Contents

Rust Tour

Linux

(The command below will make sure rust is installed)

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
```

Full toolchain is installed which includes rustc, cargo, rust-std.

This script installs rustup, the Rust toolchain installer, which in turn installs:

- rustc – the Rust compiler
- cargo – Rust's package manager and build tool
- rust-std – the standard library

Toolchain Management with rustup:

If you've installed rustup in the past, you can update your installation by running:

```
rust update
```

Configuring the PATH:

```
export PATH="$HOME/.cargo/bin:$PATH"
```

Uninstall Rust:

```
rustup self uninstall
```

Windows

Go to the following page https://forge.rust-lang.org/infra/other-installation-methods.html and download the msi package. These installers come with rustc, cargo, rustdoc

For your reference:

- x86_64-pc-windows-msvc : uses Microsoft Visual C++ (MSVC) ABI, MSVCRT (Microsoft C Runtime Library), Full compatibility with Windows apps ,Optimized for Windows and Native Windows apps, Visual Studio projects
- aarch64-pc-windows-msvc: for ARM64 CPU's
- x86_64-pc-windows-gnu: uses GNU (MinGW) ABI, uses libgcc and libstdc++ (GNU libraries) for c runtimes, Requires MinGW installation (separate toolchain) and Portable apps, cross-compiling from Linux

Code Editors

If you are using vscode download here https://code.visualstudio.com/download and install the following extensions: rust-analyzer, even better TOML, crates-io

If you want to use Zed written in Rust click here Download link, you don't have to add any additional plugin. (However, you might need to build it yourself for Windows systems. Hence, either build it yourself before coming to the workshop or just use VSCode.)

Working with Projects

create new project :

```
cargo new <name_of_project>
```

After creation add bevy to the project by adding it to the dependency:

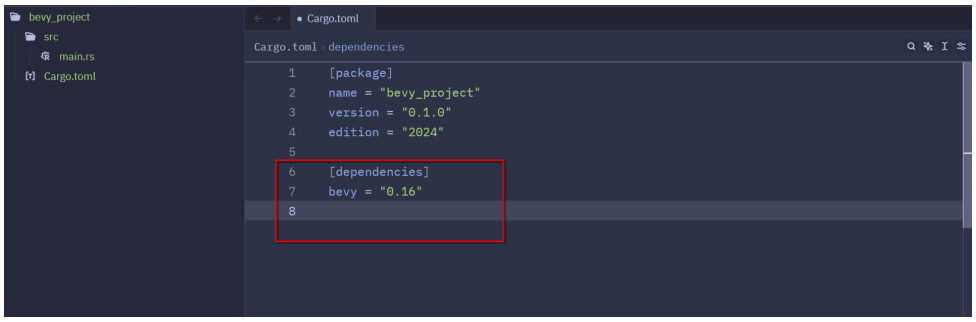

Figure 1: Adding Dependency to Toml

- Before you run make sure the following packages are installed in the system as bevy requires them in order to compile bevy:
  - libudev-dev (libudev-devel for fedora/centos)
  - libx11-dev libasound2-dev libudev-dev libxkbcommon-x11-0
  - For Windows Download & install the following: **Microsoft C++ Build Tools**
- Adding dependency in .toml file as shown in figure 1 above and run the following command:

