# Chapter 6 – Functions

Functions are the basic building blocks in V code: they are very so versatile that we can say that V has a lot of characteristics of a functional language. Functions are also a kind of data: they are themselves values and have types. Functions are first class features in V, like in Go.

In this chapter we start with elaborating on the elementary function-description in ch 4.

## 6.1 Introduction

Every program consists of a number of functions: a function is the basic code block.

All functions in a module are private by default, pub makes them public:

pub fn public\_function() {

}

fn private\_function() {

}

Because V code is compiled, the order in which the functions are written in the program does not matter. Functions can be used before their declaration This is true for all declarations in V and eliminates the need of header files or thinking about the order of files and declarations.

However for readability it is better to start with main() and write the functions in a logical order (for example the calling order), or alphabetical order.

If you want to make useof the fact that main() can be left out in V, you must write the main code after all the other functions.

The main purpose of functions is to break a large problem which requires many code lines into a number of smaller tasks (functions). Also the same task can be invoked several times, so a function promotes code reuse.

(In fact a good program honors the DRY-principle, Don’t Repeat Yourself: this means that the code which performs a certain task can only appear once in the program.)

To enhance the maintainability of your code, every function should only perform one single, related action (the SOLID or single responsibility principle, see <https://en.wikipedia.org/wiki/Single_responsibility_principle>)

In § 4.2 the main characteristics of functions were described, but now we have more material to build concrete and useful examples.

A function ends when it has executed its last statement (before }), or when it executes a return statement, which can be with or without argument(s); these arguments are the values that the function returns from its computation (see § 6.2). A simple return can thus also be used to end an infinite for-loop, or to stop a thread.

Function names:

Naming convention: snake\_case. User functions cannot start with a \_

If you try to use PascalCase for a function name, V will complain: function names cannot contain uppercase letters, use snake\_case instead.

There are 3 types of functions in V:

* Normal functions with an identifier
* Anonymous or lambda functions (see § 6.7)
* Methods (see § 10.6)

Any of these can have parameters and return values. The definition of all the function parameters and return values, together with their types, is called the function signature.

A function is called or invoked in code, in a general format like: mod1.func1(arg1,arg2,…,argn)

where func1 is a function defined in module mod1, and arg1, and so on are the arguments: the values which are passed into the parameters of the function (see § 6.2). When a function is invoked copies of the arguments are made and these are then passed to the called function.

The calling or invocation happens in the code of another function: the calling function. A function can call other functions as much as needed, and these in turn can call other functions, and this can go on with theoretically no limit (unless the stack upon which these function calls are placed is exhausted).

Here is a simple example of a function calling another function:

Listing 6.1 – greeting.v:

fn greeting() {

  println('In greeting: Hi!!!!!')

}

// start main code:

println('In main before calling greeting')

greeting()

println('In main after calling greeting')

Output: In main before calling greeting

In greeting: Hi!!!!!

In main after calling greeting

A function call can have another function call as its argument, provided that this function has the same number and types of arguments in the correct order that the first function needs, e.g.:

Suppose f1 needs 3 int arguments f1(a, b, c int), and f2 returns 3 arguments:

f2(a, b int) (int, int, int), then this can be a call to f1: f1(f2(a, b))

In the same way, one or several of the arguments can be replaced by a function call.

Function overloading, that is coding two or more functions in a program with the same function name but a different parameter list and/or a different return-type(s), is not allowed in V (the same as in C and Go).

This simplifies the code and improves maintainability and readability.

In our example, overloading fn greeting() gives the compiler error: redefinition of "greeting"

The main reason is that overloading functions requires additional type matching of its arguments, which is sometimes impossible only at compile time. No overloading means only a simple function dispatch is needed.

So you need to give your functions appropriate unique names, probably named according to their signature.

Here is another simple example (taken from the Docs page of the V website):

Listing 6.2 – add\_sub.v:

fn add(x, y int) int {

  return x + y

}

fn sub(x, y int) int {

  return x - y

}

println(add(77, 33)) // 110

println(sub(100, 50)) // 50

Multiple arguments of the same type can be clustered together like in: x, y int

Exercise: Use a fn main() that calls add and sub before they are defined.

