The `Rust` Programming Language: Notes

Moritz M. Konarski

August 23, 2020

Contents

Disclaimer:
These notes follow the book *The Rust Programming Language* about Rust.
They are simply my personal notes that I take as I go along.

Common Programming Concepts

Variables and Mutability

- default is immutable

```
let x = 5;
```

- is safer and simpler to work with
- designating a variable as mutable makes it changeable

```
let mut x = 5;
```

- the `mut` makes it clear that the variable is supposed to change at some point in the future

Immutables vs Constants

- constants are not the same as variables without `mut`
- you can never change a constant
- to declare a constant you say

```
const x: u32 = 123;
```

- `const` declares the constant and the data type must be annotated
- constants cant be set to results of functions or thing only computed at runtime

Shadowing

- we can declare a new variable with the same name as a previous variable
- the first variable is *shadowed* by the second one, its data is accessed with the identifier
- shadowing can be used to change the value of a variable without making it `mut`:

```
let x = 5;
let x = x + 1;
let x = x * 2;
```

- it can also be used to convert between data types but keep the name:

```
let spaces: String = "   ";
let spaces: u32    = spaces.len();
```

Data Types

- every value in Rust is of a specific data type
- Rust is *statically typed*, it must know the data types at compile time
- when more than one data type is possible, the programmer must specify which one should be used:

```
let guess: u32 = "42".parse()
    .expect("Not a number!");
```

Scalar Types

- single value
- four primary types: integers, floating-point numbers, booleans, characters

Integer Types

- whole number without fractional component, standard is `i32`
- signed numbers are stored using *two's complement*
- all integers except for the byte literal excepts a type suffix such as

```
57u8
```

and underscore as a visual separator like

```
1_000
```

- list of integer sizes:

Length	Signed	Unsigned
8-bit	i8	u8
16-bit	i16	u16
32-bit	i32	u32
64-bit	i64	u64
128-bit	i128	u128
arch	isize	usize

- list of integer literals:

Number Literals	Example
Decimal	98_222
Hex	0xff
Octal	0o77
Binary	0b1111_0000
Byte (u8 only)	b'A'

- integer overflow is still a thing

Floating-Point Types

- Rust has `f32` and `f64` floating-point types

- the standard is `f64`

Arithmetic Operations

Operation	Example
Addition	`let sum  = 5 + 10;`
Subtraction	`let diff = 95.5 - 4.3;`
Multiplication	`let prod = 4 * 30;`
Division	`let quot = 56.7 / 32.2;`
Remainder	`let rem  = 43 % 5;`

Boolean Type

- `true` or `false`, takes up one byte in rust

```
let t = true;
let f: bool = false;
```

Character Type

- `char` is the most basic type
- chars are 4 bytes in size and represent unicode values, are specified with single quotes

```
let c = 'z';
let d: char = 'H';
```

- unicode has a lot more than just simple characters so it might be somewhat confusing as to what `char` can store

Compound Types

- combine multiple values into one type
- Rust has two primitive compound types

Tuple Type

- groups together a variety of types into one compound type
- once declared, their size is fixed
- create tuples by writing comma separated values in parenthesis

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
let tup = (32, 64.6, 3);
```

- to access the members of a tuple, *destructuring* pattern matching can be used

```
let tup = (500, 6.4, 1);
let (x, y, z) = tup;
```

- indices can can also be used to access elements of tuples

```
let tup: (i32, f64, u8) = (500, 6.4, 1);
let five_hundred = tup.0;
let one = tup.2;
```

Array Type

- compound type that holds multiples of the same type of value
- arrays in Rust have a fixed length

```
let a = [1, 2, 3, 4, 5];
```

- data here will be allocated on the stack
- because of the fixed length they are useful for values that do not change in number, e.g. months in a year
- declaring length and type of an array works like this:

```
let a: [i32; 5] = [1, 2, 3, 4, 5];
```

- alternatively one can declare an array with e.g. 5 elements and all of them are 15

```
let a = [15; 5];
```

Accessing Array Elements

- access elements using indexes in square brackets

```
let a = [1, 2, 3, 4, 5];

let first = a[0];
```

Invalid Array Element Access

- if the index is out of bounds, a runtime error will occur
- the access is stopped to make the program safer and more stable

Functions

- pervasive in Rust code
- `fn main()` is the most important one, it's the entry point for many programs
- other functions are declared at any point in the file

```
fn another_function() {
    println!("Another function!");
}
```

- calling a function is simple too

```
fn main() {
    another_function();
}
```

Function Parameters

- the are part of the function definition

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

- defining multiple parameters works with commas

```
fn another_function(x: i32, message: String) {
    println!("The value of x is {}, {}", x, message);
}
```

Function Bodies, Statements, Expressions

- *Statements* are instructions that perform an action and don't return a value

```
let y = 6;
```

- *Expressions* evaluate to a resulting value
- assignments are not expressions in Rust, so this **won't** work