  ```
  cargo build
  ```
- You may add /target to your .ignore file as rust builds will be created there and you dont need to add them to your repo(optional)
- For further information go to the following link: **Bevy Info**

A Friend in need is a Friend Indeed

Consider Rust compiler as your friend which can help you and pin point the location of every error in your code.

Figure 2: Rustc pointintg error

- as you can see in figure 2 rustc will point out directly and will explain you where exactly the error lies and what fixes you can do.

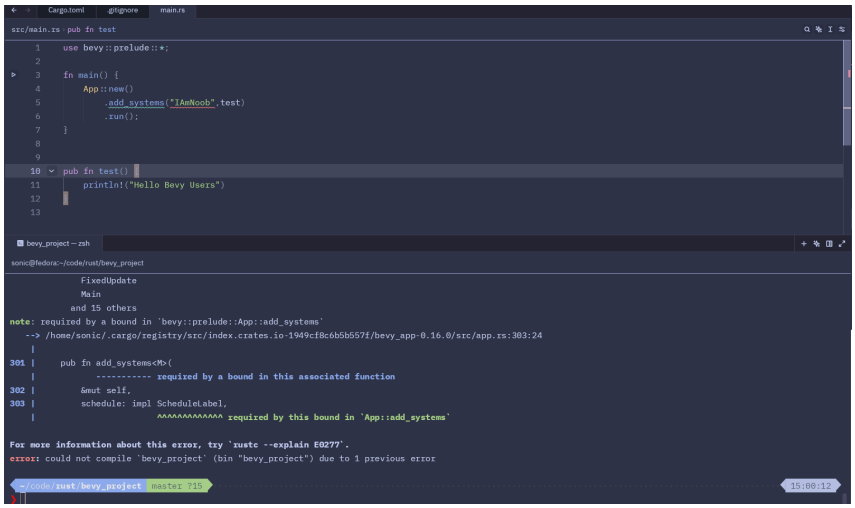

Figure 3: Rustc explainations

- However if you need further more understanding of error rust come up with an extra help and exaplination of error by typing in the command as mentioned in the second last line in the shell of figure 3 which is "`rustc --explain E0277`"

Figure 4: Rustc explainations

- As shown in Figure 4 this is how the further explaination looks like which can help you debug and understand your problem in a better way.

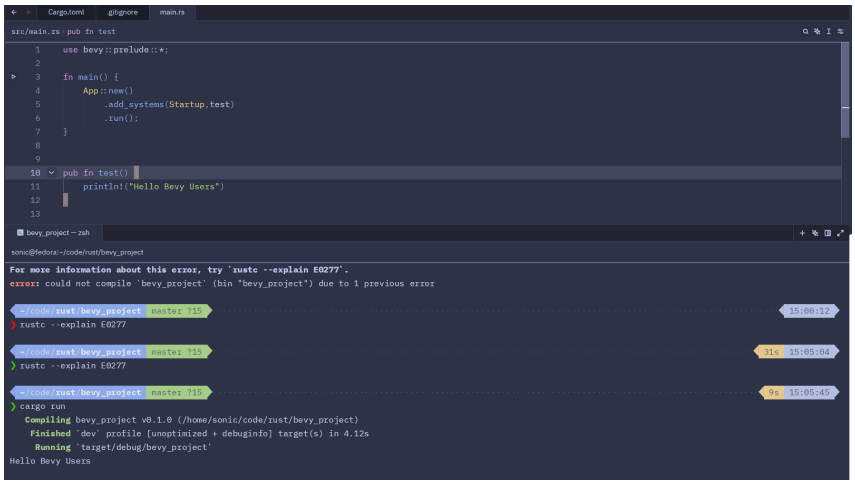

Figure 5: Rustc explainations

- After fixing the error the program was able to compile without any issues as shown in figure 5.

Rust Data Types

Scalar Types

These represent a single value

Type	Description	Example
i8–i128	Signed integers (8 to 128 bits)	`let x: i32 = -42;`
u8–u128	Unsigned integers (8 to 128 bits)	`let x: u64 = 42;`
isize	Signed integer, pointer-sized	`let x: isize = 10;`
usize	Unsigned, pointer-sized	`let x: usize = 100;`
f32	32-bit floating point	`let x: f32 = 3.14;`
f64	64-bit floating point (default)	`let x: f64 = 2.718;`
bool	Boolean	`let is_ready: bool = true;`
char	A Unicode scalar value	`let letter: char = '🦀';`

Str type is a sequence of char type which is in double quotes for example:

“ `let s: &str = "Rust";` ”

Type	Description
&str	String slice (view into a string)
u8–u128	Growable, heap-allocated string

Compound Types

They are Grouping multiple values into one type for example:

Tuple:

```
let tup: (i32, f64, char) = (42, 6.9, 'R');
let (x, y, z) = tup;
```

Array:

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

Custom Types:

```
struct User {
    name: String,
    age: u8,
}
```

Enums:

```
enum Result<T, E> {
    Ok(T),    // holds success value of type T
    Err(E),   // holds error value of type E
}
```

Union(unsafe)

