The Rust Cookbook

Author: Jaideep Ganguly, Sc.D. (MIT)
ganguly.jaideep@gmail.com | Github | LinkedIn

Last updated: 2024-09-17 09:30:11+05:30

Contents

Contents **2**

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 . . . 13
8 if else & iteration . . . 13
9 Data Structures & Traits . . . 14
9.1 Tuple . . . 14
9.2 struct . . . 15
9.3 Trait . . . 15
9.4 trait bound . . . 16
9.5 to_owned() and clone() . . . 16
10 Enum . . . 17
10.1 Match . . . 17
10.2 Result { OK, Err } . . . 17
10.3 ResultErrStr { OK, Err } . . . 18
10.4 Option { Some, None } . . . 18
10.5 if let . . . 19
10.6 Unwrap . . . 19
11 HashMap . . . 20
12 Generic . . . 20
13 Smart Pointers, Deref & Drop . . . 21
13.1 Smart Pointers . . . 21
13.2 Deref . . . 21
13.3 Drop . . . 22
13.4 dyn . . . 22
13.5 tokio . . . 23

14 Async . . . 24
14.1 Long Running Functions . . . 24
14.2 Arc, Mutex & Lock . . . 25
14.2.1 Arc,Mutex,Lock with HashMap . . . 25
14.2.2 Arc,Mutex,Lock with Struct . . . 27
14.3 Message Passing . . . 29
15 PostgreSQL . . . 30
16 IO . . . 31
17 gRPC . . . 33
17.1 Server . . . 33
17.2 Client . . . 35
18 Dockerfile . . . 36
18.1 Install . . . 36
18.2 . . . 36
18.3 Dockerfile . . . 36
18.4 Docker build & run . . . 36
18.5 Docker Push . . . 36
18.6 Docker Commands . . . 36
19 Kubernetes . . . 37
19.1 . . . 37
19.2 Start & Stop Kubernetes Cluster . . . 37
19.3 Kubernetes yaml file . . . 37
19.4 Kubectl commands . . . 37
19.5 Plugin Manager for kubectl . . . 38
20 Kubernetes Definitions . . . 38

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 . . . 13
16 if else . . . 13
17 iter . . . 13
18 Tuple . . . 14
19 Struct . . . 15
20 Trait . . . 15
21 Trait bound . . . 16
22 to_owned() & clone() . . . 16
23 Enum & Match . . . 17
24 Result . . . 17
25 ResultErrStr . . . 18
26 Option . . . 18
27 if let . . . 19
28 unwrap . . . 19
29 HashMap . . . 20
30 Generic . . . 20
31 Box . . . 21
32 Deref . . . 21
33 Drop . . . 22
34 dyn . . . 22
35 tokio . . . 23
36 Long Running functions . . . 24
37 Mutex on HashMap . . . 25
38 Mutex on Struct . . . 27
39 Message Passing . . . 29
40 PostgreSQL . . . 30
41 dyn . . . 31
42 Server Code . . . 33
43 Client Code . . . 35
44 Install Docker . . . 36
45 Visual Code Configuration . . . 36
46 Dockerfile . . . 36
47 Dockerfile Build & Run . . . 36

48 Docker Push 36
49 Docker Commands 36
50 Install Kubernetes 37
51 Start & Stop Kubernetes Cluster 37
52 Kubernetes yaml file 37
53 Kubectl commands 37
54 Plugin Manager for kubectl 38
55 Yaml 38
56 Namespace 38

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

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
rustup update # Update
rustup self uninstall # Uninstall
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

```
cargo new tpl # creates a new directory named tpl
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.

```
[package]
name = "tpl"
version = "0.1.0"
edition = "2021"

[dependencies]
```

3.2.1 Build & Release

```
cd tpl
cargo build # debug
cargo build --release
./tpl
```

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

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

3.3 Dependencies

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

3.4 Cargo.toml - an example

```toml
[package]
name = "tpl"    # template
version = "0.1.0"
edition = "2021"

# See more keys and their definitions at https://doc.rust-lang.org/cargo/reference/manifest.html

[dependencies]
lazy_static = "1" ✅
futures = "0.3.30" ✅
serde_json = "1.0.128" ✅
serde = {version = "1.0", features = ["derive"]} ✅
tokio = {version = "1.40", features = ["full"]} ✅
mysql = "25.0.1" ✅
postgres = "0.19" ✅
chrono = "0.4" ✅
psutil = "3.3.0" ✅
tokio-postgres = "0.7" ✅

[[bin]]
name = "tpl"
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

```
cargo install cargo-edit # First install cargo-edit.
```

```
cargo upgrade # upgrade all dependencies
cargo upgrade serde # upgrade a specific dependency
cargo upgrade --major-versions # might include breaking changes
cargo upgrade --incompatible --verbose # update to latest, may break
cargo install cargo-outdated  # first install cargo-outdated
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."

