

# The Rust Cookbook

Author: Jaideep Ganguly, Sc.D. (MIT)

✉ [ganguly.jaideep@gmail.com](mailto:ganguly.jaideep@gmail.com) | Github  | LinkedIn 

Last updated: 2024-09-29 08:06:49+05:30



# Contents

<b>Contents</b>	<b>2</b>
1 The Rust Cookbook: Your Essential Guide . . . . .	6
2 Installation . . . . .	7
3 Toml: Cargo Rust's Build System & Package Manager . . . . .	7
3.1 Terminology . . . . .	7
3.2 Create New Package . . . . .	7
3.2.1 Build & Release . . . . .	7
3.3 Dependencies . . . . .	7
3.4 Cargo.toml - an example . . . . .	8
3.4.1 Cargo update . . . . .	8
3.4.2 Cargo upgrade . . . . .	8
3.5 Publish in Crate . . . . .	8
4 Module . . . . .	9
5 Data Types & Variables . . . . .	10
5.1 Data Types . . . . .	10
5.2 Variables & Mutability . . . . .	10
5.3 Ownership . . . . .	10
5.4 Borrow . . . . .	11
5.5 Lifetime . . . . .	11
6 Collections . . . . .	12
6.1 Array & Slice . . . . .	12
6.2 Closure . . . . .	12
6.3 Vector . . . . .	12
7 String . . . . .	14
8 if else & iteration . . . . .	14
9 Data Structures & Traits . . . . .	15
9.1 Tuple . . . . .	15
9.2 struct . . . . .	16
9.3 Trait . . . . .	16
9.4 trait bound . . . . .	17
9.5 to_owned() and clone() . . . . .	17
9.6 vec of struct . . . . .	18
10 Enum . . . . .	19
10.1 Match . . . . .	19
10.2 Result { OK, Err } . . . . .	19
10.3 ResultErrStr { OK, Err } . . . . .	20
10.4 Option { Some, None } . . . . .	20
10.5 if let . . . . .	21
10.6 Unwrap . . . . .	21
11 HashMap . . . . .	22
12 Generic . . . . .	22
13 Smart Pointers, Deref & Drop . . . . .	23
13.1 Smart Pointers . . . . .	23
13.2 Deref . . . . .	23
13.3 Drop . . . . .	24
13.4 dyn . . . . .	24

	13.5	tokio . . . . .	25
14	Async . . . . .		26
	14.1	Long Running Functions . . . . .	26
	14.2	Arc, Mutex & Lock . . . . .	27
		14.2.1 Arc,Mutex,Lock with HashMap . . . . .	27
		14.2.2 Arc,Mutex,Lock with Struct . . . . .	29
	14.3	Message Passing . . . . .	32
15	PostgreSQL . . . . .		33
16	IO . . . . .		34
17	gRPC . . . . .		36
	17.1	Server . . . . .	36
	17.2	Client . . . . .	38
18	Dockerfile . . . . .		40
	18.1	Install . . . . .	40
	18.2	. . . . .	40
	18.3	Dockerfile . . . . .	40
	18.4	Docker build & run . . . . .	40
	18.5	Docker Push . . . . .	40
	18.6	Docker Commands . . . . .	40
19	Kubernetes . . . . .		42
	19.1	. . . . .	42
	19.2	Start & Stop Kubernetes Cluster . . . . .	42
	19.3	Kubernetes yaml file . . . . .	42
	19.4	Kubectl commands . . . . .	42
	19.5	Plugin Manager for kubectl . . . . .	43
20	Kubernetes Definitions . . . . .		43

# Listings

1	src/mod_example_dir/mod.rs	9
2	src/mod_example_dir/file1.rs	9
3	src/mod_example_dir/sub_dir/mod.rs	9
4	src/mod_example_dir/sub_dir/file2.rs	9
5	src/main.rs	9
6	println!	9
7	Mutability	10
8	Ownership	10
9	Borrow	11
10	Borrow Mutable Reference	11
11	Lifetime	11
12	Array & Slice	12
13	Closure	12
14	Vector	12
15	String	14
16	if else	14
17	iter	15
18	Tuple	15
19	Struct	16
20	Trait	16
21	Trait bound	17
22	to_owned() & clone()	17
23	vec of struct	18
24	Enum & Match	19
25	Result	19
26	ResultErrStr	20
27	Option	20
28	if let	21
29	unwrap	21
30	HashMap	22
31	Generic	22
32	Box	23
33	Deref	23
34	Drop	24
35	dyn	24
36	tokio	25
37	Long Running functions	26
38	Mutex on HashMap	27
39	Mutex on Struct	29
40	Message Passing	32
41	PostgreSQL	33
42	dyn	34
43	Server Code	36
44	Client Code	38
45	Install Docker	40
46	Visual Code Configuration	40
47	Dockerfile	40

48	Dockerfile Build & Run . . . . .	40
49	Docker Push . . . . .	40
50	Docker Commands . . . . .	41
51	Install Kubernetes . . . . .	42
52	Start & Stop Kubernetes Cluster . . . . .	42
53	Kubernetes yaml file . . . . .	42
54	Kubectl commands . . . . .	42
55	Plugin Manager for kubectl . . . . .	43
56	Yaml . . . . .	43
57	Namespace . . . . .	43

## 1 The Rust Cookbook: Your Essential Guide

Mastering Rust offers significant advantages for developers seeking to excel in modern programming. Rust's focus on **memory safety**, **concurrency**, and **performance** makes it a powerful tool for building **reliable** and **high-performance software**. By mastering Rust, you gain the ability to write code that is not only efficient but also free from common bugs such as null pointer dereferences and data races. Rust's strong type system and strict compiler checks encourage best practices and help ensure that code is **robust** and **maintainable**.

As Rust continues to gain traction in various industries, mastering it opens up opportunities to work on cutting-edge projects and contribute to innovative solutions. Embracing Rust's unique features and paradigms equips you with skills that are increasingly valuable in today's technology landscape, positioning you as a proficient developer in a growing and dynamic field.

This **Rust Cookbook** is intended for developers who are proficient in a modern programming language and have a solid understanding of fundamental computer science concepts. It provides a deliberately concise and focused exploration of Rust, concentrating on the most useful and frequently utilized features of the language. The goal is to streamline your learning process and offer practical solutions for common challenges. Following are the objectives of this book:

- 1 **Bridging the Gap Between Theory and Practice:** Rust's concepts can be abstract. This cookbook offers concrete examples that demonstrate how to apply theoretical knowledge to real-world problems.
- 2 **Provide Idiomatic Rust Solutions:** It showcases idiomatic Rust patterns and best practices, helping developers write efficient and maintainable code.
- 3 **Address Common Challenges:** Address common pitfalls and challenges that Rust developers often encounter, saving time and frustration.
- 4 **Quick Reference** When you need to quickly look up how to perform a specific task, a cookbook can be a valuable resource.
- 5 **Encourage Experimentation:** By providing examples, a cookbook inspire developers to experiment with different approaches and learn new techniques.

Rust development was led by **Graydon Hoare** at **Mozilla Research** in 2010. With the 1.0 version officially released on May 15, **2015**, Rust has grown with contributions from a large and active open-source community, and it is now maintained by the Rust Foundation and the Rust community.

While this cookbook serves as a valuable resource, it is designed to complement the official Rust documentation. For a more comprehensive understanding and in-depth coverage of Rust's features, refer to the official Rust manual. Together, this cookbook and the official documentation will equip you with the knowledge and tools to master Rust effectively.

In summary, this Rust Cookbook serves as an essential guidebook for developers, designed to navigate the Rust ecosystem effectively. It empowers developers to write code that is not only efficient and safe but also leverages Rust's elegant features. By providing practical examples, best practices, and comprehensive solutions, this cookbook aims to streamline the development process and enhance coding proficiency in Rust.

## 2 Installation

```
1 curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
2 rustup update # Update
3 rustup self uninstall # Uninstall
4 rustc --version # Version
```

## 3 Toml: Cargo Rust's Build System & Package Manager

### 3.1 Terminology

- ① **Package**: A collection of one or more crates that are built, tested, and published together. It is a fundamental unit of distribution in Rust, managed by Cargo, Rust's build system and package manager.
- ② **Crate** - Smallest unit of code distribution and compilation. A crate can be either a library or a binary, and it is the building block for Rust packages.
- ③ **Module** - A way to organize and encapsulate related items (such as functions, structs, enums, constants, and traits) within a crate.
- ④ **use** - Bring items from modules into scope, making them accessible without needing to use their full paths.

