Rust Course notes

Razafy Rindra, "Hagamena"

August 5, 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 immutable 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 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 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)

```

```rust
8          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:

```rust
1  fn main(){
2      let x = 5;
3      let y = x; // Copy
4
5      let s1 = String::from("hello");
6      let s2 = s1.clone();
7
8      println!("{}, world", s1);
9  }
```

3.2   Ownership and functions

Example 3.0.2.

```rust
1          fn main(){
2              let s = String::from("hello);
3              takes_ownership(s);
4              println!("{}", s);
5          }
6
7          fn takes_ownership(some_string: String){
8              println!("{}", some_string);
9          }
10
```

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.

```rust
1          fn main() {
2              let s1 = gives_ownership();
3              println!("s1 = {}", s1);
4          }
5
6          fn gives_ownership(){
7              let some_string = String::from("hello");
8              some_string // returning the string moves ownership to s1
9          }
10
```

Example 3.0.4.

```rust
1          fn main() {
2      let s1 = gives_ownership();              // gives_ownership moves its return
3                                               // value into s1
4
5          let s2 = String::from("hello");      // s2 comes into scope
6
7          let s3 = takes_and_gives_back(s2);   // s2 is moved into
8                                               // takes_and_gives_back, which also
9                                               // moves its return value into s3
10     } // Here, s3 goes out of scope and is dropped. s2 was moved, so nothing
11       // happens. s1 goes out of scope and is dropped.
12
13     fn gives_ownership() -> String {                // gives_ownership will move its
14                                                     // return value into the function
15                                                     // that calls it
16
17         let some_string = String::from("yours"); // some_string comes into scope
18
19         some_string                                 // some_string is returned and
20                                                     // moves out to the calling
```

```
21                                               // function
22      }
23
24      // This function takes a String and returns one
25      fn takes_and_gives_back(a_string: String) -> String { // a_string comes into
26                                               // scope
27
28          a_string  // a_string is returned and moves out to the calling function
29      }
30
```

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

```
1      fn main() {
2      let s1 = String::from("hello");
3
4      let (s2, len) = calculate_length(s1);
5
6      println!("The length of '{}' is {}.", s2, len);
7  }
8
9  fn calculate_length(s: String) -> (String, usize) {
10     let length = s.len(); // len() returns the length of a String
11
12     (s, length)
13 }
```

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.

```
1      fn main() {
2          let s1 = String::from("hello");
3
4          let len = calculate_length(&s1);
5
6          println!("The length of '{}' is {}.", s1, len);
7      }
8
9      fn calculate_length(s: &String) -> usize {
10         let length = s.len(); // len() returns the length of a String
11         length
12     }
```

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:

```
1      fn main() {
2      let mut s1 = String::from("hello");
3
4      change(&mut s1);
5  }
6
7  fn change(some_string: &mut String) {
8      some_string.push_str(", world");
9  }
```

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. 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.

```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

```rust
//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

```rust
#[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

```
1      //Using if let syntax
2
3  fn main(){
4      let some_value = Some(3);
5  // Instead of using the match like this, when we only have on case we care about.
6      match some_value{
7          Some(3) => println!("three"),
8          _=> (),
9      }
10
11 // We can use the if-let synyax
12
13      if let Some(3) = some_value{ // Says if some_value matches with Some(3) execute the bellow code
14          println!("three");
15      }
16 }
```

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

```rust
//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

```rust
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.

```rust
 1   mod front_of_house {
 2       pub mod hosting {
 3           pub fn add_to_waitlist() {}
 4       }
 5   }
 6
 7   // 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.
 8
 9   // use self::front_of_house::hosting; relative path with "self" referenceing the current module
10
11   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"
12
13
14   pub fn eat_at_restaurant() {
15       let secret_number = rand::thread_rng().gen_range(1, 101);
16       hosting::add_to_waitlist();
17       hosting::add_to_waitlist();
18       hosting::add_to_waitlist();
19   }
20
21
22
23
24   /* Could have brought the funtion into scope to not have to use hosting::
25    But in Rust the idiomatic way to bring a function into scope is to bring it's parent module into scope
26    In order to make it clear that add_to_waitlist() function is not a local function.
27
28    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.
29
30   For example here:
31
32
33   use std::fmt;
34   use std::io;
35
36   fn function1() -> fmt::Result {
37       // --snip--
38       Ok(())
39   }
40
41   fn function2() -> io::Result<()> {
42       // --snip--
43       Ok(())
44   }
45
46   Another way of doing this is to rename our function:
47
48   use std::fmt::Result;
49   use std::io::Result as IoResult;
50
51   fn function1() -> Result {
52       // --snip--
53       Ok(())
54   }
55
56   fn function2() -> IoResult<()> {
57       // --snip--
58       Ok(())
59   }
60   */
```

Using external packages

```
1     //Use keyword
2
3     /*Instead of this:
4     use rand::Rng;
5     use rand::CryptoRng;
6     use rand::ErrorKind::Transient;
7     We can do this:
8     */
9
10    use rand::{Rng, CryptoRng, ErrorKind::Transient}; // Nested paths
11
12
13    /*
14    Instead of this:
15    use std::io
16    use std::io::Write
17
18    We can do this:
19
20    */
21
22    use std::io::{self, Write};
23
24
25    //Glob operator:
26    // 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:

```rust
// 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();
}
```

```rust
//In src/front_of_house.rs

    pub mod hosting;
```

```rust
// 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.

```rust
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.

```rust
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
```
 1    fn main() {
 2        a();
 3    }
 4
 5    fn a(){
 6        b();
 7    }
 8
 9    fn b(){
10        c(22);
11    }
12
13    fn c(num: i32){
14        if num == 22{
15            panic!("Do not pass in 22!");
16        }
17    }
```

Running it with backtrace to see where we have the bad function call
```
 1     RUST_BACKTRACE=1 cargo run
 2      Finished dev [unoptimized + debuginfo] target(s) in 0.00s
 3       Running `target/debug/errors`
 4   thread 'main' panicked at 'Do not pass in 22!', src/main.rs:15:9
 5   stack backtrace:
 6      0: rust_begin_unwind
 7               at /rustc/a8314ef7d0ec7b75c336af2c9857bfaf43002bfc/library/std/src/panicking.rs:584:5
 8      1: core::panicking::panic_fmt
 9               at /rustc/a8314ef7d0ec7b75c336af2c9857bfaf43002bfc/library/core/src/panicking.rs:142:14
10      2: errors::c
11               at ./src/main.rs:15:9
12      3: errors::b // c is being called by b
13               at ./src/main.rs:10:5
14      4: errors::a // b is being called by a
15               at ./src/main.rs:6:5
16      5: errors::main
17               at ./src/main.rs:2:5
18      6: core::ops::function::FnOnce::call_once
19               at /rustc/a8314ef7d0ec7b75c336af2c9857bfaf43002bfc/library/core/src/ops/function.rs:248:5
20   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:

```
 1    enum Result<T,E>{
 2        Ok(T), // represents success
 3        Err(E), // represents failure
 4    }
```

