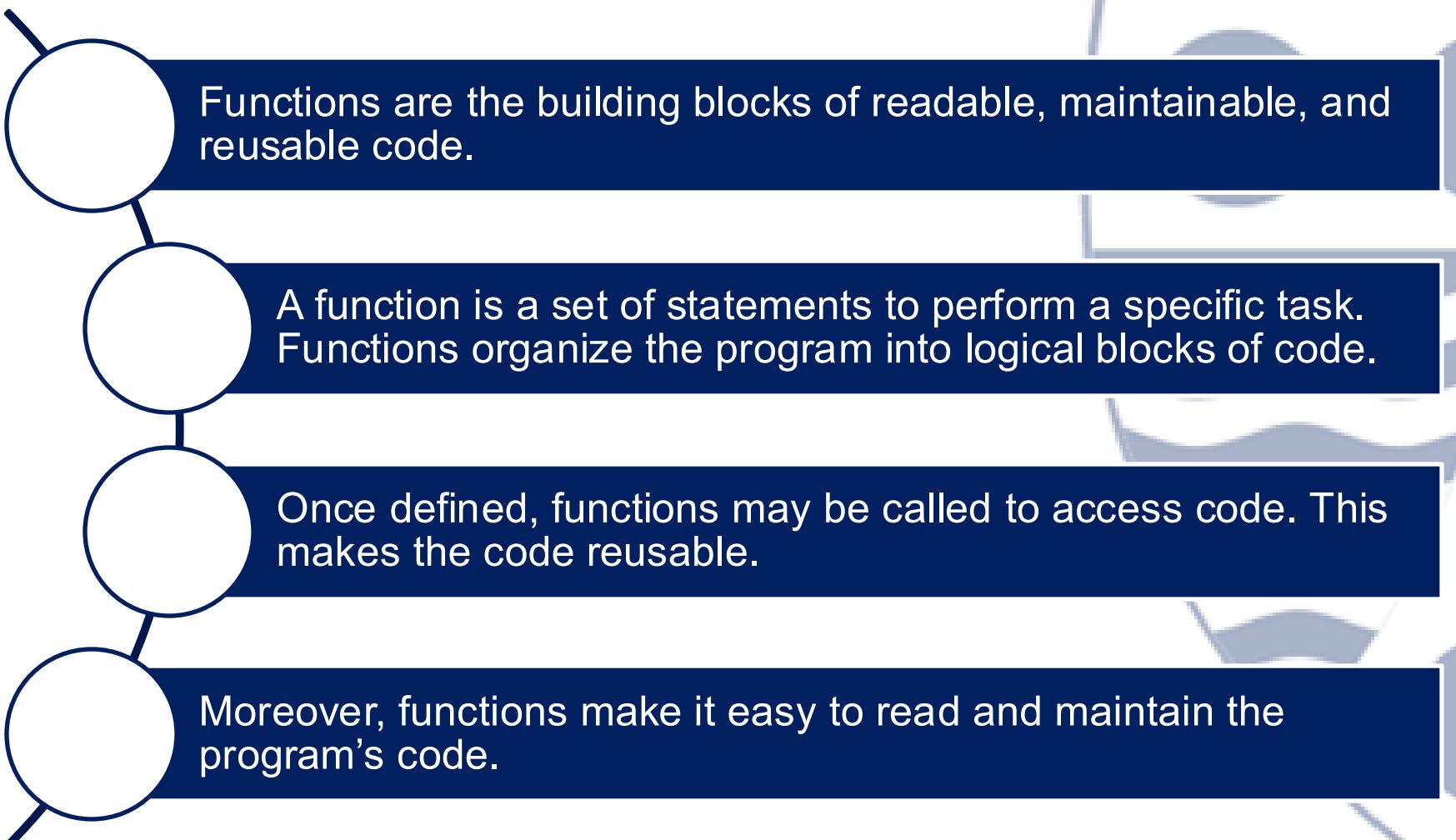


# Rust – Functions & Arrays

*Week 7*



# Rust - Functions



Functions are the building blocks of readable, maintainable, and reusable code.

A function is a set of statements to perform a specific task. Functions organize the program into logical blocks of code.

Once defined, functions may be called to access code. This makes the code reusable.

Moreover, functions make it easy to read and maintain the program's code.

# Rust - Functions



A function declaration tells the compiler about a function's name, return type, and parameters. A function definition provides the actual body of the function.