```
union MyUnion {
    i: i32,
    f: f32,
}
```

Warning: unions are a low-level, unsafe feature that let you define a type where all fields share the same memory, like C-style unions. They are rarely used in safe Rust, but they are powerful when you need tight control over memory layout, like in FFI (calling C code) or low-level systems programming

References Types

Type	Description
&T	Shared Reference
&mut T	Mutable reference

Collection Types (from std::collections)

Type	Description
Vec<T>	Growable array (vector)
HashMap<K, V>	Key-value store
HashSet<T>	Unordered set

Never Type

Type	Description
!	Used for functions that never return (panic!, loops, etc.)

Rust Copy and Non-Copy Types

What are Copy Types?

Copy types in Rust are those that can be duplicated with a simple bitwise copy, without requiring ownership transfer. When you assign a value of a Copy type to another variable, Rust performs a simple bitwise copy without invalidating the original variable.

Characteristics of Copy Types

- Fast, because they are duplicated with a bitwise copy.
- No ownership transfer, so the original value is still accessible.
- Commonly used for primitive types and types without heap allocation.

Examples of Copy Types

- Integer types: i32, u32, u8, etc.
- Floating point types: f32, f64
- Boolean: bool
- Character: char
- Arrays and tuples, but only if all elements are also Copy types.

Example Code:

```
fn main() {
    let x: i32 = 42;
    let y = x; // Copy happens here, x is still valid
    println!("x: {}, y: {}", x, y);
}
```

What are Non-Copy Types?

Non-Copy types in Rust are types that involve ownership transfer when assigned to another variable. Instead of being copied, ownership is moved to the new variable, making the original variable invalid.

Characteristics of Non-Copy Types

- Ownership transfer occurs when assigned or passed.
- Cannot be duplicated without explicitly using methods like clone().
- Commonly used for heap-allocated types and complex data structures.

Examples of Non-Copy Types

- Strings: String
- Vectors: Vec
- Boxed values: Box
- Any custom struct that does not explicitly implement Copy.

Example Code:

```
fn main() {
    let s1 = String::from("Hello, Rust!");
    let s2 = s1; // Move occurs, s1 is no longer valid
    // println!("s1: {}", s1); // Error: s1 is moved
    println!("s2: {}", s2);
}
```

Mut vs Non-Mut and Their Copy Behavior

A mut variable is one that can be modified after initialization.A non-mut variable is immutable and cannot be changed after initialization.

How mut Affects Copy Types

A mutable Copy type can be changed, but it is still copied bitwise.

```
fn main() {
    let mut x: i32 = 10;
    let y = x; // Copy occurs
    x = 20;   // x is changed, y remains the same
    println!("x: {}, y: {}", x, y);
}
```

How mut Affects Non-Copy Types

A mutable Non-Copy type can be modified, but ownership rules still apply.

```
fn main() {
    let mut s1 = String::from("Hello");
    let s2 = s1; // Move occurs, s1 is no longer valid
    // s1.push_str(", World!"); // Error: s1 is moved
    println!("s2: {}", s2);
}
```

RUST Ownership & Borrowing rules.

Introduction:

Ownership is one of Rust's unique and most powerful features, designed to ensure memory safety without a garbage collector. It allows Rust to efficiently manage memory through a set of rules.

Ownership Basics

1. Each value in Rust has a single owner.
2. When the owner goes out of scope, the value is dropped.
3. A value can be moved or copied to another owner.

Example:

```
fn main() {
    let s1 = String::from("hello");
    let s2 = s1; // s1 is moved to s2

    // println!("{}", s1); // Error: s1 is no longer valid
    println!("{}", s2);
}
```

Explanation:

- The String type in Rust is not copy-able by default.
- When s1 is assigned to s2, it is moved, making s1 invalid.

Borrowing and References

Borrowing allows a function or variable to temporarily use a value without taking ownership of it.

- Use & to create a reference.
- Use &mut for mutable references.

Example:

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

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

Explanation:

- &mut s allows the function to borrow the string and modify it
- Only one mutable reference is allowed at a time.

Borrowing Rules

- You can have one mutable reference or any number of immutable references, but not both at the same time.
- References must always be valid.

Example:

```
fn main() {
    let mut s = String::from("hello");
    let r1 = &s; // immutable borrow
    let r2 = &s; // another immutable borrow
```

```
    // let r3 = &mut s; // Error: cannot borrow as mutable

    println!("{}, {}", r1, r2);
}
```

Slices - Borrowing Parts of Data

Slices allow borrowing a part of a data structure without taking ownership.

Example:

