ARASHTAD

Title	LEARNING RUST FOR COSMWASM SMART CONTRACTS: A PERFECT GUIDE
Description	Guide
Date	May 30, 2022
Author	Arashtad
Author URI	https://Arashtad.com

Terra network smart contracts are built on top of the CosmWasm template for smart contracts, which is derived from the Cosmos ecosystem. The programming language used for writing these kinds of contracts is Rust. So, learning Rust for CosmWasm smart contracts will be essential. In this article, we are going to get started with some of the basics of learning Rust programming language for CosmWasm smart contracts. Of course, this series of tutorials are not meant to teach you all the details about Rust, but enough for understanding CosmWasm smart contracts.

WHAT ARE COSMWASM SMART CONTRACTS?

CosmWasm smart contracts are used by platforms like Cosmos, Terra, etc. In contrast to solidity, they have some major distinctions like:

1. Avoiding the reentrancy of attacks
2. If you have read our articles on solidity, you already know what safeMath is and why we use it (detecting overflow or underflows).

In CosmWasm smart contracts, Rust allows you to simply set in the overflow-checks = true Cargo manifest to abort the program if any overflow is detected. No way to opt-out of safe math.

3. We don't have Delegate Call logic in CosmWasm. And … mentioned in this documentation link.

LEARNING RUST FOR COSMWASM SMART CONTRACTS

Before we get started with CosmWasm smart contracts, we need to learn the Rust programming language.

INSTALLATION OF RUST LANGUAGE

The installation of Rust on Linux or Mac OS is by using the following command:

```
curl https://sh.rustup.rs -sSf | sh
```

On windows you can use Visual Studio and also install rustup tool for windows and follow the steps on this link. Getting started with Scripts: We get started with the most minimalisitic scripts, Hello World!:

```
fn main() {
 println!("Hello, world!");
}
```

Save the file as FirstScript.rs and open a terminal in that directory and enter the following command to run this script:

```
rustc FirstScript.rs ./t
```

Result

```
"Hello, world!"
```

LEARNING RUST BASICS: COMMENTS

Here is how we write the comments in Rust

```
//This is a comment
/* This is a long,
Multi-line comment
*/
```

Result:
no output

LEARNING RUST BASICS: VARIABLE DECLARATIONS AND DATA TYPES

1) 1. Integers:

we have different types of integers in Rust. Below you can see, the unsigned integer 32, integer 32, usize which is unsigned integer and the number of bytes depends on the the architecture of the machine (32 or 64) and isize which is a signed integer with a number of bytes depending on the architecture of the system(32 or 64):

```
fn main() {
 let _a:u32 = 35;
 let _b:i32 = -25;
 let _ab:u32 = 10;
 let _c = 200; // i32 by default
 let _d:usize = 20; // The size of data depends on the architecture
of Machine
 let _cd:usize = 220; // The size of data depends on the architecture
of Machine
 let _e:isize = 94; // The size of data depends on the architecture
of Machine
 println!("a plus b equals{}",_ab);
 println!("sum of {} and {} equals {}",_c,_d,_cd);
 println!("e is dclared as isize and it has the value of {}",_e);
}
```

```
rustc t.rs ./t
```

```
a plus b equals10 sum of 200 and 20 equals 220 e is dclared as isize
and it has the value of 94
```

LEARNING RUST BASICS: DATA OVERFLOW

When writing CosmWasm smart contracts using Rust, since we are dealing with great numbers, sometimes even u16 integers are overflown. As a result, it is important that we get notified. Rust does this for us:

```
fn main() {
 let a:u8 = 255;
 let a:u8 = 256;
 let b:u8 = 257;
 println!("a equals {} ",a);
 println!("b equals {}",b);
 println!("c equals {}",c);
 println!("d equals {}",d);
 }
```

Result

```
error[E0425]: cannot find value `c` in this scope --> t.rs:9:28 | 9
| println!("c equals {}",c); | ^ help: a local variable with a
similar name exists: `a` error[E0425]: cannot find value `d` in this
scope --> t.rs:10:28 | 10 | println!("d equals {}",d); | ^ help: a
local variable with a similar name exists: `a` error: aborting due
to 2 previous errors For more information about this error, try
`rustc --explain E0425`.
```

2. Strings, Booleans, and Floating points:

```
fn main() {
 let boolean = true; //boolean
 let name = "Alice"; // string
 let age = 32; //int
 let balance:f32 = 12.456; //float , you can declare it as let
balance = 12.456
 println!("{} is {} ",name,age);
 println!("Is She an active user? {}",boolean);
 println!("Her net worth is {}",balance);
 }
```

```
rustc t.rs ./t
```