```
cargo login <TOKEN>
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`


```rust
pub mod sub_dir;
pub mod file1;
```

Listing 1: src/mod_example_dir/mod.rs

```rust
pub fn test1a() {
    println!("file1.test1a");
}

pub fn test1b() {
    println!("file1.test1b");
}
```

Listing 2: src/mod_example_dir/file1.rs

```rust
pub mod file2;
```

Listing 3: src/mod_example_dir/sub_dir/mod.rs

```rust
pub fn test2() {
    println!("file2.test2");
}
```

Listing 4: src/mod_example_dir/sub_dir/file2.rs

```rust
mod mod_example_dir;
use mod_example_dir::file1::test1a;

fn main() {
    test1a(); // use specifies the path to test1a()
    mod_example_dir::file1::test1b();         // use has not specified the path
    mod_example_dir::sub_dir::file2::test2(); // use has not specified the path
}
```

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.

```rust
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.

```rust
fn main() {
    let x = 50;
    let mut s = String::from("hello");
}
```

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.

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

Listing 8: Ownership

5.4 Borrow

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

```rust
fn main() {
    let s = String::from("hello");
    let len = calc_len(&s); // Pass an immutable reference to ‘calc_len‘
    println!("The length of {} is {}", s, len);
}

fn calc_len(s: &str) -> usize {
    s.len() // Return the length of the string slice
}
```

Listing 9: Borrow

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

```rust

fn main() {
    let mut s = String::from("hello");
    let s2: &mut String = &mut s; // Borrow a mutable ref to ’s’, must be initialized

    let len = calc_len(&s2); // Pass an immutable reference to ‘calc_len‘
    println!("The length of ’{}’ is {}", s2, len);
}

fn calc_len(s: &str) -> usize {
    s.len() // Return the length of the string slice
}
```

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.

```rust
fn longest<’a>(s1: &’a str, s2: &’a str) -> &’a str {
    if s1.len() > s2.len() {
        s1
    } else {
        s2
    }
}

fn main() {
    let string1 = String::from("hello");
    let string2 = String::from("world!");

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

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

```rust
fn main() {
    let arr1 = [1, 2, 3, 4, 5]; // array. Must have a known length,
    println!("{:?}",arr1);      // all elements must be initialized.

    let arr2:[i32; 3] = [1; 3]; // create an array of length 3, all elements set to 1
    println!("{:?}",arr2);

    let slice = &arr1[1 .. 4];  // create a slice, the last index not included
    println!("{:?}",slice);     // excludes arr1[3]
}
```

Listing 12: Array & Slice

6.2 Closure

```rust
fn main() {
    let y = 3;
    let add = |x| {
        x + y
    };

    let result = add(5);
    println!("Result: {}", result);
    let result = receive_closure(add, 6); // pass the closure and i32
    println!("Result: {}", result);
}

fn receive_closure<F>(f: F, x: i32) -> i32 // arguments are the closure and i32
    where F: Fn(i32) -> i32 { // a specific closure signature
        f(x) // as i32
    }
```

Listing 13: Closure

6.3 Vector

```rust
fn main() {
    let mut v = vec!["Hello","how","are","you"]; // create a vector
    v.push("Sir");        // push
    v.pop();              // pop
    v.insert(4, "Madam"); // insert a value at a specific index
    v.remove(0);          // remove a value at a specific index
    let index = v.iter().position(|x| x == &"Madam" ).unwrap(); // find index
    v.remove(index);

    let result = v.iter().enumerate().find(|(_, &x)| x.eq("you"));  // find Tuple
    if let Some(value) = result {
        let (i, s) = value;
        println!("position:{:?} value:{:?}", i,s);
    }
    for i in &v { // iterate, &v because of move due to implicit call to .into_iter()
        println!("{}", i);
    }
    for (i, elem) in v.iter().enumerate() { // iterate using enumerate
        println!("Element at position {}: {:?}", i, elem);
    }
}
```

Listing 14: Vector

7 String

```rust
fn main() {
    let mut s = String::from("Jaideep");
    s = s.replace("Jaideep","Mone");    // not in place, returns new String
    println!("replaced = {:?}", s);

    let mut litstr = "Ganguly";
    s = litstr.to_string();             // convert literal string to String
    litstr = s.as_str();                // convert to literal
    println!("literal = {:?}", s);

    s.push('G');              // append a char
    s.push_str("anguly");     // append a slice
    println!("len = {}",s.len());  // length
    println!("trimmed {}",s.trim()); // remove leading and trailing spaces
    let s = s.clear();        // clear

    let s = "Jaideep Ganguly";
    let v: Vec<&str> = s.split_whitespace().collect(); // split by whitesp.
    for token in v {
        println!("{}",token);
    }

    let s = "Jaideep.Ganguly";  // split
    let x = s.split(".");
    for token in x {
        println!("{}",token);
    }

    let x = s.chars();          // chars
    for token in x {
        println!("{}",token);
    }
    let mut s1 = String::from("Jaideep");   // concatenation
    let s2 = String::from("Ganguly");
    let mut s1 = s1 + &s2; // not s1 + s2 because of ownership rules

    let s1 = String::from("Jaideep");   // concatenation
    let s2 = String::from("Ganguly");

    let mut s3 = format!("{} - {}",s1,s2);
    s3 = s.to_uppercase();               // uppercase
    s3 = s.to_lowercase();               // lowercase
}
```

Listing 15: String

8 if else & iteration

```rust
fn main() {
    let x = 5;
    if x > 10 {
        println!("x is greater than 10");
    } else if x > 5 {
        println!("x is greater than 5");
    } else {
        println!("x is less than or equal to 5");
    }
}
```

Listing 16: if else

```rust
fn main() {
    for i in (0..10).step_by(2){
        println!("i: {}", i);
    }

    let mut count = 0;
    while count < 5 {
        println!("Count: {}", count);
        count += 1;
    }

    let mut is_running = true;
    loop {  // infinite loop
        if !is_running {
            break;
        }
        // Do something here, based on that set is_running = false
    }

    let arr2:[i32; 3] = [1; 3]; // create an array of length 3, all elements set to 1
    for item in arr2.iter().enumerate() {
        let (i,x):(usize,&i32) = item;
        println!("array[{i}] = {x}");
    }
}
```

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.