```
fn main() {
    let s = String::from("hello world");
    let hello = &s[0..5]; // slice
    println!("{}", hello); ; // valid
    println!("{}", s); ; // valid
}
```

RUST Functions

Introduction

Functions are a core building block in Rust, allowing you to organize code into reusable sections. Functions take inputs, perform operations, and may return values.

Defining Functions

A function in Rust is defined using the fn keyword followed by the function name, parameters (if any), and the return type (if any). Functions are declared within a scope (like main) or at the module level.

Example:

```
fn greet(name: &str) -> String {
    format!("Hello, {}!", name)
}

fn main() {
    let message = greet("Alice");
    println!("{}", message);
}
```

Explanation:

- fn is the keyword used to define a function.
- greet is the function name.
- (name: &str) is a parameter of type string slice.
- -> String indicates that the function returns a String type
- format! is a macro that formats text, returning a string.

Parameters and Return Types

- Parameters are defined within parentheses and must specify their type.
- Functions can return values using the -> symbol.
- If a function does not explicitly return a value, it returns the unit type ().

Example with Multiple Parameters:

```
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Statements vs. Expressions

- Rust functions primarily use expressions, which are pieces of code that return a value.
- Statements do not return values (e.g., variable declarations).

Example:

```
fn main() {
    let x = 5;
    let y = { // This is an expression block
        let temp = 3;
        temp + 1
    };
    println!("x = {}, y = {}", x, y);
}
```

Function Documentation

Rust provides a system for documenting functions using doc comments (

Example:

```
/// Adds two numbers together.
///
/// # Arguments
/// * `a` - The first number.
/// * `b` - The second number.
fn add(a: i32, b: i32) -> i32 {
    a + b
}
```

Running "`cargo doc --open`" will compile and open a documentation for your project.

Rust Structs

What is a Struct?

A struct (short for structure) in Rust is a custom data type that lets you group related data together. Structs are like objects in OOP languages, but without methods unless you implement them separately.

```
struct Book {
    title: String,
    author: String,
    pages: u32,
    published: bool,
}
```

Creating Instance

```
let book1 = Book {
    title: String::from("Bevy Workshop"),
    author: String::from("Rust Malaysia"),
    pages: 10,
    published: true,
};
```

In the Rust programming language, String::from() is a function that creates a String (mutable string) from a given &str (string slice). It essentially converts a string literal or a string slice into a String data type, which allows for modification and ownership. For more info checkout struct strings

```
let book2 = Book {
    title: String::from("Bevy workshop"),
    ..book1
};
```

This copies all remaining fields from book1. book1 will no longer be valid if any fields that don't implement Copy (like String) are moved. Which means if we try to access book1.author it will give an error as we already borrowed via "..book1" . However book1.title will still be accessible because we never borrowed it.

Accessing struct fields

```
println!("Title:{}",book.title)
```

Interesting fact: println! is a macro in Rust (note the !), not a function. And "{}" is a placeholder or like a template which will be replaced by the value book1.title.

Tuple Structs

Tuple structs are useful without named fields:

```
struct Color(u8, u8, u8);
let red = Color(255, 0, 0);
```

To access the struct we can do red.0 or red.1.

Unit-Like Stricts

```
struct Marker;
let _m = Marker;
```

A unit-like struct is a struct with no fields, just a name. the underscore is to let the compiler know that i am not going to use this variable to avoid the warning.

Implementing Methods

Implements are use to Implement methods for sturcts or enums using the impl block. This is a way to attach behaviour to your types which is similar to methods in classes in other languages.

```
struct Counter {
    count: u32,
}

impl Counter {
    fn new() -> Self {
        Counter { count: 0 }
    }

    fn increment(&mut self) {
        self.count += 1;
    }

    fn get(&self) -> u32 {
        self.count
    }
}

fn main() {
    let mut c = Counter::new();
    c.increment();
    println!("Count: {}", c.get()); // prints: Count: 1
}
```

Using impl Block with the same name as struct you can associate it to the struct.

