Rust Course notes

Razafy Rindra, “Hagamena”

July 24, 2022

Contents

1 Introduction

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

2 Common Programming Concepts in Rust

2.1 Variables and Mutability

2.1.1 Mutable variables

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

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

2.1.2 Constants

We can also declare constants in Rust like so:

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

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

2.1.3 Shadowing

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

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

2.2 Data Types

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

Types of scalar data types

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

Type of compound data types

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

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

```

2. Arrays, in Rust they are fixed length.

2.3 Functions

Functions are defined with an fn keyword like so:

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

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

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

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

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

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

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

2.4.2 Loops

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

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

We can also use the while statement:

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

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

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

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

We can also loop over a range:

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

The last number of the range is excluded.

3 Ownership

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

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

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

2. Manual memory management

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

   Notice that these two have opposite trade-offs.

3. Ownership model

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

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)

```

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

```

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.

```
    fn main() {
    let s1 = gives_ownership();         // gives_ownership moves its return
                                        // value into s1

    let s2 = String::from("hello");     // s2 comes into scope

    let s3 = takes_and_gives_back(s2);  // s2 is moved into
                                        // takes_and_gives_back, which also
                                        // moves its return value into s3
} // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
  // happens. s1 goes out of scope and is dropped.

fn gives_ownership() -> String {             // gives_ownership will move its
                                             // return value into the function
                                             // that calls it

    let some_string = String::from("yours"); // some_string comes into scope

    some_string                              // some_string is returned and
                                             // moves out to the calling
```

```
                                                        // function
    }

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

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

```

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

3.3 References

Let us see how references fix the following situation

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

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

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

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

    (s, length)
}
```

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

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

        let len = calculate_length(&s1);

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

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

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

Definition 3.1. Passing in references as function paramaters 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.

```rust
    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 referece already exists. Immuatble reference don’t expect the underlying data to change. But we can have many Immuatble references, since we don’t expect the underlying value to change.

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

Rules of Refences

1. At any given time, we can either have one mutable reference or any number of immuatble reference.

2. References must always be valid.

3.4 Slices

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

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

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

    s.len()
}

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:

```rust
    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[..]
    }

}
```

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

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

fn main() {
    //Creating an instance of struct
    let mut user1 = User {
        email: String::from("user@email.com"),
        username: String::from("user123"),
        active: true,
        sign_in_count: 1
    };
    let name = user1.username; // Get values from struct with the . operator
    user1.username = String::from("user234"); // modify specific values with . operator

    // Creating a new instance of struct with function
    let user2 = build_user(
        String::from("user345@email.com"),
        String::from("user345")
    );

    //We can also create new instances of a struct using existing instances.

    let user3 = User{
        email: String::from("user456@email.com"),
        username:String::from("user456"),
        ..user2 // Takes the remaining field from user2
    };

    // We can also create structs without naming fields, these are called tuple structs

    struct Colour(i32,i32,i32);
    struct Point(i32,i32,i32);

    // Colour and Point have the same fields, but have different type,
    //so we can't pass a Colour to a function that expects a Point.

}

fn build_user(email: String, username: String) -> User {
    User{
        email, // Since functions arguments have same name as field names, can write it like this...
        username, //...This is called the field init shorthand syntax
        active: true,
        sign_in_count : 1,
    }
}
```

4.2 Methods and associated functions

```
#[derive(Debug)] // Needed to print out, we will look more into this later.
struct Rectangle{
    width: u32,
    height: u32
}

// Creating implementation block for Rectangle struct,
// This will house function and methodes associated with struct

impl Rectangle{
    fn area(&self) -> u32{ // First arguement in method is always self, the instance the method is being called on
            self.width*self.height
    }

    fn can_hold(&self, other: &Rectangle) ->bool{
        self.width > other.width && self.height>other.height
    }
}

//Struct allow us to have multiple impl blocs

// We can also def associated function, unlike methods they aren't tied to an instance of our struct.

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

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 adresses, an IP adresse 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 adresses in code.

```rust
enum IpAddrKind {
    V4(u8, u8, u8, u8), // We can male this code more precise by adding data in the variant
    V6(String),   // Enum variants can store different types of data
}

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

//Just like strucs we can define methods and associated functions on enum type.

impl Message{
    fn some_fn(){
        println!("Ya-hoo");
    }
}


struct  IpAddr{
    kind: IpAddrKind,
    address: String,
}

fn main() {
    let four = IpAddrKind::V4;
    let six = IpAddrKind::V6;

    let localhost = IpAddrKind::V4(127,0,0,1);
}

//Since V4 and V6 are both variants of IpAddrKind, we can define a function that takes in
//our enum type, and it can take in either V4 or V6

fn route(ip_kind: IpAddrKind) {}

```

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.

