

**CoE 164** 

Computing Platforms



01c: Rust Functions and Control Statements





### STATEMENTS AND EXPRESSIONS

A **statement** performs some action and does not return a value. On the other hand, an **expression** evaluates to return some value.

Note that expressions do not have a semicolon at the end. Statements do.

```
let x = 7; // statement
let y = 2 + 3; // RHS is an expression

let r = {
    let z = x + y;
    z - 2
};
```



### VARIABLE SCOPE

Variables have a *lifetime* or **scope** - that is, it is valid only up to a certain point in the program. Usually, a scope is defined within curly braces.

If a variable is defined or "captured" within a scope, it is said to have become *valid*. Conversely, if it goes *out of scope*, it is said to have become *invalid*.

```
let v = 3; // v is valid here
   // do something...
}
// v is no longer valid here
```



Variables can be **redeclared** later on in the code using the let keyword.

On the other hand, variable names can be reused in inner scopes. This overshadows any same variable names in outer scopes. When the scope ends, any variable values are reverted to their initial ones before the scope started.

```
let v = 5;
let v = v + 1;
println!("Old v is {v}");
    let v = v + 3;
    println!("v inner is {v}");
println!("New v is {v}");
```

## **FUNCTIONS**

A **function** is a subroutine in a program. It can be *called* with some *parameters* to run a certain task or process the parameters in some way.





### **FUNCTIONS: BASICS**

Declare a function by writing the fn keyword followed by the name. Contents are placed inside curly braces.

Functions are named using *snake case* - all words should be in *lowercase* and possibly separated by underscores.

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



### **FUNCTIONS: PARAMETERS**

Function parameters are defined inside the parentheses after the name. Each parameter should be annotated with a type and each one should be separated with a comma.

```
fn print_pairs(a: i64, b: i64) {
    println!("{}, {}", a, b);
}
print_pairs(4, 5);
```



# **FUNCTIONS: RETURN VALUES**

A function can be annotated to return a value by writing an arrow and the data type of the return value after the function signature.

A function can return an explicit value using the return keyword, or implicitly return the last expression. Note the lack of semicolon for the implicit return.

```
fn add_me(a: i64, b: i64) -> i64 {
    return a + b;
}

fn add_me_v2(a: i64, b: i64) -> i64 {
    a + b
}
```



# **FUNCTIONS: RETURN VALUES**

A function can return "multiple data" by returning a tuple with those values.

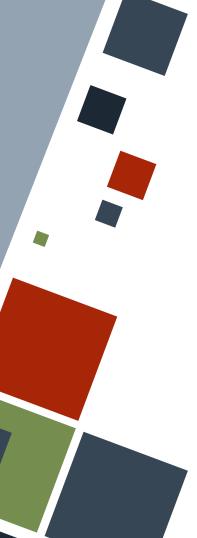
```
fn add_me(a: i64, b: i64) -> (i64, i64) {
    return (a + 3, b + 7);
}
```



### **FUNCTIONS: MAIN**

The main entry point to a Rust program is through a function named main(). It does not have any parameters. Compiled executables will start by finding this function and executing statements inside of it.

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



## **FUNCTIONS: NESTED**

It is possible to write function definitions within functions. It is usually used when the inner function will not be used anywhere outside of it.

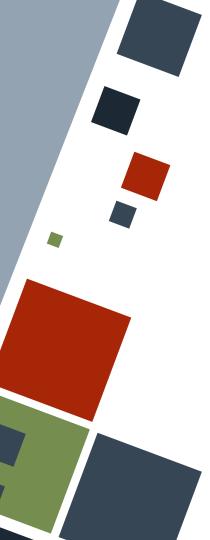
```
fn add me(a: i64, b: i64) -> (i64, i64)
    fn print_first(a: i64) {
    print first(a);
    print first(b);
   return (a + 3, b + 7);
```

# CONTROL STATEMENTS

Rust has the following basic control statements:

- if/else if statements
- loop, while, and for loops



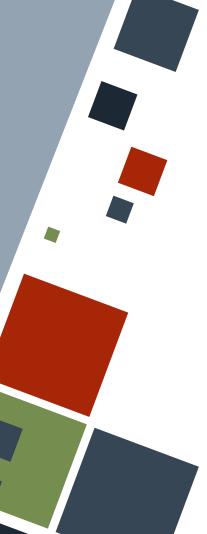


### **CONTROL: IF**

Conditional statements in Rust are built using the keywords if, else if, and else.