```
let y = (let x = 6);
```

- math operations, numbers, macros, functions, scopes are expressions

```
let y = {
    let x = 3;
    x + 1
}
```

- expressions **do not** end in semicolons

Functions with Return Values

- the type of return values is declared after `->` after the function signature
- the return value is the same as the last expression in a code block
- `return` can be used to return explicitly or early, most returns are implicit and on the last line

```
fn five() -> i32 {
    5
}

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

Comments

- simple comment

```
// hello world
```

- comments are generally above the line of code they are commenting on

```
// minimum age to buy alcohol
let drinking_age = 21;
```

Control Flow

- things that make programming easier by conditionally or repeatedly running code

`if` Expressions

- branches the code depending on certain boolean conditions, elements of the statement are sometimes called arms

```
let number = 3;

if number < 5 {
    println!("condition is true");
} else {
    println!("condition is false");
}
```

Multiple conditions with `else if`

```
let number = 6;

if number % 4 == 0 {
    println!("divisible by 4");
} else if number % 3 == 0 {
    println!("divisible by 3");
}
```

Using `if` in a `let` statement

- `if` is an expression, so it can be used in assignments

```
let condition = true;
let number = if condition {
    5
} else {
    6
};
```

- the types of all arms need to be the same

Repetition with Loops

- `loop`, `while`, `for` can execute blocks of code more than once

Repeating code with `loop`

- repeat something forever until explicit stop

```
loop {
    println!("again!");
}
```

- use `break` in a `loop` to break out of it normally

Returning values from Loops

- `loop` is an expression that can return values

```
let mut counter = 0;

let result = loop {
    counter += 1;

    if counter == 10 {
        break counter * 2;
    }
};
```

Conditional Loops with `while`

- loop with built-in test and break statements

```
let mut number = 3;

while number != 0 {
    println!("{}!", number);

    number -= 1;
}
```

- this eliminates a lot of nesting

Looping through a Collection with `for`

- `while` can loop through a collection of elements

```
let a = [10, 20, 30, 40, 50];
let mut index = 0;

while index < 5 {
    println!("the value is {}!", a[index]);

    index += 1;
}
```

- a more concise and safe way is to use a `for` loop, indices will always work

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

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

- to use a `for` loop a specified number of times, including the first and excluding the last, use

```
// (1..4) gives [1, 2, 3]
// rev() reverses the order of the numbers
for number in (1..4).rev() {
    // code
}
```

Understanding Ownership

- ownership is meant to make memory safe without having a garbage collector
- this chapter will cover ownership, borrowing, slices, data in memory layouts

What is Ownership

- *ownership* is central to the way Rust works and it's simple to explain
- all programs have to manage a computer's memory for running
- some use garbage collectors that constantly check for unused memory, some need the programmer to manually allocate memory
- rust uses a system that checks rules at compile time and thus does not slow down the program when it is running
- this chapter will cover strings as an example

The Stack and the Heap

- in many programming scenarios the stack and heap are not that important, but for systems programming and rust they are very important
- where data is stored influences the behavior of the language as well as its speed
- stack: memory that stores data in order and returns them in the opposite order, last int, first out
- data stored on the stack must have a known size at compile time, unknown or changing sizes must be stored on the heap
- heap: less organized, a certain amount of space is requested to store data, OS finds the space and returns a pointer (address of its location) to it
- pushing to the stack is faster than allocating on the heap because for the stack no location large enough has to be found and then kept in order
- accessing data on the heap is slower and jumping between data is also slower than working on one piece of data at a time
- when a function is called, the values passed to the function are all pushed onto the stack – to return the values they are popped off the stack
- ownership addresses what code is using data on the heap, cleaning up unused data on the heap etc

Ownership Rules

- each value in Rust has a variable that's called its *owner*
- there can only be one owner at a time
- when the owner goes out of scope, the value will be dropped

Variable Scope

- range in a program for which an item is valid
- when a variable comes *into scope* it is valid, when is goes *out of scope* it becomes invalid
- scopes are generally encapsulated by or related to curly brackets

```
{                      // s comes into scope
    let s = "hello";

                       // s is valid

}                      // s goes out of scope
```

The `String` Type

- simple data types are stored on the stack and popped off when they go out of scope
- more complex data types are stored on the heap and must be cleaned up after use
- `String` will be the example used here insofar as it relates to ownership
- string literals are not always convenient because they are immutable and hard coded
- `String` is allocated on the heap and can change at runtime, they can be created from string literals

```
let s = String::from("hello");
```

- the resulting type can be modified:

```
let mut s = String::from("hello");
s.push_str(", world!");         // appends to s
```

- the difference between `String` and string literals is the way they deal with memory