Three things to notice here:

- function new() return self and doesn't take any self. This is an associated function (static with no self parameter. And if you notice its being accessed with double colon. These functions are usually used as a constructor , helper functions ,factory methods(return different variants) or name spacing to keep the related logic grouped with a type.
- Method increment() takes a mutable self which means it will modify the existing instance.
- Method get() is a self which will return the current value of the instance.

Cheat Sheet

… You may use this cheat sheet
for your own good.

Basic Types & Variables

Types
bool — Boolean

Unsigned integers
u8, u16, u32, u64, u128

Signed integers
i8, i16, i32, i64, i128

Floating point
f32, f64

Platform-specific
usize: Unsinged integer, Same number of bits as the platform's pointer type.

isize: signed Integer. Same number of bits as the platform's pointer type.

char — Unicode Scalar Value
&str — String slice
String — Owned string

Tuple

```
let coordinates = (82, 64);
let score = ("Team A", 12);
```

Array & Slice

```
let array = [1, 2, 3, 4, 5];
let array2 = [0; 3]; // [0, 0, 0]
let slice = &array[1..3];
```

HashMap

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

let mut newHash = HashMap::new();
newHash.insert(String::from("RustMY"), 100000);
newHash.entry("".to_owned())
    .or_insert(3);
```

Mutability

```
let mut x = 5;
x = 6
```

Struct

```
//Definition
struct User {
  username: String,
  active: bool,
}

//Instantiation
let user1 = User {
  username: String::from("bogdan"),
  active: true,
};

//Tuple Struct
struct Color(i32, i32, i32);
let black = Color(0, 0, 0);
```

Enum

```
//Definition
enum Command {
    Quit,
    Move { x: i32, y: i32 },
    Speak(String),
    ChangeBGColor(i32, i32, i32),
}
//Instatniation
let msg1 = Command::Quit;
let msg2 = Command::Move{ x: 1, y: 2 };
let msg3 = Command::Speak("Rust".to_owned());
let msg4 = Command::ChangeBGColor(0, 0, 0);
```

Constant

```
const MAX_POINTS: u32 = 100_000;
```

Static Variable

```
// Unlike constants static variables are
// stored in a dedicated memory location
// and can be mutated.
static MAJOR_VERSION: u32 = 1;
static mut COUNTER: u32 = 0;
```

Type alias

```
// `NanoSecond` is a new name for `u64`.
type NanoSecond = u64;
```

Control Flow

if & if let

```
let num = Some(22);
if num.is_some() {
println!("number is: {}", num.unwrap());
}
// match pattern and assign variable
if let Some(i) = num {
println!("number is: {}", i);
}
```

loop

```
let mut count = 0;
loop {
count += 1;
if count == 5 {
    break; // Exit loop
  }
}
```

Nested loops & labels

```
outer: loop {
  inner: loop {
    // This breaks the inner loop
    break;
  }
  // This breaks the outer loop
  break 'outer;
}
```

Returning from loops

```
let mut count = 0;
let result = loop {
  count += 1;
  if count == 10 {
    break count;
  }
};
```

while & while let

```
while n < 101 {
  n += 1;
}
let mut optional = Some(0);
while let Some(i) = optional
{
  print!("{}", i);
}
```

for loop

```
for n in 1..101 {
  println!("{}", n);
}
let nms = vec!["Ivn","Han","Nik","Nix","Jef"];
for name in nms.iter() {
  println!("{}", name);
}
```

match

```
let optional = Some(0);
match optional {
    Some(i) => println!("{}", i),
    None => println!("No value.")
}
```

References, Ownership & Borrowing

Ownership rules

1 - Every value in Rust is owned by a single variable, which acts as its “owner.”
2- Ownership is exclusive—only one variable can own a value at a time.
3 - When the owner variable goes out of scope (is no longer used), the value is automatically removed from memory.
4- This system ensures efficient memory management without requiring manual cleanup.

Borrowing rules

1 - At any moment, you can either have a single mutable reference or multiple immutable references to a value.
2 - References must remain valid and never point to invalid or deallocated memory.

Creating references

```
let s1 = String::from("hello world!");
let s1_ref = &s1; // immutable reference
let mut s2 = String::from("hello");
let s2_ref = &mut s2; // mutable reference
s2_ref.push_str(" world!");
```

Copy, Move & Clone

