different part of program execute concurrently and **parallel programming** is when different part of the code execute at the same time. In these notes when we say concurrency, we mean both concurrency and parallel.

Definition 20.2. Within a program we can have multiple parts that run at the same time, the features that run these parts of the program are called **threads**.

Challenges

- Race conditions, threads are accessing data/resources in inconsistent order
- Dead locks, if two threads both waiting for a resource that another thread has
- Hard to reproduce and fix bugs, since execution order is non-determinastic.

Main kinds of threads

- 1 to 1 threads, aka OS threads, etc..; when we create a thread in our program it maps to our OS thread. So there is a 1 to 1 mapping between the two.
- Green threads, aka user threads, program threads, etc..; special implementation of threads provided by the programming language. They don't have a 1 to 1 mapping to OS threads.

Rust aims to have an extremly small runtime, so it must sacrifice features. Since Green threads will require a larger language runtime, Rust only includes 1 to 1 threads in the standard library. If we want to use green threads we must use crates.

20.1 Implementing threads

```
thread::spawn(||{
    for i in 1..10{
        println!("hi number {} from the spawned thread!", i);
        thread::sleep(Duration::from_millis(1)); // Pause execution of thread for a milisecond.

}
});

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

thread::sleep(Duration::from_millis(1));
}
```

```
Output

| hi number 1 from the main thread!
| hi number 2 from the spawned thread!
| hi number 2 from the spawned thread!
| hi number 3 from the main thread!
| hi number 3 from the spawned thread!
| hi number 4 from the spawned thread!
| hi number 5 from the spawned thread!
```

Note the main thread finished printing all of it's numbers, but the spawned thread didn't. This is because when the main thread ends the spawned thread stops.

How do we fix the code to make sure that the spawned thread finished execution.

```
Code use std::{thread, time::Duration};

fn main() {
    let handle = thread::spawn(||{
        for i in 1..10{
            println!("hi number {} from the spawned thread!", i);
            thread::sleep(Duration::from_millis(1)); // Pause execution of thread for a milisecond.
    }
}

});

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

thread::sleep(Duration::from_millis(1));

handle.join().unwrap();

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

Calling join() blocks the thread currently running(main thread in this case) until the thread associated with join (i.e. spawned thread) terminates. Blocking a thread prevents it from doing any further work or exiting.

```
Output

hi number 1 from the main thread!
hi number 2 from the spawned thread!
hi number 2 from the spawned thread!
hi number 3 from the main thread!
hi number 3 from the spawned thread!
hi number 4 from the spawned thread!
hi number 5 from the spawned thread!
hi number 6 from the spawned thread!
hi number 7 from the spawned thread!
hi number 8 from the spawned thread!
hi number 9 from the spawned thread!
```

If we move our handle.join() right after our spawn thread is created, the main thread will wait until the spawn thread is finished executing.

```
Output if we move handle

| hi number 1 from the spawned thread! |
| hi number 3 from the spawned thread! |
| hi number 4 from the spawned thread! |
| hi number 5 from the spawned thread! |
| hi number 6 from the spawned thread! |
| hi number 7 from the spawned thread! |
| hi number 8 from the spawned thread! |
| hi number 9 from the spawned thread! |
| hi number 1 from the main thread! |
| hi number 2 from the main thread! |
| hi number 3 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
| hi number 4 from the main thread! |
```

So where the join method is called can affect if our threads run at the same time

20.2 Move closures

Up until this point, the thread did not depend on any variable outside of our thread.

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

4    let handle = thread::spawn(||{ //Compile time error: the closure may outlive the current function, but
    borrows v which is owned by current function.
    println!("Here's a vector: {:?}", v);
});

drop(v); //oh no!

handle.join().unwrap();

}
```

The rust compiler doesn't know if v is always a valid reference. If our code was allowed to run, we could enter the main thread drop the value v, then switch back to the spawned thread, at which point v is invalid.

So Rust doesn't allow us to take a reference to v, our thread must take ownership with the move keyword.

```
move keyword to force closure to take ownership
fn main() {
   let v = vec![1,2,3];

   let handle = thread::spawn(move || { //closure takes ownership of v
        println!("Here's a vector: {:?}", v);
   };

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

20.3 Using data to pass messages between threads

One popular approach to insure safe concurrency is message passing. Where we have threads(actors) passing messagees to eachother which contains data. One tool rusts provides in the std library to pass message is a channel. Like a channel of water, we can pass something on the stream it will travel downstream to the end of the water way.

A channel in programming has two halves the transceiver (upstream) and the reciever (downstream). The channel is closed if the reciever or

```
Implementing a channel
    use std::sync::mpsc; // mpsc = multi-producer, single consumer.

use std::thread;