Memory and Allocation

- string literals are hardcoded into the program because they are known at compile time – they are fast efficient
- it is not possible to reserve blobs of memory at compile time for each string that might change
- `String` is growable, so: its memory must be requested from the OS at runtime; the memory must be returned to the OS when the `String` is done
- the programmer does the allocation manually

```
String::from("text")
```

- normally memory is either freed by a garbage collector or manually by the programmer, in Rust it is freed when the variable goes out of scope
- when `s` goes out of scope the `drop` function associated with it is automatically called by Rust to free the memory
- this seems simple now, but it can be more complicated in more complicated code

Ways Variables and Data Interact: Move

- if two primitive data types are set equal, the data is copied and then there are two variables with two copies of the same data, both are on the stack

```
let x = 5;
let y = x;
```

- for `String` this is different

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

- `s1` is made up of a `ptr`, `len`, and `capacity`, the pointer points to the first element of the string in memory, `len` is the amount of bytes of memory that the string is currently using and `capacity` is the total amount of memory allocated by the OS
- when `s1` is assigned to `s2`, the three pieces of data are copied, but the data on the stack remains the same, it is not copied and the two pointers point to the same place in memory
- in the example above Rust moves the data from `s1` to `s2` and invalidates `s1` so it is no longer valid
- invalidating `s1` will mean that when `s2` goes out of scope the memory is only freed once and thus does not generate a double free error
- additionally, Rust will never automatically make deep and expensive copies of anything – it will be fast by default

Ways Variables and Data Interact: Clone

- if we do want a deep copy of the data on the heap we use `clone`

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

- `clone` is something that is expensive to call

Stack-Only Data: Copy

- if a type has the `copy` trait, an older version of the variable is still valid after copying, like with integers

```
let x = 5;
let y = x;
```

- a type can't have the `copy` trait if any of its parts implement `drop`
- all simple or primitive types are copy

Ownership and Functions

- passing a variable to a function is similar to assigning values to variables, thus the same rules apply

```
fn main() {
    let s = String::from("hello");      // s comes into scope
    takes_ownership(s);                 // value of s moves into
```

```
                                        // function
                                        // it's no longer valid

    let x = 5;                          // x comes into scope
    makes_copy(x);                      // x is Copy and is thus
                                        // still valid
}   // x and then s go out of scope
    // nothing special happens to s because it is already invalid

fn takes_ownership(s: String) {         // s comes into scope
    println!("{}", s);
}   // s goes out of scope and drop is called, memory is freed

fn makes_copy(i: i32) {                 // i comes into scope
    println!("{}", i);
}   // i goes out of scope, not affecting x
```

- if `s` were to be used after the `takes_ownership(s)` was called, a compile time error would happen

Return Values and Scope

- returning values can also transfer ownership

```
fn main() {
    let s1 = give_ownership();          // fn moves its return
                                        // value to s1

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

    let s3 = takes_and_gives_back(s2);  // s2 moved into fn
                                        // return value moved to s3
}   // s3 goes out of scope and is dropped, so does s1.
    // s2 is already out of scope, so nothing happens

fn gives_ownership() -> String {        // will move return value
                                        // into calling fn
    let s = String::from("hello");      // s comes into scope
    s                                   // s is returned and moves
                                        // to the calling function
}   // nothing goes out of scope

fn takes_and_gives_back(s: String) -> String {
                                        // s comes into scope
    s                                   // s is returned and moves
                                        // to the calling fn
}   // nothing goes out of scope
```

- assigning the value of a variable to another moves it
- when an active variable goes out of scope, it is dropped

- one option for returning ownership of the argument plus a result is to return a tuple from a function – a better way to do it is to use *references*

References and Borrowing

- if one uses a function that takes ownership and then has to return ownership so the argument can be used afterwards
- passing references to functions instead of taking ownership is the solution to that

```
fn main() {
    let s1 = String::from("hello");
    let len = calculate_length(&s1);
    println!("The length of '{}' is {}.", s1, len);
}

fn calculate_length(s: &String) -> usize {
    s.len()
}
```

- ampersands '`&`' are *references* and enable referring to values without taking ownership
- above, `s` points to `s1` which points to the actual value
- dereferencing is done with *
- `&s1` refers to the value of `s1` but does now own it – the value will not be dropped when `s` goes out of scope
- when functions have references as parameters it is called *borrowing*
- references are immutable by default

Mutable References

- creating a mutable string and then passing a mutable reference to a function allows variables to be modified using their references

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

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

- big restriction: there can only be one mutable reference to a particular piece of data in a particular scope
- this only allows restricted mutation – less than most other languages
- Rust can thus prevent *data races* at compile time – these three things need to be true: two or more pointers access data at the same time, at least one of the pointers is used to write to the data, there are no mechanisms to synchronize the access to the data
- data races are undefined and difficult to diagnose