### 3.2 Create New Package

```
1 cargo new tpl # creates a new directory named tpl
2 cargo new tpllib --lib # creates a library, write code in src/lib.rs
```

It generates a file **Cargo.toml** in the directory **tpl** with the following contents.

```
1 [package]
2 name = "tpl"
3 version = "0.1.0"
4 edition = "2021"
5
6 [dependencies]
```

#### 3.2.1 Build & Release

```
1 cd tpl
2 cargo build # debug
3 cargo build --release
4 ./tpl
```

Build produces a binary **tpl**, if it is a library, build produces **tpllib.rlib**. It can be used as:

```
1 [dependencies]
2 tpllib = { path = "../tpllib" } # .rlib should NOT be specified
```

### 3.3 Dependencies

```
1 [dependencies]
2 serde = "*" # any version
3 serde = "1.2.3" # exact version
4 serde = "^1.0" # greater than or equal to 1.0.0 and less than 2.0.0
```

### 3.4 Cargo.toml - an example

```
2  [package]
3  name = "tpl"      # template
4  version = "0.1.0"
5  edition = "2021"
6
7  # See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html
8
9  [dependencies]
10 lazy_static = "1" ✓
11 futures = "0.3.30" ✓
12 serde_json = "1.0.128" ✓
13 serde = {version = "1.0", features = ["derive"]} ✓
14 tokio = {version = "1.40", features = ["full"]} ✓
15 mysql = "25.0.1" ✓
16 postgres = "0.19" ✓
17 chrono = "0.4" ✓
18 psutil = "3.3.0" ✓
19 tokio-postgres = "0.7" ✓
20
21 [[bin]]
22 name = "tpl"
23 path = "src/main.rs"
```

#### 3.4.1 Cargo update

**cargo update** is used to update dependencies in Cargo.toml file. It checks the latest versions available for each dependency and **updates the specified version constraints** in Cargo.toml. It updates the Cargo.lock file based on existing constraints in Cargo.toml, keeping version specifications unchanged.

#### 3.4.2 Cargo upgrade

```
1 cargo install cargo-edit # First install cargo-edit.
1 cargo upgrade # upgrade all dependencies
2 cargo upgrade serde # upgrade a specific dependency
3 cargo upgrade --major-versions # might include breaking changes
4 cargo upgrade --incompatible --verbose # update to latest, may break
5 cargo install cargo-outdated # first install cargo-outdated
6 cargo outdated # Check for outdated dependencies
```

**cargo upgrade** updates the versions of dependencies in the Cargo.toml to the latest versions within the constraints specified, and optionally allows for new major versions and breaking changes. It updates both the Cargo.toml and Cargo.lock files, allowing you to specify new version constraints and **upgrade to potentially breaking versions** if desired.

### 3.5 Publish in Crate

- 1 Log in to **crates.io**.
- 2 Click on your username at the top right and select "Account Settings."
- 3 Scroll down to the "API Access" section.
- 4 Click "New Token."

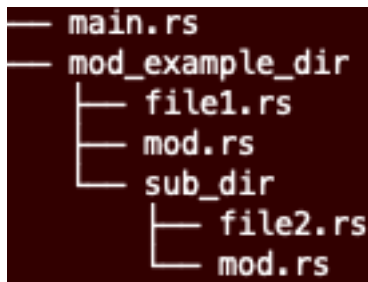
```
1 cargo login <TOKEN>
2 cargo publish
```



## 4 Module

Rust doesn't see files as files, but it sees them as **modules and files inside folders as sub-modules**. You can't reference them directly with a simple import. You need to create a tree of **mod.rs** that lists the contents of the directory. The **::** syntax is used to access associated functions, constants, and types of a module, as well as to call associated functions on a type. The **use** keyword is used to bring paths, types, functions, structs, enums, or traits into scope. The keyword **pub** makes any module, function, or data structure **accessible** from inside of external modules.

Consider the following file structure under **src**



```
1 pub mod sub_dir;
2 pub mod file1;
```

Listing 1: src/mod\_example\_dir/mod.rs

```
1 pub fn test1a() {
2     println!("file1.test1a");
3 }
4
5 pub fn test1b() {
6     println!("file1.test1b");
7 }
```

Listing 2: src/mod\_example\_dir/file1.rs

```
1 pub mod file2;
```

Listing 3: src/mod\_example\_dir/sub\_dir/mod.rs

```
1 pub fn test2() {
2     println!("file2.test2");
3 }
```

Listing 4: src/mod\_example\_dir/sub\_dir/file2.rs

```
1 mod mod_example_dir;
2 use mod_example_dir::file1::test1a;
3
4 fn main() {
5     test1a(); // use specifies the path to test1a()
6     mod_example_dir::file1::test1b(); // use has not specified the path
7     mod_example_dir::sub_dir::file2::test2(); // use has not specified the path
8 }
```

Listing 5: src/main.rs

We use the following **macro** to print some variable x, the **{:?}** placeholder acts as a generic format specifier, allowing Rust to automatically determine the appropriate way to represent the value of x.

```
1 println!("{:?},x)
```

Listing 6: println!

## 5 Data Types & Variables

### 5.1 Data Types

- ① Scalar Types: integers: `i8`, `i16`, `i32`, `i64` unsigned: `u8`, `u16`, `u32`, `u64` floating-point: `f32`, `f64` Unicode character: `char` Boolean: `bool` true or false
- ② Compound Types: `Tuples`, `Arrays`, `Slices`, `Vectors`, `Structs`, `Traits` (Interfaces), `Enums`, `Strings` (UTF-8 encoded text), `References` (Pointers to other values)
- ③ Note: `&str` - String literal, aka string slice, is a sequence of Unicode characters, value is known at compile time, static by default, guaranteed to be valid for the duration of the entire program. `String` is a growable Mutable Collection. String values are allocated on the heap at runtime.
- ④ Other Types: `Raw Pointers`, `Functions`, `Iterators` (Lazy sequences of values), `Closures`

### 5.2 Variables & Mutability

Variables are immutable by default. They are declared using the `let` keyword, with bindings evaluated at runtime. If mutability is required, the `mut` keyword is used to make a variable mutable. Constants, on the other hand, are declared using the `const` keyword and are evaluated at compile time.

```
1 fn main() {  
2     let x = 50;  
3     let mut s = String::from("hello");  
4 }
```

Listing 7: Mutability

### 5.3 Ownership

**Ownership** governs memory safety without needing a garbage collector. It ensures there are no dangling pointers or memory leaks by enforcing strict rules at compile time. The principles:

- ① **Owner:** Each value in Rust has a variable that's its owner. **There can only be one owner at a time. When the owner goes out of scope, the value is dropped.**
- ② **Move semantics:** Variables like `String` don't have multiple owners. When ownership is transferred, the previous owner becomes invalid.
- ③ **Borrow:** Rust allows references to a value through borrowing, either as immutable or mutable, but you **cannot have mutable and immutable references at the same time and there can be only one mutable reference to a particular piece of data in a particular scope.**
- ④ **Ownership rules** apply differently to types that implement the `Copy` trait, such as integers. Primitive types like integers, floats, and other simple types are stored on the stack, and their values are copied rather than moved when assigned or passed to a function.
- ⑤ **String literals** `&str`, `&'static str`, implement the `Copy` trait. This is because string literals are stored in the read-only memory section of the binary, and their data does not change at runtime. Since the memory location is fixed, copying a string literal is both safe and efficient.

```
1 fn main() {  
2     let x = 50;  
3     let y = x + 1;  
4     let s1 = String::from("Hello"); // s1 owns the string  
5     let s2 = s1;                     // Ownership of the string is moved to s2  
6     // println!("{:?}", s1);         // This would result in a compile-time error
```

```
7 }
```

Listing 8: Ownership

## 5.4 Borrow

Many resources are too expensive in terms of time or memory to be copied for every reassignment. In these cases, Rust offers the option to borrow with the ampersand `&` character before the variable.

```
1 fn main() {
2     let s = String::from("hello");
3     let len = calc_len(&s); // Pass an immutable reference to 'calc_len'
4     println!("The length of {} is {}", s, len);
5 }
6
7 fn calc_len(s: &str) -> usize {
8     s.len() // Return the length of the string slice
9 }
```

Listing 9: Borrow

If the variable is mutable, then borrow from the variable using `&mut` before the variable.

```
1
2 fn main() {
3     let mut s = String::from("hello");
4     let s2: &mut String = &mut s; // Borrow a mutable ref to 's', must be initialized
5
6     let len = calc_len(&s2); // Pass an immutable reference to 'calc_len'
7     println!("The length of '{}' is {}", s2, len);
8 }
9
10 fn calc_len(s: &str) -> usize {
11     s.len() // Return the length of the string slice
12 }
```

Listing 10: Borrow Mutable Reference

## 5.5 Lifetime

This construct is used by the compiler to ensure that references are valid as long as they're needed. It helps prevent dangling references, which can occur when a reference outlives the data it points to.