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

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.

```rust
#[derive(Debug)]
pub struct Rect<'a> {
    pub id: &'a str,    // string literals need a lifetime parameter unlike int
    pub width: i32,     // need the ',' unlike Golang struct
    pub length: i32
}

impl<'a> Rect<'a> {
    pub fn new(id: &'a str, width: i32, length: i32) -> Rect<'a> { // Constructor like fn
        Rect { id, width, length }                  // 'Self' will work, but it doesn't
    }                                               // accept parameters such as lifetime
    pub fn area(&self) -> i32 {
        self.width * self.length
    }
    pub fn volume(&self, height: i32) -> i32 {
        self.area()*height
    }
}

fn main() {
    let r1 = Rect {id:"id100",width:10, length:20}; // constructor
    let r2 = Rect::new("id100", 10,25);  // fn used as constructor, any name of fn is ok
    println!("{:?} {:?} {:?} {:?}",r1.width, r1.length, r2.width, r2.length);
    println!("Area = {} Volume = {}",r1.area(),r1.volume(10));
}
```

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.

```rust
pub struct Herbivore;
pub struct Carnivore;

pub trait TrAnimal {
    fn eat(&self, _n:i32);  // signature
}
impl TrAnimal for Herbivore{
    fn eat(&self, n:i32) {  // specific implementation
        println!("I eat plants");
    }
}
impl TrAnimal for Carnivore { // specific implementation
    fn eat(&self, n:i32) {
        println!("{} {}","I eat meat",n);
    }
}
fn main() {
    let h = Herbivore;
    h.eat(3);
    let c = Carnivore;
    c.eat(4);
}
```

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**.

```rust
pub trait TrActivity {
    fn fly(&self);
}

#[derive(Debug)]
pub struct Eagle;

impl TrActivity for Eagle {
    fn fly(&self) {
        println!("{:?} is flying",&self);
    }
}

pub fn activity<T: TrActivity + std::fmt::Debug>(bird: T) {
    println!("I fly as {:?}",bird);
}

fn main() {
    let eagle = Eagle;
    eagle.fly();
    activity(eagle);
}
```

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.

```rust
fn main() {
    let s = "Hello, world!";
    let s1 = s.to_owned(); // Creates a new owned String
    let s2 = s.clone(); // Also creates a new owned String

    let v = vec![1, 2, 3];
    let v1 = v.to_owned(); // Creates a new owned vector
    let v2 = v.clone(); // Also creates a new owned vector
    println!("{:?} {:?}",v1,v2);
}
```

Listing 22: to_owned() & clone()

10 Enum

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

10.1 Match

```rust
enum Shape {
    Rectangle { width: u32, height: u32 },
    Circle { radius: f64 },
    Square(u32)
}

fn main() {
    let mut shape: Shape;
    shape  = Shape::Rectangle { width: 5, height: 10 };

    let mut w: u32=0;
    let mut h: u32=0;
    let mut r: f64=0.0;
    let mut s: u32=0;

    match shape {
        Shape::Rectangle { width, height } => {
            println!("Rectangle: width = {}, height = {}", width, height);
            w = width;
            h = height;
        },
        Shape::Circle { radius } => {
            println!("Circle: radius = {}", radius);
            r = radius;
        },
        Shape::Square(side) => {
            println!("Square: side = {}", side);
            s = side;
        },
    }
    println!("w={:?} h={:?}",w,h);
}
```

Listing 23: Enum & Match

10.2 Result { OK, Err }

```rust
use std::result::Result;

fn main() {
    let result = get_result("Hi".to_string());
    println!("{:?}",result);

    match result {
        Ok(value) => {
            // Use the extracted value
            println!("Matched value: {}", value);
        }
        Err(error) => {
            // Handle the error case
            println!("No Match: {}", error);
        }
    }
}


```

```rust
fn get_result(s: String) -> Result<String,String> {
    if s.eq("Hello") {
        Ok(s)
    } else {
        Err(String::from("not Hello")) // Err(String::from("not Hello"))
    }
}
```

Listing 24: Result

10.3 ResultErrStr { OK, Err }

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

```rust
type ResultErrStr<T> = std::result::Result<T, &'static str>;

fn get_result(s: String) -> ResultErrStr<String> {
    if s.eq("Hello") {
        Ok(s)
    } else {
        Err("not Hello")    // literal, not String
    }
}

fn main() {
    let s = String::from("Hi");
    let result = get_result(s);
    println!("{:?}",result );

    match result {
        Ok(value) => {
            // Use the extracted value
            println!("Matched value: {}", value);
        }
        Err(error) => {
            // Handle the error case
            println!("No Match: {}", error);
        }
    }
}
```