- new scopes allow for more mutable references, just not simultaneous ones
- we also cannot borrow data as mutable if it is also borrowed as immutable
- if an immutable reference is being used it is not expected that the value changes at the same time
- multiple immutable references are ok because nobody can change any of the data
- some intricacies are: if immutable references are no longer used a mutable one can be created even if the other ones are technically still in scope
- borrowing errors are annoying, but they prevent bugs at compile time

Dangling References

- these are created when a reference to a non-existent memory exists, producing undefined behavior

```
fn main() {
    let ref_to_nothing = dangle();
}

fn dangle() -> &String {               // returns ref to str
    let s = String::from("hello");     // s is new string

    &s                                 // return ref to str
}   // s goes out of scope here and is dropped
    // the reference now refers to nothing
```

- the simple solution here is to return the string with ownership instead

The Rules of References

- at any given time, you can have *either* one mutable reference *or* any number of immutable references
- references must always be valid

The Slice Type

- slices do not have ownership
- slices reference a contiguous sequence of elements in a collection rather than the whole collection
- imagine a function that returns the first word in a string, or the index of the end of the word

```
fn first_word(s: &String) -> usize {
    let bytes = s.as_bytes();       // convert to array of bytes

    // iterate over the bytes, iter returns each element
    // in a collection, enumerate returns an index and a
    // reference as a tuple
    for (i, &item) in bytes.iter().enumerate() {
        // compare the byte using the byte literal syntax
        if item == b' ' {
```

```
            return i;
        }
    }

    s.len()
}
```

- the problem with this is that the returned `usize` is not connected to the sting but meaningless without it
- having to worry about the index is stupid, even more so when two indices are to be returned – the solution? string slices

String Slices

- strings slices are references to parts of strings

```
let s = String::from("hello world");

let hello = &s[0..5];
let world = &s[6..11];
```

- similar to a string with extra indices for the beginning and the end
- they are constructed as

```
[starting_index..ending_index]      // ending_index is one more
                                    // than the last position
```

- the slices stores the starting position and the length of the slice
- range syntax in Rust allows the first 0 to be omitted `[0..2] == [..2]`
- the last byte of the string can also be omitted `[0..len] == [..]`
- the rewritten function from above is

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

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

    &s[..]
}
```

- with these slices it is impossible to get disconnected values that have nothing to do with each other
- a compiler error will occur because the slice is an immutable burrow and any modification would necessitate a mutable borrow, which is illegal

String Literals Are Slices

- string literals are slices that point to specific areas of the binary where the literal is stored
- their type is `&str`

String Slices as Parameters

- using `&str` as parameter means that both string slices and String can be used with it

```
let my_string = String::from("hello world!");

let word = first_word(&my_string[..]);

let my_literal = "literal";

let word = first_word(&my_literal[..]);

let word = first_word(my_literal);
```

Other Slices

- slices work on other data types too, like arrays – their type is `&[i32]`

```
let a = [1,2,3,4,5];

let slice = &a[1..3];
```

Using Structs to Structure Related Data

- a struct is a custom data type that packages multiple related values and makes them a meaningful group
- `struct` is like an objects data attributes
- how do structs and tuples differ, usages, function use
- structs and enums are the heart of Rust's way of creating new types

Defining and Instantiating Structs

- the data in a struct can be of different types
- each piece of data is named to make things clear
- data is entered in fields whose name and type are specified

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

- an instance of the struct needs to be created by giving values for the fields
- the instance is created with the fields in `key: value` pairs, or just values if the order is the same as in the definition

```
let user1 = User {
    email: String::from("someone@example.com"),
    username: String::from("somename"),
    active: true,
    sign_in_count: 1,
};
```

- to access and to change (if struct is mut) the dot notation is used

```
let name = user.name;

user1.email = String::from("changed@example.com");
```

- functions can return structs

```
fn build_user(email: String, username: String) -> User {
    User {
        email: email,
        username: username,
        active: true,
        sign_in_count: 1,
    }
}
```

- repeating `username` and `email` every time is tedious

Using the Field Init Shorthand when Variables and Fields Have the Same Name

- we can rewrite the function above to make it shorter

```
fn build_user(email: String, username: String) -> User {
    User {
        email,
        username,
        active: true,
        sign_in_count: 1,
    }
}
```

Creating Instances From Other Instances With Struct Update Syntax

- often one wants to copy parts of an existing struct and change some values
- the struct update syntax makes is short, `..` implies that the other fields should be takes from the specified instance

```
let user2 = User {
    email: String::from("newmail"),
    username: String::from("newuser"),
    ..user1
};
```

Using Tuple Structs without Name dFiels to Create Different Types

- tuple structs are structs that look similar to tuples
- they are for cases where the struct naming is useful but naming each part of the struct is superflous