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

```
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

```
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

```
1  #![allow(unused)]
2
3  use std::fs::File;
4  use std::io::{self, Read};
5  fn read_username_from_file() -> Result<String, io::Error> {
6      let f = File::open("hello.txt");
7
8      let mut f = match f {
9          Ok(file) => file, //If opening the file succeds we take that file and store it in f
10         Err(e) => return Err(e), // If opening the file fails we take the error and return it
11     };
12
13     let mut s = String::new();
14
15     match f.read_to_string(&mut s) { // Read the contents of the file and store it in the string
16         Ok(_) => Ok(s),
17         Err(e) => Err(e),
18     } //read_to_string returns a result type, if we have success we return the string in error we return
           error.
19
20     }
21     fn main() {}
```

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

```
1
2  #![allow(unused)]
3  use std::fs::File;
4  use std::io::{self, Read};
5
6  fn read_username_from_file() -> Result<String, io::Error> {
7      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
8      let mut s = String::new();
9
10     f.read_to_string(&mut s)?; // If this call fails, the function will return error
11     Ok(s) //If we get to this line, read_to_string succeeded so we return the string
12 }
13 fn main() {}
```

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

Reducing our codes to three lines

```
1
2  #![allow(unused)]
3  use std::fs::File;
4  use std::io::{self, Read};
5
6  fn read_username_from_file() -> Result<String, io::Error> {
7      let mut s = String::new();
8      File::open("hello.txt")?.read_to_string(&mut s)?;
9      Ok(s)
10 }
11 fn main() {}
```

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

Ultimate implematation

```
1
2  #![allow(unused)]
3  use std::fs::{self, File};
4  use std::io::{self, Read};
5
6  fn read_username_from_file() -> Result<String, io::Error> {
7      fs::read_to_string("hello.txt")
8  }
9
10 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:

```
1
2  fn main(){
3      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
```
1      #![allow(unused)]
2      use std::error::Error;
3      use std::fs::File;
4
5      fn main() -> Result<(), Box<dyn Error>>{ //In the error case we return a "Trait" object, which we will
        talk about in chap 17
6          let f = File::open("hello.txt")?;
7
8          Ok(())
9
10     }
```

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:
```
1      fn main() {
2      let number_list = vec![34, 50, 25, 100, 65];
3
4      let mut largest = number_list[0];
5
6      for number in number_list {
7          if number > largest {
8              largest = number;
9          }
10     }
11
12     println!("The largest number is {}", largest); // prints 100
13 }
```

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:

```rust
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

```rust
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

```rust
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.

```rust
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

```rust
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

```rust
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:

```rust
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?

```rust
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

```
1    // Put in previous Implementations of Tweet, NewsArticle, Summary are to be put here
2
3    pub fn notify(item: &impl Summary){ // This function takes in a reference to something that implements
       summary
4        println!("Breaking news! {}", item.summarize());
5    }
6
7
8    fn main(){
9        \\ Put in the previous defintions of tweet and article
10       notify(&article);
11   }
```

This code outputs

```
1    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

```
1
2    // Put in previous Implementations of Tweet, NewsArticle, Summary are to be put here
3    pub fn notify<T:Summary>(item: &T){ // Generic that is limited to something that interprets a Summary
       trait.
4        println!("Breaking news! {}", item.summarize());
5  } // 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

```
1    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.
2        //...
3  }
4
5    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.
6        //...
7  }
```

Specifying multiple traits and the where clause

```
1    // impl syntax:
2
3
4    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
5        //...
6  }
7
8    // //Trait bound syntax
9
10 pub fn notify<T:Summary+Display>(item1: &T, item2: &T){
11        //....
12 }
13
14   // Specifying multiple trait bounds can hinder readablity, for example:
15
16 fn some_function<T:Display+Clone, U: Clone + Debug>(t:&T, u:&U) ->i32{....}
17
18
19   // To make it more readible we can use the "where"-clause
20
21 fn some_function<T,U>(t:&T,u:&U) ->i32
22        where T:Display+Clone,
23              U: Clone+Debug
24    {
25      ///..
26    }
```

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(){
```

```rust
    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.

```rust
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);
}
```

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

```
#[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

```
    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

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

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

```
 8  use super::*;
 9      #[test]
10      fn greeting_contains_name(){
11          let result = greeting("Carol");
12          assert!(
13              result.contains("Carol"),
14              "Greeting did not contain name, value was {}", result // Custom failure message
15          );
16      }
17  }
```

Failure message

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

Asserting a function that panics

```
 1      pub struct  Guess{
 2      value: i32,
 3  }
 4
 5
 6  impl Guess{
 7      pub fn new(value:i32) -> Guess{
 8          if value < 1 || value >100{
 9              panic!("Guess value must be between 1 and 100, got {}.", value);
10          }
11
12          Guess {value}
13      }
14  }
15
16  #[cfg(test)]
17  mod tests {
18  use super::*;
19      #[test]
20      #[should_panic] // Asserts that the test panics
21      fn greater_than_100(){
22          Guess::new(200);
23      }
24  }
```

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

Many panics

```
 1      pub struct  Guess{
 2      value: i32,
 3  }
 4
 5
 6  impl Guess{
 7      pub fn new(value:i32) -> Guess{
 8          if value < 1{
 9              panic!("Guess value must be greater than or equal to 1, got {}", value);
10          }else if value > 100{
11              panic!("Guess value must be less than or equal to 100, got {},", value)
12          }
13
14          Guess {value}
15      }
16  }
17
18  #[cfg(test)]
19  mod tests {
20  use super::*;
21      #[test]
22      #[should_panic(expected = "Guess value must be less than or equal to 100")]
23      fn greater_than_100(){
24          Guess::new(200);
25      }
26  }
```