S/No	Function & Description
1	<b>Defining a function</b> A function definition specifies what and how a specific task would be done.
2	<b>Calling or invoking a Function</b> A function must be called so as to execute it.
3	<b>Returning Functions</b> Functions may also return value along with control, back to the caller.
4	<b>Parameterized Function</b> Parameters are a mechanism to pass values to functions.

# Defining a Function

- A function definition specifies what and how a specific task would be done. Before using a function, it must be defined.
- The function body contains code that should be executed by the function.
- The rules for naming a function are similar to that of a variable.
- Functions are defined using the **fn** keyword.

## Syntax:

```
fn function_name(param1, param2... paramN)
{
    // function body
}
```

# Invoking a Function

A function must be called so as to execute it. This process is termed as **function invocation**. Values for parameters should be passed when a function is invoked. The function that invokes another function is called the **caller function**.

## Practice 1: week-7/practice\_1/src/main.rs

```
main.rs          x
fn my_grade(){
    // function body
    println!("Greetings from function my_grade!");
}

fn main() {
    //calling a function
    my_grade();
}
```

# Invoking a Function

```
main.rs          x
use std::io;

fn checker(){

    let mut input = String::new();
    println!("Enter a character:");
    io::stdin().read_line(&mut input).expect("Failed to read input");
    let ch:char = input.trim().parse().expect("Invalid input");

    if ch >= '0' && ch <= '9'
    {
        println!("Character '{}' is a digit",ch);
    }
    else
    {
        println!("Character '{}' is not a digit",ch);
    }
}

fn main() {
    // calling function
    println!("Welcome! This program checks whether a character variable
contains a digit or not");
    checker()
}
```

**Practice 2: week-7/practice\_2/src/main.rs**

# Returning Value from a Function

Functions may also return a value along with control, back to the caller. Such functions are called returning functions.

Syntax:

```
fn function_name() -> return_type {  
    //statements return value;  
}
```

main.rs

**Practice 3: week-7/practice\_3/src/main.rs**

```
fn main(){  
    println!("pi value is {}",get_pi());  
}  
  
fn get_pi()->f64 {  
    let a:f64 = 22.0;  
    let b:f64 = 7.0;  
    let c:f64 = a/b;  
    return c;  
}
```

# Function with Parameters

- Parameters are a mechanism to pass values to functions.
- Parameters form a part of the function's signature. The parameter values are passed to the function during its invocation.
- Unless explicitly specified, the number of values passed to a function must match the number of parameters defined.
- Parameters can be passed to a function using:

Pass by  
Value

Pass by  
Reference

# Function with Parameters

main.rs

x

**Practice 4: week-7/practice\_4/src/main.rs**

```
use std::io;

fn add(a: i32, b: i32) {
    let sum = a + b;

    println!("Sum of A and B = {}", sum);
}

fn main() {

    let mut input1 = String::new();
    println!("Enter input for parameter A:");
    io::stdin().read_line(&mut input1).expect("Failed to read input");
    let a:i32 = input1.trim().parse().expect("Invalid input");

    let mut input2 = String::new();
    println!("Enter input for parameter B:");
    io::stdin().read_line(&mut input2).expect("Failed to read input");
    let b:i32 = input2.trim().parse().expect("Invalid input");

    // call add function with arguments
    add(a, b);
}
```

# Pass by Value

When a method is invoked, a new storage location is created for each value parameter. The values of the actual parameters are copied into them. Hence, the changes made to the parameter inside the invoked method have no effect on the argument.

**Practice\_5** declares a variable **num** which is initially **5**. The variable is passed as parameter (by value) to the **mutate\_num\_to\_zero()**function, which changes the value to **0**. After the function call, when control returns back to main method the value will be the same.

# Pass by Value

Practice 5: `week-7/practice_5/src/main.rs`

```
fn main(){
    let num:i32 = 5;
    mutate_num_to_zero(num);
    println!("The value of no is:{}" ,num);
}

fn mutate_num_to_zero(mut param_num: i32) {
    param_num = param_num*0;
    println!("param_num value is :{}" ,param_num);
}
```

**Output:**

```
C:\Users\Moruson\Documents\d.morusonCSC101\week-6\practice_5>cargo run
Compiling practice_5 v0.1.0 (C:\Users\Moruson\Documents\d.morusonCSC101\week-6\practice_5)
  Finished dev [unoptimized + debuginfo] target(s) in 0.96s
    Running `target\debug\practice_5.exe`
param_num value is :0
The value of no is:5
```

# Pass by Reference

- When you pass parameters by reference, unlike value parameters, a new storage location is not created for these parameters. The reference parameters represent the same memory location as the actual parameters that are supplied to the method. Parameter values can be passed by reference by prefixing the variable name with an & .
- In the **Practice\_6**, we have a variable **num**, which is initially **5**. A reference to the variable num is passed to the **mutate\_num\_to\_zero()** function. The function operates on the original variable. After the function call, when control returns back to main method, the value of the original variable will be the zero.