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

    thread::spawn(move || { // We need to use move keyword to move tx into the closure, in order to pass msg into tx.
    let msg = String::from("hi");
    tx.send(msg).unwrap(); // send() returns a result type, if the recieving end is dropped then send
    () returns an error in real life we want to handle this more gracefully
```

20.3.1 Ownership rules

Ownership rules help us, prevent errors in our concurrent code. When we call send and pass in our value, send takes ownership of our value. So we can't modify or drop our value after it is being passed to another thread.

20.3.2 Passing multiple messages

We will pass multiple messages in order to prove that our code is running concurrently

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

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

    thread::spawn(move || {
        let vals = vee! |
            String::from("hi"),
            String::from("fthe"),
            String::from("the"),
            String::from("the"),
            String::from("thead"),
            |;
            for val in vals {
                tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
            }
            });

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

```
Output | Got: hi | Got: from | Got: the | Got: thread
```

20.3.3 Creating multiple producers

Note our program is non-determinastic, when we run it may recive the messages in a different order

```
Output 1
| Got: hi |
| Got: more |
| Got: messages |
| Got: messages |
| Got: for |
| Got: the |
| Got: you |
| Got: thread |
```

```
Output 2
Got: hi
Got: more
Got: messages
Got: from
Got: for
Got: the
Got: you
Got: thread
```

20.4 Transfering data with shared state

Unlike with the channel where we are passing ownership of a piece of data form one thread to another, with shared state concurrency we have some piece of data in memory that multiple threads can read and write to.

20.4.1 Mutex

Definition 20.3. Mutex means mutual exclusion, this means that only one thread can access a piece of data at any time. We achieve this with a locking system. The lock is a data structure that keeps track on what thread can use that piece of data. Once the thread is done with the piece of data it can unlock the data allowing other threads to have access to it.

We need to remember two rules when we use mutex:

- 1. We need to aquire a lock before we have access to data.
- 2. We have to release that lock when we are done with the data, so that other threads can have access.

Simple implementation use std::sync::Mutex; fn main() { let m = Mutex::new(5); 6 {

```
let mut num = m.lock().unwrap(); //Block current thread until that lock is aquired. Calling lock
will fail if there is already another thread that has a lock to data and it panics, then our call will
fail. unwrap() makes us panic.

*num = 6;

println!("m = {:?}", m); // value stored in m is 6

println!("m = {:?}", m); // value stored in m is 6
```

num is of type MutexGuard smart pointer. When MutexGuard goes out of scope it calls drop trait which will release lock to the data.

So we want is something exactly like Rc but thread safe. We will use atomic reference counting smart pointer

Note counter is immutable but we can get mutable reference to value inside since Mutex uses interior mutability

Remark. Mutex may cause deadlocks.

20.5 Building concurrency features

While the rust language provide few concurrency features, it does provide the building blocks for us to build our own or use features built by someone else.

Two basic concurrency concepts provided by std lib are the send and sync traits.

21 Object Oriented Programming features in Rust

When it comes to OOP there are three characteristics that all OOP languages share:

- Objects
- Encapsulation
- Inheritance

Let us talk about each of these characteristics and if they are supported in rust. Objects are made out of data and methods that operate on that data.

21.1 Objects

In Rust structs and enums hold data and we can define impl blocks to create methods for them. So even if they are not called objects they function in the same way.

21.2 Encapsulation

Definition 21.1. Encapsulation means that implementation details of an object are hidden from the code using that object. Code outside of the object can only interact with object through the public API.

Previously we learned how to use the pub keyword to decide which modules, types, function and methodes are public since everything is private by default.

We can change everything that is private (for example using hashmap instead of vec), and as long as the signatures of the public functions stay the same code using our struct doesn't have to care.

21.2.1 Inheritance

Definition 21.2. Inheritance is the ability of an object to inherit from another object's definition gaining the data and behaviour of that other object without having to define the data and behaviour itself.

Rust doesn't have this ability, you can't define a struct that inherits fields and methods from another struct. Rust has other tools we can use depending on why we are reaching for inheritance.

There are two main reason to use inheritance

- Code sharing, we can implement behaviour on one type and all other types that inherit from it can reuse that behaviour. In rust we can do this by using default trait method implementations. There is a limit, traits can only define methods not fields, but there is a proposal for them to define fields.
- Polymorphims, allows us to substitute multiple objects at run time if they share certain characteristic, in classical inhertence that is a parent class. For example we can have base class Vehicles, and have subclasses inherit from that such as Truck, Car, etc.. And if we have a function that takes in a Vehicle at runtime we can pass in a Truck, Car, etc.. to that function. Rust takes in a different approach, in Rust we use generic to abstract away concrete types and we use trait bounds to restrict the characteristic of those types. Rust also provides trait objects which are similar to generics but they use dynamic dispatch while generics use static dispatch. We will see more about this in the next part.

22 Trait objects

Imagine we were building a GUI library, where the goal is to take in a list of visual elements (i.e. buttons, text boxes, etc..) and draw them on the screen. We want users to be able to create their own visual components that can be drawn on the screen. At compile time we don't know the breath of objects used at compile time, but we know they will have a method called Draw.

If we were using a language with classical inheritance we may create a base class called visualComponents with a Draw method, where each visual component will inherit, or override with their own implement.

In Rust we define shared behaviour using traits.

Definition 22.1. Traits are define by first specifying some Pointer (like ref, or smart pointer) and use the dyn keyword followed by the relevant trait.