Result

```
Alice is 32 Is She an active user? true Her net worth is 12.456
```

3. Strings in details:

```
fn main(){
 let mut network = "Terra".to_string();
 network.push_str("Network");
 println!("{}",network);
 }
```

```
rustc t.rs
./t
```

Result

```
TerraNetwork
```

LEARNING RUST BASICS: DATA OVERFLOW

```
fn main() {
 let network = " Cosmos";
 println!("length is {}",network.len());
 }
```

Result

```
rustc t.rs ./t
```

Result

```
length is 7
```

LEARNING RUST BASICS: VECTORS AND SPLIT IN STRINGS

```
fn main() {
 let networks = "Cosmos, Terra, Ethereum, BSC, AVAX";
 for platform in networks.split(","){
 println!("platform is {}",platform);
 }
 println!("\n");
 let platforms:Vec<&str>= networks.split(",").collect();
 println!("platform is {}",platforms[0]);
 println!("platform is {}",platforms[1]);
 }
```

```
rustc t.rs ./t
```

Result

```
platform is Cosmos platform is Terra platform is Ethereum platform is
BSC platform is AVAX platform is Cosmos platform is Terra
```

FUNCTIONS, TUPLES, ARRAYS, AND STRUCTS IN RUST LANGUAGE

Now, we are going to learn how to use functions, tuples, arrays, and structs. There is also an important concept of mutable variables which is very useful when dealing with different circumstances.

LEARNING RUST FOR COSMWASM: FUNCTIONS AND MUTABLE PARAMETERS

If you declare the variable inside the function as mutable, disregarding the initial value that it has been given, it can be changed any number of times that you want to change it.

```
fn main(){
 let number:i32 = 5;
 mutate_number(number);
 println!("The value of number is:{}",number);
}
fn mutate_number(mut mutable_number: i32) {
 mutable_number = mutable_number*0;
 println!("mutable_number value is :{}",mutable_number);
}
```

In the above code, we have defined the 32 bytes integer variable called the “number” and we have given the value 5 to it. After that, we have used the function called the “mutate number” inside which, we have defined a mutable number of the same type as a number (i32). Then, inside the function, we have to change the number by multiplying it by 0 and then printed the result. After that, outside the function, we print the number which was not mutable. The result should show different outputs.
Since the number which was defined outside the function was not mutable, the mutable_number which was inside the function was mutable. So, the mutable_number inside the function must change (5*0 = 0) whereas the number outside of the function must not change (5 = 5). Now, let’s run the code using the following command:

```
rustc practice.rs ./t
```

Result:

```
mutable_number value is :0 The value of number is:5
```

As you can the see, the output of the 2 prints differ from each other. If you want to change the number outside the function, you'll need to declare the variable as mutable.

```
fn main() {
 let mut number:i32 = 5;
 mutate_number(&mut number);
 println!("The value of number is:{}",number);
}
fn mutate_number(mutable_number:&mut i32){
 *mutable_number = 0;
}
```

Now, let's run the code using the following command:

```
rustc practice.rs ./t
```

Result:

```
The value of number is:0
```

And as you can see, the output outside the function has changed to zero. Another useful syntax for functions that makes Rust similar to Solidity, is this type that identifies the type of the returned variable. Besides, the variable that you want to return must have no semicolon.

```
fn function_name() -> return_type {
 value //no semicolon means this value is returned
}
```

LEARNING RUST FOR COSMWASM: TUPLES

In Rust, tuples are simply defined like the below examples.

```
fn main() {
 let tuple:(i32,f64,u8) = (243,-36.1,22);
 println!("{:?}",tuple);
}
```

In the above code, we have defined a tuple with different items in it such as i32, f64 and u8. Then we have assigned di0fferent numbers according to their type. As a result, in a tuple, we can define any type of data to any item of it.
Now, let's run the code using the following command:

```
rustc practice.rs ./t
```

Result:

```
(243, -36.1, 22)
```

We can also call an item of a tuple by using the tuple.n script, in which n is the nth member of the tuple. In the below example, we try to retrieve the first member of the tuple.

```
fn main() {
 let tuple:(i32,f64,u8) = (-325,4.9,22);
 println!("integer is :{:?}",tuple.0);
 println!("float is :{:?}",tuple.1);
 println!("unsigned integer is :{:?}",tuple.2);
}
```