# Pass by Reference

## Practice 6: week-7/practice\_6/src/main.rs

```
main.rs *  
  
fn main() {  
    let mut num:i32 = 5;  
    mutate_num_to_zero(&mut num);  
    println!("The value of num is:{}" ,num);  
}  
  
fn mutate_num_to_zero(param_num:&mut i32){  
    *param_num = *param_num*0; //de reference  
    println!("param_num value is :{}" ,param_num);  
}
```

## Output:

```
C:\Users\Moruson\Documents\d.morusonCSC101\week-6\practice_6>cargo run  
Compiling practice_6 v0.1.0 (C:\Users\Moruson\Documents\d.morusonCSC101\week-6\practice_6)  
  Finished dev [unoptimized + debuginfo] target(s) in 0.92s  
    Running `target\debug\practice_6.exe`  
param_num value is :0  
The value of num is:0
```

# Rust - Arrays

An array is a homogeneous collection of values. Simply put, an array is a collection of values of the same data type.

## Features of an Array

- An array declaration allocates sequential memory blocks.
- Arrays are static. This means that an array once initialized cannot be resized.
- Each memory block represents an array element.
- Array elements are identified by a unique integer called the subscript/ index of the element.
- Populating the array elements is known as array initialization.
- Array element values can be updated or modified but cannot be deleted.

# Declaring & Initializing Arrays

The **{:?}** syntax of the **println!()** function is used to print all values in the array.

The **len()** function is used to compute the size of the array.

**//Syntax1**

```
let variable_name = [value1,value2,value3];
```

**//Syntax2**

```
let variable_name:[dataType;size] = [value1,value2,value3];
```

**//Syntax3**

```
let variable_name:[dataType;size] = [default_value_for_elements,size];
```

# Declaring & Initializing Arrays

Practice 7: week-7/practice\_7/src/main.rs

```
fn main(){

    // Array with data type (explicit integer datatype)
    let arr1:[i32;4] = [10,20,30,40];
    println!("\\nArray with data type");
    println!("array is {:?}",arr1);
    println!("array size is :{}",arr1.len());

    // Array without data type (implicit float datatype)
    let arr2 = [10.4,20.7,30.4,40.9,51.2,72.2];
    println!("\\nArray without data type");
    println!("array is {:?}",arr2);
    println!("array size is :{}",arr2.len());

    // Array with default values that creates and
    // initializes all its elements with a default value of -1.
    let arr3:[i32;8] = [-1;8];
    println!("\\nArray with default values");
    println!("array is {:?}",arr3);
    println!("array size is :{}",arr3.len());
}
```

# Array with For Loop

## Practice 8: week-7/practice\_8/src/main.rs

```
fn main(){

    let city_arr:[&str;5] = ["Abuja","Portharcourt","Maiduguri","Kano","Lagos"];
    println!("array is {:?}",city_arr);
    println!("array size is {}",city_arr.len());

    for index in 0..5 {
        println!("City index {} is located in : {}",index,city_arr[index]);
    }
}
```

## Practice 9: week-7/practice\_8/src/main.rs ....(using *iter()* function)

```
// The iter() function fetches values of all elements in an array.

fn main(){

    let arr:[i32;4] = [10,20,30,40];
    println!("array is {:?}",arr);
    println!("array size is {}",arr.len());

    for val in arr.iter(){
        println!("value is {}",val);
    }
}
```

# Class Project



# Project I

Your MTH 101 Professor has asked you to develop a Rust program that can calculate the area or volume of different shapes, depending on the user's choice:

Area of Trapezium formula =  $\text{height}/2 * (\text{base1} + \text{base2})$

Area of the Rhombus formula =  $\frac{1}{2} \times \text{diagonal1} \times \text{diagonal2}$

Area of Parallelogram formula =  $\text{base} \times \text{altitude}$

Area of Cube formula =  $6 \times (\text{length of the side})^2$

Volume of Cylinder formula =  $\pi * \text{radius}^2 * \text{height}$

Using your knowledge of Rust Functions, develop the program that prompts a user to select an equation, reads inputs and then performs the corresponding calculations.