- ① **Each reference has a lifetime**, which is the scope for which the reference is valid.
- ② **Lifetimes are checked at compile time**, ensures that references will never point to invalid data.
- ③ **Explicit lifetimes are usually needed in function signatures when references are involved**, to define how long each reference is valid.

```
1 fn longest<'a>(s1: &'a str, s2: &'a str) -> &'a str {
2     if s1.len() > s2.len() {
3         s1
4     } else {
5         s2
6     }
7 }
8
9 fn main() {
10     let string1 = String::from("hello");
11     let string2 = String::from("world!");
```

```

12
13     let result = longest(&string1, &string2);
14     println!("The longest string is: {}", result);
15 }

```

Listing 11: Lifetime

The `'a` syntax is a lifetime annotation. The returned reference will live at least as long as `s1` and `s2`.

## 6 Collections

### 6.1 Array & Slice

```

1 fn main() {
2     let arr1 = [1, 2, 3, 4, 5]; // array. Must have a known length,
3     println!("{:?}", arr1);    // all elements must be initialized.
4
5     let arr2: [i32; 3] = [1; 3]; // create an array of length 3, all elements set to 1
6     println!("{:?}", arr2);
7
8     let slice = &arr1[1 .. 4]; // create a slice, the last index not included
9     println!("{:?}", slice);   // excludes arr1[3]
10 }

```

Listing 12: Array & Slice

### 6.2 Closure

```

1 fn main() {
2     let y = 3;
3     let add = |x| {
4         x + y
5     };
6
7     let result = add(5);
8     println!("Result: {}", result);
9     let result = receive_closure(add, 6); // pass the closure and i32
10    println!("Result: {}", result);
11 }
12
13 fn receive_closure<F>(f: F, x: i32) -> i32 // arguments are the closure and i32
14     where F: Fn(i32) -> i32 { // a specific closure signature
15     f(x) // as i32
16 }

```

Listing 13: Closure

### 6.3 Vector

```

1 fn main() {
2     let mut v = vec!["Hello", "how", "are", "you"]; // create a vector
3     v.push("Sir"); // push
4     v.pop(); // pop
5     v.insert(4, "Madam"); // insert a value at a specific index
6     v.remove(0); // remove a value at a specific index
7     let index = v.iter().position(|x| x == &"Madam").unwrap(); // find index
8     v.remove(index);
9
10    let result = v.iter().enumerate().find(|(_, &x)| x.eq("you")); // find Tuple
11    if let Some(value) = result {
12        let (i, s) = value;
13        println!("position: {:?} value: {:?}", i, s);
14    }

```

```

15  for i in &v { // iterate, &v because of move due to implicit call to .into_iter()
16      println!("{}", i);
17  }
18  for (i, elem) in v.iter().enumerate() { // iterate using enumerate
19      println!("Element at position {}: {:?}", i, elem);
20  }
21
22  /*
23      When you call v.into_iter(), it consumes the vector v, which means v is no
24      longer available afterward. use v.iter() or v.iter_mut() if you want to
25      borrow to access v later.
26  */
27
28  // Print the sorted vector
29  println!("Sort");
30  v.sort();
31  v.sort_by(|a, b| b.cmp(a));
32  for (i, elem) in v.iter().enumerate() { // iterate using enumerate
33      println!("Element at position {}: {:?}", i, elem);
34  }
35
36  // Filter by "how" and create a new vector. In &&x, the first & comes from v.iter(),
37  // The second & because the elements in the vector are &str
38  println!("Filtered new list");
39  let filtered_v: Vec<_> = v.iter().filter(|&&x| x == "how").cloned().collect();
40  for (i, elem) in filtered_v.iter().enumerate() { // iterate using enumerate
41      println!("Element at position {}: {:?}", i, elem);
42  }
43
44  // Retain only "how" value in the same vector
45  println!("Filtered in same list");
46  v.retain(|&x| x == "how");
47  for (i, elem) in v.iter().enumerate() { // iterate using enumerate
48      println!("Element at position {}: {:?}", i, elem);
49  }
50
51
52 }

```

Listing 14: Vector

## 7 String

```
1 fn main() {
2     let mut s = String::from("Jaideep");
3     s = s.replace("Jaideep","Mone");    // not in place, returns new String
4     println!("replaced = {:?}", s);
5
6     let mut litstr = "Ganguly";
7     s = litstr.to_string();              // convert literal string to String
8     litstr = s.as_str();                 // convert to literal
9     println!("literal = {:?}", s);
10
11    s.push('G');                          // append a char
12    s.push_str("anguly");                 // append a slice
13    println!("len = {}",s.len());        // length
14    println!("trimmed {}",s.trim());     // remove leading and trailing spaces
15    let s = s.clear();                   // clear
16
17    let s = "Jaideep Ganguly";
18    let v: Vec<&str> = s.split_whitespace().collect(); // split by whitesp.
19    for token in v {
20        println!("{}",token);
21    }
22
23    let s = "Jaideep.Ganguly"; // split
24    let x = s.split(".");
25    for token in x {
26        println!("{}",token);
27    }
28
29    let x = s.chars();                  // chars
30    for token in x {
31        println!("{}",token);
32    }
33    let mut s1 = String::from("Jaideep"); // concatenation
34    let s2 = String::from("Ganguly");
35    let mut s1 = s1 + &s2; // not s1 + s2 because of ownership rules
36
37    let s1 = String::from("Jaideep"); // concatenation
38    let s2 = String::from("Ganguly");
39
40    let mut s3 = format!("{}", - {},s1,s2);
41    s3 = s.to_uppercase();              // uppercase
42    s3 = s.to_lowercase();              // lowercase
43 }
```

Listing 15: String

## 8 if else & iteration

```
1 fn main() {
2     let x = 5;
3     if x > 10 {
4         println!("x is greater than 10");
5     } else if x > 5 {
6         println!("x is greater than 5");
7     } else {
8         println!("x is less than or equal to 5");
9     }
10 }
```

Listing 16: if else

```

1 fn main() {
2     for i in (0..10).step_by(2){
3         println!("i: {}", i);
4     }
5
6     let mut count = 0;
7     while count < 5 {
8         println!("Count: {}", count);
9         count += 1;
10    }
11
12    let mut is_running = true;
13    loop { // infinite loop
14        if !is_running {
15            break;
16        }
17        // Do something here, based on that set is_running = false
18    }
19
20    let arr2:[i32; 3] = [1; 3]; // create an array of length 3, all elements set to 1
21    for item in arr2.iter().enumerate() {
22        let (i,x):(usize,&i32) = item;
23        println!("array[{}] = {}", i, x);
24    }
25 }

```

Listing 17: iter

## 9 Data Structures & Traits

### 9.1 Tuple

**Tuple** is a fixed-size collection of values of different types. It is often used to group related data together. Unlike arrays, tuples can contain elements of different types.

```

1 fn main() {
2     let tup1: (i32,f64,String) = (10,200.32,String::from("Jai")); // tuple is mutable
3     let tup2: (i32,f64,&str) = (10,200.32,"Ganguly");
4     println!("{:?} {:?}",tup1.0, tup2.2); // ? implements std::fmt::Display
5 }

```

Listing 18: Tuple

## 9.2 struct

A **struct** is a custom data type that allows you to group related data together into a single unit. A struct can hold multiple pieces of data, each called a field, and each field can have its own type.

```
1 #[derive(Debug)]
2 pub struct Rect<'a> {
3     pub id: &'a str,    // string literals need a lifetime parameter unlike int
4     pub width: i32,     // need the ',' unlike Golang struct
5     pub length: i32
6 }
7 impl<'a> Rect<'a> {
8     pub fn new(id: &'a str, width: i32, length: i32) -> Rect<'a> { // Constructor like
9         ↪ fn
10         Rect { id, width, length }                                // 'Self' will work, but it doesn't
                                                                    // accept parameters such as
11     ↪ lifetime
12     pub fn area(&self) -> i32 {
13         self.width * self.length
14     }
15     pub fn volume(&self, height: i32) -> i32 {
16         self.area()*height
17     }
18 }
19 fn main() {
20     let r1 = Rect {id:"id100",width:10, length:20}; // constructor
21     let r2 = Rect::new("id100", 10,25); // fn used as constructor, any name of fn is ok
22     println!("{:?}", r1.width, r1.length, r2.width, r2.length);
23     println!("Area = {} Volume = {}",r1.area(),r1.volume(10));
24 }
```