```rust
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.

```
    //Using Match Expressions

fn main(){
    value_in_cents(Coin::Quarter(UsState::Alaska));
}

#[derive(Debug)]
enum UsState{
    Alabama,
    Alaska,
    Arizona,
    Arkansas,
    California,
    // ...
}

enum Coin{
    Penny,
    Nickel,
    Dime,
    Quarter(UsState),
}

fn value_in_cents(coin:Coin) -> u8{
    match coin{
        Coin::Penny => {
            println!("Lucky penny!");
            1
        }
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter(state) => {
            println!("State quarter from {:?}!", state);
            25
        }
    }
}
```

This program will output:

```
State quarter from Alaska!
```

```
    //Using Match Expressions with Option<T> enum

fn main(){
    let five = Some(5);
    let six = plus_one(5);
    let none = plus_one(None);
}

fn plus_one(x: Option<i32>) -> Option<i32>{
    match x{
        Some(i) => Some(i+1),
        None => None
    }
}
/*
Note Match expressions are exhaustive so we need to match all possible values. So if we
have a lot of types of values, we use the underscore placeholder.
    match x{
        Some(i) => Some(i+1),
        _=> None // underscore means if it is any other pattern execute this code.
    }
*/
```

5.4 Using if let syntax

```rust
    //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.
Crates come in two flavours:

- Binary crate: Code you 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 Definintions, Paths and Privacy

```
//Our goal is to create library that helps run a restaurant.

//We seperate the restaurant in two parts, the front of the house is where the customers are
//Back of the house is where food is being made, dishes are clean and where manager is.

mod front_of_house { // module are specified with mod keyword
// In side this module we have two other modules, modules can contain other modules, structs, enums, constants, traits, etc...

pub mod hosting { //By default a child module and anything inside of it is private from the point of view of parent module.
// So need to put the "pub" keyword, so that front_of_house can see hosting and it's contents.
    pub fn add_to_waitlist() {}

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}


// If you want to reference an item in module tree (like a funciton), need to specifyt a path to function
pub fn eat_at_restaurant(){
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    //Relative path, start from the current module
    front_of_house::hosting::add_to_waitlist();

}

fn serve_order(){}

mod back_of_house{
    fn fix_incorrect_order(){
        cook_order();
        super::serve_order(); // super:: allows us to reference the parent module, in this case crate.
    }

    fn cook_order() {}
}
```

6.3 Privacy rules when it comes to structs

```rust
//Privacy rules when it comes to structs

mod back_of_house {
    pub struct Breakfast { // Need to pub keyword to access it.
    pub toast: String, // Even though struct is pub, by default fields are private
    seasonal_fruit: String,
    }

    impl Breakfast {
        pub fn summer(toast: &str) -> Breakfast {
            Breakfast {
                toast: String::from(toast),
                seasonal_fruit: String::from("peaches"),
            }
        }
    }
}

pub fn eat_at_restaurant() {
    let mut meal =
        back_of_house::Breakfast::summer("Rye");

    meal.toast = String::from("Wheat");

    // Note we need to use summer function to make Breakfast struct.
    // We can't create one directly, since it contains a private field, seasonal_fruit.

    /*
    This gives us an error

    let meal2 = back_of_house::Breakfast{
        toast: String::from("Wheat"),
        seasonal_fruit: String::from("peaches")
    };
    */
}
```

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

Bringing paths into scope with the use keyword.

```
    mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}

// In order to not have to specify the whole path to add_to_waitlist each time we call it, we can use the
    "use" keyword to bring a path in to scope.

// use self::front_of_house::hosting; relative path with "self" referenceing the current module

pub use crate::front_of_house::hosting; // If we want code that is external to this file to reference
    hosting::, we need to mark this "pub"


pub fn eat_at_restaurant() {
    let secret_number = rand::thread_rng().gen_range(1, 101);
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
    hosting::add_to_waitlist();
}




/*  Could have brought the funtion into scope to not have to use hosting::
 But in Rust the idiomatic way to bring a function into scope is to bring it's parent module into scope
 In order to make it clear that add_to_waitlist() function is not a local function.