```
pub components: Vec<Box<dyn Draw>>,
```

When we define our trait object, at compile time Rust will make sure everything in the vector implements the Draw trait.

Why shouldn't we use generics rather than trait objects? When we use generics we are limited to one type!

```
pub struct Screen<T: Draw>{
    pub components: Vec<T>,
}
```

Then we can only store one type in our vector, either all buttons or all text boxes. Using trait objects we can store buttons, textboxes, etc... in the same vector.

22.1 Static vs Dynamic dispatch

Definition 22.2. Monomorphism, is a process when the compiler will generate non-generic implementation of function based on concrete types used in place of generic types.

For example, we have a function add, which takes two generic parameters and adds them, if we use that function with floats and integers the compile will create a function for integer_add and float_add and find the call sites of the add method and replace it with the concrete implementation.

This is called **Static dispatch**; the compiler knows the concrete functions we are calling at compile time. The opposite is **dynamic dispatch**, the compile figures out what concrete methods we are running at run time.

When using trait objects, Rust compiler uses dynamic dispatch since it doesn't know all the concrete objects used at compile time. There is a runtime performance cost in using this, but we get to write flexible code that accepts any object that implements a certain trait.

22.2 Object Safety

We can only make object safe traits into trait bounds. A trait is object safe if all methods implemented on that trait have these properties

Definition 22.3.

Return type is not &self

There are no generic parameters

If the trait doesn't have these properties, the compiler can't figure out the concrete type of that trait and doesn't know what method to call.

23 Implementing the state object oriented pattern

In the state pattern we have some value that has internal state, represented by state object. Each state object is responsible for it's own behaviour and when to change state. The object containing the state objects knows nothing about the different behaviours or state or when to transition.

To understand how the state pattern works, we will implement a blog-post workflow in Rust.

```
pub fn request_review(&mut self){ // Method is the same no matter what state we are in. if let Some(state) = self.state.take(){ // take() takes value out of option and leaves None instead. Since Rust doesn't allow unpopulated struct fields, we need to wrap our field inside an
Option to move it.
instead. Since Rust doesn't allow unpopulated struct fields, we need to wrap our field inside an
```

```
main.rs
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salade for lunch today");
    assert_eq!("", post.content());

    post.request_review();
    assert_eq!("", post.content());

post.approve();
    assert_eq!("I ate a salade for lunch today", post.content());

assert_eq!("I ate a salade for lunch today", post.content());

assert_eq!("I ate a salade for lunch today", post.content());
```

Exercise 1

- Add another method called reject which takes post under review and returns it back to draft state.
- Make it so that two apportals are required before a post is published
- Make it so that can only add text to a post when it is in draft mode.

One of the downsides of State Pattern is that some states are coupled to eachother, if we wanted to add a state between PendingReview and Published, then we would need to update the PendingReview state so that it doesn't transition to Published state.

Another downside is duplication, we have very similar implementation for request_review and approve methods. By implementating the state pattern exactly as in OOP we are not taking full advantage of Rust, let's try a different approach. We will encode different states as different types.

```
main.rs
use blog::Post;

fn main() {
    let mut post = Post::new();

    post.add_text("I ate a salade for lunch today");

    let post = post.request_review();

    let post = post.approve();
    assert_eq!("I ate a salade for lunch today", post.content());
}
```

This new implementation doesn't follow the OOP state pattern, but invalid states are now impossible due to they type system and type checking.

24 Patterns

24.1 Patterns and Matching

Definition 24.1. Patterns are a special syntax in rust that is used to match against the structure of types. They are consisted of litterals, Destructed (Arrays, Enums, Struct or tuples), variables, wildcards, placeholders.

These components describe the shape of the data we are working with which we match against values to determine if our program have the correct data to run a piece of code.

Patterns in match Expressions We use patterns inside the arms of match expressions, for example:

Recall that match expressions have to be exhaustive, so we need to use the catch all pattern denoted by an underscore. Note that the underscore doesn't bind to the variable we are matching on we can instead use a variable like so:

```
enum Language {
    English,
    Spanish,
    Russian,
    Japenese
}

let language = Language:: English;

match language {
    Language:: English ⇒ println!("Hello World!"),
    Language:: Spanish ⇒ println!("Hola Mundo!"),
    lang ⇒ println!("Unsuported Language! {:?}", lang)
}
```

Patterns in conditional if let expressions Recall we can use if let expressions if we want to match on some variable, but we only care about one case.

