Functional Programming

Hitesh Kumar Sharma
Instructor, Pluralsight

Agenda

- Function Definition
- Functions Calling
- Returning Value from Function
- Function Parameterization
- Generic Functions
- Summary

Introduction to Rust Programming

A function is a collection of statements that carry out a particular activity. The program is divided into logical code chunks by functions. Functions can be invoked after being defined to access code. The code is now reused as a result. Additionally, functions make the program's code simple to read and maintain.

A Function has following components:-

- Function Definition
- Function Invocation
- Function Return Statement
- Function Parameterization

Function Definition

A function needs to be specified before it can be used. Code that should be executed by the function is contained in the function body. Similar to name standards for variables, there exist rules for naming functions. Fn is a keyword used to define functions. Below is the syntax for defining a standard function.

Syntax

```
fn function_name(param: param_data_type) {

   // logic

}
```

A function declaration can optionally contain parameters/arguments. To pass the values into a function, parameters are used.

Function Definition Example

In this example we have defined two functions in Rust Programming named as:

- **greeting_from_pluralsight()**
- **add_two_numbers(num1: i32, num2: i32)**

```
// define a function
fn greeting_from_pluralsight() {
    println!("Hello, PluralSight! Welcome to Rust Programming");
}


// function to add two numbers
fn add_two_numbers(num1: i32, num2: i32) {
    let sum = num1+ num2;
    println!("Sum of num1 and dnum2 = {}", sum);
}
```

Function Calling

To run a function, it must be invoked/called. Function calling is the name given to this process.

When calling a function, parameters' values should be provided along. The caller function is the one that calls another function

```
Syntax

function_name(val1,val2,valN)
```

Function Calling Example

In this example we have called two functions those we have defined in function definition slide:

- **greeting_from_pluralsight()**
- **add_two_numbers()**

```
fn main(){
  //calling a function
  greeting_from_pluralsight();
  add_two-numbers(num1,num2);
}
```

Here, the *main()* is the caller function.

Function Definition and Calling (Single Example)

The following example defines a function ***pluralsight_fn()***. The function prints a message to the console. The ***main()*** function invokes the ***pluralsight_fn()***.function.

```
fn main(){
   //calling a function
   pluralsight_fn();
}
//Defining a function
fn pluralsight_fn(){
   println!(“Hello from PluralSight Function");
}
```

Output
Hello from PluralSight Function

Returning Value from Function

In addition to giving the caller control back, functions can also return a value. These are referred to as returning functions.

Syntax:

Either of the following syntax can be used to define a function with return type.

With return statement

```
fn function_name() -> return_type {
  //statement1
 // Statement 2
  return value;
}
```

Without return statement

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

Returning Value from Function (Example)

The following example is used to demonstrate the returning a value from a function ***sum()*** The function returns the value of addition of two given numbers to the console. The ***main()*** function invokes the ***sum()*** function.

```
fn main() {
    let a:i32 = 5;
    let b:i32 = 3;
    let result:i32 = sum(a, b);
    println!("Sum is: {} + {} = {}", a, b, result);
}
fn sum(a: i32, b: i32) -> i32 {
    return a + b;
}
```

```
fn main() {
    let a:i32 = 5;
    let b:i32 = 3;
    let result:i32 = sum(a, b);
    println!("Sum is: {} + {} = {}", a, b, result);
}
fn sum(a: i32, b: i32) -> i32 {
    a + b
}
```

Function Parameterization

Functions can be passed values through parameters. The signature of the function includes parameters. The function is called with the parameter values supplied to it. The number of values provided to a function must match the number of stated parameters, unless otherwise specified.

You can pass parameters to a function by utilizing one of the following methods:

1. **Passing Parameters by Value**

2. **Passing Parameters by Reference**

Function Parameterization
(Passing by Value)

Each value parameter receives its own storage place when a method is called. They receive copies of the actual parameter values. As a result, the argument is unaffected by changes made to the parameter inside the invoked method.

Example:

```
fn main(){
   let number:i32 = 7;
   convert_to_even(number);
   println!("The value after calling the function is:{}", number);
}
fn  convert_to_even (mut my_num: i32) {
 my_num = my_num *2;
   println!(" The value on calling the function is :{}", my_num);
}
```

Output:

The value on calling the function is :14

The value after calling the function is :7

Function Parameterization
(Passing by Reference)

In contrast to value parameters, when you pass parameters by reference, a new storage location is not made for these parameters. The actual parameters that are passed to the method's reference parameters represent the same memory address. The variable name can be prefixed with a & to send parameter values via reference.

Example:

```
fn main(){
   let number:i32 = 7;
   convert_to_even(&mut number);
   println!("The value after calling the function is:{}", number);
}
fn  convert_to_even (my_num:&mut i32) {
 *my_num = *my_num *2;
   println!(" The value on calling the function is :{}", my_num);
}
```

Output:

The value on calling the function is :14

The value after calling the function is :14

The * operator is used to access value stored in the memory location that the variable **my_num** points to. This is also known as dereferencing.

Calling Function inside another Function

In this example we have demonstrated the calling of one function in another function.

Example:

```
fn main() {
    PluralSight_result(  5,   3);
}
fn PluralSight_result(a: i32, b: i32) {
    println!("{} + {} = {}", a, b, add(a, b));
    println!("{} - {} = {}", a, b, sub(a, b));
}
fn add(a: i32, b: i32) -> i32 {
    return a + b;
}
fn sub(a: i32, b: i32) -> i32 {
    return a - b;
}
}
```

Function: Summary

- A function definition explains to Rust the purpose and appearance of the function.
- When we actually use a function, we make a function call.
- Functions have the ability to receive arguments that let a user provide them values.
- When we pass by value, the original value is unaffected and a copy is made to a new memory location.
- When we pass by reference, the original value is impacted and the same memory address is overwritten.
- Whether or not the return keyword is used, a function can output a value.
- It is possible to call functions from within the definition of other functions.

End of Module