```
struct Color(i32, i32, i32);
struct Point(i32, i32, i32);

let origin = Point(0, 0, 0);
```

- Color and Point are different types even though they store the same kind of data
- they can otherwise be treated like tuples

Unit-Like Structs Without Any Fields

- one can define dstruct that do not have any fields
- they are called unit-like structs because they are similar to the unit type
- they can be useful in situation where a type is supposed to have a trait but that type should not store any data – sounds like an attribute

Ownership of Struct Data

- a struct own all of its data if the data types are not owned by something else
- using string slices is not possible without the use of lifetimes

An Example Program Using Structs

- to understand the use of structs we'll write a program that finds the area of a rectangle
- starting with a program that only uses variables and then refactoring stuff until there are only structs left

```
fn main() {
    let width1 = 30;
    let height1 = 50;

    println!("The area of the rectangle is {} square pixels",
        area(width1, height1));
}

fn area(width: u32, height: u32) -> u32 {
    width * height
}
```

- while it works, the parameters of `area()` don't have an obvious connection
- the previously discussed tuple type could be useful here

```
fn main() {
    let rect = (30, 50);

    println!("The area of the rectangle is {} square pixels",
        area(rect));
}

fn area(dimensions: (u32, u32)) -> u32 {
    dimensions.0 * dimensions.1
}
```

- this version is both more and less clear
- the calls are shorter but the calculations are less clear; the meaning of the data is not clear

```
struct Rectangle {
    width: u32,
```

```
    height: u32,
}

fn main() {
    let rect = Rectangle {width: 30, height: 50};

    println!("The area of the rectangle is {} square pixels",
        area(&rect));
}

fn area(rectangle: &Rectangle) -> u32 {
    rectangle.height * rectangle.width
}
```

- this version of the code is much clearer and more understandable

Adding Useful Functionality with Derived Traits

- it would be nice to be able to print and instance of `Rectangle` and see the values of all of its fields
- standard printing does not work because it is not implemented
- the `{:?}` syntax is from Debug and it has to be opted in by putting the below code before the struct definition

```
#[derive(Debug)]
```

- we see `rect1 is Rectangle { width: 30, height: 50 }`, when we use `{:#?}` the output is

```
rect1 is Rectangle {
    width: 30,
    height: 50
}
```

- for more types and traits and behaviors we can derive see Appendix C
- it would be useful to be able to tie the `area` function to the `Rectangle` type so that it turns into a *method* of the type

Method Syntax

- methods are similar to functions, just that they are defined in the context of a struct, enum, or trait object and their first parameter is always self
- `self` represents the instance of the struct the method is being called on

Defining Methods

- changing the area function yields the following result

```
impl Rectangle {
    fn area(&self) -> u32 {
        self.width * self.height
```

```
    }
}
```

- `impl` means implementation and starts the context of Rectangle
- the first argument of a method is `&self` followed by all necessary arguments
- self in this case is immuatbly borrowed, but it can also be mutably borrowed or owned
- using the `impl` block makes assessing the features of Rectangle simple because all of the functionality is in one place
- Rust does not use `->` because it uses automatic referencing and dereferencing which is possible because the `self` signature makes it clear which one is needed so the code can be cleaner and Rust takes care of the rest

Methods with More Parameters

- new method that checks if one rectangle can fit completely into another one

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

Associated Functions

- associated functions are functions defined in `impl` that *don't* take self a parameter
- those functions are associated with the struct, hence the name, see `String::from`
- associated functions can be used as constructors that return a new instance of the struct

```
fn build_square(side_length: u32) -> Rectangle {
    Rectangle {width: side_length, height: side_length}
}
```

Multiple `impl` Blocks

- multiple blocks can be used even if there is no real reason to

Summary

- structs let you create custom types that are meaningful for your code

Enums and Pattern Matching

- enums are enumerations – types defined by their possible variants
- enums, the `Option` enum, `match` expression, `if let` construct

Defining an Enum

- used for a known number of different cases, e.g. IPv4 and IPv6
- the enumerated things are mutually exclusive

```
enum IpAddrKind {
    V4,
    V6,
}
```

- this is now a custom data type

Enum Values

- instances of an enum can be created like this

```
let four = IpAddrKind::V4;
```

- the variants are namespaced under the enum, functions can accept them

```
fn route(ip_kind: IpAddrKind) {
    ...
}
```

- now we can say which type something is but we don't have a way to store the associated data
- the naive approach would be an enum with `IpAddrKind` and `String`, but the better alternative is just an enum

```
enum IpAddr {
    V4(String),
    V6(String),
}

let home = IpAddr::V4(String::from("192.168.1.1"));
```

- enums can also handle tuple cases for the associated data type

```
enum IpAddr {
    V4(u8, u8, u8, u8),
    V6(String),
}
```

- enums can even hold struct or enum data themselves

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

- and like structs, methods can be associated with them