 If we are bringing enums, struct, other items into scope then we specify the whole path. Except if we are
     bringing two items with the same name into scope.

For example here:


use std::fmt;
use std::io;

fn function1() -> fmt::Result {
    // --snip--
    Ok(())
}

fn function2() -> io::Result<()> {
    // --snip--
    Ok(())
}

Another way of doing this is to rename our function:

use std::fmt::Result;
use std::io::Result as IoResult;

fn function1() -> Result {
    // --snip--
    Ok(())
}

fn function2() -> IoResult<()> {
    // --snip--
    Ok(())
}
*/
```

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::{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();
}
```

```
//In src/front_of_house.rs

    pub mod hosting;
```

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

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

7 Common Collections

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

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

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

7.1 Vectors

```rust
    fn main() {
//Vectors
    let a = [1,2,3];
    let mut v:Vec<i32> = Vec::new(); //Vectors can hold any values, so we need to specify the type.
    v.push(1); // Vectors can grow in size, we can add eleements to it with .push()
    v.push(2);
    v.push(3);

    {
        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.

//Acessing elemetents in vectors

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

    let third = &v[2];
    //  v.push(6);  Error since v is borred as immutable on previous line. If we have an immutable reference to something we expect the underlying value to not change. But if we take a mutable reference to the same thing, the underlying value could change.
    // When we push an element to a vector, we may need to allocated more memory for the new element, so we would move all elements in vector to new location, and our element at the previous line will be pointing to something else.
    println!("The third element is {}", third);

    // If we try to run with third = &v[20], we will get a runtime error, unlike with arrays, we don't get a compile time error. Since at compile time we don't know the size of vector.

    //If we want to access data gracefully, so that the program doesn't crash at runtime if an invalid index is used.

    match v.get(20) {
        Some(third) => println!("The third element is {}", third),
        None => println!("There is no third element."),
    }

//Iterating over vector.

    for i in &mut v{ // for loop
        *i += 50; // Derefencing operator
        println!("{}", i);
    }

// Storing enum variables

    enum SpreadsheetCell {
        Int(i32),
        Float(f64),
        Text(String),
    }

    let row = vec![
        SpreadsheetCell::Int(3),
        SpreadsheetCell::Text(String::from("blue")),
        SpreadsheetCell::Float(10.12),
    ];

    match &row[1] {
        SpreadsheetCell::Int(i) => println!("{}", i),
        _=> println!("Not an integer!")
    };
}
```

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.

```
    use std::collections::HashMap;

fn main() {
    let blue = String::from("Blue");
    let yellow = String::from("Yellow");

    let mut scores = HashMap::new();

    scores.insert(blue, 10); //Note we are not passing the strings as references, so we transfer ownership into the hashmap
    scores.insert(yellow,50);

    let team_name = String::from("Blue");
    let score = scores.get(&team_name); //Get method takes a reference to a key and gives us an option, since we can't guarentee a value will be returned, if we pass an invalide key it will return None.

    for (key, value) in &scores{
        println!("{}: {}", key, value);
    }

    println!("--------");

// Updating our Hashmap

    scores.insert(String::from("Blue"), 20); // Overwrites the Blue key with the value 20.

    //If we don't run to overwrite the existing values:

    scores.entry(String::from("Green")).or_insert(30); //If there isn't entry for key insert key with value 30 unless do nothing.
    scores.entry(String::from("Green")).or_insert(40); //Nothing happens here.

    for (key, value) in &scores{
        println!("{}: {}", key, value);
    }

    println!("---");

// Updating values based on old value

    let text = "hello world wonderful world";
    // value is how many times the key appears in the string.
    let mut map = HashMap::new();


    //["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 key. If the word doesn't exists it adds it to the hashmap and initialises the value to 0. If it does exists it doesn't do anything.
        *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.
    }

    println!("{:?}", map);

}
```