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
```
1  #[cfg(test)]
2  mod tests {
3      #[test]
4      fn it_works() -> Result<(), String>{
5          if 2+2 == 4{
6              Ok(())
7          } else {
8              Err(String::from("two plus two not equal to four!"))
9          }
10     }
11 }
```

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:

```
1    cargo test —help
```

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

```
1    cargo test — —help
```

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

```
1        cargo test — —test-threads=1
2
```

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
```
1    cargo test — —show-output
```

```
1   fn prints_and_returns_10(a: i32) -> i32{
2       println!("I got the value {}", a);
3       10
4  }
5      #[cfg(test)]
6      mod tests {
7          use super::*;
8
9          #[test]
10         fn this_test_will_pass(){
11             let value = prints_and_returns_10(4); // This code will print out if we use above command but
    not by default
12             assert_eq!(10,value);
13         }
14
15
16         #[test]
17         fn this_test_will_fail(){
18             let value = prints_and_returns_10(8); // This code will always print since the test fails
19             assert_eq!(5, value);
20         }
21     }
```

Running a subset of tests
```
1      pub fn add_two(a:i32)->i32{
2      a+2
3  }
4  #[cfg(test)]
5  mod tests {
6      use super::*;
```

```
 7
 8        #[ test ]
 9        fn  add_two_and_two(){
10            assert_eq!(4,  add_two(2));
11        }
12
13        #[ test ]
14        fn  add_three_and_two(){
15            assert_eq!(5,  add_two(3));
16        }
17
18        #[ test ]
19        fn  one_hundred(){
20            assert_eq!(102,  add_two(100));
21        }
22
23 }
```

In this example we can:

- Only run a specific test by specifiying it's name:

```
1            cargo  test  one_hundred
2
```

- Run tests with a common part of their name:

```
1            cargo  test  add
2
```

This will run tests for add_two_and_two and add_three_and_two

- Run tests by specifying the module:

```
1            cargo  test  tests::
2
```

Which will run all three tests in the module.

Ignoring tests

```
 1        #[cfg(test)]
 2 mod tests {
 3        #[ test ]
 4        fn  it_works(){
 5            assert_eq!(2+2,4);
 6        }
 7
 8        #[ test ]
 9        #[ignore]  //  This  makes  it  so  that  we  ignore  this  tests  when  we  run  cargo  test
10        fn  expensive_test(){
11            //  code  that  takes  an  hour  to  run
12        }
13 }
```

To run ignored test

```
1      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.

```
1    pub fn add_two(a:i32)->i32{
2        internal_adder(a,2)
3    }
4
5    fn internal_adder(a:i32, b:i32) ->i32{
6        a+b
7    }
8
9    // Convention that in the same file as our product code we have a module named test, that runs tests.
10   #[cfg(test)] // This makes it so that we only compile the bellow code when we run test.
11   mod tests { //Test module, holds our test.
12       use super::*;
13
14       #[test]
15       fn internal(){
16           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.
17       }
18   }
```

Integration Test

```
1    //In the test folder in the root file, file path is adder/tests/integration_test.rs
2    use adder;
3
4    #[test]
5    fn it_adds_two(){
6        assert_eq!(4, adder::add_two(2)); // Need to call the public API, can't call the inner adder function
7    }
```

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

```
1        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

```
1        use adder;
2
3    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
4
5    #[test]
6    fn it_adds_two(){
7        common::setup(); //Using the module here
8        assert_eq!(4, adder::add_two(2));
```

mod.rs file

```
1    // Path /adder/tests/common/mod.rs
2        pub fn setup(){
3        // set up code
4    }
```

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

```rust
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

```rust
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

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

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

```
 6
 7      let config= Config::new(&args);
 8
 9      println!("Searching for {}", config.query);
10      println!("In file {}", config.filename);
11
12      let contents = fs::read_to_string(config.filename)
13      .expect("Something went wrong reading the file");
14
15      println!("With text:\n{}", contents);
16 }
17
18 struct Config{
19      query: String,
20      filename: String,
21 }
22
23 impl Config{
24      fn new(args: &[String]) -> Config{ // Calling this function new is convention for constructor
         functions.
25          let query = args[1].clone();
26          let filename = args[2].clone();
27
28          Config {query, filename}
29      }
30 }
```

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.

```
1      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:

```
1      fn new(args: &[String]) -> Config{
2          if args.len()<3{
3              panic!("not enough arguments");
4          }
5
6          let query = args[1].clone();
7          let filename = args[2].clone();
8
9          Config {query, filename}
10     }
```

Now the error message is

```
1      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
```
1      use std::env;
2  use std::fs;
3  use std::process; // Let's us exit program without panicking.
4
5  fn main() {
6      let args: Vec<String> = env::args().collect();
7
8      let config= Config::new(&args).unwrap_or_else(|err|{
9          println!("Problem parsing arguments {}", err);
10         process::exit(1);
11     });
12
13     println!("Searching for {}", config.query);
14     println!("In file {}", config.filename);
15
16     let contents = fs::read_to_string(config.filename)
17     .expect("Something went wrong reading the file");
18
19     println!("With text:\n{}", contents);
20 }
21
```

```rust
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

```rust
    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
```rust
    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);
```

```rust
     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
```rust
     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
```rust
     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);
     });
```

```
14        println!("Searching for {}", config.query);
15        println!("In file {}", config.filename);
16
17        if let Err(e) = minigrep::run(config){
18            println!("Application error: {}", e); //If run returns error then we exectrue this code block.
19            process::exit(1);
20        }
21
22 }
```

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
```
1        #[cfg(test)]
2 mod test{
3        use super::*;
4
5        #[test]
6        fn one_result(){
7            let query = "duct";
8            let contents = "\
9 Rust
10 safe, fast, productive.
11 Pick three";
12            assert_eq!(vec!["safe, fast, productive"], search(query, contents)); \\Search function doesn't
       exits yet
13        }
14 }
```

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

```
1        pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str>{// Recall we need to put in
       lifetimes
2        vec![]
3 }
```

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
```
1        pub fn search<'a>(query: &str, contents: &'a str) -> Vec<&'a str>{// Recall we need to put in
       lifetimes
2        let mut results = Vec::new(); // Create our results vector, making it mutable.
3        for line in contents.lines(){  // loop over the lines in our content str
4            if line.contains(query){
5                results.push(line); // push line in resutls if it contains he query
6            }
7        }
8        results
9 }
```