Listing 19: Struct

## 9.3 Trait

A **trait** is a collection of methods that are defined for a particular type or a group of types. It allows you to define a set of methods that can be implemented by multiple types, similar to interfaces.

```
1 pub struct Herbivore;
2 pub struct Carnivore;
3 pub trait TrAnimal {
4     fn eat(&self, _n:i32); // signature
5 }
6 impl TrAnimal for Herbivore{
7     fn eat(&self, n:i32) { // specific implementation
8         println!("I eat plants");
9     }
10 }
11 impl TrAnimal for Carnivore { // specific implementation
12     fn eat(&self, n:i32) {
13         println!("{}", "I eat meat",n);
14     }
15 }
16 fn main() {
17     let h = Herbivore;
18     h.eat(3);
19     let c = Carnivore;
20     c.eat(4);
21 }
```

Listing 20: Trait



## 9.4 trait bound

A **trait bound** is a constraint on a generic type that specifies that the type must implement a particular trait (or traits). Trait bounds are used to guarantee that a generic type supports specific functionality. **This allows Rust to enforce compile-time checks, ensuring that only types that implement the required traits can be passed to the generic function or used in a generic struct.**

```
1 pub trait TrActivity {
2     fn fly(&self);
3 }
4
5 #[derive(Debug)]
6 pub struct Eagle;
7
8 impl TrActivity for Eagle {
9     fn fly(&self) {
10         println!("{:?} is flying",&self);
11     }
12 }
13
14 pub fn activity<T: TrActivity + std::fmt::Debug>(bird: T) {
15     println!("I fly as {:?}",bird);
16 }
17
18 fn main() {
19     let eagle = Eagle;
20     eagle.fly();
21     activity(eagle);
22 }
```

Listing 21: Trait bound

This function **activity** is generic over a type **T**. The **trait bound** **T: TrActivity + std::fmt::Debug** specifies that the type **T** must implement both the **TrActivity** and **Debug** traits.

## 9.5 to\_owned() and clone()

The **to\_owned()** method is specifically designed for converting borrowed data (like a **&str** or **&[T]**) into owned data (like **String** or **Vec<T>**). It is defined by the **ToOwned** trait, which is implemented for types that can produce an owned version of themselves.

```
1 fn main() {
2     let s = "Hello, world!";
3     let s1 = s.to_owned(); // Creates a new owned String
4     let s2 = s.clone(); // Also creates a new owned String
5
6     let v = vec![1, 2, 3];
7     let v1 = v.to_owned(); // Creates a new owned vector
8     let v2 = v.clone(); // Also creates a new owned vector
9     println!("{:?} {:?}",v1,v2);
10 }
```

Listing 22: to\_owned() & clone()

## 9.6 vec of struct

```
1 use std::sync::{Arc, Mutex};
2 use itertools::Itertools;
3
4 #[derive(Debug, Clone)]
5 struct MyStruct<'a> {
6     id: i32,
7     label: &'a str,
8 }
9
10 fn main() {
11     // Create a vector of 10 instances of 'MyStruct'
12     let v = vec![
13         MyStruct { id: 1, label: "a" },
14         MyStruct { id: 2, label: "b" },
15         MyStruct { id: 3, label: "c" },
16         MyStruct { id: 4, label: "a" },
17         MyStruct { id: 5, label: "b" },
18         MyStruct { id: 6, label: "c" },
19         MyStruct { id: 7, label: "a" },
20         MyStruct { id: 8, label: "b" },
21         MyStruct { id: 9, label: "c" },
22         MyStruct { id: 10, label: "b" },
23     ];
24
25     let v2 = v.clone();
26     // Filter by label "b" and create a mutable vector
27     let filtered_v: Vec<_> = v.into_iter().filter(|x| x.label == "b").collect();
28
29     // Sort the filtered vector by 'id' in descending order
30     let mut sorted_filtered_v = filtered_v.clone(); // Clone the filtered vector
31     sorted_filtered_v.sort_by(|a, b| b.id.cmp(&a.id));
32
33     // combined statement
34     let sorted_filtered_v: Vec<_> = v2.iter()
35         .filter(|x| x.label == "b")
36         .sorted_by(|a, b| b.id.cmp(&a.id))
37         .collect();
38
39     for instance in &sorted_filtered_v {
40         println!("{:?}", instance);
41     }
42     println!();
43
44     // To lock and add, wrap filtered vector in Arc<Mutex<>>
45     // for shared ownership & thread safety
46     let shared_v = Arc::new(Mutex::new(filtered_v));
47     {
48         // Lock the mutex to get mutable access and add an 11th instance
49         let mut locked_v = shared_v.lock().unwrap();
50         locked_v.push(MyStruct { id: 11, label: "x" });
51     } // Mutex lock is released here automatically
52
53     // Now, let's lock and print the shared and modified vector
54     let locked_v = shared_v.lock().unwrap();
55     for instance in locked_v.iter() {
56         println!("{:?}", instance);
57     }
58 }
```

Listing 23: vec of struct

## 10 Enum

Enums are a way to define a type that can have one of several possible variants with data.

### 10.1 Match

```
1 enum Shape {
2     Rectangle { width: u32, height: u32 },
3     Circle { radius: f64 },
4     Square(u32)
5 }
6
7 fn main() {
8     let mut shape: Shape;
9     shape = Shape::Rectangle { width: 5, height: 10 };
10
11     let mut w: u32=0;
12     let mut h: u32=0;
13     let mut r: f64=0.0;
14     let mut s: u32=0;
15
16     match shape {
17         Shape::Rectangle { width, height } => {
18             println!("Rectangle: width = {}, height = {}", width, height);
19             w = width;
20             h = height;
21         },
22         Shape::Circle { radius } => {
23             println!("Circle: radius = {}", radius);
24             r = radius;
25         },
26         Shape::Square(side) => {
27             println!("Square: side = {}", side);
28             s = side;
29         },
30     }
31     println!("w={:?} h={:?}",w,h);
32 }
```

Listing 24: Enum & Match

### 10.2 Result { OK, Err }

```
1 use std::result::Result;
2
3 fn main() {
4     let result = get_result("Hi".to_string());
5     println!("{:?}",result);
6
7     match result {
8         Ok(value) => {
9             // Use the extracted value
10            println!("Matched value: {}", value);
11        }
12        Err(error) => {
13            // Handle the error case
14            println!("No Match: {}", error);
15        }
16    }
17 }
18
```

```

19
20 fn get_result(s: String) -> Result<String,String> {
21     if s.eq("Hello") {
22         Ok(s)
23     } else {
24         Err(String::from("not Hello")) // Err(String::from("not Hello"))
25     }
26 }

```

Listing 25: Result

### 10.3 ResultErrStr { OK, Err }

This is convenient, the matched value in **Err** is a literal, not a string.

```

1 type ResultErrStr<T> = std::result::Result<T, &'static str>;
2
3 fn get_result(s: String) -> ResultErrStr<String> {
4     if s.eq("Hello") {
5         Ok(s)
6     } else {
7         Err("not Hello") // literal, not String
8     }
9 }
10
11 fn main() {
12     let s = String::from("Hi");
13     let result = get_result(s);
14     println!("{:?}", result);
15
16     match result {
17         Ok(value) => {
18             // Use the extracted value
19             println!("Matched value: {}", value);
20         }
21         Err(error) => {
22             // Handle the error case
23             println!("No Match: {}", error);
24         }
25     }
26 }

```

Listing 26: ResultErrStr

### 10.4 Option { Some, None }

```

1 use std::option::Option;
2
3 fn main() {
4     let option = get_option(10,0);
5
6     match option {
7         Option::Some(value) => {
8             // Use the extracted value
9             println!("value: {}", value);
10        }
11        Option::None => {
12            // Handle the error case
13            println!("None");
14        }
15    }

```

```

16 }
17
18 pub fn get_option(x: i32, y:i32) -> Option<i32> {
19     if y != 0 {
20         Option::Some(x/y)
21     } else {
22         Option::None
23     }
24 }

```

Listing 27: Option

## 10.5 if let

The **if let** construct is a concise way to match a value against a pattern and conditionally execute code if the pattern matches. It is used when you only care about one specific pattern and ignore others.

```

1 fn main() {
2     let mut s: String = "Hi".to_string();
3     let result = get_result(s);
4
5     if let Ok(value) = result {
6         s = value.to_owned();
7         println!("Extracted value s is: {}", s);
8     } else if let Err(error) = result {
9         println!("Error: {}", error);
10    }
11 }
12
13 fn get_result(s: String) -> Result<String,String> {
14     if s.eq("Hello") {
15         Ok(s)
16     } else {
17         Err("not Hello".to_string()) // Err(String::from("not Hello"))
18     }
19 }