Any combination of if, if-else if, and if-else statements are valid in Rust.

```
if num == 3 {
    println!("This is a three");
else if num == 5 {
    println!("This is a five");
else {
    println!("This is {}", num);
```



### **CONTROL: IF ASSIGNMENT**

Conditional statements can return expressions that can consequently be assigned to a variable. Note that each block should *implicitly* return some value and all condition cases are enumerated.

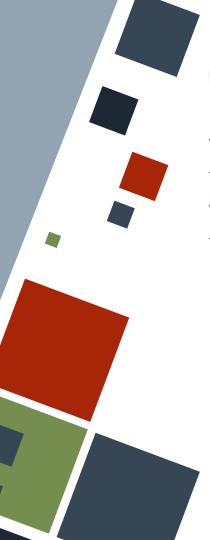
```
Example
let z = if num == 3 {
    "three"
else if num == 5 {
    "five"
else
    "unknown"
```



# **CONTROL: INFINITE LOOP**

A group of statements can be set to run infinitely many times by encapsulating them in a loop block.

```
// Print forever
loop {
    println!("hello!");
}
```



### **CONTROL: LOOP BREAKS**

We can break out of a loop using the break keyword.

We can skip execution of the rest of the loop and restart it from the top using the continue keyword.

```
let mut z = 0;
loop {
    if z > 3 {
        break;
    z += 1;
    if z % 2 == 0 {
        continue;
    println!("hi! {z}");
```



A loop block can return expressions that can consequently be assigned to a variable. Note that the loop *should* terminate and its return value provided as a break statement.

```
let mut i = 0;
let n = 5;
let z = loop {
     i += 1;
     if i >= n {
         break i;
```



### **CONTROL: LOOP LABELS**

We can label loops by specifying a *loop label*. The label is placed beside the loop keyword.

We can break or continue relative to that loop label by writing the label name after the relevant keyword.

```
let n = 3;
let m = 5;
let mut ans = 0;
'out: loop {
    loop {
        ans += m * n;
        if ans > 1000 {
            break 'out;
        continue;
    m += 1;
```



# **CONTROL: WHILE LOOP**

A while loop can be used if looping through a statement while checking whether a certain condition holds is needed.

```
let z = 0;
while z <= 3 {
    z += 1;
    println("hi! {z}");
}</pre>
```



A for loop can be used if there is a collection of data with a fixed size to iterate through.

We can loop through a range of integers by using two dots (..) in between the two ends of the range. The end range is *not* included in the loop.

```
Example
let a = [7, 1, 3];
let mut ans = 0;
for each elm in a {
    ans += each elm;
let n = 5;
let mut tri num = 0;
for i in 0..n {
     tri num += i;
```

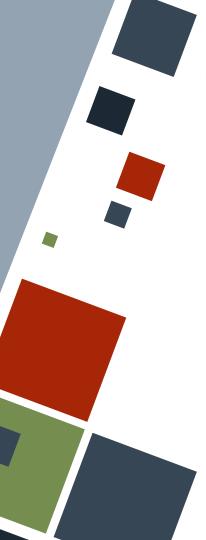


### **CONTROL: RANGES**

A range is denoted by a start and end points with two dots (..) in between. In this case, the end point is *not* included in the range.

For a range to include the end point, an equal sign (...=) should be placed after the two dots.

```
let r = 3;
let x = 0..10; // 0 to 9 inclusive
let z = 0..=r; // 0 to 3 inclusive
let stwo = (12..24).step_by(2); // skip by 2
let revved = (0..5).rev(); // 4 to 0 inclusive
```



### **CONTROL: RANGES**

Ranges can omit the start and end points. However, only ranges that have a starting point can be used in for loops.

```
Example
for x in 0..5 {
    println!("{x}");
for x in 3.. {
    println!("{x}");
    if x >= 10 {
        break;
for x in ..5 {
     break;
```



**CoE 164** 

Computing Platforms



01c: Rust Control Statements and Functions