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

new run function
```
1        pub fn run(config: Config) -> Result<(), Box<dyn Error>>{
2        let contents = fs::read_to_string(config.filename)?;
3
4            for line in search(&config.query,&contents){
5                println!("{}", line);
6            }
7
8        Ok(())
9 }
```

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

Output of our program with query: dog
```
1     Searching for dog
2     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
```
1  #[cfg(test)]
2  mod test{
3      use super::*;
4
5      #[test]
6      fn case_sensitive(){
7          let query = "duct";
8          let contents = "\
9  Rust
10 safe , fast , productive.
11 Pick three
12 Duct tape.";
13         assert_eq!(vec!["safe , fast , productive."], search(query, contents));
14     }
15
16     #[test]
17     fn case_insensitive() {
18         let query = "rUsT";
19         let contents = "\
20 Rust:
21 safe , fast , productive.
22 Pick three.
23 Trust me.";
24
25         assert_eq!(
26             vec!["Rust:", "Trust me."],
27             search_case_insensitive(query, contents)
28         );
29     }
30 }
```

Creating our failling function
```
1      pub fn search_case_insensitive<'a>(
2      query: &str ,
3      contents: &'a str ,
4  ) -> Vec<&'a str >{
5      vec![]
6  }
```

Making this test pass
```
1      pub fn search_case_insensitive<'a>(
2      query: &str ,
3      contents: &'a str ,
4  ) -> Vec<&'a str >{
5      let query = query.to_lowercase(); // Note to_lowercase() returns a new string , so we are not modifying
        strings .
6      let mut results = Vec::new();
7      for line in contents.lines(){
8          if line.to_lowercase().contains(&query){
9              results.push(line);
10         }
11     }
12     results
13 }
```

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.

```
1    pub fn run(config: Config) -> Result<(), Box<dyn Error>>{
2        let contents = fs::read_to_string(config.filename)?;
3
4        let results = if config.case_sensitive{
5            search(&config.query, &contents)
6        }else{
7            search_case_insensitive(&config.query, &contents)
8        };
9
10       for line in results{
11               println!("{}", line);
12       }
13
14       Ok(())
15   }
16
17   pub struct Config{
18       pub query: String,
19       pub filename: String,
20       pub case_sensitive: bool,
21   }
22
23   impl Config{
24       pub fn new(args: &[String]) -> Result<Config, &str>{
25           if args.len()<3{
26               return Err("not enough arguments");
27           }
28
29           let query = args[1].clone();
30           let filename = args[2].clone();
31
32           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
33           // .is_err() returns a boolean. If Result was Ok it returns true, otherwise it returns false.
34
35           Ok(Config {query, filename, case_sensitive})
36       }
37   }
```

Now if we run our code normally:

cargo run to poem.txt
```
1    Searching for to
2    In file poem.txt
3    Are you nobody, too?
4    How dreary to be somebody!
```

Now set CASE_INSENSITIVE to true:

```
1        export CASE_INSENSITIVE=true
```

cargo run to poem.txt
```
1        Searching for to
2        In file poem.txt
3        Are you nobody, too?
4        How dreary to be somebody!
5        To tell your name the livelong day
6        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
```
1    Problem parsing arguments not enough arguments
```

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

eprintln!
```
1     use std::env;
2  use std::process;
3
4  use minigrep::Config;
5
6  fn main() {
7      let args: Vec<String> = env::args().collect();
8
9      let config= Config::new(&args).unwrap_or_else(|err|{
10         eprintln!("Problem parsing arguments {}", err); // eprintln! prints to the standard error stream
11         process::exit(1);
12     });
13
14     println!("Searching for {}", config.query);
15     println!("In file {}", config.filename);
16
17     if let Err(e) = minigrep::run(config){
18         eprintln!("Application error: {}", e);
19         process::exit(1);
20     }
21 }
```

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
```
1    Searching for to
2  In file poem.txt
3  Are you nobody, too?
4  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
```
1     struct Cacher<T>
2  where
3      T: Fn(u32)->u32, // Any function that takes in an u32 and returns a u32
4  {
5      calculation: T,
6      value: Option<u32>,
7  }
```

Closuures can't have many types infered for each parameters
```
1     let example_closure = |x| x;
2     let s = example_closure(String::from("hello")); // Compiler infers that the example closure takes in a string and outputs a string.
3     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
```

```rust
    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 implemantation 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:

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

```
4        let equal_to_x = |z| z == x; // Even though x is defined outside of our closure our function still
     has access.
5
6        let y = 4;
7
8        assert!(equal_to_x(y));
9    }
```

Now what happens if we change closure to function?

```
1  fn main(){
2      let x = 4;
3
4      fn equal_to_x(z: i32) -> bool {
5          z == x //Error: can't capture dynamic environment in a fn item use the `|| { ... }` closure form
     instead
6      }
7
8
9      let y = 4;
10
11     assert!(equal_to_x(y));
12 }
```

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 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.

```
1   fn main(){
2       let x = vec![1,2,3];
3
4       let equal_to_x = move |z| z == x; // Closure takes ownership of x
5
6       println!("Can't use x here: {:?}", x); // Get error here since we are using a borrowed value after
     it has been moved.
7
8       let y = vec![1,2,3];
9
10      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

```
1    fn main() {
2        let v1 = vec![1,2,3];
3        let v1_iter = v1.iter();
4
5        for value in v1_iter{
6            println!("Got {}", value);
7        }
8    }
```

Output

```
1   Got 1
2   Got 2
3   Got 3
```

15.1   Iterator trait

How do iterators work? All iterators implement the iterator trait

```
1    pub trait Iterator{
2        type Item; // associated type
3        fn next(&mut self) -> Option<Self::Item>;
4
5        // methods with default implematation elided
6    }
```

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

Test example

```
1   #[test]
2   fn iterator_demonstration(){
3       let v1 = vec![1,2,3];
4
5       let mut v1_iter = v1.iter();
6
7       // .iter immutatable reference
8       // .iter_mut mutable reference
9       // into_iter to own types
10
11      assert_eq!(v1_iter.next(), Some(&1));
12      assert_eq!(v1_iter.next(), Some(&2));
13      assert_eq!(v1_iter.next(), Some(&3));
14      assert_eq!(v1_iter.next(), None);
15   }
```

Definition 15.2. There are two types of methods on iterators

1. **Adaptators** they take in an iterator and return another iterator

2. **Consumer** they take in a iterator and returns another type.

The consumer method, sum