Now, let's run the code using the following command:

```
rustc practice.rs ./t
```

Result

```
integer is :-325 float is :4.9 unsigned integer is :22
```

Another example; this time using a tuple from within a function:

```
fn main(){
 let b:(u32,bool,f64) = (110,true,-21.78);
 print(b);
}
fn print(z:(u32,bool,f64)){
 println!("Inside print method");
 println!("{:?}",z);
}
```

Now, let's run the code using the following command:

```
rustc practice.rs ./t
```

Result

```
Inside print method (110, true, -21.78)
```

In the example below, we have defined a tuple and stored each of its items in separate variables, then we print the result:

```
fn main(){
 let b:(i32,bool,f64) = (110,true,-21.78);
 print(b);
}
fn print(z:(i32,bool,f64)){
 println!("Inside print method");
 let (age,is_male,cgpa) = z;
 println!("Age is {} , isMale? {},cgpa is {}",age,is_male,cgpa);
}
```

Now, let's run the code using the following command:

```
rustc practice.rs ./t
```

Result:

```
Inside print method Age is 110 , isMale? true,cgpa is -21.78
```

LEARNING RUST FOR COSMWASM: ARRAYS

In Rust, arrays are simply defined like the below example. As you can see, we use the syntax NameOfTheArray:[type:size_of_the_array] to define an array. In the example below, we first define the array, and then print its contents and size.

```
fn main(){
 let arr:[i64;4] = [9,22,-67,4];
 println!("array is {:?}",arr);
 println!("array size is :{}",arr.len());
}
```

Now, let's run the code using the following command:

```
rustc practice.rs ./t
```

Result:

```
array is [10, 20, 30, 40] array size is :4
```

In the following example, firstly we do the same as we did in the last example, and secondly, print every one of the items inside the array with their indices.

```
fn main(){
 let arr:[i64;4] = [9,22,-67,4];
 println!("array is {:?}",arr);
 println!("array size is :{}",arr.len());
 for index in 0..4 {
 println!("index is: {} & value is : {}",index,arr[index]);
 }
}
```

Now, let's run the code using the following command in the terminal or command prompt:

```
rustc practice.rs ./t
```

Result:

```
array is [9, 22, -67, 4] array size is :4 index is: 0 & value is : 9
index is: 1 & value is : 22 index is: 2 & value is : -67 index is: 3
& value is : 4
```

LEARNING RUST FOR COSMWASM: STRUCTS

Structs in Rust are like classes in python or other object-oriented languages. First, we define a struct, then we use it inside of a function. You can see in the below struct, we have first defined the category of the data for each person and then in the main function, we have defined the emp1 (or employee1) using the struct we have defined.

```
struct Employee {
 name:String,
 company:String,
 age:u32
}
fn main() {
 let emp1 = Employee {
 company:String::from("Microsoft"),
 name:String::from("Sarah"),
 age:32
 };
 println!("Name is :{} company is {} age is
{}",emp1.name,emp1.company,emp1.age);
}
```

Now, let's run the code using the following command in the console or the terminal.

```
rustc practice.rs ./t
```

Result:

```
Name is :Sarah company is Microsoft age is 32
```

And here goes a complete example of using functions and structs. In this example, we have done the same job as the previous example with the difference that we have used the struct for 2 employees (emp1 and emp2). In the end, we have displayed the data of the 2 employees using the display function.

```
struct Employee {
 name:String,
 company:String,
 age:u32
}
fn main() {
 let emp1 = Employee {
 company:String::from("Microsoft"),
 name:String::from("Sarah"),
 age:32
 };
 let emp2 = Employee{
 company:String::from("Amazon"),
 name:String::from("John"),
 age:28
 };
 display(emp1);
 display(emp2);
}
fn display( emp:Employee){
 println!("Name is :{} company is {} age is
 {}",emp.name,emp.company,emp.age);
}
```

Now, let's run the code using the following command in the terminal:

```
rustc practice.rs ./t
```

Result:

```
Name is :Sarah company is Microsoft age is 32 Name is :John
company is Amazon age is 28
```

ENUMS AND COLLECTIONS IN RUST LANGUAGE

I how Enums and collections like vectors, hash sets, and hash maps are used in Rust programming language. We are going to get deeper into using them by writing some scripts.

LEARNING RUST: ENUMS

Enum is nearly the same as the Enum that we had in Solidity. Here we use it for defining certain stuff like categories and types of objects.