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

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

fn a(){
    b();
}

fn b(){
    c(22);
}

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

```
    RUST_BACKTRACE=1 cargo run
    Finished dev [unoptimized + debuginfo] target(s) in 0.00s
     Running `target/debug/errors`
thread 'main' panicked at 'Do not pass in 22!', src/main.rs:15:9
stack backtrace:
   0: rust_begin_unwind
             at /rustc/a8314ef7d0ec7b75c336af2c9857bfaf43002bfc/library/std/src/panicking.rs:584:5
   1: core::panicking::panic_fmt
             at /rustc/a8314ef7d0ec7b75c336af2c9857bfaf43002bfc/library/core/src/panicking.rs:142:14
   2: errors::c
             at ./src/main.rs:15:9
   3: errors::b // c is being called by b
             at ./src/main.rs:10:5
   4: errors::a // b is being called by a
             at ./src/main.rs:6:5
   5: errors::main
             at ./src/main.rs:2:5
   6: core::ops::function::FnOnce::call_once
             at /rustc/a8314ef7d0ec7b75c336af2c9857bfaf43002bfc/library/core/src/ops/function.rs:248:5
note: Some details are omitted, run with `RUST_BACKTRACE=full` for a verbose backtrace.
```

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:

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

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

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

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

    let f = match f{  //Need to resolve what we do with the different enum variables
        Ok(file) =>file, // Assign the file to f
        Err(error) => match error.kind(){ // If there is an error match based on the kind of error
                ErrorKind::NotFound => match File::create("hello.txt"){ // If file doesn't exists create the file
                    Ok(fc) => fc, // If no error we can create the file
                    Err(e) => panic!("Problem creating the file: {:?}", e) // If there is error panic!
                },
                other_error=>{ // If we get an error not of the file not found kind panic!
                    panic!("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| {
                panic!("Problem creating the file: {:?}", error);
            })
        } else {
            panic!("Problem opening the file: {:?}", error);
        }
    });
}
```

Useful functions on the Result enum

```rust
    use std::fs::File;

fn main(){
//  let f = File::open("new_hello.txt").unwrap();

    let f = File::open("new_hello.txt").expect("Failed to open new_hello.txt");


//we can use unwrap() to do the same this as the below match expression.
//we can use the expect() method, if we also want to specify the error message that is passed when we panic!

    /*
        let f = match f{
            Ok(file) => file,
            Err(error) => panic!("Problem opening 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

```rust
#![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.

    }
    fn main() {}
```

A cleaner way to use error propagation, with the ? operator

```rust

#![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 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

```rust

#![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

```rust

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

fn read_username_from_file() -> Result<String, io::Error> {
    fs::read_to_string("hello.txt")
}

fn main() {}
```

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(())

    }
```

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

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

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

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

We may also use expect or unwrap in test code.

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

9 Generics

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

9.1 A first example

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

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

    let mut largest = number_list[0];

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

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

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

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

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

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

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

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

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


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

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

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

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

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

    let 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
}
```

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

We will talk more about traits in next section.

Generic Types in Structs

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


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

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

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

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

Genercis 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 p1 = Point {x: 5.0, y:10.0}; // both x() and y() are available.
}
```

A more complex example

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


impl<T, U> Point<T, U> {
    fn mixup<V,W>(self, other: Point<V,W>) -> Point<T,W>{
        Point{
            x: self.x,
            y: other.y,
        }
    }
}


fn main(){
    let p1 = Point{x: 5, y:10.4};
    let p2 = Point{x:"Hello", y: 'c'};

    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y); // p3.x = 5, p3.y = c
}
```

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

```
enum Option<T>{
    Some(T),
    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

How are Traits implemented?

```
use std::iter::Sum;
pub struct NewsArticle {
    pub headline: String,
    pub author: String,
    pub content: String,
}

 impl Summary for NewsArticle{
    fn summarize_author(&self) -> String {
        format!("{}", self.author)
    }
    // Don't need to implement summarize since we will use the default implematation.
 }

pub struct Tweet{
    pub username: String,
    pub content: String,
    pub reply: bool,
    pub retweet: bool,
}

impl Summary for Tweet{
    fn summarize_author(&self) -> String {// We need to implement summarize_author since the function has
    no default implementation.
        format!("@{}", self.username)
    }

    fn summarize(&self) -> String {
        format!("{}: {}", self.username, self.content)
    }
}

pub trait Summary{ //Defines a shared methods for our types
    fn summarize_author(&self) -> String;

    fn summarize(&self) -> String{
        format!("(Read more from {}...)", self.summarize_author()) //Default implementation, if our struct
     doesn't implement this function it will use this implementation
    }
}
fn main(){
    let tweet = Tweet{
        username: String::from("@user"),
        content: String::from("Hello world"),
        reply: false,
        retweet: false
    };
    let article = NewsArticle{
        author: String::from("The Author"),
        headline: String::from("I wrote an article!"),
        content: String::from("This is my article")
    };
    println!("Tweet summary: {}", tweet.summarize());
    println!("Article summary: {}", article.summarize());
}
```

This code outputs