```
1   #[test]
2   fn iterator_sum(){
3       let v1 = vec![1,2,3];
4
5       let v1_iter = v1.iter();
6
7       let total: i32 = v1_iter.sum();
8
9       assert_eq!(total,6);
10   }
```

The adaptor method, map

```
1    fn main() {
2        let v1: Vec<i32> = vec![1,2,3];
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

```
1  fn main() {
2      let v1: Vec<i32> = vec![1,2,3];
3      let v2: Vec<_> = v1.iter().map(|x| x+1).collect();
4
5      assert_eq!(v2,vec![2,3,4]); // true
6  }
```

Example

```
1      #[derive(PartialEq,Debug)]
2  struct Shoe{
3      size: u32,
4      style: String,
5  }
6
7  fn shoes_in_my_size(shoes: Vec<Shoe>, shoe_size: u32) -> Vec<Shoe>{
8      shoes.into_iter().filter(|s| s.size == shoe_size).collect()
9      // into_iter -> create an iterator that takes ownership
10     // filter takes in closure and creates a new iterator, with only the elements that return true when
       passed into the closure
11     // collect creates vector of shoe.
12 }
13
14 #[cfg(test)]
15 mod tests {
16     use super::*;
17
18     #[test]
19     fn filters_by_size() {
20         let shoes = vec![
21             Shoe {
22                 size: 10,
23                 style: String::from("sneaker"),
24             },
25             Shoe {
26                 size: 13,
27                 style: String::from("sandal"),
28             },
29             Shoe {
30                 size: 10,
31                 style: String::from("boot"),
32             },
33         ];
34
35         let in_my_size = shoes_in_my_size(shoes, 10);
36
37         assert_eq!(
38             in_my_size,
39             vec![
40                 Shoe {
41                     size: 10,
42                     style: String::from("sneaker")
43                 },
44                 Shoe {
45                     size: 10,
46                     style: String::from("boot")
47                 },
48             ]
49         );
50     }
51 }
```

15.2   Creating our own iterators

Iterator on Counter Struct

```
1      struct Counter{
2      count: u32
3  }
4
5  impl Counter{
6      fn new() -> Counter{
7          Counter {count: 0}
8      }
```

```
 9  }
10
11  impl Iterator for Counter{
12      type Item = u32;  // See more about assocaited types later.
13
14      fn next(&mut self) -> Option<Self::Item>{ // next is the only method we need to implement.
15          if self.count < 5{
16              self.count += 1;
17              Some(self.count)
18          }else {
19              None
20          }
21      }
22  }
23
24  #[test]
25  fn calling_next_directly(){
26      let mut counter = Counter::new();
27
28      assert_eq!(counter.next(), Some(1));
29      assert_eq!(counter.next(), Some(2));
30      assert_eq!(counter.next(), Some(3));
31      assert_eq!(counter.next(), Some(4));
32      assert_eq!(counter.next(), Some(5));
33      assert_eq!(counter.next(), None);
34  }
35
36  fn main() {}
```

Default implementations for other methods on our iterator
```
1      #[test]
2  fn using_other_iterator_trait_methods(){
3      let sum: u32 = Counter::new()// Creating a new counter
4          .zip(Counter::new().skip(1))
5          .map(|(a,b)| a*b) // map takes a closure and calls for each item in the iterator.
6          .filter(|x|x%3 == 0)// Filter to only items divisible by three
7          .sum(); // take the sum
8      assert_eq!(18,sum);
9  }
```

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

15.3   Using Iterators to refactor the CLI program we made

Note in our CLI program we created an vector of strings called args using env::args.collect(), but env::args is an iterator.

In our implementation block for config we had to use the clone() method to get the query and filename since args is a reference of to a [String] so we can't take ownership. Since we have ownership of iteratros, let us use them to remove this clone operation.

Improved Config
```
1      impl Config{
2      pub fn new(mut args: env::Args) -> Result<Config, &'static str>{
3          args.next(); // Returns our first command line argument, we are discarding it since it is the path
      to our program which we don't care about.
4
5          let query = match args.next(){ //query is taking ownership of it's string
6           Some(arg) => arg,
7           None => return Err("Didn't get a query string"),
8          };
9
10         let filename = match args.next(){//filename is taking ownership of it's string
11          Some(arg) => arg,
12          None => return Err("Didn't get a filename"),
13         };
14
15         let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
16
17         Ok(Config {query, filename, case_sensitive}) //Ok is taking ownershiup of the strings.
18      }
```

Improved search function with iterator adaptors

```rust
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.
}
```

full lib.rs file

```rust
use std::fs;
use std::error::Error;
use std::env;
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(mut args: env::Args) -> Result<Config, &'static str>{
        args.next();
        let query = match args.next(){
         Some(arg) => arg,
         None => return Err("Didn't get a query string"),
        };
        let filename = match args.next(){
         Some(arg) => arg,
         None => return Err("Didn't get a filename"),
        };
         let case_sensitive = env::var("CASE_INSENSITIVE").is_err();
         Ok(Config {query, filename, case_sensitive})
    }
}
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.
}

pub fn search_case_insensitive<'a>(
    query: &str,
    contents: &'a str,
) -> Vec<&'a str>{
    let query = query.to_lowercase();
    let mut results = Vec::new();
    for line in contents.lines(){
        if line.to_lowercase().contains(&query){
            results.push(line);
        }
    }
    results
}
// Tests here
}
```

full main.rs file

```rust
use std::env;
use std::process;
use iterators::Config;
fn main() {
    let config= Config::new(env::args()).unwrap_or_else(|err|{
        eprintln!("Problem parsing arguments {}", err);
        process::exit(1);
    });
    println!("Searching for {}", config.query);
    println!("In file {}", config.filename);
    if let Err(e) = iterators::run(config){
        eprintln!("Application error: {}", e);
        process::exit(1);
    }
}
```

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.

```
1  [package]
2  name = "my_crate"
3  version = "0.1.0"
4  edition = "2021"
5
6  # See more keys and their definitions at https://doc.rust−lang.org/cargo/reference/manifest.html
7
8  [dependencies]
9
10
11 [profile.dev]
12 opt−level = 0
13
14 [profile.release]
15 opt−level = 3
```

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

16.2   Uploading code to crates.io

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

16.2.1   Preparing our code

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

Example of documentation comment
```
1  /// Adds one to the number given.
2  ///
3  /// # Examples
4  ///
5  /// ```
6  /// let arg = 5;
7  /// let answer = my_crate::add_one(arg);
8  ///
9  /// assert_eq!(6, answer);
10 /// ```
11 pub fn add_one(x: i32) −> i32 {
12     x + 1
13 }
```

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 documetation to be in-synced with the code.