Exercise: Write a function that gives you the output for fizzbuzz for an integer (see Exercise 5.7): *fizzbuzz2.v*

Exercise: Write a function that takes 2 boolean values and prints out the &&, || and ! result (see *logical\_operations.v*)

Exercises:

 Write a V program to find the square of any number using a function: *square.v*

 Write a V program to check a given number is even or odd using a function: *odd\_even.v*

 Write a V program to convert decimal number to binary number using a function: *convert\_decimal\_binary.v*

 Write a V program to check whether a number is a prime number or not using a function: *is\_prime.v*

A function cannot be declared inside another function (no nesting), but this can be mimicked by using anonymous functions.

A function fn1 is defined in a module (say mod1). When the module is imported with import mod1, the function fn1 can be called inside this source file as an explicit import: mod1.fn1(), like in Go and Oberon. This avoids name-conflicts, and always makes it clear in which module a function is defined. Other languages like Nim allow importing functions into the global namespace and simply calling them as fn1(), which can become a huge problem when working on large code bases.

## 6.2 Parameters and return values

A function can take parameters to use in its code, and it can return zero or more values. This is also a great improvement compared to C, C++, Java and C#, where function can have only one return value. V functions often return an optional result instead of returning both a result and an error like in Go (see ch. 12).

Returning (a) value(s) is done with the keyword return . In fact every function that returns at least 1 value must end with return.

Code after return in the same block is not executed anymore. If return is used, then every possible code-path in the function must end with a return statement.

A function with no parameters is sometimes called a niladic function, like main() in module main, or main.main().

**6.2.1 Parameters: call by value or call by reference**

When you pass a variable arg1 as an argument to a function like this:

fn function1(arg1 Type1) {

...

}

Or in case of a method (see § 9.6) like this:

fn (foo Foo) bar\_method() {

...

}

the V compiler can decide whether to pass arg1 *by value* (a copy is made of the variable) or *by reference* (the address of the variable is passed on to the function. In the first case, the function works with and possibly changes the copy, the original value is not changed. If the variable takes up a lot of memory, then the second case can be better for saving memory resources (passing a pointer (a 32 bit or 64 bit value) is in many cases cheaper than making a copy of the object. But also in this case, the variable cannot be changed by the function.

If you want to ensure that the struct is always passed by reference you can add &:

fn function1(arg1 &Type1) {

...

}

Or in case of a method

fn (foo &Foo) bar\_method() {

println(foo.abc)

}

But even then, arg1 or foo cannot be changed through the function.

If you want a function to be able to change the value of arg1 itself (‘in place’), you have to indicate that variable as mutable mut: function1(mut arg1).

See this in action in the program mutate\_array.v in § 6.2.3.

Some functions just perform a task, and do not return values: they perform what is called a *side-effect*, like printing to the console, sending a mail, logging an error, and so on. For an example, see function greeting() in Listing 6.1

In the following program simple\_function.v the function displays a side-effect: it takes 3 int parameters a, b and c, all passed by value and returns nothing:

Listing 6.2 – simple\_function1.v:

fn multiply\_3nums(a, b, c int) {

  println('Multiply $a \* $b \* $c = ${a \* b \*c}') // Multiply 2 \* 5 \* 6 = 60

}

multiply\_3nums(2, 5, 6)

### 6.2.2 Return values

Most functions return values to be used later in the program.

The following program simple\_function2.v the function returns the product of the arguments as an int parameter:

Listing 6.2b – simple\_function2.v:

fn multiply\_3nums(a, b, c int) int {

  // product := a \* b \* c

  // return product

  return a \* b \* c

}

i1 := multiply\_3nums(2, 5, 6)

println('Multiply 2 \* 5 \* 6 = $i1')

println('Multiply 2 \* 5 \* 6 = ${multiply\_3nums(2, 5, 6)}')

/\* Output:

Multiply 2 \* 5 \* 6 = 60

Multiply 2 \* 5 \* 6 = 60

\*/

(the commented lines show more verbose alternative code, where a local variable product is used in the function)

A function can have multiple return values, of the same or different types.