```
fn main() {
    let authorization_status: Option<&str> = None;
    let is_admin = false;
    let group_id: Result<u8, > = "34".parse();

if let Some(status) = authorization_status{
        println!("Authorization status: {}", status);
} else if is_admin{
        println!("Authorization status: admin");
} else if let Ok(group_id) = group_id{
        if group_id > 30{
            println!("Authorization status: priviledged");
} else{
        println!("Authorization status: basic");
} } else {
        println!("Authorization status: guest");
}
}else {
        println!("Authorization status: guest");
}
```

The downside of the if let expression is that the compiler doesn't enforce that they are exhaustive, so we can ommit the last else case and our program will compile.

Patterns in while let expressions Similarly to the if let, the while let syntax lets us run a loop as long as the pattern specified continue to match.

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

stack.push(1);
stack.push(2);
stack.push(3);

while let Some(top) = stack.pop(){
    println!("{}", top);
}
```

```
Patterns in for loops

| let v = vec!['a','b','c'];

| for (index, value) in v.iter().enumerate() { // enumerate() returns a tuple of index and value println!("{} is at index {}", value, index);

| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is at index {}", value, index);
| println!("{} is
```

Patterns in let statements When we write a let statement we are actually writing:

```
1 //let PATTERN = EXPRESSION
```

For example rust will check that this pattern and expression match:

```
let (x,y,z) = (1,2,3); 

// let (x,y) = (1,2,3) This causes an error since compiler expect a pattern that matches 3 ints. 

let (x,y,\_) = (1,2,3); // If we want to ignore 3, we need to use an underscore.
```

```
| fn main() {
| let point = (3,5);
| print_coordinates(&point);
| 4 | }
| 6 | fn print_coordinates(&(x,y):&(i32,i32)) {
| println!("Current locationL: ({{}}, {{}})", x,y);
| 8 | }
```

This will also work with closures.

Irrefutable and refutable patterns

Definition 24.2. Irrefutable patterns are patterns that will always match for example:

```
x = 5;
```

Refutable patterns may not match, for example:

```
let x: Option<&str> = None;
if let Some(x) = x{
    println!("{}", x);
}
```

Function paramaters, let statements and for loops only accept Irrefutable patterns. For example:

```
let x: Option<&str> = None;
// let Some(x) = x; Error, since x is a None variant this will never match.

if let x = 5{ // Compiler warning; 'This pattern will always match, so this pattern is useless'
println!("{}",x);
}
```

24.2 Pattern Syntax

We are going to talk about all the valid syntax we can use in patterns.

The first pattern we can match on is matching against litterals:

This pattern is useful when we want our code to take action when it receives a named value.

Next pattern we can match on is named variables:

25 Advanced topic

25.1 Unsafe Rust

So far the code we have written has been forced to follow Rust's memory safety guarentee at compile time. But if we wan to ignore these memory safety guarentees we can use **Unsafe Rust**. This exists for two reasons:

- Rust will reject program if it can't guarentee that the program is memory safe even if we know it is.
- The underlying hardware is inheritantly unsafe, so if had to stay safe there is some task we couldn't do which we wan to if we are doing low-level system programming for example.

To use usafe rust we must use the unsafe keyword, gives 5 abilities don't have in safe world:

- 1. Deref raw pointer
- 2 Call unsafe function or method
- 3. Access or modify a mutable static variable

- 4. implement unsafe trait
- 5. Access fields of unions.

Unsafe doesn't turn off borrow-checker or disable safety checks. If we have reference in unsafe code it will still be checked.

The onus is on the programmer to make sure that memory inside unsafe blocks is handled properly. If we keep blocks small and isolated it will make the code easier to debug, in case of an error. We can also wrap unsafe code in a safe abstraction and provide a safe API.