```
impl Message {
    fn call(&self) {
        // method body
    }
}

let m = Message::Write(String::from("hello"));
m.call();
```

The `Option` Enum and Its Advantages Over Null Values

- `Option` encodes the common scenario where something can be something or nothing
- with this type the compiler can check if all the cases are handled correctly, preventing bugs
- Rust does not have `Null` as a conscious feature because if you treat a null value like it is not-null an error will occur and this is super common
- the concept as a whole is pretty good though, it expresses a currently invalid value, Rust has the `Option<T>` enum

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

- this enum is so useful that it is automatically included
- the `<T>` means that `Some(T)` can hold any type, strings, numbers, etc
- if `None` is used we need to explicitly specify the type because it is not inferrable

```
let absent_num: Option<i32> = None;
```

- the thing that makes `Option<T>` better than `Null` is that the `Option<T>` and `T` are different values and `Option<T>` cannot be used like a valid value
- generally speaking `Option<T>` needs to be converted to `<T>` before it can be used
- accordingly, when a value is not `Option<T>` it can be safely assumed that it is not `Null`
- the `match` expression takes care of the handling of the valid and invalid cases

The `match` Control Flow Operator

- match can handle a ton of different cases and the compiler makes sure that all possible values are handled

```
enum Coin {
    Penny,
    Nickel,
    Dime,
    Quarter,
}

fn value_in_cents(coin: Coin) -> u8 {
    match coin {
        Coin::Penny => 1,
        Coin::Nickel => 5,
        Coin::Dime => 10,
        Coin::Quarter => 25,
    }
}
```

- how it works: `match` followed by any kind of expression and then braces
- then come the match arms: the have a pattern and come code, e.g. the pattern is `Coin::Penny` and the code is `1`, they are separated by `=>`, the arms are separated by commas
- the comparisons of the value with the arms occurs in order
- if it matches, the associated code is executed
- the code for each arm is an expression and the result is returned by the match expression – multiple commands per arm are possible

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

Patterns that Bind to Values

- match arms can bind to parts of the values that match the pattern – this is how data can be extracted from enums, e.g.

```
#[derive(Debug)] // so we can inspect the state in a minute
enum UsState {
    Alabama,
    Alaska,
```

```
    // --snip--
}

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

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

Matching with `Option<T>`

- this works the same as in the example above

```
fn plus_one(x: Option<i32>) -> Option<i32> {
    match x {
        None => None,
        Some(i) => Some(i + 1),
    }
}
```

Matches Are Exhaustive

- every possible case must be handled, otherwise the program won't compile

The _ Placeholder

- the _ placeholder can be used as a `default` that includes all cases that are not explicitly specified

```
let some_u8_value = 0u8;
match some_u8_value {
    1 => println!("one"),
    3 => println!("three"),
    5 => println!("five"),
    7 => println!("seven"),
    _ => (),
}
```

- `match` can be a bit wordy in case only one case is wanted

Concise Control Flow with `if let`

- if only one type of result of a match expression is of interest, it can be verbose

```
let some_u8_value = Some(0u8);
match some_u8_value {
    Some(3) => println!("three"),
    _ => (),
}
```

- this can be written in a shorter form

```
if let Some(3) = some_u8_value {
    println!("three");
}
```

- it works by having `if let <pattern> = <expression> { <code> }`
- `if let` has the disadvantage that it does not have exhaustive checking
- the `if let` statement also accepts an else statement, making these two pieces of code equivalent

```
let mut count = 0;
match coin {
    Coin::Quarter(state) =>
        println!("State quarter from {:?}!", state),
    _ => count += 1,
}
// is the same as
let mut count = 0;
if let Coin::Quarter(state) = coin {
    println!("State quarter from {:?}!", state);
} else {
    count += 1;
}
```

Projects with Packages, Crates, and Modules

- with large projects organization becomes super important to stay on top of things
- until now we've written programs in one module in one file
- later it can be split into multiple modules and multiple files
- a package can contain multiple binary crates and optionally one library crate
- large packages can have parts extracted into separate crates that become external dependencies
- public interfaces to code make it simpler to use other code and keep the implementation hidden
- scope management is another important aspect, one needs to manage how to keep things organized and working
- part of Rust's module system are:
  - Packages: Cargo feature to build, test, share crates
  - Crates: three of modules producing a library or executable
  - Modules and use: let you control the organization, scope, and privacy of paths
  - Paths: ways of naming items (structs, functions, modules)

Packages and Crates

- crate: binary or library
- crate root: source file rustc starts from and that makes up the root module of the crate
- package: one or more crates that provide functionality, contains a Cargo.toml file describing how to build them
- a package must contain one or zero library crates, it must contain at least one binary crate or more
- `cargo new` creates a package, `src/main.rs` is the crate root or a binary crate
- if there is a file called `src/lib.rs` it's the crate root of a library crate
- if a package has both, it also works
- if there are more binary files, they reside in `src/bin` – each will be a binary crate
- a crate groups related functionality together so its easy to access, `rand` from the first program is an example of that
- keeping scopes clear is important so conflicts are avoided – namespacing