Here is an example with 2 return values of the same type:

Listing 6.2c – simple\_function2B.v:

fn add\_sub(x int, y int) (int, int) {

  return x + y, x - y

}

n1, n2 := add\_sub(77, 33)

println('n1 is $n1, n2 is $n2') // n1 is 110, n2 is 44

Another example function header returning different types: fn mult\_ret\_types() (int, string) { … }

When it becomes necessary to return 3 or more values from a function, it is best to to this in the form of an array (see chapter 7) if the values are of the same type (homogeneous). Use a pointer to a struct (see chapter 10) if they are of different type (heterogeneous). Passing a pointer like that is cheap and allows to modify the data in place.

### 6.2.3 Partially pure functions

V functions are pure by default, meaning that their return values are only determined by their arguments, and their evaluation has no side effects.

This is achieved by lack of global variables and because all function arguments are immutable by default.

V is not a pure functional language however: you can modify function arguments by prefixing them with the keyword mut, but the arguments have to be arrays, maps, and structs. To change simple values, return the changed value, so instead of func(n mut int) write a func(n int) int.

So this cannot work: part\_pure.v:

fn non\_pure (a mut int) int {

        b := a \* 5

        return b

}

n := non\_pure(10)

println(n)

/\* compile error:

mutable arguments are only allowed for arrays, maps, and structs.

return values instead: `foo(n mut int)` => `foo(n int) int`

\*/

Instead, you have to write it as:

fn non\_pure (a int) int {

    b := a \* 5

    return b

}

n := non\_pure(10)

println(n) // => 50

It is preferable to return values instead of modifying arguments. Modifying arguments should only be done in performance-critical parts of your application to reduce allocations and copying.

In chapters 7, 8 and 9 we’ll see other examples.

Exercise: Change the first element of an array through a function modify: see change\_array.v

Exercise: Pick uniformly a random integer in [a..b]

// Pick a random integer greater than or equals to a, inferior or equals to b. Precondition : a < b.

(see pick\_uniformly\_a\_random\_int\_point\_number\_in.v)

### 6.2.4 Variadic functions

These are functions that can take a variable number of arguments (varargs). They have the signature:

fn fun1(params ...Type) {}

This … means here that the 2nd parameter is a variable number of Type. That number can be 0, 1 or more.

The variadic function collects these arguments into an array params, which is than processed inside the function.

Example: varnumpar.v

fn min(a ...int) int {

  if a.len == 0 { return 0 }

  mut min := a[0]

  for v in a {

    if v < min { min = v }

  }

  return min

}

x := min(1, 3, -8, 2, 0)

println('The minimum is: $x') // The minimum is: -8

/\* doesn't work yet ??:

arr := [7, 9, -8, 3, 5, 1]

y := min(arr...) // expected type `[]int`, but got `int`

println('The minimum in the array arr is: $y')

\*/

Example: varnumpar2.v

fn print\_out(test ...string) {

    for txt in test {

        println(txt)

    }

}

print\_out("V", "is", "the", "best", "lang" , "ever")

/\* Output:

V

is

the

best

lang

ever

\*/

## 6.3 Control constructs in functions

### 6.3.1 Using if and return in functions

An if can leave a function by using return.

The idiom in V-code is to omit the else-clause when the if ends in a break, continue or return statement.

***IDIOM*** if condition {

return x

}

// next statement presumes the else branch

When returning different values x and y whether or not a condition is true, use the following:

***IDIOM*** if condition {

return x

}

return y

Example: compare\_values.v:

fn compare\_values(a, b int) string {

  if a > b {

    return 'Bigger'

  }

  return 'Equal or Smaller'

}

println(compare\_values(3, 10)) // => Equal or Smaller

### 6.3.2 Using match with return in functions

Return can also be used to return a value from a function which contains a match. As an example, see season\_of\_month.v:

fn season\_of\_month(month int) string {

  match month {

    12,1,2  { return 'Winter' }

    3,4,5   { return 'Spring'}

    6,7,8   { return 'Summer'}

    9,10,11 { return 'Autumn'}

  }

  return 'Season unknown'

}

println(season\_of\_month(2)) // => Winter

### 6.3.3 Using defer in functions

Here we can see how defer works when called within functions:

Listing 6.9 – defer.v :

fn fun1() {

  println("In fun1 at the top")

  defer { fun2() }

  println("In fun1 at the bottom!")