Listing 25: ResultErrStr

10.4 Option { Some, None }

```rust
use std::option::Option;

fn main() {
    let option = get_option(10,0);

    match option {
        Option::Some(value) => {
            // Use the extracted value
            println!("value: {}", value);
        }
        Option::None => {
            // Handle the error case
            println!("None");
        }
    }
}

```

```rust
pub fn get_option(x: i32, y:i32) -> Option<i32> {
    if y != 0  {
        Option::Some(x/y)
    } else {
        Option::None
    }
}
```

Listing 26: 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.

```rust
fn main() {
    let mut s: String = "Hi".to_string();
    let result = get_result(s);

    if let Ok(value) = result {
        s = value.to_owned();
        println!("Extracted value s is: {}", s);
    } else if let Err(error) = result {
        println!("Error: {}", error);
    }
}

fn get_result(s: String) -> Result<String,String> {
    if s.eq("Hello") {
        Ok(s)
    } else {
        Err("not Hello".to_string()) // Err(String::from("not Hello"))
    }
}
```

Listing 27: 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**.

```rust
fn main() {
    let mut s: String = "Hi".to_string();
    let result = get_result(s.clone());
    s = result.unwrap_or_else(|_| "Error".to_string());
    println!("Final value of s is: {}", s);
}

fn get_result(s: String) -> Result<String, String> {
    if s.eq("Hello") {
        Ok(s)
    } else {
        Err("not Hello".to_string()) // Err(String::from("not Hello"))
    }
}
```

Listing 28: unwrap

11 HashMap

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

fn main() {

    // create a hashmap
    let mut hm: HashMap<String,String> = HashMap::new();

    // insert key
    hm.insert("MA".to_string(),"Massachusetts".to_string());
    hm.insert("NY".to_string(),"New York".to_string());
    hm.insert("CA".to_string(),"California".to_string());

    // iterate over key, value
    for (key, val) in hm.iter() {
        println!("key: {} val: {}", key, val);
    }

    // find value corresponding to a key
    let key = "MA".to_string();
    match hm.get(&key) {
        Some(value) => println!("The value for key '{}' is: {}", key, value),
        None => println!("Key '{}' not found in the HashMap.", key),
    }

    // remove a key (and value)
    hm.remove("CA");
    for (key, val) in hm.iter() {
        println!("key: {} val: {}", key, val);
    }

    // length of hashmap
    println!("{:?}", hm.len());

}
```

Listing 29: HashMap

12 Generic

```rust

fn main() {
    struct Data<T> {
        value: T,
    }

    let t:Data<i32> = Data{value:350};                    // i32
    println!("value is :{}",t.value);

    let t2:Data<String> = Data{value:"Tom".to_string()};    // String
    println!("value is :{}",t2.value);
}
```

Listing 30: 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.

```rust
fn main() {
    let x = Box::new(100);
    println!("x = {}", x);
}
```

Listing 31: 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.

```rust
use std::ops::Deref;

fn main() {
    let person = MyBox( Person { age:50, name:"Ram".to_string() } );
    println!("{} {}",person.age,person.name); // Use deref to access fields of boxed Person
    person.greet();
}

struct Person {
    age: u32,
    name: String,
}

impl Person {
    fn greet(&self) {
        println!("Hello, my name is {} and I'm {} years old.", self.name, self.age);
    }
}

```

```rust
struct MyBox<T>(T); // Create a wrapper struct that will implement Deref

impl<T> Deref for MyBox<T> {
    type Target = T;

    fn deref(&self) -> &Self::Target {
        &self.0
    }
}
```

Listing 32: 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.

```rust
struct Resource {
    name: String,
}

impl Drop for Resource {
    fn drop(&mut self) {
        println!("Dropping resource: {}", self.name);
    }
}

fn main() {
    {
        let res1 = Resource {
            name: String::from("Resource 1"),
        };
        let res2 = Resource {
            name: String::from("Resource 2"),
        };
        println!("Resources created.");
    } // res1 and res2 go out of scope here, triggering the drop method
    // At this point, the drop method for res1 and res2 has been called.
}
```

Listing 33: 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.

```rust
use std::error::Error;

fn main() -> Result<(), Box<dyn Error>> {
    let result = get_result();

    match result {
        Ok(_) => println!("Success!"),
        Err(e) => println!("Error: {}", e),
    }
    Ok(())
```

```rust
}

fn get_result() -> Result<(), Box<dyn Error>> {
    let result: Result<(), Box<dyn Error>> = Ok(());
    // let result: Result<(), Box<dyn Error>> = Err("An error occurred".into());
    result
}
```

Listing 34: 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.