Example 1

In example 1, we define the enum called GenderCategory and then define a struct called person, inside which, we use the GendeCategory enum and then instantiate from the struct Person inside of our main function.

```
#[derive(Debug)]
enum GenderCategory {
 Male,Female
}
#[derive(Debug)]
struct Person {
 name:String,
 gender:GenderCategory
}
fn main() {
 let p1 = Person {
 name:String::from("Collins"),
 gender:GenderCategory::Male
 };
 let p2 = Person {
 name:String::from("Sarah"),
 gender:GenderCategory::Female
 };
 println!("{:?}",p1);
 println!("{:?}",p2);
}
```

Now, let's run the code using the following commands in the terminal.

```
rustc t.rs
./t
```

Result:

```
Person { name: "Collins", gender: Male } Person { name: "Sarah",
gender: Female }
```

Example 2

In the 2nd example, we use enum for defining another category and use it inside of our print-size function for every clothing category with the message printed related to their size. In the end, we will print the size of every clothing according to its category in the main function.

```
enum Clothe {
 Formal,
 Informal
}
fn print_size(cloth:Clothe) {
 match cloth {
 Clothe::Formal => {
 println!("long clothe");
 },
 Clothe::Informal => {
 println!("short clothe");
 }
 }
}
fn main(){
 print_size(Clothe::Formal);
 print_size(Clothe::Informal);
}
```

Now, let's run the code using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
long clothe short clothe
```

LEARNING RUST: COLLECTIONS

1. Vector:

Vector is a resizable array that grows or shrinks in size and it has 5 main methods as follows:

1. new(): creates a new vector.
2. push(): appends an element to the back of a collection.
3. remove(): removes and returns the element with a certain index.
4. contains(): checks whether the vector has an element. And returns true or false according to the input.
5. len(): returns the length of the vector.

In the next example, we define a mutable vector called v and then we will push new data to it. As we push new data to the vector, it increases in size. We can print its size with the vector_name.len() function and also the vector itself to see the change after each push.

```
fn main() {
 let mut v = Vec::new();
 v.push(34);
 println!("size of vector is :{}",v.len());
 println!("{:?}",v);
 v.push(25);
 println!("size of vector is :{}",v.len());
 println!("{:?}",v);
 v.push(12);
 println!("size of vector is :{}",v.len());
 println!("{:?}",v);
}
```

Now, let's run the code using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
size of vector is :1 [34] size of vector is :2 [34, 25] size of
vector is :3 [34, 25, 12]
```

Another Example

In this example, at first, we define a vector with 3 items in it and then use the v.contains() function to check if it has a certain item in it or not.

```
fn main() {
 let v = vec![10,20,30];
 if v.contains(&10) {
 println!("found 10");
 }
 println!("{:?}",v);
}
```

Now, let's run the code using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
Found 10 [10, 20, 30]
```

2. Hash map:

Hash map in Rust works like a dictionary in python or mapping in Solidity. Meaning that it is a collection of key-value pairs. Hash maps have the following methods:

1. insert()
2. len()
3. get ()
4. iter()
5. contains_key()
6. remove()

Example1

In this example, we define a new hash map that is mutable and then we add different properties with their values in it.

```
use std::collections::HashMap;
fn main() {
 let mut Users = HashMap::new();
 Users.insert("name","Sarah");
 Users.insert("Balance","1000");
 println!("size of map is {}",Users.len());
}
```

Now, let's run the code using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
size of map is 2
```

Example 2

In the 2nd example, we define another hash map and then try to remove one of the properties inside it using the .remove function.

```
use std::collections::HashMap;
fn main() {
 let mut Users = HashMap::new();
 Users.insert("name","Sarah");
 Users.insert("Balance","1000");
 Users.insert("Email","Sarah@ymail.com");
 println!("length of the hashmap {}",Users.len());
 Users.remove(&"Email");
 println!("length of the hashmap after remove() {}",Users.len());
}
```

Now, let's run the code using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
length of the hashmap 3 length of the hashmap after remove() 2
```

3. HashSets:

Hash set is another type of collection with unique values and it has the same methods as the Map sets. Let's see an example:

```
use std::collections::HashSet;
fn main() {
 let mut names = HashSet::new();
 names.insert("Sarah");
 names.insert("Tim");
 names.insert("Mike");
 names.insert("Sarah");
 println!("{:?}",names);
}
```

As you can see in the above example, we do not have mappings as opposed to hash maps where we had a mapping between the properties and their values.
Now, let's run the code using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
{"Sarah", "Tim", "Mike"}
```

As you can see we do not have the repeated name Sarah in the result. Because the items should be unique in the hash sets.