```
// Simple values which implement the Copy trait
are copied by value
let x = 5;
let y = x;
println!("{}", x); // x is still valid
// The string is moved to s2 and s1 is
invalidated
let s1 = String::from("Welcometo Workshop");
let s2 = s1; // Shallow copy a.k.a move
println!("{}", s1); // Error: s1 is invalid
let s1 = String::from("Welcometo Workshop");
let s2 = s1.clone(); // Deep copy
// Valid because s1 isn't moved
println!("{}", s1);
```

Ownership & Functions

```
fn main() {
    let x = 5;
    takes_copy(x); // x is copied by value
    let s = String::from("Welcome to Workshop");
    // s is moved into the function
    takes_ownership(s);
    // return value is moved into s1
    let s1 = gives_ownership();
    let s2 = String::from("RustMY");
    let s3 = takes_and_gives_back(s2);
  }

  fn takes_copy(some_integer: i32) {
    println!("{}", some_integer);
  }

  fn takes_ownership(some_string: String) {
    println!("{}", some_string);
  } // some_string goes out of scope and drop is
called. The backing memory is freed.

  fn gives_ownership() -> String {
    let some_string = String::from("WorkShop");
    some_string
  }

  fn takes_and_gives_back(some_string: String)->
String {
    some_string
  }
```

Pattern Matching

Basics

```
let x = 5;
match x {
// matching literals
    1 => println!("one"),
    // matching multiple patterns
    2 | 3 => println!("two or three"),
    // matching ranges
    4..=9 => println!("within range"),
    // matching named variables
    x => println!("{}", x),
    // default case (ignores value)
    _ => println!("default Case")
}
```

Destructing

```
  struct Point {
    x: i32,
    y: i32,
  }
  let p = Point { x: 0, y: 7 };
  match p {
  Point { x, y: 0 } => {
      println!("{}" , x);
    },
    Point { x, y } => {
      println!("{} {}" , x, y);
    },
  }

  enum Shape {
    Rectangle { width: i32, height: i32 },
    Circle(i32),
  }
  let shape = Shape::Circle(10);
  match shape {
    Shape::Rectangle { x, y } => //...
    Shape::Circle(radius) => //...
  }
```

Ignoring Values

```
struct SemVer(i32, i32, i32);
 let version = SemVer(1, 32, 2);
 match version {
   SemVer(major, _, _) => {
     println!("{}", major);
   }
 }
 let numbers = (2, 4, 8, 16, 32);
 match numbers {
   (first, .., last) => {
     println!("{}, {}", first, last);
   }
 }
```

Match guards

```
let num = Some(4);
match num {
  Some(x) if x < 5 => println!("less than five:
{}", x),
  Some(x) => println!("{}", x),
  None => (),
}
```

Bindings

```
struct User {
    id: i32
  }
let user = User { id: 5 };
match user {
User {
  id: id_variable @ 3..=7,
    } => println!("id: {}", id_variable),
  User { id: 10..=12 } => {
    println!("within range");
  },
  User { id } => println!("id: {}", id),
}
```

Iterators

Usage

```
// Methods that consume iterators
let v1 = vec![1, 2, 3];
let v1_iter = v1.iter();
let total: i32 = v1_iter.sum();
// Methods that produce new iterators
let v1: Vec<i32> = vec![1, 2, 3];
let iter = v1.iter().map(|x| x + 1);
// Turning iterators into a collection
let v1: Vec<i32> = vec![1, 2, 3];
let v2: Vec<_> = v1.iter().map(|x| x +
1).collect();
```

Implementing the Iterator trait

```
struct Counter {
  count: u32,
}
impl Counter {
  fn new() -> Counter {
    Counter { count: 0 }
    }
}
impl Iterator for Counter {
  type Item = u32;
}
fn next(&mut self) -> Option<Self::Item> {
  if self.count < 5 {
    self.count += 1;
    Some(self.count)
  } else {
    None
  }
}
```

Error Handling

Throw unrecoverable error

```
panic!("Critical error! Exiting!");
```

Option enum

```
fn get_user_id(name: &str) -> Option<u32> {
  if database.user_exists(name) {
    return Some(database.get_id(name))
  }
  None
}
```

Result enum

```
fn get_user(id: u32) -> Result<User, Error> {
  if is_logged_in_as(id) {
   return Ok(get_user_object(id))
  }
  Err(Error { msg: "not logged in" })
}
```

? Operator