```

Listing 28: if let

## 10.6 Unwrap

**unwrap\_or\_else** is a concise way to handle the Result value result and assign a default value to s if the Result is an Err. Just using **unwrap()** can cause **panic**.

```

1 fn main() {
2     let mut s: String = "Hi".to_string();
3     let result = get_result(s.clone());
4     s = result.unwrap_or_else(|_| "Error".to_string());
5     println!("Final value of s is: {}", s);
6 }
7
8 fn get_result(s: String) -> Result<String, String> {
9     if s.eq("Hello") {
10        Ok(s)
11    } else {
12        Err("not Hello".to_string()) // Err(String::from("not Hello"))
13    }
14 }

```

Listing 29: unwrap

## 11 HashMap

```
1 use std::collections::HashMap;
2
3 fn main() {
4
5     // create a hashmap
6     let mut hm: HashMap<String,String> = HashMap::new();
7
8     // insert key
9     hm.insert("MA".to_string(),"Massachusetts".to_string());
10    hm.insert("NY".to_string(),"New York".to_string());
11    hm.insert("CA".to_string(),"California".to_string());
12
13    // iterate over key, value
14    for (key, val) in hm.iter() {
15        println!("key: {} val: {}", key, val);
16    }
17
18    // find value corresponding to a key
19    let key = "MA".to_string();
20    match hm.get(&key) {
21        Some(value) => println!("The value for key '{}' is: {}", key, value),
22        None => println!("Key '{}' not found in the HashMap.", key),
23    }
24
25    // remove a key (and value)
26    hm.remove("CA");
27    for (key, val) in hm.iter() {
28        println!("key: {} val: {}", key, val);
29    }
30
31    // length of hashmap
32    println!("{:?}", hm.len());
33
34 }
```

Listing 30: HashMap

## 12 Generic

```
1
2 fn main() {
3     struct Data<T> {
4         value: T,
5     }
6
7     let t:Data<i32> = Data{value:350};           // i32
8     println!("value is {:?}",t.value);
9
10    let t2:Data<String> = Data{value:"Tom".to_string()}; // String
11    println!("value is {:?}",t2.value);
12 }
```

Listing 31: Generic

## 13 Smart Pointers, Deref & Drop

### 13.1 Smart Pointers

In Rust there is an additional difference between references and smart pointers in that references are pointers that only borrow data while smart pointers, in many cases, own the data they point to. Smart pointers are mostly implemented using structs. These structs implement the **Deref** and **Drop** traits.

The most straightforward smart pointer is a box, whose type is written **Box<T>**. Boxes allow you to store data on the heap rather than the stack. What remains on the stack is the pointer to the heap data. Boxes do not have any performance overhead, other than storing their data on the heap instead of on the stack. Box is useful under the following circumstances.

- 1 A type whose size is unknown at compile time and we want to use a value of that type in a context that requires an exact size.
- 2 A large amount of data, we want to **transfer ownership but do not want the data to be copied**.
- 3 Own a value and its type must implement a certain trait rather being of a particular type.

```
1 fn main() {  
2     let x = Box::new(100);  
3     println!("x = {}", x);  
4 }
```

Listing 32: Box

### 13.2 Deref

**Dereferencing lets you access the value a reference points to without owning the value itself.** The **Deref** trait allows you to customize how dereferencing works using the **\*** operator. When a type implements the Deref trait, Rust lets you treat that type like a reference and access its contents as if it were a simple pointer.

For example, **smart pointers** like **Box**, **Rc**, and **Arc** implement **Deref** to give you more functionality while still behaving like normal references. **Deref coercion** automatically **converts a reference to a type with Deref into a reference to its target type**.

The syntax **type Target = T;** in a **Deref implementation** tells Rust what type the smart pointer should dereference to. **For example, in MyBox<T>, it means that MyBox<T> will dereference to T.** This simplifies accessing the actual value inside smart pointers.

```
1 use std::ops::Deref;  
2  
3 fn main() {  
4     let person = MyBox( Person { age:50, name:"Ram".to_string() } );  
5     println!("{}",person.age,person.name); // Use deref to access fields of boxed  
6     ↪ Person  
7     person.greet();  
8 }  
9 struct Person {  
10     age: u32,  
11     name: String,  
12 }  
13  
14 impl Person {  
15     fn greet(&self) {  
16         println!("Hello, my name is {} and I'm {} years old.", self.name, self.age);  
17     }  
18 }
```

```

17     }
18 }
19
20 struct MyBox<T>(T); // Create a wrapper struct that will implement Deref
21
22 impl<T> Deref for MyBox<T> {
23     type Target = T;
24
25     fn deref(&self) -> &Self::Target {
26         &self.0
27     }
28 }

```

Listing 33: Deref

### 13.3 Drop

Implementing **Drop** allows defining custom cleanup behavior when a smart pointer goes out of scope. This ensures that resources like memory, DB connections, etc., are properly released, preventing leaks. Smart pointers use **Drop** to manage ownership and resource cleanup efficiently.

```

1 struct Resource {
2     name: String,
3 }
4
5 impl Drop for Resource {
6     fn drop(&mut self) {
7         println!("Dropping resource: {}", self.name);
8     }
9 }
10
11 fn main() {
12     {
13         let res1 = Resource {
14             name: String::from("Resource 1"),
15         };
16         let res2 = Resource {
17             name: String::from("Resource 2"),
18         };
19         println!("Resources created.");
20     } // res1 and res2 go out of scope here, triggering the drop method
21     // At this point, the drop method for res1 and res2 has been called.
22 }

```

Listing 34: Drop

### 13.4 dyn

The **dyn** keyword denotes dynamically sized types, specifically for creating trait objects. **Box** is used to store a **trait** object of type **Error**. This indicates that while the exact type of the error is not known at compile time, it must implement the **Error** trait. The **dyn** keyword facilitates the creation of **trait** objects, enabling dynamic dispatch, i.e., method implementations to be resolved at runtime.

```

1 use std::error::Error;
2
3 fn main() -> Result<(), Box<dyn Error>> {
4     let result = get_result();
5
6     match result {

```



```

7      Ok(_) => println!("Success!"),
8      Err(e) => println!("Error: {}", e),
9  }
10 Ok(())
11 }
12
13 fn get_result() -> Result<(), Box<dyn Error>> {
14     let result: Result<(), Box<dyn Error>> = Ok(());
15     // let result: Result<(), Box<dyn Error>> = Err("An error occurred".into());
16     result
17 }

```

Listing 35: dyn

### 13.5 tokio

The `#[tokio::main]` attribute is used when working with the `Tokio` asynchronous runtime. It marks the main function of your application as an asynchronous entry point, enabling the use of asynchronous code within it.

```

1 use std::error::Error;
2
3 #[tokio::main]
4 async fn main() -> Result<(), Box<dyn Error>> {
5
6 }

```

Listing 36: tokio

## 14 Async

### 14.1 Long Running Functions

```
1 use std::error::Error;
2 use std::thread;
3 use tokio::time::{sleep, Duration, timeout, Instant};
4
5 pub async fn long_running_fn_1(x: &mut i32) -> i32 {
6     sleep(Duration::from_secs(10)).await; // don't use thread::sleep
7     *x = *x + 1;
8     println!("Task 1 completed successfully with output: {}", *x);
9     return (*x);
10 }
11
12 pub async fn long_running_fn_2() -> i32 {
13     sleep(Duration::from_secs(1)).await;
14     println!("Task 2 completed successfully with output: {}", 42);
15     return(42);
16 }
17
18 #[tokio::main]
19 async fn main() -> Result<(), Box<dyn Error>> {
20     let mut x = 100;
21
22     let task1 = tokio::spawn(async move { long_running_fn_1(&mut x).await }); // spawn
23     let task2 = tokio::spawn(async move { long_running_fn_2().await }); // spawn
24
25     let timeout_duration1 = Duration::from_secs(15); // timeout
26     let timeout_duration2 = Duration::from_secs(5); // timeout
27
28     let result1 = tokio::time::timeout(timeout_duration1, task1);
29     let result2 = tokio::time::timeout(timeout_duration1, task2);
30
31     let (result1, result2) = tokio::join!(result1, result2); // run concurrently & join
32
33     match result1 { // Handle results for task1
34         Ok(task_result1) => match task_result1 {
35             Ok(output) => println!("Task 1 success with output: {}", output),
36             Err(error) => println!("Task 1 error during execution: {:?}", error),
37         },
38         Err(_) => println!("Task 1 timedout after {} sec", timeout_duration1.as_secs()),
39     }
40
41     match result2 { // Handle results for task2
42         Ok(task_result2) => match task_result2 {
43             Ok(output) => println!("Task 2 success with output: {}", output),
44             Err(error) => println!("Task 2 error during execution: {:?}", error),
45         },
46         Err(_) => println!("Task 2 timedout after {} sec", timeout_duration2.as_secs()),
47     }
48
49     Ok(())
50 }
```

Listing 37: Long Running functions

## 14.2 Arc, Mutex & Lock

- 1 **Arc** is an **Atomic Reference Counted** pointer, a smart pointer that enables multiple ownership of the same data. **Arc** enables sharing of data safely across threads. The reference count is updated atomically, ensuring that the data remains valid as long as there are references to it.
- 2 A **Mutex** is a synchronization primitive that provides mutual exclusion, allowing only one thread to access the data at a time. When a **Mutex** is locked, it returns a **MutexGuard**, which represents the locked state. The **MutexGuard** ensures that the **Mutex** remains locked for the duration of its scope, automatically releasing the lock when it goes out of scope. **This prevent deadlocks and ensures that the Mutex is always properly unlocked.**
- 3 **Lock** represents the result of acquiring a Mutex. It provides methods like deref to access the protected data and unlock to release the lock and guarantees exclusive access to the data while the lock is held.