```rust
use std::error::Error;

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {

}
```

Listing 35: tokio

14 Async

14.1 Long Running Functions

```rust
use std::error::Error;
use std::thread;
use tokio::time::{sleep, Duration, timeout, Instant};

pub async fn long_running_fn_1(x: &mut i32) -> i32 {
    sleep(Duration::from_secs(10)).await; // don't use thread::sleep
    *x = *x + 1;
    println!("Task 1 completed successfully with output: {}", *x);
    return (*x);
}

pub async fn long_running_fn_2() -> i32 {
    sleep(Duration::from_secs(1)).await;
    println!("Task 2 completed successfully with output: {}", 42);
    return(42);
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn Error>> {
    let mut x = 100;

    let task1 = tokio::spawn(async move { long_running_fn_1(&mut x).await }); // spawn
    let task2 = tokio::spawn(async move { long_running_fn_2().await });       // spawn

    let timeout_duration1 = Duration::from_secs(15); // timeout
    let timeout_duration2 = Duration::from_secs(5);  // timeout

    let result1 = tokio::time::timeout(timeout_duration1, task1);
    let result2 = tokio::time::timeout(timeout_duration1, task2);

    let (result1, result2) = tokio::join!(result1, result2); // run concurrently & join

    match result1 { // Handle results for task1
        Ok(task_result1) => match task_result1 {
            Ok(output) => println!("Task 1 success with output: {}", output),
            Err(error) => println!("Task 1 error during execution: {:?}", error),
        },
        Err(_) => println!("Task 1 timedout after {} sec", timeout_duration1.as_secs()),
    }

    match result2 { // Handle results for task2
        Ok(task_result2) => match task_result2 {
            Ok(output) => println!("Task 2 success with output: {}", output),
            Err(error) => println!("Task 2 error during execution: {:?}", error),
        },
        Err(_) => println!("Task 2 timedout after {} sec", timeout_duration2.as_secs()),
    }

    Ok(())
}
```

Listing 36: 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 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

```rust
use tokio::sync::Mutex;
use std::collections::HashMap;
use std::sync::Arc;
use std::time::Duration;

// future generated needs to be safely sent across threads, Send + Sync
async fn long_running_fn_1(x: &mut i32) -> Result<String, Box<dyn std::error::Error + Send + Sync>> {
    tokio::time::sleep(Duration::from_secs(10)).await; // Simulate long-running computation
    *x += 1;
    Ok(format!("{}", x))
}

// future generated needs to be safely sent across threads, Send + Sync
async fn long_running_fn_2() -> Result<String, Box<dyn std::error::Error + Send + Sync>> {
    tokio::time::sleep(Duration::from_secs(2)).await; // long-running computation
    Ok("".to_string())
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let mut x = 100;

    let shared_map = Arc::new(Mutex::new(HashMap::new())); // clone below does not create
    let shared_map_clone1 = Arc::clone(&shared_map);       // deep copy; it increments
    let shared_map_clone2 = Arc::clone(&shared_map);       // ref count of shared object

    let timeout_duration1 = Duration::from_secs(30);
    let timeout_duration2 = Duration::from_secs(1);

    let task1 = tokio::spawn(async move { // Task 1
        let mut x = x; // x is now owned by this task
        let result = long_running_fn_1(&mut x).await;
        let mut shared_map = shared_map_clone1.lock().await;

```

```
        match result {
            Ok(output) => {
                shared_map.insert("task1".to_string(), output);
                // println!("{:?}",shared_map);
                println!("Task 1 inserted result into map");
            },
            Err(error) => {
                shared_map.insert("task1".to_string(), format!("Error: {:?}", error));
                println!("Task 1 encountered an error: {:?}", error);
            },
        }
    });

    let task2 = tokio::spawn(async move { // Task 2
        let result = long_running_fn_2().await;
        let mut shared_map = shared_map_clone2.lock().await;
        match result {
            Ok(output) => {
                shared_map.insert("task2".to_string(), output);
                // println!("{:?}",shared_map);
                println!("Task 2 inserted result into map");
            },
            Err(error) => {
                shared_map.insert("task2".to_string(), format!("Error: {:?}", error));
                println!("Task 2 encountered an error: {:?}", error);
            },
        }
    });

    // Apply timeouts
    let result1 = tokio::time::timeout(timeout_duration1, task1);
    let result2 = tokio::time::timeout(timeout_duration2, task2);

    // Use join to concurrently wait for both results
    let (result1, result2) = tokio::join!(result1, result2);

    match result1 { // Handle task 1 result
        Ok(Ok(_)) => println!("Task 1 completed"),
        Ok(Err(error)) => println!("Task 1 error during execution: {:?}", error),
        Err(_) => println!("Task 1 timed out after {} sec", timeout_duration1.as_secs()),
    }

    match result2 { // Handle task 2 result
        Ok(Ok(_)) => println!("Task 2 completed"),
        Ok(Err(error)) => println!("Task 2 error during execution: {:?}", error),
        Err(_) => println!("Task 2 timed out after {} sec", timeout_duration2.as_secs()),
    }

    // Print the shared map contents
    let shared_map = shared_map.lock().await; // Lock the shared map here
    if shared_map.is_empty() {
        println!("Map is empty");
    } else {
        for (key, value) in shared_map.iter() {
            println!("{}: {}", key, value);
        }
    }
    Ok(())
}
```

Listing 37: Mutex on HashMap

14.2.2 Arc,Mutex,Lock with Struct

```rust
use tokio::sync::Mutex;
use std::sync::Arc;
use std::time::Duration;