}

fn fun2() {

  println("fun2: Deferred until the end of the calling function fun1!")

}

fun1()

/\* Output:

In fun1 at the top

In fun1 at the bottom!

fun2: Deferred until the end of the calling function fun1!

\*/

(compare the output when defer is removed)

If defer has arguments, they are evaluated at the line of the defer-statement; this is illustrated in the following snippet, where the defer will print 0:

fn a() {

    mut i := 0

    defer { println(i) }

    i++

    return

}

a() // => 1

defer allows us to guarantee that certain clean-up tasks are performed before we return from a function, for example:

1. closing a file stream:

// open a file

defer file.Close() (see § 12.2)

1. unlocking a locked resource (a mutex):

mu.Lock()  
defer mu.Unlock() (see § 9.3)

1. printing a footer in a report:

printHeader()  
defer printFooter()

1. closing a database connection:

// open a database connection

defer disconnectFromDB()

It can be helpful to keep the code cleaner and so often shorter.

The following listing simulates case 4):

Listing 6.10 – defer\_dbconn.v :

fn connect\_to\_db() {

  println("ok, connected to db")

}

fn disconnect\_from\_db() {

  println("ok, disconnected from db")

}

fn do\_db\_operations() {

  connect\_to\_db()

  println("Defering the database disconnect.")

  defer { disconnect\_from\_db() }

  println("Doing some DB operations ...")

  println("Oops! some crash or network error ...")

  println("Returning from function here!")

  return //terminate the program

  // deferred function executed here just before actually returning,

  // even if there is a return or abnormal termination before

}

do\_db\_operations()

/\* Output:

ok, connected to db

Defering the database disconnect.

Doing some DB operations ...

Oops! some crash or network error ...

Returning from function here!

ok, disconnected from db

\*/

Tracing with defer:

A primitive but sometimes effective way of tracing the execution of a program is printing a message when entering and leaving certain functions. This can be done with the following 2 functions:

fn trace(s string)   { println('entering: $s') }

fn untrace(s string) { println('leaving:  $s') }

where we will call untraced with the defer keyword, as in the following program

Listing 6.11 – \_defer\_tracing.v:

fn trace(s string)   { println('entering: $s') }

fn untrace(s string) { println('leaving:  $s') }

fn a() {

  trace("a")

  defer { untrace("a") }

  println("in a")

}

fn b() {

  trace("b")

  defer { untrace("b") }

  println("in b")

  a()

}

b()

/\* Output:

entering: b

in b

entering: a

in a

leaving: a

leaving: b

\*/

## 6.4 Higher order or first class functions

Function calls can be *chained*, see for example: arr\_len := os.get\_line().trim\_space().int() (In array\_readin.v ch 6), also in function calls, where arguments can be (chained) function calls: rand.seed(time.now().uni)

You can pass functions as arguments, return them, and call them later. What you pass or return is not the function itself, but the it’s address in code space, or in other words: a function pointer.

This is the syntax for function pointers:

**type** fun1 **fn** (a, b int) string

it consists of defining a type with a the signature (a, b int) string of a function (fn), whose name is fun1.

(to get your mind around it, make a sentence like the previous one which explains every aspect of the definition)

Another example of a function fun2, which takes a function a with signature fn(int) int as argument, and returns a function pointer to a function with signature fn(string):

fn fun2(a fn(int) int) fn(string) {}

Here is an example of passing that function as an argument to another function named tests:

fn test(f fun1 (int, int) string) {

f(0, 1)

}

Or defining a struct with a field that is a function pointer:

struct Foo { f fun1 }

Here is a working example:

Listing 6.6B: function\_param.v

fn call(fun fn (int) string) {

  println(fun(25))

}

fn say(n int) string {

  return 'I said it!'

}

call(say)

// I said it!

Exercise: see function\_param2.v

Make a function run that takes an integer value as argument, and a function that takes an int and returns an int. run calls its 2nd argument (the function) with its 1st  argument. Use sqr as test function: fn sqr(a int) int

Types can be passed as arguments: ?? example.