The **Send** and **Sync** traits are crucial for ensuring safe concurrent programming. The **Send** trait allows a **type** to be safely transferred from one thread to another. If a type implements Send, it means ownership of that type can be passed between threads. The **Sync** trait allows a type to be safely shared between multiple threads, meaning that multiple threads can reference the same value concurrently. A **type** that is **Sync** is also **Send**, but the **converse is not always true**.

### 14.2.1 Arc, Mutex, Lock with HashMap

```
1 use tokio::sync::Mutex;
2 use std::collections::HashMap;
3 use std::sync::Arc;
4 use std::time::Duration;
5
6 // future generated needs to be safely sent across threads, Send + Sync
7 async fn long_running_fn_1(x: &mut i32) -> Result<String, Box<dyn std::error::Error +
  ↳ Send + Sync>> {
8     tokio::time::sleep(Duration::from_secs(10)).await; // Simulate long-running
  ↳ computation
9     *x += 1;
10    Ok(format!("{}", x))
11 }
12
13 // future generated needs to be safely sent across threads, Send + Sync
14 async fn long_running_fn_2() -> Result<String, Box<dyn std::error::Error + Send +
  ↳ Sync>> {
15     tokio::time::sleep(Duration::from_secs(2)).await; // long-running computation
16     Ok("").to_string()
17 }
18
19 #[tokio::main]
20 async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
21     let mut x = 100;
22
23     let shared_map = Arc::new(Mutex::new(HashMap::new())); // clone below does not
  ↳ create deep copy; it increments ref count of shared object
24     let shared_map_clone1 = Arc::clone(&shared_map);
25     let shared_map_clone2 = Arc::clone(&shared_map);
26
27     let timeout_duration1 = Duration::from_secs(30);
28     let timeout_duration2 = Duration::from_secs(1);
29
30     let task1 = tokio::spawn(async move { // Task 1
31         let mut x = x; // x is now owned by this task
```

```

32     let result = long_running_fn_1(&mut x).await;
33     let mut shared_map = shared_map_clone1.lock().await;
34
35     match result {
36         Ok(output) => {
37             shared_map.insert("task1".to_string(), output);
38             // println!("{:?}", shared_map);
39             println!("Task 1 inserted result into map");
40         },
41         Err(error) => {
42             shared_map.insert("task1".to_string(), format!("Error: {:?}", error));
43             println!("Task 1 encountered an error: {:?}", error);
44         },
45     }
46 });
47
48 let task2 = tokio::spawn(async move { // Task 2
49     let result = long_running_fn_2().await;
50     let mut shared_map = shared_map_clone2.lock().await;
51     match result {
52         Ok(output) => {
53             shared_map.insert("task2".to_string(), output);
54             // println!("{:?}", shared_map);
55             println!("Task 2 inserted result into map");
56         },
57         Err(error) => {
58             shared_map.insert("task2".to_string(), format!("Error: {:?}", error));
59             println!("Task 2 encountered an error: {:?}", error);
60         },
61     }
62 });
63
64 // Apply timeouts
65 let result1 = tokio::time::timeout(timeout_duration1, task1);
66 let result2 = tokio::time::timeout(timeout_duration2, task2);
67
68 // Use join to concurrently wait for both results
69 let (result1, result2) = tokio::join!(result1, result2);
70
71 match result1 { // Handle task 1 result
72     Ok(Ok(_)) => println!("Task 1 completed"),
73     Ok(Err(error)) => println!("Task 1 error during execution: {:?}", error),
74     Err(_) => println!("Task 1 timed out after {} sec",
↪ timeout_duration1.as_secs()),
75 }
76
77 match result2 { // Handle task 2 result
78     Ok(Ok(_)) => println!("Task 2 completed"),
79     Ok(Err(error)) => println!("Task 2 error during execution: {:?}", error),
80     Err(_) => println!("Task 2 timed out after {} sec",
↪ timeout_duration2.as_secs()),
81 }
82
83 // Print the shared map contents
84 let shared_map = shared_map.lock().await; // Lock the shared map here
85 if shared_map.is_empty() {
86     println!("Map is empty");
87 } else {
88     for (key, value) in shared_map.iter() {
89         println!("{:}: {:?}", key, value);
90     }

```

```

91     }
92     Ok(())
93 }

```

Listing 38: Mutex on HashMap

### 14.2.2 Arc, Mutex, Lock with Struct

```

1 use tokio::sync::Mutex;
2 use std::sync::Arc;
3 use std::time::Duration;
4
5 // Struct to hold the results from both tasks
6 struct SharedResults {
7     task1_result: Option<String>,
8     task2_result: Option<String>,
9 }
10
11 impl SharedResults {
12     fn new() -> Self {
13         SharedResults {
14             task1_result: None,
15             task2_result: None,
16         }
17     }
18 }
19
20 // future generated needs to be safely sent across threads, Send + Sync
21 async fn long_running_fn_1(x: &mut i32) -> Result<String, Box<dyn std::error::Error +
    ↪ Send + Sync>> {
22     tokio::time::sleep(Duration::from_secs(10)).await; // long-running computation
23     *x += 1;
24     Ok(format!("{}", x))
25 }
26
27 // future generated needs to be safely sent across threads, Send + Sync
28 async fn long_running_fn_2() -> Result<String, Box<dyn std::error::Error + Send +
    ↪ Sync>> {
29     tokio::time::sleep(Duration::from_secs(2)).await; // long-running computation
30     Ok("").to_string()
31 }
32
33 #[tokio::main]
34 async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
35     let mut x = 100;
36
37     let shared_results = Arc::new(Mutex::new(SharedResults::new())); // clone below
    ↪ does not create deep copy; it increments ref count of shared object
38     let shared_results_clone1 = Arc::clone(&shared_results);
39     let shared_results_clone2 = Arc::clone(&shared_results);
40
41     let timeout_duration1 = Duration::from_secs(30);
42     let timeout_duration2 = Duration::from_secs(1);
43
44     let task1 = tokio::spawn(async move { // Task 1
45         let mut x = x; // x is owned by this task
46         let result = long_running_fn_1(&mut x).await;
47         let mut shared_results = shared_results_clone1.lock().await;
48         match result {
49             Ok(output) => {
50                 shared_results.task1_result = Some(output);
51                 println!("Task 1 inserted result into shared struct");

```

```

52     },
53     Err(error) => {
54         shared_results.task1_result = Some(format!("Error: {:?}", error));
55         println!("Task 1 encountered an error: {:?}", error);
56     },
57 }
58 });
59
60 let task2 = tokio::spawn(async move { // Task 2
61     let result = long_running_fn_2().await;
62     let mut shared_results = shared_results_clone2.lock().await;
63     match result {
64         Ok(output) => {
65             shared_results.task2_result = Some(output);
66             println!("Task 2 inserted result into shared struct");
67         },
68         Err(error) => {
69             shared_results.task2_result = Some(format!("Error: {:?}", error));
70             println!("Task 2 encountered an error: {:?}", error);
71         },
72     }
73 });
74
75 // Apply timeouts
76 let result1 = tokio::time::timeout(timeout_duration1, task1);
77 let result2 = tokio::time::timeout(timeout_duration2, task2);
78
79 // Use join to concurrently wait for both results
80 let (result1, result2) = tokio::join!(result1, result2);
81
82 match result1 { // Handle task 1 result
83     Ok(Ok(_)) => println!("Task 1 completed"),
84     Ok(Err(error)) => println!("Task 1 error during execution: {:?}", error),
85     Err(_) => println!("Task 1 timed out after {} sec",
86 ↪ timeout_duration1.as_secs()),
87 }
88
89 match result2 { // Handle task 2 result
90     Ok(Ok(_)) => println!("Task 2 completed"),
91     Ok(Err(error)) => println!("Task 2 error during execution: {:?}", error),
92     Err(_) => println!("Task 2 timed out after {} sec",
93 ↪ timeout_duration2.as_secs()),
94 }
95
96 // Print the shared struct contents
97 let shared_results = shared_results.lock().await; // Lock the shared struct here
98 if shared_results.task1_result.is_none() && shared_results.task2_result.is_none() {
99     println!("No results in the shared struct");
100 } else {
101     if let Some(result1) = &shared_results.task1_result {
102         println!("Task 1 Result: {}", result1);
103     }
104     if let Some(result2) = &shared_results.task2_result {
105         println!("Task 2 Result: {}", result2);
106     }
107 }
108
109 println!("Completed");
110 Ok(())