// Struct to hold the results from both tasks
struct SharedResults {
    task1_result: Option<String>,
    task2_result: Option<String>,
}

impl SharedResults {
    fn new() -> Self {
        SharedResults {
            task1_result: None,
            task2_result: None,
        }
    }
}

// future generated needs to be safely sent across threads, Send + Sync
async fn long_running_fn_1(x: &mut i32) -> Result<String, Box<dyn std::error::Error + Send + Sync>> {
    tokio::time::sleep(Duration::from_secs(10)).await; // long-running computation
    *x += 1;
    Ok(format!("{}", x))
}

// future generated needs to be safely sent across threads, Send + Sync
async fn long_running_fn_2() -> Result<String, Box<dyn std::error::Error + Send + Sync>> {
    tokio::time::sleep(Duration::from_secs(2)).await; // long-running computation
    Ok("".to_string())
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
    let mut x = 100;

    let shared_results = Arc::new(Mutex::new(SharedResults::new())); // clone not create
    let shared_results_clone1 = Arc::clone(&shared_results); // deep copy; it increments
    let shared_results_clone2 = Arc::clone(&shared_results); // ref count of shared object

    let timeout_duration1 = Duration::from_secs(30);
    let timeout_duration2 = Duration::from_secs(1);

    let task1 = tokio::spawn(async move { // Task 1
        let mut x = x; // x is owned by this task
        let result = long_running_fn_1(&mut x).await;
        let mut shared_results = shared_results_clone1.lock().await;
        match result {
            Ok(output) => {
                shared_results.task1_result = Some(output);
                println!("Task 1 inserted result into shared struct");
            },
            Err(error) => {
                shared_results.task1_result = Some(format!("Error: {:?}", error));
                println!("Task 1 encountered an error: {:?}", error);
            },
        }
    });
```

```rust

    let task2 = tokio::spawn(async move { // Task 2
        let result = long_running_fn_2().await;
        let mut shared_results = shared_results_clone2.lock().await;
        match result {
            Ok(output) => {
                shared_results.task2_result = Some(output);
                println!("Task 2 inserted result into shared struct");
            },
            Err(error) => {
                shared_results.task2_result = Some(format!("Error: {:?}", error));
                println!("Task 2 encountered an error: {:?}", error);
            },
        }
    });

    // Apply timeouts
    let result1 = tokio::time::timeout(timeout_duration1, task1);
    let result2 = tokio::time::timeout(timeout_duration2, task2);

    // Use join to concurrently wait for both results
    let (result1, result2) = tokio::join!(result1, result2);

    match result1 { // Handle task 1 result
        Ok(Ok(_)) => println!("Task 1 completed"),
        Ok(Err(error)) => println!("Task 1 error during execution: {:?}", error),
        Err(_) => println!("Task 1 timed out after {} sec", timeout_duration1.as_secs()),
    }

    match result2 { // Handle task 2 result
        Ok(Ok(_)) => println!("Task 2 completed"),
        Ok(Err(error)) => println!("Task 2 error during execution: {:?}", error),
        Err(_) => println!("Task 2 timed out after {} sec", timeout_duration2.as_secs()),
    }

    // Print the shared struct contents
    let shared_results = shared_results.lock().await; // Lock the shared struct here
    if shared_results.task1_result.is_none() && shared_results.task2_result.is_none() {
        println!("No results in the shared struct");
    } else {
        if let Some(result1) = &shared_results.task1_result {
            println!("Task 1 Result: {}", result1);
        }
        if let Some(result2) = &shared_results.task2_result {
            println!("Task 2 Result: {}", result2);
        }
    }

    println!("Completed");
    Ok(())
}
```

Listing 38: Mutex on Struct

14.3 Message Passing

```rust
use std::process;
use std::sync::mpsc;
use std::thread;
use std::time::Duration;

fn main() {
    // Channel to send and receive messages between concurrent sections of  code; has two halves, a transmitter and a receiver.
    let (tx, rx) = mpsc::channel();     // multiple producer, 1 consumer
    let tx2 = mpsc::Sender::clone(&tx); // clone a second producer

    // spawn a thread, move the transmitter into the closure
    // spawned thread will now own the transmitter
    thread::spawn( move || {
        let vals = vec![
            String::from("Hello"),
            String::from("from"),
            String::from("thread-1"),
        ];

        for val in vals {
            tx.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    thread::spawn( move || { // same comments as above
        let vals = vec![
            String::from("Hi"),
            String::from("there"),
            String::from("thread-2"),
        ];
        for val in vals {
            tx2.send(val).unwrap();
            thread::sleep(Duration::from_secs(1));
        }
    });

    // receive the result, timeout beyond 1 sec
    let result = rx.recv_timeout(Duration::from_millis(1000));
    match result {
        Err(e) => {
            println!("{:?}",e);
            process::exit(0);
        },
        Ok(x) => {
            for received in rx {
                println!("Got: {}", received);
            }
        }
    }
}
```

Listing 39: Message Passing

15 PostgreSQL

```rust
use postgres::{Client, NoTls, Error};
use chrono::prelude::*; // For date and time

#[derive(Debug, PartialEq, Eq)]
struct Tab {
    cat: String,
    tsk: String,
}

fn main() -> Result<(), Error> {
    let connection_string = "postgresql://imsrole:imspass@localhost:5432/postgres";

    let mut client = Client::connect(connection_string, NoTls)?;

