readme.md

RESOURCES :

There are some of the resources I found VERRRYYYYY useful when learning rust the incredible language!

- Module Systems

Ultimate Rust cheat sheet :

Tuples

- let new_tup = ("name_str", 32);
- let (name,age) = new_tup //destructuring
- new_tup.1 //Dot notation for indexing, indexing start from 0
- Only has debug print trait implemented

Structs

- Basic syntax for defining a struct :

```
struct Message{
    from: String,
    to: String,
    content: Vec<i32>, //Vector for bytes
    typ: MessageType
}
```

- A struct can be built and is reffered to as creating and "instance", NOTE : To construct a struct you must provide all fields.

```
let msg1 = Message{
        from: String::from("Navin"),
        to: String::from("User2"),
        content: vec![72,101,108,108,111,44,32,87,111,114,108,100,33],
        typ: MessageType::Message,
};
```

- fields of a stuct are read using dot notation
- Entire instance must be mutable, individual fields can not be mutable by themselves.
- [Struct Update Syntax] Another instance can be created from existing ones, all new field value must be explicitly mentioned before the ..old_struct, like so : .

```
let _msg3 = Message{
        typ: MessageType::File,
        ..msg1 //NOTE : All values of msg1 that do not have copy trait are mov
    };
    //println!("Messag was sent from {:?} to {:?} with content : {:?}",msg1.fr
```

Be very vary of the move happening in the "Struct Update Syntax", as the name suggests, this must be used for updating a struct not creating new ones.

- Rust also has tuples struct, where the field names are not given:

```
struct Color (i32,i32,i32);
struct Coord_3d (i32,i32,i32); //Can destructure and index using . notation
```

The choice if String instead for &str was a deliberate choice above, this is cus Rust expects a struct to own all it's values. If a struct should store a reference then we need lifetime specifiers which we will see later.

- Structs by themselves do not have debug print trait, and hence needs to be derived using #[devive(Debug)]

> NOTE : Every field of the struct must have the same set of derived traits.

- Struct has have methods closely tied to themselves in Rust, These are called implmentations.

Enums and Options : The alternative to the billion dollar mistake. How Enums tie it all together.

- Basic enum definition syntax :

```
#[derive(Debug)]
struct Day{
    day_in_words: String,
    date: i32,
    month: i32,
    year: i32
}
#[derive(Debug)]
enum DayType{
    Weekday(Day),
    Weekend(Day)
}

impl DayType{
    fn print_pretty(self: &Self){
        println!("{:#?}",self);
    }
}
```

- Options :

```
let x: Option<i32> = Some(31);
let _y = x+34; //Gives error, cant add Option and i32
```

That is hwo rust makes sure that a null reference can never happen, Option is more like indicating that a variable can be of typer None, meaning before you can do any operation on it some check have to be done. If the variable is not of type option is can never be None.

> Note : None is analougous to NULL is other subject, but ofc better.

- This is where we get `match` to help us, NOTE : match has to be exhaustive at all times or have _ arm. Using match with Options :

```
x = match x{
    None => {
        println!("Not initiliased");
        None
    },
    Some(x)=>{
        Some(x+31)
```

```
    }
}
```

- Using match with above enums :

```
impl DayType{
    fn print_pretty(self: &Self){
        println!("{:#?}",self);
    }
    fn is_tueday(self : &Self){
        match self{
            DayType::Weekday(day)=>{
                if day.day_in_words == "tuesday"
                {
                    println!("It is tueday!");
                }else{
                    println!("It is not tuesday :(");
                }
            },
            _=>println!("It is not tuesday :(")
        }
    }
}
```

> NOTE : the Enums-03 module is very very well written! refer it.

Modules

- Rust only sees the "crate root", that is the main.rs files in binary crate and lib.rs in library crate. Any file that is neeeded to be used must be explicitly added in the module tree. This can be done using the mod keyword, but as said before rust only sees main.rs, hence we use the mod keyword in main.rs file.
- Everything is private in rust, we can make it public.
- If ther exists modules in a folder, the folder must have a mod.rs which define all the modules in the folder and also pub or not.
- Once added in the Module tree, the given module can be accessed from other files using the correct path.