```

## Listing 39: Mutex on Struct

### 14.3 Message Passing

```
1 use std::process;
2 use std::sync::mpsc;
3 use std::thread;
4 use std::time::Duration;
5
6 fn main() {
7     // Channel to send and receive messages between concurrent sections of code; has
8     ↪ two halves, a transmitter and a receiver.
9     let (tx, rx) = mpsc::channel(); // multiple producer, 1 consumer
10    let tx2 = mpsc::Sender::clone(&tx); // clone a second producer
11
12    // spawn a thread, move the transmitter into the closure
13    // spawned thread will now own the transmitter
14    thread::spawn( move || {
15        let vals = vec![
16            String::from("Hello"),
17            String::from("from"),
18            String::from("thread-1"),
19        ];
20        for val in vals {
21            tx.send(val).unwrap();
22            thread::sleep(Duration::from_secs(1));
23        }
24    });
25
26    thread::spawn( move || { // same comments as above
27        let vals = vec![
28            String::from("Hi"),
29            String::from("there"),
30            String::from("thread-2"),
31        ];
32        for val in vals {
33            tx2.send(val).unwrap();
34            thread::sleep(Duration::from_secs(1));
35        }
36    });
37
38    // receive the result, timeout beyond 1 sec
39    let result = rx.recv_timeout(Duration::from_millis(1000));
40    match result {
41        Err(e) => {
42            println!("{:?}", e);
43            process::exit(0);
44        },
45        Ok(x) => {
46            for received in rx {
47                println!("Got: {}", received);
48            }
49        }
50    }
51 }
```

Listing 40: Message Passing



## 15 PostgreSQL

```
1 use postgres::{Client, NoTls, Error};
2 use chrono::prelude::*; // For date and time
3
4 #[derive(Debug, PartialEq, Eq)]
5 struct Tab {
6     cat: String,
7     tsk: String,
8 }
9
10 fn main() -> Result<(), Error> {
11     let connection_string = "postgresql://imsrole:imspass@localhost:5432/postgres";
12
13     let mut client = Client::connect(connection_string, NoTls)?;
14
15     // SELECT query
16     let selected_tab = client.query(
17         "SELECT cat, tsk FROM ims.tmp",
18         &[],
19     )?;
20
21     let tabs: Vec<Tab> = selected_tab.iter().map(|row| {
22         Tab {
23             cat: row.get(0),
24             tsk: row.get(1),
25         }
26     }).collect();
27
28     for r in &tabs {
29         println!("{}", r.cat, r.tsk);
30     }
31
32     // INSERT query
33     let rows = vec![
34         Tab { cat: "Test-11".to_string(), tsk: "Test-12".to_string() },
35         Tab { cat: "Test-21".to_string(), tsk: "Test-22".to_string() },
36     ];
37
38     for row in rows {
39         client.execute(
40             "INSERT INTO ims.tmp (cat, tsk) VALUES ($1, $2)",
41             [&row.cat, &row.tsk],
42         )?;
43     }
44
45     Ok(())
46 }
```

Listing 41: PostgreSQL

## 16 IO

```
1 use std::fs;
2 use std::fs::{File,OpenOptions};
3 use std::io::{Read,Write};
4
5 pub fn fn_stdin() {
6     let mut line = String::new();
7     println!("Please enter your name:");
8     let nb = std::io::stdin().read_line(&mut line).unwrap();
9     println!("Hi {}", line);
10    println!("# of bytes read , {}", nb);
11 }
12
13 pub fn fn_stdout() {
14     let b1 = std::io::stdout().write("Hi ".as_bytes()).unwrap();
15     let b2 = std::io::stdout().write(String::from("There\n").as_bytes()).unwrap();
16     std::io::stdout().write(format!("#bytes written {}",(b1+b2)).as_bytes()).unwrap();
17 }
18
19 pub fn fn_cmdline() {
20     let cmd_line = std::env::args();
21     println!("# of command line arguments:{}",cmd_line.len());
22     for arg in cmd_line {
23         println!("{}",arg);
24     }
25 }
26
27 pub fn fn_fileread(filename: &str){
28     let mut file = std::fs::File::open(filename).unwrap();
29     let mut contents = String::new();
30     file.read_to_string(&mut contents).unwrap();
31     let _ = file.sync_all(); // Explicitly close the file by syncing the changes
32     print!("{}", contents);
33 }
34
35 pub fn fn_filewrite(filename: &str, s: &str) {
36     let mut file = std::fs::File::create(filename).expect("Create failed");
37     file.write_all(s.as_bytes()).expect("write failed");
38     println!("Write completed" );
39 }
40
41 pub fn fn_fileappend(filename: &str, s: &str) {
42     let mut file = OpenOptions::new().append(true).open(filename).expect("Not found");
43     file.write_all(s.as_bytes()).expect("write failure");
44     println!("Appended file {}",filename);
45 }
46
47 pub fn fn_filecopy(src: &str, des: &str) {
48     let mut file_inp = std::fs::File::open(src).unwrap();
49     let mut file_out = std::fs::File::create(des).unwrap();
50     let mut buffer = [0u8; 4096];
51     loop {
52         let nbytes = file_inp.read(&mut buffer).unwrap();
53         file_out.write(&buffer[..nbytes]).unwrap();
54         if nbytes < buffer.len() {
55             break;
56         }
57     }
58 }
59 }
```

```

60 pub fn fn_filedelete(filename: &str) {
61     fs::remove_file(filename).expect("Unable to delete file");
62     println!("Deleted file {}", filename);
63 }
64
65 fn main() {
66     fn_stdin();
67     fn_stdout();
68     fn_cmdline();
69     fn_fileread("/Users/jganguy/Per/GitHub/rustlearn/plg/Cargo.toml");
70     fn_filewrite("io_write", "Write test");
71     fn_fileappend("io_write", "More stuff");
72     fn_filecopy("io_write", "io_write_copy");
73     fn_filedelete("io_write");
74 }

```

Listing 42: dyn

## 17 gRPC

Note the use of `Pin` which is a wrapper type provided by the standard library to ensure that the value it wraps cannot be moved. This is particularly useful for asynchronous programming and when dealing with self-referential structs, where moving a value can invalidate references or cause other issues.

### 17.1 Server

```
1 use tonic::{transport::Server, Request, Response, Status};
2 use tokio_stream::wrappers::ReceiverStream;
3 use tokio::sync::mpsc;
4 use futures::{Stream, StreamExt}; // for Stream
5 use std::pin::Pin;
6 use tokio::time::{sleep, Duration};
7
8 // Import the generated protobuf code
9 pub mod tpl {
10     tonic::include_proto!("tpl"); // Ensure this matches the package name in tpl.pro
11 }
12
13 use tpl::example_service_server::{ExampleService, ExampleServiceServer};
14 use tpl::{RequestStruct, ResponseStruct};
15
16 #[derive(Default)]
17 pub struct MyExampleService;
18
19
20 #[tonic::async_trait]
21 impl ExampleService for MyExampleService {
22     type SomeMethodStream = Pin<Box<dyn Stream<Item = Result<ResponseStruct, Status>> +
23         ↪ Send + Sync>>; // Use futures::Stream
24
25     // Implement the bidirectional streaming SomeMethod
26     async fn some_method( &self, request: Request<tonic::Streaming<RequestStruct>>) ->
27         ↪ Result<Response<Self::SomeMethodStream>, Status> {
28
29         let mut stream = request.into_inner();
30         let (tx, rx) = mpsc::channel(4); // Create channel to send responses, buffer = 4
31
32         // Create a vector to store the received items
33         let mut items: Vec<RequestStruct> = Vec::new();
34
35         tokio::spawn(async move {
36             // read an item from stream, process (gets automatically removed)
37             while let Some(req) = stream.next().await {
38                 match req {
39                     Ok(req_struct) => {
40                         dbg!(&req_struct); // won't print in "cargo build --release"
41                         let id = req_struct.id;
42                         items.push(req_struct);
43                         sleep(Duration::from_secs(2)).await;
44
45                         let response = ResponseStruct { // Process, create response
46                             message: format!("Processed ID: {}", id),
47                         };
48
49                         if let Err(_) = tx.send(Ok(response)).await { // Send response
50                             eprintln!("Client disconnected");
51                         }
52                     }
53                     _ => {}
54                 }
55             }
56         });
57     }
58 }
```

```

50         return;
51     }
52 }
53
54     Err(e) => {
55         eprintln!("Error receiving stream: {:?}", e);
56         return;
57     }
58 }
59 }
60 });
61
62
63 let response_stream = ReceiverStream::new(rx); // Return response stream
64
65 Ok(Response::new(
66     Box::pin(response_stream) as Self::SomeMethodStream
67 ))
68 }
69 }
70
71 #[tokio::main]
72 async fn main() -> Result<(), Box<dyn std::error::Error>> {
73
74     let addr = "[::1]:50051".parse()?; // Define the address for the serve
75     let example_service = MyExampleService::default();
76     println!("gRPC Server listening on {}", addr);
77
78     // Start the gRPC server
79     Server::builder()
80         .add_service(ExampleServiceServer::new(example_service))
81         .serve(addr)
82         .await?;
83
84     Ok(())
85 }