25.1.1 Raw Pointers

Unsafe rust has two types of raw pointers that are similar to references, immutable and mutable raw pointers. The difference between raw pointers and references/smart pointers, is that raw pointers:

- bypass borrow rules,
- not guarenteed to point to valid mem.
- are allowed to be null.
- don't implement automatic cleanup

```
let mut num = 5;
let r1 = &mum as *const i32; // immutable raw pointer, can't be direcely assigned after being deref
let r2 = &mut num as *mut i32; // mutable raw pointer

// Rust allows us to create raw pointers but we can't deref them if we are not in unsafe block.

unsafe {
    println!("r1 is: {}", *r1);
    println!("r2 is: {}", *r2);
}
```

25.1.2 Unsafe methods and function

```
unsafe fn dangerous (){} // unsafe means that when calling functions we need to put in the correct arguments, it implies we read the function's documentation and are taking responsibility for upholding the function contract.

unsafe{ // If we remove the unsafe block it will call an error. The body of unsafe function is an unsafe block so we don't have to write an unsafe block inside function.

dangerous();
}
```

Safe abstraction of unsafe code Just because a function contains unsafe code doesn't make it an unsafe function. We can wrap unsafe code inside a safe function.

```
//Example: split_at_mut(), is a safe method that split a slice into two slices along index passed in.

let mut v = vec![1,2,3,4,5,6];

let r = &mut v[..];

let (a,b) = r.split_at_mut(3);

assert_eq!(a, &mut [1,2,3]);
assert_eq!(b, &mut [4,5,6]);
```

```
Naïve implementation of split at mut

fn split_at_mut(slice: &mut [i32], mid: usize) -> (&mut [i32], &mut [i32]) {

let len = slice.len();

assert!(mid <= len);

(&mut slice[..mid], &mut slice[mid..]) // Error: Cannot borrow slice as mutable more than once at a time.

time.
```

The borrow checker can't understand that we are borrowing different parts of our slice. We know our code is ok but Rust can't figure it out. We need to use unsafe block:

Note even if function has unsafe block it is safe.

25.1.3 Functions that call external code

sometimes we need to interact code in another language. In this case Rust has the extern keyword, which let's use a foreign language interface, ffi. A ffi is a way for a prog to define a function that another language can call.

```
Calling a function from C

extern "C" { // "C" defines which application binary interface, ABI the external function uses. ABI defines how to call the function at the assembly level.

// abs function from C std library.

fn abs(input: i32) -> i32; // Specify the name and signature of foreign function we want to call.

// calling function in extern block is always unsafe.

unsafe {

println!("Absolute value of -3 according to C: {}", abs(-3));

}
```

```
Allowing other languages to use our code | #[no_mangle] // no_mangle is to let the compile know not mangle(when the compiler changes the name of our function) our function name | pub extern "C" fn call_from_c() { | println!("Just called a Rust function from C!"); | }
```

25.1.4 Accessing and modifying mutable static variables

Definition 25.1. Global variables in rust are supported but can be problematic with ownership rules. In rust global variables are called **static variables**. The convention is to write the variable in screaming snakecase. We must also annotate the life time of variable and it must have a static lifetime.

```
static HELLO_WORLD; &str = "Hello, world!"; // In this case don't have to specify lifetime since Rust infers the lifetime of &str

fn main() {
    println!("name is: {}", HELLO_WORLD);
}
```

The differences with immutable static and constants is that: static variables have fixed address in memory, constants can duplicate their data whenever they are used. If we are referencing constant in code base, the compiler can replace constant with concrete value

Also, static variables can be mutable but accessing an modifying mutable static variables in unsafe.

25.1.5 Implementing an unsafe trait

A trait is unsafe if at least one its methods is unsafe

```
unsafe trait Foo{
// methods go here
}

unsafe impl Foo for i32{
// method implementation goes here.
}
```

25.1.6 Unions

The fifth ability with unsafe code is able to access field of unions.

A union is similar to a struct but only one field is used in each instance. Primarily used to interface with C unions, it is unsafe to access field of a union since Rust can't guarrentee what type stored in union is for a given instance.

25.2 Advanced Traits

Definition 25.2. Associated types are placeholders added to traits and methods can use. For example:

```
pub trait Iterator{
type Item; // associated type

fn next(&mut self) -> Option<Self::Item>;
}
```

What is the difference between an associated type and generic? They both allow us to define a type without specifying concrete value. In associated type we have only one concrete type per implementation, but generics can have many concrete types

```
Associated Types implementation

struct Counter{}

impl Iterator for Counter{
    type Item = u32; // Can't have another implementation where item is different

fn next(&mut self) -> Option<Self::Item>{
    Some(0)

}

// We can't have a second implementation like this:

impl Iterator for Counter{
    type Item = u16;

fn next(&mut self) -> Option<Self::Item>{
    Some(0)

}
```

```
20 }
21
```

We can have many implementations with generics:

```
pub trait GenericIterator<T>{
    fn next(&mut self) -> Option<T>;
}

// Two implementation of GenericIterator.

impl GenericIterator<u32> for Counter{
    fn next(&mut self) -> Option<u32>{
        Some(0)

}

impl GenericIterator<u16> for Counter{
    fn next(&mut self) -> Option<u16>{
        Some(0)

}

impl GenericIterator<u16> for Counter{
    fn next(&mut self) -> Option<u16>{
        Some(0)

}

some(0)

}

some(0)

}

reflections
```

When deciding between generics and iterators we need to know if we need multiple implementation for a single type.