Vectors : Allocated in the heap

- let mut vec = Vec::new();
- vec.push(1) //Only a single value

- let mut vec2 = vec!['H','E','L','L','O'];
- vec.len()
- vec[2] //indexble
- vec.get(2) //same as indexing
- let vec_slice = &vec[2..5]; //Slices are ALWAYS refrences, the last index is not included
- vectors are indexable, but instead of using [] and letting the code panic when the element does not exists, we can use .get method that return an Option type (Some() or None). Code :

```
//----------Vectors----------
let mut v1 : Vec<i32> = Vec::new();
let mut v2 = vec![2,4,7];
v1.push(3);
v2.push(3);
v2.pop();
println!("First element : {}",v1[0]);
println!("First element : {}",v1.get(0).unwrap()); //Safer way of fetching ele
```

This can lead to cleaner code as such :

```
let mut input : String = String::new();
std::io::stdout().write(b"Enter an index to fetch from vector 1 : ").unwrap();
std::io::stdout().flush().unwrap();
std::io::stdin().read_line(&mut input).unwrap();
let input_i32 : usize = match input.trim().parse(){
    Ok(n) => n,
    _=>panic!("Invalid input for index")
};
match v1.get(input_i32){
    Some(n) => println!("Element found : {}",n),
    None => println!("Element not found at that index")
};
```

Incredible piece of code, no?

- You can not have a push operation where there is a refrence (mut or imut) before hand. To make this easier to remember just assume that .push() method takes a mutable reference of vectors to write to it. Simply Rust will not let you modify anything as long as a refernce exists.

```
//Works :
let v1_ref = &v1[0];
v1.push(2);

//Wont work :
let v1_ref2 = &v1[1];
v1.push(4);
//println!("Printing a refrence that appeared before push : {}",v1_ref2); //Wi
```

- We can iterate over vectors with mut or imut refrences, like so :

```
for i in &v1{
    println!("Item from v1 : {}",i);
}
for i in &mut v1{
    *i=*i+50;
}
for i in &v1{
    println!("Item from v1 : {}",i);
}
```

- Vectors are sadly homogeneous in Rust (This also is a degin choice to security), but we can still do something like this :

```
#[derive(Debug)]
enum UniversalType{
    Text(String),
    Number(i32),
    Charecter(char),
    Decimal(f64)
}

...

let v3 = vec![UniversalType::Text(String::from("hello")),UniversalType::Number
println!("from v3 : {:?}",v3.get(1).unwrap());
```

String

- Can convert from &str to String using .to_string(), ownership is no transffered. Contat using + does transffer ownership depending on refrence of not. Code like so :

```
//----------String-----------
let str1 : &str = "Hello, World!";
let mut string1 : String = str1.to_string();
string1.push_str(&str1); //Take refrence of str1 thus ownership of str1 is mai
println!("{}",str1);
let string2 : String = String::from("Hello world, twice!");
let string3 = string2 + &string1; //ownership of string2 is transffered. strin
println!("{}",string3);
```

that thing with + sign is confusing, no? Well in the stad def the code is such that u can add a String with &str, but &string1 is a &String, how does it work then? Rust does something called defer coerce, it converts the &String to &str.

> Notes : You can not add two String (String + String) in rust. Its this property that makes the usage of + for concat of String very weird, hence the format! macro.

- format macro:

```
let temp_str1 = String::from("5");
let temo_str2 = String::from("7");
let string4 : String = format!("{}\"{}\' tall",&temp_str1,&temo_str2);
println!("Your height : {}",string4);
```

- Idexing string using [] is not allowed. Simply put its because Rust doesn't want us fools accessing a single byte when in some cases utf-8 characters are encoded with 2 bytes. Rust book explains is much better here.
- If you still want to index them, you can make references index or slices, both of which will return raw bytes and have no assurity.

HashMaps

- HashMaps by default store a <String,i32> key-value pair, to define anything different :

```
#[derive(Debug)]
struct Data{
    player_id:i32,
    age:i32,
    team:String,
    averagescore:f64
```

```
}

let mut data : HashMap<String,Data> = HashMap::new();
data.insert(String::from("Player1"),Data { player_id: 237, age: 23, team: Stri
```

- there does not exist a macro to build hashmaps, we must depend on .insert() method. We could also use .collect() method that works on iter, like so:

```
let players : Vec<String> = vec![String::from("Player2"),String::from("Player3
let player_data : Vec<Data> = vec![Data { player_id: 238, age: 32, team: Strin
let data_map2 : HashMap<_,_> = players.into_iter().zip(player_data.into_iter()
println!("{:#?}",data_map2);
```

the HashMap<,> is done such that the compier can fill in those values at compile time looking at the right hand side.