LEARNING HOW TO USE GENERICS AND TRAITS IN RUST LANGUAGE

learning Rust for CosmWasm smart contracts will lead us into knowing what Generics and traits are and how we can use them in a script. We are also going to work with some input and output functions to be able to interact with the user.

Learning Rust for CosmWasm: Generics

Generics is a way to write code for multiple types of data such as integers or strings. Suppose we want to store a set of data with different types. If we use a vector, the problem that we face is that we cannot push different types from the ones we had defined at first and then we can only push the type of data that is the same as the first declared variable.

In the below example we define a simple vector with all the data in it of the same type.

```
fn main(){
 let mut dataset: Vec = vec![12,13];
 dataset.push(14);
 println!("{:?}",dataset);
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result:

```
[12, 13, 14]
```

Now if we want to push a string data to the vector, an error will be thrown.

```
fn main(){
 let mut dataset: Vec = vec![12,13];
 dataset.push(14);
 dataset.push("Mike");
 println!("{:?}",dataset);
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result:

```
error[E0308]: mismatched types --> t.rs:4:17 | 4 |
dataset.push("Mike"); | ^^^^^^ expected `i32`, found `&str` error:
aborting due to previous error For more information about this error,
try `rustc --explain E0308`.
```

We can fix this by declaring a struct inside which we declare the type T as a general type for every data type value.

```
struct Dataset {
 value:T,
}
fn main() {
 //generic type of i32
 let t:Dataset = Dataset{value:12};
 println!("value is :{} ",t.value);
 //generic type of String
 let t2:Dataset = Dataset{value:"Mike".to_string()};
 println!("value is :{} ",t2.value);
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result:

```
value is :12 value is :Mike
```

LEARNING RUST FOR COSMWASM: TRAITS

Traits are like interfaces in object-oriented programming. First, we define a trait and then we implement it using impl. In the following example, we define a structure for our data; Employees. Then, we declare the trait containing the function that should be implemented and then the implementation of the trait for the data structure. Now if we define the main function, we can enter any data with the structure Employees, and then we apply the function inside the implementation.
In the below example, you can see that we have trait an impl (implementation of the trait). We also have a struct called Employee using which we define employee b1 in the main function, and we hire the b1 employee. Notice that in the impl we apply a trait on a struct, doing this makes the program written in Rust have a more beautiful structure

```
fn main(){
 let b1 = Employee {
 id:86,
 name:"John"
 };
 b1.HireTheEmplyee();
}
struct Employee {
 name:&'static str,
 id:u32
}
trait Hire {
 fn HireTheEmplyee(&self);
}
impl Hire for Employee {
 fn HireTheEmplyee(&self){
 println!("Hiring Employee with id:{} and name
{}",self.id,self.name)
 }
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result:

```
Hiring Employee with id:86 and name John
```

LEARNING RUST FOR COSMWASM: INPUT AND OUTPUT

Reading from the Command Line:
The following code reads from the command line interface and prints it with an additional sentence. In the below example, we will define a new string which is a mutable called line. This string is going to be used to get data from the user in the terminal. Then, we will use let b1 = std::io::stdin().read_line(&mut line).unwrap() to get data from the user and then we will print it next a sentence.

```
fn main(){
 let mut line = String::new();
 println!("Enter your job title :");
 let b1 = std::io::stdin().read_line(&mut line).unwrap();
 println!("Your job title is , {}", line);
 println!("number of bytes is , {}", b1);
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result:

```
Enter your job title : Developer Your job title is , Developer
number of bytes is , 10
```

LEARNING RUST FOR COSMWASM: WRITING

Write is a new way to print something on the console and can be used as an alternative for "println!". You can see the use case of write in the below example where we use write instead of "println!".