```

Listing 43: Server Code

## 17.2 Client

```
1 use tonic::transport::Channel;
2 use tonic::Request;
3 use futures::StreamExt; // for StreamExt::next
4 use futures::stream; // Import for stream::iter
5 use chrono::{Utc, TimeZone};
6 use prost_types::Timestamp;
7 use tokio::time::{sleep, Duration};
8
9 pub mod tpl {
10     tonic::include_proto!("tpl"); // Ensure this matches the package name in tpl.pro
11 }
12
13 use tpl::example_service_client::ExampleServiceClient;
14 use tpl::{RequestStruct, ResponseStruct};
15
16 // Function to convert a chrono::DateTime to prost_types::Timestamp
17 fn datetime_to_timestamp(datetime: chrono::DateTime<Utc>) -> Timestamp {
18     Timestamp {
19         seconds: datetime.timestamp(),
20         nanos: datetime.timestamp_subsec_nanos() as i32,
21     }
22 }
23
24
25 #[tokio::main]
26 async fn main() -> Result<(), Box<dyn std::error::Error>> {
27
28     // create gRPC channel & client
29     let channel = Channel::from_static("http://[::1]:50051").connect().await?;
30     let mut client = ExampleServiceClient::new(channel);
31
32     let ts = datetime_to_timestamp( Utc::now());
33     let r1 = RequestStruct {
34         id: 1,
35         value: 1.0,
36         name: "Item1".into(),
37         datetime: Some(ts)
38     };
39
40     sleep(Duration::from_secs(2)).await;
41     let ts = datetime_to_timestamp( Utc::now());
42     let r2 = RequestStruct {
43         id: 2,
44         value: 2.0,
45         name: "Item2".into(),
46         datetime: Some(ts)
47     };
48
49 };
50
51 let request_stream = stream::iter(vec![ r1, r2 ]);
52
53 // Call the gRPC method "SomeMethod" in proto, i.e., some_method in gRPC server
54 let response_stream = client.some_method(Request::new(request_stream)).await?;
55
56 // Process the responses
57 let mut response_stream = response_stream.into_inner();
58 while let Some(response) = response_stream.next().await {
59     match response {
60         Ok(response_struct) => {
```

```

60         println!("Received response: {:?}", response_struct);
61     }
62     Err(e) => {
63         eprintln!("Error: {:?}", e);
64     }
65 }
66 }
67
68 ok(())
69 }

```

Listing 44: Client Code

## 18 Dockerfile

### 18.1 Install

#### [Install Docker Desktop](#)

```
1 brew install docker
2 docker --version
```

Listing 45: Install Docker

### 18.2

```
1 open ~/.docker/config.json
2 # make these edits
3 {
4     "auths": {
5         "https://index.docker.io/v2/": {}
6     },
7     "credsStore": "osxkeychain"
8 }
9
10 export PATH="/Applications/Docker.app/Contents/Resources/bin/docker:${PATH}" # add path
11 sudo ln -s
12     ↪ /Applications/Docker.app/Contents/Resources/bin/docker-credential-osxkeychain
13     ↪ /usr/local/bin/docker-credential-osxkeychain # add symbolic link
14
15 docker-credential-osxkeychain version # verify
```

Listing 46: Visual Code Configuration

### 18.3 Dockerfile

```
1 FROM rust:1.80.1
2 WORKDIR /app
3 COPY Cargo.toml Cargo.lock ./
4 RUN cargo build --release
5 RUN cargo clean
6 COPY src ./src
7 RUN cargo build --release
8 CMD ["/target/release/tpl"]
```

Listing 47: Dockerfile

### 18.4 Docker build & run

```
1 docker build . -t dtpl
2 docker run --rm dtpl # remove the container when it exits. This means that the container
3 # will be deleted immediately after it finishes running and not accumulate.
```

Listing 48: Dockerfile Build & Run

### 18.5 Docker Push

```
1 docker login
2 jganguly          # userid
3 *****@gmail.com # email
4 *****          # password
5
6 docker tag dtpl:latest jganguly/dtpl # update the tag before pushing
7 docker push jganguly/dtpl
```

Listing 49: Docker Push

### 18.6 Docker Commands



```
1 docker images
2 docker rmi $(docker images -q)
3 docker ps -a
4 docker rm container-id
```

Listing 50: Docker Commands

## 19 Kubernetes

### 19.1

```
1 brew install minikube
2 brew unlink minikube
3 brew link minikube
```

Listing 51: Install Kubernetes

### 19.2 Start & Stop Kubernetes Cluster

```
1 minikube start --driver=docker
2 minikube status
3 kubectl config view
4
5 kubectl version
6 kubectl get node
7
8 minikube stop
```

Listing 52: Start & Stop Kubernetes Cluster

### 19.3 Kubernetes yaml file

```
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4   name: dtpl
5   labels:
6     app: dtpl
7 spec:
8   replicas: 1
9   selector:
10    matchLabels:
11      app: dtpl
12   template:
13     metadata:
14       labels:
15         app: dtpl
16     spec:
17       containers:
18       - name: app
19         image: jganguly/dtpl
20         resources:
21           requests:
22             memory: "64Mi" # Request 64 megabytes of memory
23             cpu: "0.5" # Request 0.5 CPU cores
24           limits:
25             memory: "128Mi" # Limit memory usage to 128 megabytes
26             cpu: "1" # Limit CPU usage to 1 CPU core
27       env:
28       - name: PING_URLS
29         value: https://google.com
```

Listing 53: Kubernetes yaml file

### 19.4 Kubectl commands

```
1 # apply yaml
2 kubectl apply -f dtpl.yaml
3
4 # log data
5 kubectl get pod
```

```

6 kubectl logs --tail=5 dtpl-66b6fcfbf4-2rr5k # the service name is appended with pod id
7
8 kubectl apply -f
  ↪ https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml
9
10 # metrics
11 kubectl top node
12 kubectl top pod
13 kubectl top pod --all-namespaces

```

Listing 54: Kubectl commands

[Get CPU and Memory Usage of NODES and PODS - Kubectl | K8s](#)

## 19.5 Plugin Manager for kubectl

**Krew** is a plugin manager for **kubectl** command-line utility. [Install krew](#)

```

1 kubectl krew version # verify
2 kubectl krew install resource-capacity # Install resource-capacity plugin using Krew
  ↪ and Kubectl
3 kubectl krew list
4
5 kubectl resource-capacity --sort cpu.util --util --pods --containers

```

Listing 55: Plugin Manager for kubectl

## 20 Kubernetes Definitions

- ① A **container** is a lightweight, standalone, and executable package that includes everything needed to run a piece of software, including the code, runtime, system tools, libraries, and settings.
- ② A **pod** is the smallest deployable unit in Kubernetes and represents a single instance of a running process in a cluster. It can contain one or more containers.\*\* In Kubernetes, containers are defined within the context of a Pod, which is the smallest deployable unit in Kubernetes.

To define a container in Kubernetes, you typically create a Pod specification in a YAML file, and within that specification, you specify the container(s) you want to run. Example:

```

1 apiVersion: v1
2 kind: Pod
3 metadata:
4   name: my-pod
5 spec:
6   containers:
7   - name: my-container
8     image: nginx:latest

```

Listing 56: Yaml

In Kubernetes, a **namespace** is a logical and virtual cluster within a physical Kubernetes cluster. It is a way to partition and isolate resources within the cluster, and it provides a scope for names and objects within the cluster.

```

1 kubectl resource-capacity -n kube-system -p -c
2 # or
3 kubectl resource-capacity -n kube-system --pods --containers
4
5 kubectl resource-capacity --node-labels node.kubernetes.io/instance-type=t3a.large

```

Listing 57: Namespace