```
fn get_salary(db: Database, id: i32) ->
Option<u32> {
  Some(db.get_user(id)?.get_job()?.salary)
}

fn connect(db: Database) -> Result<Connection,
Error> {
  // ? works if the return type is Result
  let conn =
db.get_active_instance()?.connect()?;
  Ok(conn)
}
```

Combinators

.map

```
let some_string = Some("RustMy".to_owned());
let some_len = some_string.map(|s| s.len());

struct Error { msg: String }
struct User { name: String }

let string_result: Result<String, Error> =
Ok("Bogdan".to_owned());

let user_result: Result<User, Error> =
string_result.map(|name| {
  User { name }
});
```

.and_then

```
let vec = Some(vec![1, 2, 3]);
let first_element = vec.and_then(
  |vec| vec.into_iter().next()
);
let string_result: Result<&'static str, _>
= Ok("5");
let number_result =
  string_result
  and_then(|s| s.parse::<u32>());
```

Multiple Error Types

Define custom error type

```
type Result<T> = std::result::Result<T,
CustomError>;

#[derive(Debug, Clone)]
struct CustomError;

impl fmt::Display for CustomError {
  fn fmt(&self, f: &mut fmt::Formatter) ->
fmt::Result {
    write!(f, "custom error message")
  }
}
```

Boxing errors

```
use std::error;
type Result<T> =
  std::result::Result<T, Box<dyn
error::Error>>;
```

Iterating Over Errors

Ignore failed items with filter_map()

```
let strings = vec!["RustMY", "22", "7"];
let numbers: Vec<_> = strings
  .into_iter()
  .filter_map(|s| s.parse::<i32>().ok())
  .collect();
```

Failed the entire operation with collect()

```
let strings = vec!["RustMY", "22", "7"];
let numbers: Result<Vec<_>, _> = strings
  .into_iter()
  .map(|s| s.parse::<i32>())
  .collect();
```

Collect all valid values & failures with partition()

```
let strings = vec!["RustMY", "22", "7"];
let (numbers, errors): (Vec<_>, Vec<_>) =
strings
  .into_iter()
  .map(|s| s.parse::<i32>())
  .partition(Result::is_ok);

let numbers: Vec<_> = numbers
  .into_iter()
  .map(Result::unwrap)
  .collect();

let errors: Vec<_> = errors
  .into_iter()
  .map(Result::unwrap_err)
  .collect();
```

Using generics

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

impl<T, U> Point<T, U> {
  fn mixup<V, W>(self, other: Point<V, W>) ->
Point<T, W> {
      Point {
          x: self.x,
          y: other.y,
      }
  }
}
```

Defining traits

```
trait Animal {
    fn new(name: &'static str) -> Self;
    fn noise(&self) -> &'static str {
        ""
    }
}

struct Dog {
    name: &'static str,
}

impl Dog {
    fn fetch() {
        // ... implementation
    }
}

impl Animal for Dog {
    fn new(name: &'static str) -> Dog {
        Dog { name }
    }
    fn noise(&self) -> &'static str {
        "woof!"
    }
}
```

Default Implementations with Derive

```
// A tuple struct that can be printed
#[derive(Debug)]
struct Inches(i32);
```

Trait Bounds


```
fn largest<T: PartialOrd + Copy>(list: &[T]) ->
T {
    let mut largest = list[0];
    for &item in list {
        if item > largest {
            largest = item;
        }
    }
    largest
}
```

impl trait

```
fn m_function(y: i32) -> impl Fn(i32) -> i32 {
  let closure = move |x: i32| { x + y };
  closure
}
```

Trait Objects

```
pub struct Screen {
  pub components: Vec<Box<dyn Draw>>,
}
```

Opertor Overloading

```
use std::ops::Add;

#[derive(Debug, Copy, Clone, PartialEq)]
struct Point {
    x: i32,
    y: i32,
}
impl Add for Point {
    type Output = Point;
    fn add(self, other: Point) -> Point {
        Point {
            x: self.x + other.x,
            y: self.y + other.y,
        }
    }
}
```

SuperTraits

```
use std::fmt;
trait Log: fmt::Display {
    fn log(&self) {
        let output = self.to_string();
        println!("Logging: {}", output);
    }
}
```