Defining Modules to Control Scope and Privacy

- modules allow the organization of code into groups for readability and reuse
- they also control the privacy of items – whether it is available to the outside (*public*) or a hidden implementation detail (*private*)
- the restaurant example will have empty function so the focus is on the organization of the code
- nested modules can be used to express real life organizations

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

        fn seat_at_table() {}
    }

    mod serving {
        fn take_order() {}

        fn serve_order() {}

        fn take_payment() {}
    }
}
```

- modules are defined using the `mod` keyword and specify its name, then curly brackets
- modules can hold other things such as structs, enums, constants, traits
- they group together related definitions – making it simpler to use
- the `src/lib.rs` file is the crate root because it is at the root

```
crate
└── front_of_house
    ├── hosting
    |   |
    |   ├── add_to_waitlist
    |   |
    |   └── seat_at_table
    └── serving
        ├── take_order
        ├── serve_order
        └── take_payment
```

- modules nest in each other, some of them are *siblings* to each other, modules are *children* of others or *parents* of others

Paths for Referring to an Item in the Module Tree

- paths are used to tell Rust where to find items – just like a filesystem
- to call a function, its path needs to be known

  - *absolute path*: starts from the crate root with the crate name or a literal crate
  - *relative path*: starts from the current module and uses `self`, `super`, or an identifier in the current module
- paths are followed by one or more identifiers separated by `::`
- now we add a public function to our library

```
mod front_of_house {
    mod hosting {
        fn add_to_waitlist() {}
    }
}
pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
    front_of_house::hosting::add_to_waitlist();
}
```

- this code won't work because it is using the `hosting` module in a function but the module is not public – it is hidden from the function
- the absolute path starts at crate because it is the root
- the relative path takes advantage of the fact that `front_of_house` is on the same level as itself
- which naming convention to use is up the design plans but absolute paths can be safer
- to make things private, they are put into modules – all items are private by default
- parent module items can't use child modules' items but child modules can use parent modules' items – basically normal inheritance
- using `pub` on inner implementations makes them visible to parent modules

Exposing Paths with the `pub` Keyword

- to fix the previous listing we need to make `hosting` public to expose it to its parent modules
- even this does not do the trick because `add_to_waitlist` is still private and thus not accessible
- making its module private did not expose the function in it
- after making the function public too the code compiles

```
mod front_of_house {
    pub mod hosting {
        pub fn add_to_waitlist() {}
    }
}
pub fn eat_at_restaurant() {
    // Absolute path
    crate::front_of_house::hosting::add_to_waitlist();

    // Relative path
```

```
        front_of_house::hosting::add_to_waitlist();
    }
```

- siblings have automatic visibility of each other – anything below them in the tree is private, anything above is visible

Starting Relative Paths with `super`

- `super` at the start of a path means that it begins at the parent module
- this allows the access of an item that is one level above the current one

```
fn serve_order() {}

mod back_of_house {
    fn fix_incorrect_order() {
        cook_order();
        super::serve_order();
    }

    fn cook_order() {}
}
```

Making Structs and Enums Public

- `pub` has the same effect on enums and structs that is has on functions
- using pub before a struct definition, the struct will be public but its fields will not be – each of the fields can be either public or private

```
mod back_of_house {
    pub struct Breakfast {
        pub toast: String,
        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() {
    // Order a breakfast in the summer with Rye toast
    let mut meal = back_of_house::Breakfast::summer("Rye");
    // Change our mind about what bread we'd like
    meal.toast = String::from("Wheat");
    println!("I'd like {} toast please", meal.toast);
```

```
    // The next line won't compile if we uncomment it; we're not allowed
    // to see or modify the seasonal fruit that comes with the meal
    // meal.seasonal_fruit = String::from("blueberries");
}
```

- if a struct has a private field, it needs a public function to create it because otherwise the private field could not be set
- if an `enum` is made public, all of its elements are made public

```
mod back_of_house {
    pub enum Appetizer {
        Soup,
        Salad,
    }
}

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

Bringing Paths into Scope with the `use` Keyword

- all the paths we have used thus far have been long and cumbersome, they can be shortened with the help of the `use` keyword
- bringing a path into scope will substantially shorten the function calls

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

use crate::front_of_house::hosting;

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

- items can also be brought into scope with a relative path

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

use front_of_house::hosting;
```

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

Creating Idiomatic `use` Paths

- the idiomatic way to bring functions into scope is to bring their parent modules into scope and then call them with their parent modules attached
- this makes it clear that the function is not defined locally
- when using enums or structs on the other hand, the full path is specified and only their name is used
- the only hard line is bringing two items with the same name into scope, in that case the parent module must be specified

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

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

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

- it is also not allowed to fully import two modules with the same names for the same reason

Providing New Names with the `as` Keyword

- another solution to the problem above is to give the item a new local name

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

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

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

- both solutions are idiomatic so it depends on the programmer

Re-exporting Names with `pub use`

- when a name is brought into scope with `use` it is private

- sometimes it is useful to import code into a program and at the same time make it available to others calling our code – this is called re-exporting

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

pub use crate::front_of_house::hosting;

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

- the above code allows external code to also call the `hosting` module and the `add_to_waitlist` function
- this allows the library to behave differently and in an optimized way – depending on what is necessary

Using External Packages

- in Chapter 2 of the rust book we used the `rand` crate