```
use std::io::Write;
fn main() {
 let b1 = std::io::stdout().write("Rust ".as_bytes()).unwrap();
 let b2 =
std::io::stdout().write(String::from("Tutorials").as_bytes()).unwrap
();
 std::io::stdout().write(format!("\nbytes written {}",
(b1+b2)).as_bytes()).unwrap();
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result:

```
Rust Tutorials bytes written 14
```

Another Example

In the example below, the std::env::args() returns the command line arguments.

```
fn main(){
 let cmd_line = std::env::args();
 println!("No of elements in arguments is
 :{}",cmd_line.len());
 let mut sum = 0;
 let mut has_read_first_arg = false;
 for arg in cmd_line {
 if has_read_first_arg {
 sum += arg.parse::().unwrap();
 }
 has_read_first_arg = true;
 }
 println!("sum is {}",sum);
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result:

```
No of elements in arguments is :1 sum is 0
```

FILES (WRITE, READ, APPEND, DELETE, AND COPY) IN RUST LANGUAGE

In Rust, we have different modules for executing different operations on the files. These methods include opening, creating, reading, appending, and writing files. You can see the list of these modules and their code below:

1. std::fs::File
Methods:
open()
create()
2. std::fs::remove_file
Methods:
remove_file()
3. std::fs::OpenOptions
Methods:
append()
4. std::io::Writes
Methods:
write_all()
5. std::io::Read
Methods:
read_to_string()

Writing to A File

In the coming example, we are going to create a file called data.txt and enter some data in it. We will also check whether we face any errors during the creation of the file or not.

```
use std::io::Write;
fn main() {
 let mut file = std::fs::File::create("data.txt").expect("create
failed");
 file.write_all("This is a file written".as_bytes()).expect("write
failed");
 file.write_all("\n In Rust Programming
language".as_bytes()).expect("write
failed");
 println!("Successfully written to file" );
}
```

Now, let's run the above script using the following command in the terminal:

```
rustc t.rs
./t
```

Result in the Terminal:

```
Successfully written to file
```

Result in the data.txt text file:

```
This is a file written In Rust Programming languag
```

Reading from A File

Now, we are going to read from the file we have just created and written to. We print the result in the terminal

```
use std::io::Read;
fn main(){
 let mut file = std::fs::File::open("data.txt").unwrap();
 let mut contents = String::new();
 file.read_to_string(&mut contents).unwrap();
 print!("{}", contents);
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result in the Terminal:

```
This is a file written In Rust Programming language
```

Append Data to A File

Now, we are going to append new data to the data.txt file and update it that way. Notice the Error handling script for OpenOptions and Write using the .expect() function.

```
use std::fs::OpenOptions;
use std::io::Write;
fn main() {
 let mut file =
OpenOptions::new().append(true).open("data.txt").expect(
 "cannot open file");
 file.write_all("\nCongrats!".as_bytes()).expect("write failed");
 file.write_all("\nYou have successfully appended the new
text.".as_bytes()).expect("write failed");
 println!("file append success");
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result in the Terminal:

```
file append success
```

Result in the data.txt text file:

```
This is a file written In Rust Programming language Congrats! You
have successfully appended the new text.
```

Delete A File

Deleting a file is similar to the way we applied other functions on the files. We first write the use std::fs; and then use its method fs::remove_file.

```
use std::fs;
fn main() {
 fs::remove_file("data.txt").expect("could not remove file");
 println!("file is removed");
}
```

Now, let's run the code using the following commands in the terminal:

```
rustc t.rs
./t
```

Result in the Terminal:

```
file is removed
```

If you look at the directory, you will see that the file data.txt has indeed been removed.

Let's run the above script using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
Is 27 an odd number? True
```

Example 2

In this simple example, we have defined a closure that adds up the input with 27. We will use this closure inside of the main function (opposite to the functions that were defined outside of the main function) and print the result for the input 12.

```
fn main(){
 let val = 27;

 let SimpleClosure = |x| {
 x + val
 };
 println!("{}",SimpleClosure(12));
}
```

Let's run the above script using the following command in the terminal:

```
rustc t.rs
./t
```

Result:

```
39
```

WHAT DID WE LEARN?

Firstly, we have got familiar with CosmWasm smart contracts as well as the Rust programming language and some of the basics of the Rust programming language for CosmWasm smart contracts. We have also learned about the syntax and data types in Rust.
Secondly, we have learned how to use functions, tuples, arrays, and structs. We have also learned about an important concept of mutable variables which is very useful when dealing with different situations.
Thirdly, we have got familiar with the syntax used for enums, vectors, hash sets, and hash maps. With the aid of examples, the use cases for each of them have been made easier to understand.
Fourthly, we have learned about Generics, Generic functions, traits, inputs and outputs, writings, etc. Through the examples, these concepts have become much clear.
And finally, we have learned how to interact with files in Rust. The functions that we used helped us read, open, create, remove, write and append to files. We have also got familiar with the iterators and closure concept in Rust with the aid of some insightful examples.