In the case of Iterators it makes sense to use associated type, since for any implementation we want next() to return the same type.

Default Generic Type parameters and type overloading Generic type parameters can specify a default concrete type so that implementers only need to specify a concrete type if it is not the default.

For example when customizing the behaviour of operator, aka operator overlead. Rust can let us customise the semantics of certain op with associated traits in std lib, for example Add operator.

```
How is the Add Trait implemented?

trait OurAdd<Rhs=Self>{
    type Output;

fn add(self,rhs:Rhs) -> Self::Output;
}
```

Add trait has generic called Rhs(right hand side), it's default concrete type is self, the object implenting the add trait. So we don't have to specify it in our Points example. Here is an example when we do need to specify it:

Specifying generic

```
struct Millimeters(u32);
struct Meters(u32);

impl OurAdd<Meters> for Millimeters{
    type Output = Millimeters;

fn add(self,rhs:Meters) -> Millimeters {
    Millimeters(self.0+(rhs.0*1000))
}

Millimeters(self.0+(rhs.0*1000))
}
```

In general we can use default generic type param for two reasons:

- Extend a type without breaking a code.
- Adding customization for code that most users won't need.

Now we move on to calling methods with the same name. We can have two traits with same methods and implement both those traits on one type. We can also implement on a type itself with the same name as in another method.

```
trait Pilot {
    fn fly(&self);
    fn fly(&self);
}

trait Wizard {
    fn fly(&self);
}

struct Human;

impl Human {
    fn fly(&self) {
        println!("*waiving arms furiously*");
    }
}

impl Pilot for Human {
    fn fly(&self) {
        println!("This is your captain speaking");
    }
}

limpl Wizard for Human {
    fn fly(&self) {
        println!("This is your captain speaking");
    }
}

fn main() {
    println!("Up!");
    }
}

fn main() {
    preson.fly(); // When we run we get *waiving arms furiously* the method on Human struct

Wizard::fly(&person); // To run the Pilot implementation
    Wizard::fly(&person); // To run the Wizard implementation
}
```

What about if we are using associated functions? Since fly method takes self as a parameter, if we had two structs implementing Wizard trait we know which method to call. This is not true for associated function.

Supertraits

Definition 25.3. Trait dependent on functionality on other trait, the trait we rely on is called **supertrait**. If we parvely define a trait like so:

```
trait OutlinePrint {
    fn outline_print(&self) {
        let output = self.to_string();
        let len = output.len();
        println!("{}", "*".repeat(len+4));
        println!("*{}*", " ".repeat(len+2));
        println!("* {} *", output);
        println!("*{}*", " ".repeat(len+2));
        println!("*{}*", " ".repeat(len+2));
        println!("{}", "*".repeat(len+4));
        }
        }
        }
}
```

We get an error, since calling to string since don't know that self implements to string. Since to string is defined in Display trait so we want to make sure anything that implements OutlinePrint implements Display.

```
Correct implementation
```

```
| use std::fmt; // This is where Display is defined.
| trait OutlinePrint: fmt::Display { // This says that our trait depends on Display trait. |
| fn outline_print(&self) { | let output = self.to_string(); |
| let len = output.len(); |
| println!("{}", "*".repeat(len+4)); |
| println!("*{}*", " ".repeat(len+2)); |
| println!("*{}*", " ".repeat(len+2)); |
| println!("*{}*", " ".repeat(len+2)); |
| println!("{}*", "*".repeat(len+2)); |
| println!("*", "*".repeat(len+2)); |
| println
```

```
14 }
15 }
```

What if we implement OutlinePrint but not Display? We get an error! The correct implementation is like so:

```
impl OutlinePrint for Point{}

impl fmt::Display for Point{
    fn fmt(&self, f:&mut fmt::Formatter) -> fmt::Result{
        write!(f, "({{}}, {{}})", self.x, self.y)
    }
}
```

Finally we will look at the new type pattern. We learned about Orphan rule, we can define trait on a type as long as triat or type is defined in our crate, the newtype pattern allows us to get around this. We don this by create a tuple with one field which is the type we are wrapping.

For example, how do we implement Display trait for Vector? They are both defined outside of our crate. Answer: We wrap vector in another object.

```
use std::fmt;
struct Wrapper(Vec<String>);
impl fmt::Display for Wrapper{
    fn fmt(&self, f: &mut fmt::Formatter<'_>) -> fmt::Result {
        write!(f, "[{{}}]", self.0.join(", "))
    }
}

fn main() {
    let w = Wrapper(
        vec![String::from("Hello"), String::from("world")]
    );
}

println!("w = {{}}", w);

// Output: w = [Hello, world]
}
```

The problem is that the Wrapper type doesn't implement the methods implemented for vector. If we did want the new type to implement every mothod on type it is holding we can implement deref trait so that deref wrapper returns the value. If we don't want the wrapper to have all the methods we need to implement each method individually.