```
    Tweet summary: @user: Hello world
Article summary: (Read more from The Author...)
```

10.2 Trait Bounds

&impl syntax and trait bound

```
    // Put in previous Implementations of Tweet, NewsArticle, Summary are to be put here

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


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

This code outputs

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

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

Trait Bound

```

    // Put in previous Implementations of Tweet, NewsArticle, Summary are to be put here
    pub fn notify<T:Summary>(item: &T){ // Generic that is limited to something that interprets a Summary
    trait.
        println!("Breaking news! {}", item.summarize());
 } // Does the same as the &impl syntax
```

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

Multiple inputs

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

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

Specifying multiple traits and the where clause

```
   // impl syntax:


   pub fn notify(item1: &(impl Summary + Display), item2: &impl: Summary){ //item1 is anything that
    implements both Summary and Display, while item2 is anything that implements Summary
        //...
 }

   // //Trait bound syntax

 pub fn notify<T:Summary+Display>(item1: &T, item2: &T){
        //....
}

   // Specifying multiple trait bounds can hinder readablity, for example:

fn some_function<T: Display+Clone, U: Clone + Debug>(t:&T, u:&U) ->i32{....}


   // To make it more readible we can use the "where"-clause

fn some_function<T,U>(t:&T,u:&U) ->i32
        where T: Display+Clone,
              U: Clone+Debug
    {
       ///..
   }
```

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.

```
    // Put in previous Implementations of Tweet, NewsArticle, Summary are to be put here

fn returns_summarizable() -> impl Summary  {//We return any type that implements the summary trait
    Tweet {
        username: String::from("horse_ebooks"),
        content: String::from(
            "of course, as you probably already know, people",
        ),
        reply: false,
        retweet: false,
    }
} // This is very useful with closures and iterators which we will learn more about later.

// While this function can return any type that implements Summary it has it's limits.  While we can return any type implementing Summary, we can only return one type at a time.

 fn return_bool(switch: bool) ->impl Summary{ // This code does not compile.
     if switch{
         NewsArticle{
             ///
         }
     } else{
         Tweet{
             ///
         }
     }
}

fn main(){
    println!("{}", returns_summarizable().summarize());
}
```

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.

Dangling referece

```
    fn main() {
    let r;           //---------------+----'a, lifetime of r denoted by 'a
    {                              //    |
        let x = 5; //--+--'b,        |         lifetime of x denoted by 'b
        r = &x;    //--|             |
    }              //--+             |           x dies here.
                                 //  |
    println!("r: {}", r);          //|
}                                    +         r dies here
```

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.

```
    fn main() {

        let x = 5;           // ---------------------+--'b
        let r = &x;          //--+--'a               |
    println!("r: {}", r);    //  |                   |
}                            //--+-------------------+
```

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

Problematic code

```
    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); // As the borrow checker how would we know that result is not dangling reference?

    fn longest(x:&str, y:&str) -> &str{
        if x.len()> y.len(){
            x
        }else{
            y
        }
    }
}
```

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
        }
    }
}
```

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.

Example with different lifetimes

```
    fn main() {
    let string1=String::from("abcd");

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

        let result = longest(string1.as_str(), string2.as_str()); //result is valid until the end of string2's lifetime.
        println!("The longest string is {}", result);

    }
}

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

Error

```
    fn main() {
    let string1=String::from("abcd");
    let result;
    {
        let string2 = String::from("xyz");

        result = longest(string1.as_str(), string2.as_str()); //result only last until string2 dies
    }
    println!("The longest string is {}", result); //Error since string2 is not valid, so result isn't
}

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

If lifetime tied only to x

```
 fn main() {
    let string1=String::from("abcd");
    let result;
    {
        let string2 = String::from("xyz");
        result = longest(string1.as_str(), string2.as_str());
    }