    // SELECT query
    let selected_tab = client.query(
        "SELECT cat, tsk FROM ims.tmp",
        &[],
    )?;

    let tabs: Vec<Tab> = selected_tab.iter().map(|row| {
        Tab {
            cat: row.get(0),
            tsk: row.get(1),
        }
    }).collect();

    for r in &tabs {
        println!("{}: {}", r.cat, r.tsk);
    }

    // INSERT query
    let rows = vec![
        Tab { cat: "Test-11".to_string(), tsk: "Test-12".to_string() },
        Tab { cat: "Test-21".to_string(), tsk: "Test-22".to_string() },
    ];

    for row in rows {
        client.execute(
            "INSERT INTO ims.tmp (cat, tsk) VALUES ($1, $2)",
            &[&row.cat, &row.tsk],
        )?;
    }

    Ok(())
}
```

Listing 40: PostgreSQL

16 IO

```rust
use std::fs;
use std::fs::{File,OpenOptions};
use std::io::{Read,Write};

pub fn fn_stdin() {
    let mut line = String::new();
    println!("Please enter your name:");
    let nb = std::io::stdin().read_line(&mut line).unwrap();
    println!("Hi {}", line);
    println!("# of bytes read , {}", nb);
}

pub fn fn_stdout() {
    let b1 = std::io::stdout().write("Hi ".as_bytes()).unwrap();
    let b2 = std::io::stdout().write(String::from("There\n").as_bytes()).unwrap();
    std::io::stdout().write(format!("#bytes written {}",(b1+b2)).as_bytes()).unwrap();
}

pub fn fn_cmdline() {
    let cmd_line = std::env::args();
    println!("# of command line arguments:{}",cmd_line.len());
    for arg in cmd_line {
        println!("{}",arg);
    }
}

pub fn fn_fileread(filename: &str){
    let mut file = std::fs::File::open(filename).unwrap();
    let mut contents = String::new();
    file.read_to_string(&mut contents).unwrap();
    let _ = file.sync_all(); // Explicitly close the file by syncing the changes
    print!("{}", contents);
}

pub fn fn_filewrite(filename: &str, s: &str) {
    let mut file = std::fs::File::create(filename).expect("Create failed");
    file.write_all(s.as_bytes()).expect("write failed");
    println!("Write completed" );
}

pub fn fn_fileappend(filename: &str, s: &str) {
    let mut file = OpenOptions::new().append(true).open(filename).expect("Not found");
    file.write_all(s.as_bytes()).expect("write failure");
    println!("Appended file {}",filename);
}

pub fn fn_filecopy(src: &str, des: &str) {
    let mut file_inp = std::fs::File::open(src).unwrap();
    let mut file_out = std::fs::File::create(des).unwrap();
    let mut buffer = [0u8; 4096];
    loop {
        let nbytes = file_inp.read(&mut buffer).unwrap();
        file_out.write(&buffer[..nbytes]).unwrap();
        if nbytes < buffer.len() {
            break;
        }
    }
}

```

```rust
pub fn fn_filedelete(filename: &str) {
    fs::remove_file(filename).expect("Unable to delete file");
    println!("Deleted file {}",filename);
}

fn main() {
    fn_stdin();
    fn_stdout();
    fn_cmdline();
    fn_fileread("/Users/jganguly/Per/GitHub/rustlearn/plg/Cargo.toml");
    fn_filewrite("io_write", "Write test");
    fn_fileappend("io_write", "More stuff");
    fn_filecopy("io_write", "io_write_copy");
    fn_filedelete("io_write");
}
```

Listing 41: dyn

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

```rust
use tonic::{transport::Server, Request, Response, Status};
use tokio_stream::wrappers::ReceiverStream;
use tokio::sync::mpsc;
use futures::{Stream, StreamExt}; // for Stream
use std::pin::Pin;

// Import the generated protobuf code
pub mod tpl {
    tonic::include_proto!("tpl"); // Ensure this matches the package name in tpl.pro
}

use tpl::example_service_server::{ExampleService, ExampleServiceServer};
use tpl::{RequestStruct, ResponseStruct};

#[derive(Default)]
pub struct MyExampleService;

#[tonic::async_trait]
impl ExampleService for MyExampleService {
    type SomeMethodStream = Pin<Box<dyn Stream<Item = Result<ResponseStruct, Status>> +
    ↪ Send + Sync>>; // Use futures::Stream

    // Implement the bidirectional streaming SomeMethod
    async fn some_method( &self, request: Request<tonic::Streaming<RequestStruct>>) ->
    ↪ Result<Response<Self::SomeMethodStream>, Status> {

        let mut stream = request.into_inner();
        let (tx, rx) = mpsc::channel(4); // Create channel to send responses, buffer = 4

        tokio::spawn(async move {
            while let Some(req) = stream.next().await {
                match req {
                    Ok(req_struct) => {
                        dbg!(&req_struct); // won't print in  "cargo build --release"

                        let response = ResponseStruct { // Process, create response
                            status_code: 200,
                            message: format!("Processed ID: {}", req_struct.id),
                            result: req_struct.value * 2.0,
                        };