Lifetimes in function signatures

```
fn long<'a>(x: &'a str, y: &'a str) -> &'a str {
    if x.len() > y.len() {
        x
    } else {
        y
    }
}
```

Lifetimes in struct defination

```
struct email<'a> {
  email_address: &'a str,
}
```

Static lifetimes

```
let test: &'static str = "Welcome to RustMY!";
```

Functions, Function Pointers & Closures

Static lifetimes

```
struct Point {
    x: i32,
    y: i32,
}

impl Point {
    // Associated function
    fn new(x: i32, y: i32) -> Point {
        Point { x, y }
    }

    // Method
    fn get_x(&self) -> i32 {
        self.x
    }
}
```

Function Pointers

```
fn do_tw(f: fn(i32) -> i32, v: i32) -> i32 {
  f(v) + f(v)
}
```

Creating Closures

```
let add_one = |num: u32| -> u32 {
  num + 1
};
```

Returning closures

```
fn add_one() -> impl Fn(i32) -> i32 {
    |x| x + 1
}

fn add_or_subtract(x: i32) -> Box<dyn Fn(i32) ->
i32> {
    if x > 10 {
        Box::new(move |y| y + x)
    } else {
        Box::new(move |y| y - x)
    }
}
```

Closure traits

• FnOnce - consumes the variables it captures from its enclosing scope.
• FnMut - mutably borrows values from its enclosing scope.
• Fn - immutably borrows values from its enclosing scope.

Closure traits

```
struct Cacher<T>
where
  T: Fn(u32) -> u32,
{
  calculation: T,
  value: Option<u32>,
}
```

Function that accepts closure or function pointer

```
fn do_twice<T>(f: T, x: i32) -> i32
  where T: Fn(i32) -> i32
{
  f(x) + f(x)
}
```

Pointers

References

```
let mut num = 5;
let r1 = &num; // immutable reference
let r2 = &mut num; // mutable reference
```

Raw Pointers

```
let mut num = 5;
// immutable raw pointer
let r1 = &num as *const i32;
// mutable raw pointer
let r2 = &mut num as *mut i32;
```

Smart Pointers

```
Box<T> - for allocating values on the heap
let b = Box::new(5);

Rc<T> - multiple ownership with reference
counting
let a = Rc : : new(5);
let b = Rc : : clone(&a);

Ref<T>, RefMut<T>, and RefCell<T> - enforce
borrowing rules at runtime instead of compile
time.

let num = 5;
let r1 = RefCell::new(5);
// Ref - immutable borrow
let r2 = r1.borrow();
// RefMut - mutable borrow
let r3 = r1.borrow_mut();
// RefMut - second mutable borrow
let r4 = r1.borrow_mut();
```

Multiple owners of mutable data

```
let x = Rc::new(RefCell::new(5));
```

PACKAGES, CRATES & MODULES

Defination

• Packages - A Cargo feature that lets you build,test, and share crates.
• Crates - A tree of modules that produces a library or executable.
• Modules and use - Let you control the organization, scope, and privacy of paths.
• Paths - A way of naming an item, such as a struct, function, or module.

Creating a new package with a binary crate

$ cargo new my-project

Creating a new package with a library crate

$ cargo new my-project –lib

Defining & using modules

```
fn some_function() {}

mod outer_module {
    // private module by default
    pub mod inner_module {
        // public inner module
        pub fn inner_public_function() {
            super::super::some_function(); //
calling a function from the top level
        }
        fn inner_private_function() {}
    }
}

fn main() {
    // Absolute path from crate root

crate::outer_module::inner_module::inner_public_func
    // Relative path from current scope

outer_module::inner_module::inner_public_function();
    // Bringing the module into scope with use
    use_outer(); // helper below
}

// Helper to show use
fn use_outer() {
    use crate::outer_module::inner_module;
    inner_module::inner_public_function();
}
```

Renaming with as keyword

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

Re-exporting with pub use

```
mod outer_module {
  pub mod inner_module {
    pub fn inner_public_function() {}
  }
}
pub use crate::outer_module::inner_module;
```

Defining modules in separate files

17

```
// src/lib.rs
mod my_module;
pub fn some_function() {
  my_module::my_function();
}

// src/my_module.rs
pub fn my_function() {}
```