- Ownership in hashmaps : is the variables type implements copy trait then it is copies, else moved. Using references will not move the value, but we do need valid lifetime specifiers, which we will see shortly.
- We can access HaspMap values using the two ways just like in vectors :

```
println!("{:#?}",data_map2["Player3"]);
println!("{:#?}",data_map2.get("Player2").unwrap());
```

Using .get we can also do that very nice error handling as the example shown in Vectors section.

- We can simply overwrite key value pairs. If we want to only insert if key doesn't exist, we can use .entry along with .or_insert(). .entry() return a Enums of type Entry that can differentiate between present and absetn keys, .or_insert() is am impl on type Entry that inserts if key doesn't exists, like so :

```
data_map2.entry(String::from("Player2")).or_insert(Data { player_id: 400, age:
data_map2.entry(String::from("Player1")).or_insert(Data { player_id: 237, age:
println!("{:#?}",data_map2);
```

> Notes : .entry().or_insert() return back a reference to the value present to the given key.

- Updating values present in HashMaps using refrences (This feels very much like C):

```
let mut hash3 : HashMap<String,i32> = HashMap::new();
let text = "she sells sea shells at the sea shore";
for i in text.split_whitespace(){
    let count = hash3.entry(i.to_string()).or_insert(0);
    *count+=1;
}
println!("{:#?}",hash3);
```

Generics : Function overloading alternatives. kind of

- HGnerics help with function overloading but also restricting methods bassaedon types and traits.
- Generics on functions (We'll get back to this later) : You can not compare any type T unless the type ad the std::cmp::PartialOrd trait is implmented, meaning to compare generic we need that trait.
- Genrics on struct :
    - For a given struct or fn the T should be the same type everywhere, internally the compiler converts T to its conrete type multiple types if needed, th can be see clealy wwth the given example belo. This means if you possibly wat two different types of generices u need two generics:

```
struct Pairs<T>{
    x: T,
    y: T,
}

struct PairsHetro<T,U>{
    x:T,
    y:U,
}

fn main() {
    println!("Hello, world!");
let p1 = Pairs{ x:12,y:23 }; //Works : Implicit definition of T is done by com
```

```
    //let p2 : Pairs<T> = { x:23,y:34  }; //Wont work cus main function doesnt
    let p2 : Pairs<i32> = Pairs{ x:23,y:34 }; //Works explicit definition of T
    let p3 = Pairs{ x:2.3,y:23 }; //Wont work;
    let p3 = PairsHetro{x:2.3,y:23}; //Works!

}
```

> Notes : Rust accomplishes this by performing monomorphization of the code using generics at compile time. Monomorphization is the process of turning generic code into specific code by filling in the concrete types that are used when compiled. This way we have 0...absolute 0 speed diff in runtime.

- Generics on enums :
  - Here we can see exaclty how Option enum was defined in std :

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

  - And also Result enum from std :

```
enum Result<T,E>{
    Ok(T),
    Err(E)
}
```

- Generics on imple, we can ue generic types in impl in very nice ways. You can restrict some method to when geneeic of some particula type. You can also make some methods available to eveything no matter the the generuc turned out be, like so:

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

use std::fmt::Debug;
impl<T:Debug,U:Debug> PairsHetro<T,U>{ //These methods are available to all T
    fn pretty_print(self:&Self){
    println!(" ({:#?},{:#?}) ",self.x,self.y);
    }
}

impl PairsHetro<i32,f32>{
    fn add2(self:&mut Self){
```

```
        self.x+=2;
        self.y+=2.0;
    }
}

//In all trait check happen on generics going into imp and type checks happen
//the object (struct or enu or whatev)

fn main() {

    let p3 = PairsHetro{x:2.3,y:23}; //Works!
    let mut p4 = PairsHetro{x:32,y:3.2};
    //p3.add2(); //Does not work
    p4.add2(); //works
    p3.pretty_print(); //Work
    p4.pretty_print(); //Works
}
```

> Notes : trait check go into impl generics, type checks go into object generics (strucor enum), see in above example.

- Its not compulsory that them impl methods have to use the same genrics as their objects, they can introduce new generics thenselves, the examples for the book makes this clear, like sso :

```
struct Point<X1, Y1> {
    x: X1,
    y: Y1,
}

impl<X1, Y1> Point<X1, Y1> {
    fn mixup<X2, Y2>(self, other: Point<X2, Y2>) -> Point<X1, Y2> {
        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);
}
```