We can also use the newtype pattern for safety, for example if we want to store a tuple of integers one being ID number and one being Age, in order to not get them confused we wrap the integers in corresponding struct.

```
struct Age(u32);

struct ID(u32);

// Pass in Age and ID in functions instead of a raw integer.
```

Another use of newtype pattern is to abstract away implementation details. We can create a People type which wraps a Hash maps of integers to strings. Code using People type will only have access to public API and won't have knowledge of the internals.

New type is a light weight way to achieve encapsulation. In addition to new type, we can use type Alias to give existing types new names.

```
fn main() {
    type Kilometers = i32; // Not a new type but a synonym for i32

let x: i32 = 5;
    let y: Kilometers = 5;

println!("x+y = {}", x+y); // Kilometer treated like i32

}
```

The main use of alias is to reduce repetition, for example if we have a lengthy type like in this example

The never type The never type is denoted by ! and means that a function will never return. Why is this useful? Recall the guessing game with code that parse user input into integer:

```
while game_in_progress{
    let guess: u32 = match guess.trim().parse(){
        Ok(num) => num,
        Err(_) => continue,
};
```

How does one arm return a u32 and another arm of the match statement ends with continue? continue is a never type, since one arm returns u32 and other! (can never return) Rust can conclude that this expression returns u32.

We are allowed to write this code since if we get in error arm continue will not return anything so guess will not be assigned. So guess can only be assigned to a u32.

Likewise panic! returns a never type. Another expression with never type is a loop.

```
fn main(){
    print!("forever ");

loop{
    print!("and ever ");
}
```

Here the loop never ends so the value of this expression a never type. This is not true if we had a break statement in loop.

Dynamically Sized Types(DST)

Definition 25.4. DST are types who's size can only be determined at run time. For example the str type:

```
let s1: str = "Hello there!";
let s2: str = "How's it going?";
3
```

This causes a bunch of errors since the compiler doesn't know the size of s1 and s3, we need to use &str instead of str. The string slice structure, &str stores two values an adress pointing the location of the string in memory and the length of the string. Both these propoerties have a type usize so we know their size at compile time.

In general this is how Dynamically sized types are used like this in rust. They have an extra bit of metadata storing the size of dynamic info, The golden rule is that we must store them behind some kind of pointer. For example str is behind a references in &str, but we can also store it in a box pointer for eh.

Traits are also DST, trait objects must be behind some sort of pointer. Rust has a special trait called the size trait that determines what type of whether a type size can be known at compile time. Rust implicitely adds the Sized trait bound to every generic function, so in general generic functions only work on types whoes size is known at compile time. However we can use the syntax:

```
fn generic <T: ?Sized >(t: &T) {}
```

To relax this restriction. This syntax means that T may be sized or not. Since T can be unsized we must put our argument type behind some sort of pointer.

25.3 Closures and Function Pointers

Definition 25.5. We can not only pass closures into functions, but also functions into functions with function pointers

Unlike closures, fin is a type rather than a trait, so we can specify it directly instead of using one of the function traits as a trait bound. Recall the three closure traits: Fn, FnMut and FnOnce (closure takes ownership). Function pointers implement all three closure traits, which is why it is best practice to use closures:

```
fn add_one(x: i32) -> i32{
    x+1

fn do_twice<T>(f: T, arg: i32) -> i32
    where T: Fn(i32)->i32{
        f(arg)+f(arg)
}

fn main() {
    let answer = do_twice(add_one, 5);
    println!("The answer is: {}", answer);
    //Output: The answer is: 12
}
```

As we see, if we use closures we still can use a function pointer as an argument. One place where we only want to accept function pointers instead of function pointers and closures, is when interfacing with a language that dfoesn't use closures like C.

Example **25.5.1**.

```
cother useru pattern with tuple
enum Status{
    Value(u32),
    Stop,

}

// Tuple string uses paranthesis to initialise values inside tuple like a function call. These initialiser are implemented as functions that take in args and returns tuple struct.

let list_of_statuses: Vec<Status> =
    (0u32..20).map(Status::Value).collect(); // map treats Value like a function pointer.
```

Returning closures from functions We need to use the impl syntax

The impl syntax doesn't always work:

```
fn other_returns_closure(a: i32) -> impl Fn(i32) -> i32{
    if a>0{
        move |b| a+b
    } else{
        move |b| a-b // Error: no two closures even if identical have the same type. Consider Boxing our closure.
    }
}
```

The two closures in our function have different types, and the impl trait synyax only works if we are returning one type. Instead of using the impl trait syntax we can return a trait object.

```
fn other_returns_closure(a: i32) -> Box<dyn Fn(i32) -> i32>{
    if a>0{
        Box::new(move |b| a+b)
} else {
        Box::new(move |b| a-b)
}
```

We need to wrap our closure in Box smart pointer, since rust does not know the size of our trait object.