The benchmarking module of M. Vlootman (see § 3.7) uses a function as parameter to the benchmarking functions.

## 6.5 Recursive functions

?? example in chapter about arrays!!

A function is called recursive, when it calls itself in its body. This sometimes makes for a shorter implementation.

A proverbial example is the calculation of the numbers of the Fibonacci sequence, in which each number is the sum of its two preceding numbers.

The sequence starts with:

1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181, 6765, 10946, …

<https://en.wikipedia.org/wiki/Fibonacci_number>

Let’s calculate the first 10 Fibonacci numbers imperatively. We don’t need a function, we simply store the numbers in an array fibs:

Listing 6.22A – \_ fibonacci\_imperative.v:

const (

    MAX = 10

)

fn main() {

  mut fibs := [0].repeat(MAX)

  fibs[0] = 1

  fibs[1] = 1

  for i := 2; i < MAX; i++ {

    fibs[i] = fibs[i-1] + fibs[i-2]

    println('The $i-th Fibonacci number is: ${fibs[i]}')

  }

}

/\* Output:

The 2-th Fibonacci number is: 2

The 3-th Fibonacci number is: 3

The 4-th Fibonacci number is: 5

The 5-th Fibonacci number is: 8

The 6-th Fibonacci number is: 13

The 7-th Fibonacci number is: 21

The 8-th Fibonacci number is: 34

The 9-th Fibonacci number is: 55

\*/

Here is a recursive version:

Listing 6.13 – \_ fibonacci.v:

fn fib(n int) int {

  if n <= 1 {

    return 1

  }

  return fib(n - 1) + fib(n - 2)

}

for i := 0; i < 10; i++ {

  println('fibonacci($i) is: ${fib(i)}')

}

/\* Output:

fibonacci(0) is: 1

fibonacci(1) is: 1

fibonacci(2) is: 2

fibonacci(3) is: 3

fibonacci(4) is: 5

fibonacci(5) is: 8

fibonacci(6) is: 13

fibonacci(7) is: 21

fibonacci(8) is: 34

fibonacci(9) is: 55

\*/

See also fibonacci\_i64.v for calculating a larger serie before overflow occurs:

fn fib(n int) i64 {

  if n <= 1 {

    return i64(1)

  }

  return fib(n - 1) + fib(n - 2)

}

println(fib(45))

// 1836311903

Many problems have an elegant recursive solution, like the famous Quicksort algorithm.

*Mutually recursive functions* can also be used in V: these are functions that call one another. Because of the V compilation process, these functions may be declared in any order. Here is a simple example: even calls odd, and odd calls even.

Listing 6.14 – mut\_recurs.v:

fn even(nr int) bool {

    if nr == 0 {return true}

    return odd(rev\_sign(nr) - 1)

}

fn odd(nr int) bool {

    if nr == 0 {return false}

    return even(rev\_sign(nr) - 1)

}

fn rev\_sign(nr int) int {

  if nr < 0 {return -nr}

  return nr

}

println('16 is even: is ${even(16)}')   // 16 is even: is 1

println('17 is odd: is ${odd(17)}')     // 17 is odd: is 1

println('18 is odd: is ${odd(18)}')     // 18 is odd: is 0

/\* Output:

16 is even: is 1

17 is odd: is 1

18 is odd: is 0

\*/

Here is the Tower of Hanoi problem, solved recursively in V: *hanoi.v*

An important problem to be aware of when using recursive functions is *stack overflow*: this can occur when a large number of recursive calls are needed and the program runs out of allocated stack memory. The V compiler does not implement tail call optimization for recursive function calls.

**EXERCISES:**

Exercise 6.5: 10to1\_recursive.v

Print the numbers from 10 to 1 in that order using a recursive function printrec(i int)

Exercise 6.6: factorial.v

Write a program which prints the factorial (!) of the first 30 integers

The factorial n! of a number n is defined as: n! = n \* (n – 1)! , 0!=1

So this clearly a good candidate for a recursive function!

Remark that when using type int the calculation is only correct up until 12!, this is of course because an int can only contain integers which fit in 32 bit. V doesn’t warn against this overflow-error!

See also *factorial\_imperative.v / factorial\_recursive.v*