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

}
fn longest<'a>(x:&'a str, y:&str) -> &'a str{
    x
}
```

This code compliles, since the borrow checker just makes sure that result has the same lifetime as string1.

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 determinstically 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 paramater
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 paramaters.

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.

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

impl<'a> ImportantExcerpt<'a>{
    fn return_part(&self, announcement: &str) -> &str{
        println!("Attenetion please: {15 Hours a Day of Mathematics Self-Study
        28,988 views
        Jun 14, 2022
        1.3K
        Dislike
        Share
        Thanks
        Clip
        Save

\paragraph*{Example combining generics, traits and lifetimes.}\begin{lstlisting}
    use std::fmt::Display;

fn longest_with_an_announcement<'a, T>(
    x: &'a str,
    y: &'a str, //Need to specify lifetime since more than 1 input.
    ann: T, //Generic
) ->&'a str
where
    T:Display, //Triat bound limit T to only those that implement Display
{
    println!("Announcement! {}", ann);
    if x.len() > y.len(){
        x
    }else {
        y
    }
}

fn main(){
    let announcement = String::from("Grand finale");
    let x = "we did generics, traits and lifetimes";
    let y = "how fun!!!!";
    let result = longest_with_an_announcement(x, y, announcement);
    println!("These past sections {}, {}", result, y);
}
```

Output

```
Announcement! Grand finale
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.

Testing our Rectangle struct

```rust
    #[derive(Debug)]
struct Rectangle{
    width: u32,
    height: u32,
}

impl Rectangle{
    fn can_hold(&self, other: &Rectangle) -> bool{
        self.width>other.width&&self.height>other.height
    }
}

#[cfg(test)]
mod tests {
    use super::*; // We need to bring our product code into scope

    #[test]
    fn larger_can_hold_smaller(){
        let larger = Rectangle{
            width: 8,
            height: 7,
        };
        let smaller = Rectangle{
            width: 5,
            height:1,
        };

        assert!(larger.can_hold(&smaller));
    }
    #[test]
    fn smaller_cannot_hold_larger(){
        let larger = Rectangle{
            width: 8,
            height: 7,
        };
        let smaller = Rectangle{
            width: 5,
            height:1,
        };

        assert!(!smaller.can_hold(&larger));
    }
}
```

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.

assert equal and not equal

```rust
    pub fn add_two(a:i32)->i32{
    a+3
}
#[cfg(test)]
mod tests {
   use super::*;
    #[test]
    fn it_adds_two(){
        assert_ne!(4,add_two(2)); // passes if 4 not equal to add_two(2)
    }
    fn it_adds_three(){
        assert_eq!(5,add_two(2)); // passes if 5 is equal to add_two(2)
    }
}
```

Both paramaters passed in assert_eq! and assert_ne! must implement the PartialEq and Debug traits.

Custom failure message

```rust
    use std::fmt::format;

pub fn greeting(name: &str) ->String{
    format!("Hello")
}
#[cfg(test)]
mod tests {
```

```
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
        );
    }
}
```

Failure message

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

Asserting a function that panics

```
    pub struct  Guess{
    value: i32,
}


impl Guess{
    pub fn new(value:i32) -> Guess{
        if value < 1 || value >100{
            panic!("Guess value must be between 1 and 100, got {}.", value);
        }

        Guess {value}
    }
}

#[cfg(test)]
mod tests {
use super::*;
    #[test]
    #[should_panic] // Asserts that the test panics
    fn greater_than_100(){
        Guess::new(200);
    }
}
```

Since we have the #[should_panic] our tests pass.

Many panics

```
    pub struct  Guess{
    value: i32,
}


impl Guess{
    pub fn new(value:i32) -> Guess{
        if value < 1{
            panic!("Guess value must be greater than or equal to 1, got {}", value);
        }else if value > 100{
            panic!("Guess value must be less than or equal to 100, got {},", value)
        }

        Guess {value}
    }
}

#[cfg(test)]
mod tests {
use super::*;
    #[test]
    #[should_panic(expected = "Guess value must be less than or equal to 100")]
    fn greater_than_100(){
        Guess::new(200);
    }
}
```

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.

Tests that return a result type

```
    #[cfg(test)]
mod tests {
    #[test]
    fn it_works() -> Result<(), String>{
        if 2+2 == 4{
            Ok(())
        } else {
            Err(String::from("two plus two not equal to four!"))
        }
    }
}
```

12.2 Running tests and organising into unit tests and integration test

12.2.1 Configuring tests

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

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

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

```
    cargo test --help
```

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

```
    cargo test -- --help
```

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

```
        cargo test -- --test-threads=1

```

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

Showing output

```
    cargo test -- --show-output
```

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

Running a subset of tests