```
[dependencies]
rand = "0.5.5"
```

- then we used it in our program

```
use rand::Rng();

fn main () {
    let secret_number = rand::thread_rng().gen_range(1, 101);
}
```

- https://crates.io hosts many useful crates and using them is always the same
  1. list them in the packages *Cargo.toml* file
  2. use `use` to bring them into scope
- the standard library `std` is also external to our crate but it is shipped with Rust
- standard library functions still need to be brought into scope

Using Nested Paths to Clean Up Large `use` Lists

- if multiple items defined in the same package or module are used, the `use` statements can become quite long
- instead, we specify the common parts of the path and then list the individual items in braces

```
use std::{cmp::Ordering, io};
```

- the longer the shared path, the better the nested paths look

The Glob Operator

- if all public items in a path are supposed to be brought into scope the glob operator can be used

```
use std::collections::*;
```

- this can be an issue because you don't know where certain names come from or which are even in your program

Separating Modules into Different Files

- when projects become too large, one might want to split them into multiple files
- to do this, create a file named `file.rs` in the `src` directory
- to be able to use the files contents, use

```
mod file;
```

in the file

- this makes `file.rs` available
- the names of the modules and the files must match
- one step further is to create a directory called `./file/` and in `file.rs` write

```
mod sub_file;
```

- now we can create a file `./file/sub_file.rs` that will be loaded as a child of the `file` module

Common Collections

- the standard library includes many useful data structures (*collections*)
- collections can contain multiple values while most data structures only represent on specific value
- collections are stored on the heap and their size does not need to be know at compile time, they can also dynamically grow
- this chapter will look at
  - *vector*: store a variable number of values in a kind of array
  - *string*: collection of characters
  - *hash map*: allows the association of a value with a particular key; particular implementation of a *map*

Storing Lists of Values with Vectors

- first collection type looked at is `Vec<T>` – a vector
- stores more than one value in one data structure next to each other in memory
- can only store values of the same type
- they are useful for a list of items

Creating a New Vector

- a new vector is created with the new function

```
let v: Vec<i32> = Vec::new();
```

- the type annotation is needed because no data is being inserted into the vector (can't be inferred) but its type needs to known
- to create a vector with values and type inference, the `vec!` macro is provided

```
let v = vec![1, 2, 3];
```

Updating a Vector

- to add elements to a vector, use the `push` method

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

v.push(2);
v.push(2222);
```

Dropping a Vector Drops Its Elements

- like any other struct, a vector is freed when it goes out of scope

```
{
    let v = vec![1, 2, 3];

    // use v
}   // v goes out of scope and is freed
```

- this can become complicated when references to elements of the vector are introduced

Reading Elements of Vectors

- there are two ways to read the values stored in a vector: indexing or the `get` method

```
let v = vec![1, 2, 3];

let third: &i32 = &v[2];    // get a reference to the third element

match v.get(2) {
    Some(third) => println!("The third element is {}", third),
    None => println!("There is no third element"),
}
```

- vectors are indexed starting from `0`
- using `&v` and `[]` gives a reference, using `get` gives `Option<&T>`
- there are two ways to do this because you should have the choice of how the program reacts when an index is out of bounds
- the reference and brackets method will crash if the index is too large – use this if an access out of bounds should never happen
- the get method will not panic and only return none – use this if occasionally an element out of bounds might be accessed
- if the obtained reference is valid the borrow checker will enforce the guidelines from chapter 4 of the rust book
- even if an immutable reference exists to a part of the vector that should remain unaffected by a modifying action it will not work

Iterating over the Values in a Vector

- for a loop of immutable references

```
let v = vec![100, 32, 57];
for i in &v {
    println!("{}", i);
}
```

- there can also be a loop over all elements in a mutable fashion

```
let mut v = vec![100, 32, 57];
for i in &mut v {
```

```
        *i += 50;
    }
```

- here the de-reference operator * needs to be used

Using an Enum to Store Multiple Types

- because vectors can only store one type of data, there is a trick using enums that allows multiple values to be stored
- thus the vectors takes one type of enum but the enum variants have different value types

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

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

- Rust needs to know what types will be in the vector at compile time because it needs to know how much memory is require to hold it
- because vectors are explicit about what kind of values a vector can hold errors caused by different types being incompatible with operations are eliminated
- another useful method is the `pop` method that returns the last item in the vector