25.4 Macros

Macros allow us write code that writes other code. This is metaprogramming. We can think of Macros as functions whose input and output is code.

Examples of macros we used are prinln! and vec!

The key difference between macros and functions:

- Functions must declare the number of param they accept but macros can accept a variable number of param
- Functions are called at run time, macros are expanded before function finished compiler.

Macros are stronger than functions but put a level of complexity in our code.

Two types of Macros: Declarative Macros and Procedural Macros

25.4.1 Declarative Macros

Declarative Macros are most used form of macros in Rust, they let us write something similar to a match expression

```
vec! declaritive macro // note we can pass in different types and a different amount of arguments in the macros:

let v1: Vec<u32> = vec![1,2,3]; // 3 args

let v2: Vec<&str> = vec!["a","b","c","d","e"]; // 5 args
```

Implementation of the vec! Macro is in the lib.rs file in dec-macro the folder.

25.4.2 Procedural Macros

Procedural Macros are like functions, they take code as input operate on it and produce code as output. In contrast with declarative macros which match against patterns and replace code with other code.

Kinds of procedural macros: Custom derived, attribute-like, function-like. For now procedural macros must be defined in their own crate with a custom crate type.

Example 25.5.2

Now we need to define the macro in it's own crate. Which is called hello_macro_derive.

Finally we can use this macro

```
use hello_macro::HelloMacro;
use hello_macro_derive::HelloMacro;

// Two use statements since we have two crates hello_macro and hello_macro_derive. Each crate stil has to be published seperately and code using our crate has to bring each crate into scope.

#[derive(HelloMacro)]
struct Pancakes;

fn main(){
    Pancakes::hello_macro();
}

// Output: Hello, Macro! My name is Pancakes!
```

Attribute-like macros Similar to custom derived macros expect instead of generating code for derive attribute we can create a custom attribute. This only works on struct and enums.

Function-like macros Look like function calls but more flexible. They can take a variable number of args and the operate on rust code.

```
let sql = sql!(SELECT * FROM post WHERE id=1);

#[proc_macro]
puu fn sql(input: TokenStream) -> TokenStream{

6 }
```

26 Building a Web Server

Definition 26.1. TCP transport layer protocol, describing how data is transferred from one computer to another. HTTP is application level protocol, describing how data is structured.

```
Reading request:

use std::net::TcpListener;

use std::net::TcpStream;

test std::net::TcpStream;

fn main() {

let listener =

TcpListener::bind("127.0.0.1:7878").unwrap(); //Since we are doing practice we are just going to call unwrap and not deal with error case

for stream in listener.incoming() {

let stream = stream.unwrap(); //Shadowing

handle_connection(stream);
}

}

fn handle_connection(mut stream: TcpStream) {

let mut buffer = [0; 1024]; // A buffer of 1024 bytes.

stream.read(&mut buffer).unwrap(); // Populate buffer with data from stram println!(

"Request: {}",

String::from_utf8_lossy(&buffer[..]) // converts a slice of bytes into string including invalid characters

)

}
```

After running our program we get a bunch of GET request. The structure of the request:

- The first line is the request line, it contains info such as HTTP method used, URI and HTTP version number.
- Under the request line we have various request headers and body. But since this is a get request we don't have a body.

Writting a response: A response must be in the following format

```
// status line: HTTP-Version Status-Code Reason-Phrase CRLF (carriage return line feed sequence)
// headers CRLF
// message-body
//
// ex: HTTP/1.1 200 Ok\r\n\r\n
// This response contains no headers and no message body. HTTP-version is HTTP/1.1, Status-Code is 200
(standard status code for sucess), reason phrase is OK, and a CRLF is \r\n
```

We are now going to create a multi-threaded server. Indeed we built a single threaded server, but in this case if we get two request at the same time, our server will process the first request then the second request. If our first request took a long time to process the second request will have to wait. Terrible user experience.

We will solve this with a thread pool, we have a fixed number of threads in a thread pool. When request come in server a thread will pick up the request and process it. When it is finished it will return in the thread pool to take on new requests. The reason we have a fixed number of threads, is that we don't want a person to take down our server by issuing a ton of request that spin up a ton of threads and exhaust the resources of our server.

Graceful shutdown and clean-up (Check the server folder in the github to see the code)