```
    pub fn add_two(a:i32)->i32{
    a+2
}
#[cfg(test)]
mod tests {
    use super::*;
```

```

    #[test]
    fn add_two_and_two(){
        assert_eq!(4, add_two(2));
    }

   #[test]
    fn add_three_and_two(){
        assert_eq!(5, add_two(3));
    }

    #[test]
    fn one_hundred(){
        assert_eq!(102, add_two(100));
    }

}
```

In this example we can:

- Only run a specific test by specifiying 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.

Ignoring tests

```
    #[cfg(test)]
mod tests {
    #[test]
    fn it_works(){
        assert_eq!(2+2,4);
    }

    #[test]
    #[ignore] // This makes it so that we ignore this tests when we run cargo test
    fn expensive_test(){
        // code that takes an hour to run
    }
}
```

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.

Unit test

```
    pub fn add_two(a:i32)->i32{
    internal_adder(a,2)
}

fn internal_adder(a:i32, b:i32) ->i32{
    a+b
}

// Convention that in the same file as our product code we have a module named test, that runs tests.
#[cfg(test)] // This makes it so that we only compile the bellow code when we run test.
mod tests { //Test module, holds our test.
    use super::*;

    #[test]
    fn internal(){
        assert_eq!(4, internal_adder(2, 2)); // internal_adder is private but we can call it since in rust
    , child modules can access anything in parent module including private fields.
    }
}
```

Integration Test

```
//In the test folder in the root file, file path is adder/tests/integration_test.rs
use adder;

#[test]
fn it_adds_two(){
    assert_eq!(4, adder::add_two(2)); // Need to call the public API, can't call the inner adder function
}
```

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.

Passing in arguments

```
    // Passing the arguments needle and haystack:
    cargo run needle haystack

    // Our code prints out
    ["target/debug/minigrep", "needle", "haystack"]
```

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.

Isolating query and filename

```
    use std::env;
fn main() {
    let args: Vec<String> = env::args().collect();
    let query = &args[1];
    let filename = &args[2];

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

Next step is to read in file.

Read from file

```
use std::env;
use std::fs;

fn main() {
    let args: Vec<String> = env::args().collect();
    let query = &args[1];
    let filename = &args[2];

    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);
}
```

Running this program with example file: poem.txt

```
\\ Cargo run on the file poem.txt:
cargo run the poem.txt

\\Output of our program:
Searching for the
In file poem.txt
With text:
I'm nobody! Who are you?
Are you nobody, too?
Then there's a pair of us - don't tell!
They'd banish us, you know.

How dreary to be somebody!
How public, like a frog
To tell your name the livelong day
To an admiring bog!
```

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

13.2 First problem: Extracting out our code

main.rs has too many responsiblities. 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::env;
use std::fs;

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

    let (query, filename) = parse_config(&args);

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

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

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

fn parse_config(args: &[String]) -> (&str,&str){
    let query = &args[1];
    let filename = &args[2];
    (query, filename)
}
```

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

Config struct

```
    use std::env;
use std::fs;

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

    let config= parse_config(&args);

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

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

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

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

fn parse_config(args: &[String]) -> Config{
    let query = args[1].clone(); // Use the clone method since we don't want to take ownership
    let filename = args[2].clone();

    Config {query, filename}
}
```

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);
}

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}
    }
```

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,
}

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})
    }
}
```

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;
use std::process;

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);

    run(config);

}

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

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

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

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

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.

lib.rs

```
    use std::fs;
use std::error::Error;



pub fn run(config: Config) -> Result<(), Box<dyn Error>>{ // Need to make functions and structs that we want to use ourside this file public
    let contents = fs::read_to_string(config.filename)?;

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

    Ok(())

}

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

impl Config{
    pub 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})
    }
}
```

main.rs

```
    use std::env;
use std::process;

use minigrep::Config;

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

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

```

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

    if let Err(e) = minigrep::run(config){
        println!("Application error: {}", e); //If run returns error then we execture this code block.
        process::exit(1);
    }

}
```

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.