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

```
1      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 follwing the comment it documents the item containg 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:

```
    //! # Art
//!
//! A library for modeling artistic concepts.

pub mod kinds {
    /// The primary colors according to the RYB color model.
    pub enum PrimaryColor {
        Red,
        Yellow,
        Blue,
    }

    /// The secondary colors according to the RYB color model.
    pub enum SecondaryColor {
        Orange,
        Green,
        Purple,
    }
}

pub mod utils {
    use crate::kinds::*;

    /// Combines two primary colors in equal amounts to create
    /// a secondary color.
    pub fn mix(c1: PrimaryColor, c2: PrimaryColor) -> SecondaryColor {
        // --snip--
        // ANCHOR_END: here
        SecondaryColor::Orange
        // ANCHOR: here
    }
}
```

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.

```
1  pub use self::kinds::PrimaryColor;
2  pub use self::kinds::SecondaryColor;
3  pub use self::utils::mix;
```

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

```
1  use my_crate::PrimaryColor;
2  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

```
1      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

```
1      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:

```
1      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 profiels.

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

```
1      [workspace]
2
3      members =[
4          "adder",
5          "add—one",
6      ]
```

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:

```
1  [dependencies]
2  add—one = {path = "../add—one"}
```

And in main.rs we write:

In add/adder/src/main.rs

```
1  use add_one;
2
3  fn main() {
4      let num = 10;
5      println!(
6          "Hello, world! {} plus one is {}!",
7          num,
8          add_one::add_one(num)
9      );
10 }
```

In order to run our adder package we write

```
1      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

In /add-one/src/lib.rs:

```
1      use rand;
2
3  pub fn add_one(x:i32)->i32 {
4      x+1
5  }
6
7  #[cfg(test)]
8  mod tests{
9      use super::*;
10
11     #[test]
12     fn it_works(){
13         assert_eq!(3,add_one(2));
14     }
15 }
```

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

```
1      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:

```
1      cargo install ripgrep
2
```

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 adresse 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
```
fn 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,
}
```

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.

```
1    enum Message{
2    Quit,
3    Move{x:i32, y:i32},
4    Write(String),
5    ChangeColor(i32, i32, i32),
6 }
```

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 to know how much spac ethe list enum takes!

How do we fix this?

Fixing with the Box Smart Pointer

```
1    enum List{
2    Cons(i32, Box<List>),
3    Nil,
4 }
```

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

```
1    enum List{
2    Cons(i32, Box<List>),
3    Nil,
4 }
5
6 use List::{Cons,Nil};
7
8 fn main() {
9    let list = Cons(1,Box::new(Cons(2, Box::new(Cons(3, Box::new(Nil))))));
10 }
```

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.

```
1    fn main (){
2    let x = 5;
3    let y = &x;
4
5    assert_eq!(5,x); //True
6    assert_eq!(5,*y); //True
7    assert_eq!(5,y); //Error can't compare `{integer}` with `&{integer}`
8 }
```

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

Now with the Box Smart Pointer

```
1    fn main (){
2    let x = 5;
3    let y = Box::new(x);
4
5    assert_eq!(5,x);
6    assert_eq!(5,*y); // Derefence operator works the same since Box implements the deref trait
7 }
```

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

```
1    use std::ops::Deref; // Bring Deref trait into scope
2
3  struct MyBox<T>(T);
4
5  impl <T> MyBox<T>{
6      fn new(x:T) -> MyBox<T>{
7          MyBox(x) // Note unlike MyBox, Box doesn't store data on the heap.
8      }
9  }
10
11 impl<T> Deref for MyBox<T>{
12     type Target = T; // We will talk about this later
13
14     fn deref(&self) -> &T{
15         &self.0 //Return a reference to item stored in tuple struct.
16     }
17 }
18
19 fn main (){
20     let x = 5;
21     let y = MyBox::new(x);
22
23     assert_eq!(5,x);
24     assert_eq!(5,*y);
25 }
```

The deref trait allows the Rust compiler to take any value that implememnts 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:

```
1    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

```
1  // MyBox implementation here
2  fn main (){
3      let m = MyBox::new(String::from("Rust"));
4      hello(&m);
5  }
6
7  fn hello(name: &str){
8      println!("Hello, {}!", name);
9  }
```

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

Explanation

```
1    // &MyBox<String> ->(deref) &String ->(deref) &str
```

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

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

```
1    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 follwing 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.
Dropping CustomeSmartPointer with data `other stuff`!
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.

Using c.drop() directly
```
fn main() {
    let c = CustomSmartPointer{
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer{
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
    c.drop(); // Error explicit destructor calls not allowed.
    println!("Dropped before end of main");
}
```

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:

drop(c)

```
fn main() {
    let c = CustomSmartPointer{
        data: String::from("my stuff"),
    };
    let d = CustomSmartPointer{
        data: String::from("other stuff"),
    };
    println!("CustomSmartPointers created.");
    drop(c);
    println!("Dropped before end of main");
}
```

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
}
```

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 ownede 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.

```
1  fn main() {
2      let a = Rc::new(Cons(5, Rc::new(Cons(10, Rc::new(Nil)))));
3      println!("count after creating a = {}", Rc::strong_count(&a)); // 1
4      let b = Cons(3,Rc::clone(&a));
5      println!("count after creating b = {}", Rc::strong_count(&a)); // 2
6      {
7          let c = Cons(4,Rc::clone(&a));
8          println!("count after creating c = {}", Rc::strong_count(&a)); // 3
9      }
10     println!("count after c leaves scope = {}", Rc::strong_count(&a)); // 2
11 }
```

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 mutatbility 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.

```
1
2  fn main() {
3      let a = 5;
4      let b = &mut a; //Error: a is immutable so cannot borrow as mutable
5
6      let mut c = 10;
7      let d = &c;
8      *d = 20; //Error: d is an immutable reference so the data it referes to cannot be written
9  }
```