                        if let Err(_) = tx.send(Ok(response)).await {   // Send response
                            eprintln!("Client disconnected");
                            return;
                        }
                    }

                    Err(e) => {
                        eprintln!("Error receiving stream: {:?}", e);
                        return;
                    }
```

```
                }
            }
        });


        let output_stream = ReceiverStream::new(rx); // Return stream of responses to
    ↪ client

        Ok(Response::new(
            Box::pin(output_stream) as Self::SomeMethodStream
            // Box::pin(output_stream) as Self::StreamProcessStream
        ))
    }
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {

    let addr = "[::1]:50051".parse()?;  // Define the address for the serve

    let example_service = MyExampleService::default();

    println!("Server listening on {}", addr);

    // Start the gRPC server
    Server::builder()
        .add_service(ExampleServiceServer::new(example_service))
        .serve(addr)
        .await?;

    Ok(())
}
```

Listing 42: Server Code

17.2 Client

```rust
use tonic::transport::Channel;
use tonic::Request;
use futures::StreamExt; // for StreamExt::next
use futures::stream; // Import for stream::iter

pub mod tpl {
    tonic::include_proto!("tpl"); // Ensure this matches the package name in tpl.pro
}

use tpl::example_service_client::ExampleServiceClient;
use tpl::{RequestStruct, ResponseStruct};

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {

    let channel = Channel::from_static("http://[::1]:50051").connect().await?; // Create a gRPC channel to connect to the server
    let mut client = ExampleServiceClient::new(channel); // Create a client for the ExampleService

    // Create a stream of RequestStruct
    let request_stream = stream::iter(vec![
        RequestStruct { id: 1, name: "Item1".into(), value: 10.0 },
        RequestStruct { id: 2, name: "Item2".into(), value: 20.0 },
    ]);

    // Call the gRPC method
    let response_stream = client.some_method(Request::new(request_stream)).await?;

    // Process the responses
    let mut response_stream = response_stream.into_inner();
    while let Some(response) = response_stream.next().await {
        match response {
            Ok(response_struct) => {
                println!("Received response: {:?}", response_struct);
            }
            Err(e) => {
                eprintln!("Error: {:?}", e);
            }
        }
    }

    Ok(())
}
```

Listing 43: Client Code

18 Dockerfile

18.1 Install

Install Docker Desktop

```
brew install docker
docker --version
```

Listing 44: Install Docker

18.2

```
open ~/.docker/config.json
# make these edits
{
    "auths": {
    "https://index.docker.io/v2/": {}
    },
    "credsStore": "osxkeychain"
}

export PATH="/Applications/Docker.app/Contents/Resources/bin/docker:${PATH}" # add path
sudo ln -s /Applications/Docker.app/Contents/Resources/bin/docker-credential-osxkeychain
    ↪ /usr/local/bin/docker-credential-osxkeychain # add symbolic link

docker-credential-osxkeychain version # verify
```

Listing 45: Visual Code Configuration

18.3 Dockerfile

```
FROM rust:1.80.1
WORKDIR /app
COPY Cargo.toml Cargo.lock ./
RUN cargo build --release
RUN cargo clean
COPY src ./src
RUN cargo build --release
CMD ["./target/release/tpl"]
```

Listing 46: Dockerfile

18.4 Docker build & run

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

Listing 47: Dockerfile Build & Run

18.5 Docker Push

```
docker login
jganguly          # userid
*******@gmail.com # email
*********         # password

docker tag dtpl:latest jganguly/dtpl # update the tag before pushing
docker push jganguly/dtpl
```

Listing 48: Docker Push

18.6 Docker Commands

```
docker images
docker rmi $(docker images -q)
docker ps -a
docker rm container-id
```

Listing 49: Docker Commands

19 Kubernetes

19.1

```
brew install minikube
brew unlink minikube
brew link minikube
```

Listing 50: Install Kubernetes

19.2 Start & Stop Kubernetes Cluster

```
minikube start --driver=docker
minikube status
kubectl config view

kubectl version
kubectl get node

minikube stop
```

Listing 51: Start & Stop Kubernetes Cluster

19.3 Kubernetes yaml file

```
apiVersion: apps/v1
kind: Deployment
metadata:
    name: dtpl
    labels:
    app: dtpl
spec:
    replicas: 1
    selector:
    matchLabels:
        app: dtpl
    template:
    metadata:
        labels:
        app: dtpl
    spec:
        containers:
        - name: app
        image: jganguly/dtpl
        resources:
            requests:
            memory: "64Mi"  # Request 64 megabytes of memory
            cpu: "0.5"       # Request 0.5 CPU cores
            limits:
            memory: "128Mi"  # Limit memory usage to 128 megabytes
            cpu: "1"           # Limit CPU usage to 1 CPU core
        env:
            - name: PING_URLS
            value: https://google.com
```

Listing 52: Kubernetes yaml file

19.4 Kubectl commands

```
# apply yaml
kubectl apply -f dtpl.yaml

# log data
kubectl get pod
kubectl logs --tail=5 dtpl-66b6fcfbf4-2rr5k # the service name is appended with pod id

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 53: 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 54: 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 55: 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 56: Namespace