Our test function

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

    #[test]
    fn one_result(){
        let query = "duct";
        let contents = "\
Rust
safe, fast, productive.
Pick three";
        assert_eq!(vec!["safe, fast, productive"], search(query, contents)); \\Search function doesn't
    exits yet
    }
}
```

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.

Working search function

```
    pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str>{// Recall we need to put in
    lifetimes
    let mut results = Vec::new(); // Create our results vector, making it mutable.
    for line in contents.lines(){  // loop over the lines in our content str
        if line.contains(query){
            results.push(line); // push line in resutls if it contains he query
        }
    }
    results
}
```

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(())
}
```

Output of our program with query: frog

```
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 tests

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

    #[test]
    fn case_sensitive(){
        let query = "duct";
        let contents = "\
Rust
safe, fast, productive.
Pick three
Duct tape.";
        assert_eq!(vec!["safe, fast, productive."], search(query, contents));
    }

    #[test]
    fn case_insensitive() {
        let query = "rUsT";
        let contents = "\
Rust:
safe, fast, productive.
Pick three.
Trust me.";

        assert_eq!(
            vec!["Rust:", "Trust me."],
            search_case_insensitive(query, contents)
        );
    }
}
```

Creating our failling function

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

Making this test pass

```
    pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str>{
    let query = query.to_lowercase(); // Note to_lowercase() returns a new string, so we are not modifying
     strings.
    let mut results = Vec::new();
    for line in contents.lines(){
        if line.to_lowercase().contains(&query){
            results.push(line);
        }
    }
    results
}
```

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.

Environment variables

```
    pub fn run(config: Config) -> Result<(), Box<dyn Error>>{
    let contents = fs::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 filename: String,
    pub case_sensitive: bool,
}

impl Config{
    pub 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();

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

eprintln!

```
    use std::env;
use std::process;

use minigrep::Config;

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

    let config= Config::new(&args).unwrap_or_else(|err|{
        eprintln!("Problem parsing arguments {}", err); // eprintln! prints to the standard error stream
        process::exit(1);
    });

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

    if let Err(e) = minigrep::run(config){
        eprintln!("Application error: {}", e);
        process::exit(1);
    }
}
```

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 infered 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>,
}
```

Closuures 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.

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

fn simulated_expensive_calculation(intensity: u32) -> u32{
    println!("calculating slowly....");
    thread::sleep(Duration::from_secs(2));
    intensity
}

fn main(){
    let simulated_intensity = 10;
    let simulated_randon_number = 7;

    generate_worlout(simulated_intensity, simulated_randon_number);
}

fn generate_worlout(intensity: u32, random_number: u32){
    if intensity < 25{
        println!(
            "Today, do {} pushups!",
            simulated_expensive_calculation(intensity)
        );
        println!(
            "Today, do {} situps!",
            simulated_expensive_calculation(intensity)
        );
    } else {
        if random_number == 3{
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes",
                simulated_expensive_calculation(intensity)
            );
        }
    }
}

```

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.

```rust
    fn generate_worlout(intensity: u32, random_number: u32){
    let expensive_result = simulated_expensive_calculation(intensity);
    if intensity < 25{
        println!(
            "Today, do {} pushups!",
            expensive_result
        );
        println!(
            "Today, do {} situps!",
            expensive_result
        );
    } else {
        if random_number == 3{
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes",
                expensive_result
            );
        }
    }
}
```

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

```rust

struct Cacher<T>
where
```

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

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

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


fn generate_worlout(intensity: u32, random_number: u32){
    let mut cached_result = Cacher::new(|num|{
        println!("calculating slowly....");
        thread::sleep(Duration::from_secs(2));
        num
    });
    if intensity < 25{
        println!(
            "Today, do {} pushups!",
            cached_result.value(intensity) // Call our expensive operation here
        );
        println!(
            "Today, do {} situps!",
            cached_result.value(intensity) // Don't call the expensive operation here, since the value was already stored.
        );
    } else {
        if random_number == 3{
            println!("Take a break today! Remember to stay hydrated!");
        } else {
            println!(
                "Today, run for {} minutes",
                cached_result.value(intensity)
            );
        }
    }
}
```

We might want to use our cacher in different contexts, but there are two problems prevemt 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;

```

```
        let equal_to_x = |z| z == x; // Even though x is defined outside of our closure our function still has access.

        let y = 4;

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

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));
}
```

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

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

1. Taking Ownership

2. Borrowing mutably

3. Borrowing immmutably

These three ways are encoded in the function traits:

1. FnOnce, FnOnce takes ownership of the variables inside the closures environment. Since closures can't take ownership of the same variables more than once, these closures can only be called once.

2. FnMut, Mutably borrows values.

3. Fn, Immutably borrows values.

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

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

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

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

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

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

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