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

```
1      pub trait Messenger{
2      fn send(&self, msg: &str);
3  }
4
5  pub struct LimitTracker<'a, T:Messenger>{ // add liftime annotation since we are borrowing T
6      messenger: &'a T,
7      value: usize, // pointer−sized unsigned integer type.
8      max: usize,
9  }
10
11 impl<'a, T> LimitTracker<'a, T>
12 where
13     T:Messenger,
14 {
15     pub fn new(messenger: &T, max:usize) −> LimitTracker<T>{
```

```
16          LimitTracker{
17              messenger,
18              value:0,
19              max,
20          }
21      }
22 }
23
24 pub fn set_value(&mut self, value: usize) {
25      self.value = value;
26
27      let percentage_of_max = self.value as f64 / self.max as f64;
28
29      if percentage_of_max >= 1.0 {
30          self.messenger.send("Error: You are over your quota!");
31      } else if percentage_of_max >= 0.9 {
32          self.messenger
33              .send("Urgent warning: You've used up over 90% of your quota!");
34      } else if percentage_of_max >= 0.75 {
35          self.messenger
36              .send("Warning: You've used up over 75% of your quota!");
37      }
38 }
```

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:

```
1      #[cfg(test)]
2 mod tests {
3      use super::*;
4
5      struct MockMessenger {
6          sent_messages: Vec<String>, //vector of sent message
7      }
8
9      impl MockMessenger {
10         fn new() -> MockMessenger {
11             MockMessenger {
12                 sent_messages: vec![],
13             }
14         }
15     }
16
17     impl Messenger for MockMessenger {// pushes message to the vector
18         fn send(&self, message: &str) {
19             self.sent_messages.push(String::from(message));  //Error cannot borrow `self.sent_messages` as
    mutable, as it is behind a `&` reference
20         }
21     }
22
23     #[test]
24     fn it_sends_an_over_75_percent_warning_message() {
25         let mock_messenger = MockMessenger::new();
26         let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);
27
28         limit_tracker.set_value(80);
29
30         assert_eq!(mock_messenger.sent_messages.len(), 1); //We should have one message
31     }
32 }
```

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.

Once again with RefCell

```
#[cfg(test)]
mod tests {
    use super::*;
    use std::cell::RefCell;

    struct MockMessenger {
        sent_messages: RefCell<Vec<String>>,
    }

    impl MockMessenger {
        fn new() -> MockMessenger {
            MockMessenger {
                sent_messages: RefCell::new(vec![]),
            }
        }
    }

    impl Messenger for MockMessenger {
        fn send(&self, message: &str) {
            self.sent_messages.borrow_mut().push(String::from(message));
        }
    }

    #[test]
    fn it_sends_an_over_75_percent_warning_message() {
        let mock_messenger = MockMessenger::new();
        let mut limit_tracker = LimitTracker::new(&mock_messenger, 100);

        limit_tracker.set_value(80);

        assert_eq!(mock_messenger.sent_messages.borrow().len(), 1);
    }
}// All tests pass
```

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

Expanding our list code

```
#[derive(Debug)]

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

use crate::List::{Cons, Nil};
use std::cell::RefCell;
use std::rc::Rc;

fn main(){
    let value = Rc::new(RefCell::new(5));

    let a = Rc::new(Cons(Rc::clone(&value), Rc::new(Nil)));

    let b = Cons(Rc::new(RefCell::new(3)), Rc::clone(&a));
    let c = Cons(Rc::new(RefCell::new(4)), Rc::clone(&a));

    *value.borrow_mut() += 10; // Automatic dereferencing feature.
```

```
21
22      println!("a after = {:?}", a);
23      println!("b after = {:?}", b);
24      println!("c after = {:?}", c);
25 }
```

Output
```
1 a after = Cons(RefCell { value: 15 }, Nil)
2 b after = Cons(RefCell { value: 3 }, Cons(RefCell { value: 15 }, Nil))
3 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.

Rxample of memory leak
```
1      use crate::List::{Cons,Nil};
2 use std::cell::RefCell;
3 use std::rc::Rc;
4
5 #[derive(Debug)]
6
7 enum List{
8      Cons(i32, RefCell<Rc<List>>),
9      Nil,
10 }
11
12 impl List{
13      fn tail(&self) -> Option<&RefCell<Rc<List>>>{
14          match self{
15              Cons(_, item) => Some(item),
16              Nil => None,
17          }
18      }
19 }
20
21 fn main() {
22      let a = Rc::new(Cons(5, RefCell::new(Rc::new(Nil))));
23
24      println!("a initial rc count = {}", Rc::strong_count(&a));   // 1
25      println!("a next item ={:?}", a.tail());
26
27      let b = Rc::new(Cons(10,RefCell::new(Rc::clone(&a))));
28
29      println!("a rc count after b creation = {}", Rc::strong_count(&a)); // 2
30      println!("b initial rc count = {}", Rc::strong_count(&b)); // 1
31      println!("b next item = {:?}", b.tail()); // List a
32
33
34      if let Some(link) = a.tail(){ // udr if let since we only care about Some
35          *link.borrow_mut() = Rc::clone(&b); // We get mutable ref to the data and use dereference operator
       to change the data into list b.
36      }
37      println!("b rc count after changing a = {}", Rc::strong_count(&b)); // 2
38      println!("a rc count after changing a = {}", Rc::strong_count(&a)); // 2
39 }
40 //This line proves we have a cycle, it will overflow the stack
41 println!("a next item = {:?}", a.tail());
```

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::rc::{Rc, Weak};

#[derive(Debug)]

struct Node{
    value: i32,
    parent: RefCell<Weak<Node>>,//Can't use Rc since it will cause a reference cycle.
    children: RefCell<Vec<Rc<Node>>>,
}

fn main() {
    let leaf = Rc::new(Node{
        value: 3,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![]),
    });

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

    let branch = Rc::new(Node{
        value: 5,
        parent: RefCell::new(Weak::new()),
        children: RefCell::new(vec![Rc::clone(&leaf)]),
    });

    *leaf.parent.borrow_mut() = Rc::downgrade(&branch); // downgrade turns Rc smart pointer into weak
    smart pointer.

    println!("leaf parent = {:?}", leaf.parent.borrow().upgrade());
}
```

Output
```
leaf parent = None
leaf parent = Some(Node { value: 5, parent: RefCell { value: (Weak) }, children: RefCell { value: [Node {
    value: 3, parent: RefCell { value: (Weak) }, children: RefCell { value: [] } }] } })
```

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

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

    {
        let branch = Rc::new(Node{
            value: 5,
            parent: RefCell::new(Weak::new()),
            children: RefCell::new(vec![Rc::clone(&leaf)]),
        });
```

```
20
21          *leaf.parent.borrow_mut() = Rc::downgrade(&branch); // downgrade turns Rc smart pointer into weak
     smart pointer.
22
23          println!(
24              "branch strong = {}, weak = {}",
25              Rc::strong_count(&branch), // 1
26              Rc::weak_count(&branch), // 1, since leaf has a weak reference to branch
27          );
28
29          println!(
30              "leaf strong = {}, weak = {}",
31              Rc::strong_count(&leaf), // 2, since leaf is a child of branch
32              Rc::weak_count(&leaf), // 0
33          );
34
35      }
36      println!("leaf parent = {:?}", leaf.parent.borrow().upgrade()); //None since branch is dropped
37      println!(
38          "leaf strong = {}, weak = {}",
39          Rc::strong_count(&leaf), // 1
40          Rc::weak_count(&leaf), // 0
41      );
42 }
```

Output
```
1 leaf strong = 1, weak = 0
2 branch strong = 1, weak = 1
3 leaf strong = 2, weak = 0
4 leaf parent = None
5 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 implememntation 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

Spawning a thread
```
1     use std::{thread, time::Duration};
2
3
4 fn main() {
```

```
5      thread::spawn(||{
6          for i in 1..10{
7              println!("hi number {} from the spawned thread!", i);
8              thread::sleep(Duration::from_millis(1)); // Pause execution of thread for a milisecond.
9          }
10     });
11
12     for i in 1..5{
13         println!("hi number {} from the main thread!", i);
14         thread::sleep(Duration::from_millis(1));
15     }
16 }
```

Output
```
1 hi number 1 from the main thread!
2 hi number 1 from the spawned thread!
3 hi number 2 from the spawned thread!
4 hi number 2 from the main thread!
5 hi number 3 from the spawned thread!
6 hi number 3 from the main thread!
7 hi number 4 from the spawned thread!
8 hi number 4 from the main thread!
9 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
```
1      use std::{thread, time::Duration};
2
3
4  fn main() {
5      let handle = thread::spawn(||{
6          for i in 1..10{
7              println!("hi number {} from the spawned thread!", i);
8              thread::sleep(Duration::from_millis(1)); // Pause execution of thread for a milisecond.
9          }
10     });
11
12     for i in 1..5{
13         println!("hi number {} from the main thread!", i);
14         thread::sleep(Duration::from_millis(1));
15     }
16
17     handle.join().unwrap();
18 }
```

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
```
1  hi number 1 from the main thread!
2  hi number 1 from the spawned thread!
3  hi number 2 from the main thread!
4  hi number 2 from the spawned thread!
5  hi number 3 from the main thread!
6  hi number 3 from the spawned thread!
7  hi number 4 from the main thread!
8  hi number 4 from the spawned thread!
9  hi number 5 from the spawned thread!
10 hi number 6 from the spawned thread!
11 hi number 7 from the spawned thread!
12 hi number 8 from the spawned thread!
13 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.

```
 1 | hi number 1 from the spawned thread!
 2 | hi number 2 from the spawned thread!
 3 | hi number 3 from the spawned thread!
 4 | hi number 4 from the spawned thread!
 5 | hi number 5 from the spawned thread!
 6 | hi number 6 from the spawned thread!
 7 | hi number 7 from the spawned thread!
 8 | hi number 8 from the spawned thread!
 9 | hi number 9 from the spawned thread!
10 | hi number 1 from the main thread!
11 | hi number 2 from the main thread!
12 | hi number 3 from the main thread!
13 | 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

```
 1 |     fn main() {
 2 |     let v = vec![1,2,3];
 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.
 5 |         println!("Here's a vector: {:?}", v);
 6 |     });
 7 |
 8 |     drop(v); //oh no!
 9 |
10 |     handle.join().unwrap();
11 | }
```

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

```
 1 |     fn main() {
 2 |     let v = vec![1,2,3];
 3 |
 4 |     let handle = thread::spawn(move ||{ //closure takes ownership of v
 5 |         println!("Here's a vector: {:?}", v);
 6 |     });
 7 |
 8 |     handle.join().unwrap();
 9 | }
```

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

```
 1 |     use std::sync::mpsc; // mpsc = multi-producer, single consumer.
 2 | use std::thread;
 3 |
 4 | fn main() {
 5 |     let (tx, rx) = mpsc::channel();
 6 |
 7 |     thread::spawn(move ||{ // We need to use move keyword to move tx into the closure, in order to pass
    |     msg into tx.
 8 |         let msg = String::from("hi");
 9 |         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
```

```
10        });
11
12        let recieved = rx.recv().unwrap(); // recv() blocks main thread execution while it waits for a value
          to be sent by channel. Returns a result type; if the channel closes get error.
13
14        //let recieved = rx.try_recv().unwrap(); try_recv(), doesn't block main thread execution, instead
          returns result immediately.
15
16        println!("Got: {}", recieved);
17 }
```

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

```
1      use std::sync::mpsc;
2  use std::thread;
3  use std::time::Duration;
4
5  fn main() {
6      let (tx, rx) = mpsc::channel();
7
8      thread::spawn(move ||{
9          let vals = vec![
10              String::from("hi"),
11              String::from("from"),
12              String::from("the"),
13              String::from("thread"),
14          ];
15
16          for val in vals{
17              tx.send(val).unwrap();
18              thread::sleep(Duration::from_secs(1));
19          }
20      });
21
22      for recieved in rx{
23          println!("Got: {}", recieved);
24      }
25 }
```

Output
```
1 Got: hi
2 Got: from
3 Got: the
4 Got: thread
```

20.3.3 Creating multiple producers

```
1  fn main() {
2      let (tx, rx) = mpsc::channel();
3
4      let tx2 = tx.clone();
5
6      thread::spawn(move ||{
7          let vals = vec![
8              String::from("hi"),
9              String::from("from"),
10              String::from("the"),
11              String::from("thread"),
12          ];
13
14          for val in vals{
15              tx.send(val).unwrap();
16              thread::sleep(Duration::from_secs(1));
```

```rust
        }
    });

    thread::spawn(move ||{
        let vals = vec![
            String::from("more"),
            String::from("messages"),
            String::from("for"),
            String::from("you"),
        ];

        for val in vals{
            tx2.send(val).unwrap(); //Can't use tx since already moved in previous thread.
            thread::sleep(Duration::from_secs(1));
        }
    });

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

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: from
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
```