Parameters and Return Values

This lesson will discuss the elements of functions like parameters and return values in detail.

WE'LL COVER THE FOLLOWING

Introduction

A function can take *parameters* to use in its code, and it can *return* zero or more values (when more values are returned, one of the values often speaks of a tuple of values). This is also a great improvement compared to C, C++, Java, and C#, and it is particularly handy when testing whether or not a function execution has resulted in an error (as we studied in Chapter 3).

Parameters of a function

Parameters can be *actual* parameters or *formal* parameters. The key difference between them is that actual parameters are the values that are passed to the function when it is invoked, while formal parameters are the variables defined by the function that receives values when the function is called. Remember the `greeting` function from the previous lesson. We called it using the following statement:

```
greeting(lastName)
```

And we implemented the function as:

```
func greeting(name string) {
    println("In greeting: Hi!!!!!", name)
    name = "Johnny"
    println("In greeting: Hi!!!!!", name)
}
```

Here, `lastName` is the *actual* parameter, and `name` is the *formal* parameter.

Return from a function

Returning value(s) is done with the keyword `return`. In fact, every function that returns at least 1 value must end with `return` or `panic` (see Chapter 11). The code after return in the same block is not executed anymore. If return is used, then every code-path in the function must end with a return statement.

> **Note:** A function with no parameters is called a **niladic function**, like `main.main()`.

Call by value or call by reference

The default way to call a function in Go is to pass a variable as an argument to a function by value. A copy is made of that variable (and the data in it). The function works with and possibly changes the copy; the original value is not changed:

```
Function(arg1)
```

If you want `Function` to be able to change the value of `arg1` itself (in place), you have to pass the *memory address* of that variable with **&**; this is **call (pass) by reference**:

```
Function(&arg1)
```

Effectively, a pointer is then passed to the function. If the variable that is passed is a pointer, then the pointer is copied, not the data that it points to. However, through the pointer, the function can change the original value. Passing a pointer (a 32-bit or 64-bit value) is in almost all cases cheaper than

making a copy of the object. Reference types like slices (Chapter 5), maps (Chapter 6), interfaces (Chapter 9) and channels (Chapter 12) are pass by reference by default (even though the pointer is not directly visible in the code).

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. However, most functions return values, which can be named or unnamed.

The following is a program of a simple function that takes 3 parameters and returns a single value.

```
package main
import "fmt"

func main() {
    fmt.Printf("Multiply 2 * 5 * 6 = %d\n", MultiPly3Nums(2, 5, 6))
        // var i1 int = MultiPly3Nums(2, 5, 6)
        // fmt.Printf("Multiply 2 * 5 * 6 = %d\n", i1)
}
func MultiPly3Nums(a, b, c int) int {
    // var product int = a * b * c
    // return product
    return a * b * c
}
```

Simple Function

As you can see, we declare a function `Multiply3Nums(a,b,c int)` at **line 9**. The function should return the product of three variables: `a`, `b`, and `c`. Now, there are 2 ways to return the product. One way is to declare a variable `product` and initialize it with `a*b*c` and then return the `product`. The second way is to simply return `a*b*c` without using any variables. You may have noticed, we implemented both methods but commented out the first method (from **line 10** to **line 11**). The second method is a one-line implementation (at **line 12**).

Now, look at the `main` function. In `main` we have to print the product of the numbers by calling the `Multiply3Nums(a,b,c)` function. There are 2 methods to do so. One way is to declare a variable `i1`, initialize it with

`Multiply3Nums(a,b,c)` and then print `ii`. The second way is to simply print `Multiply3Nums(a,b,c)` without using any variables. You may have noticed, we

implemented both methods but commented out the first method(from **line 6** to **line 7**). The second method is a one-line implementation (at **line 5**).

Named return variables

When there is more than 1 unnamed return variable, they must be enclosed within **()**, like **(int, int)**. Named variables used as result parameters are automatically initialized to their zero-value, and once they receive their value, a simple (empty) return statement is sufficient. Furthermore, even when there is only 1 named return variable, it has to be put inside ().

```
package main
import "fmt"

var num int = 10
var numx2, numx3 int

func main() {
    numx2, numx3 = getX2AndX3(num) // function call
    PrintValues()
    numx2, numx3 = getX2AndX3_2(num) // function call
    PrintValues()
}
func PrintValues() {
    fmt.Printf("num = %d, 2x num = %d, 3x num = %d\n", num, numx2, numx3)
}
func getX2AndX3(input int)(int, int) {
    return 2 * input, 3 * input
}
func getX2AndX3_2(input int)(x2 int, x3 int) {
    x2 = 2 * input
    x3 = 3 * input
        //return x2, x3
    return
}
```

Multiple Returns

In the above program, there are two functions `getX2AndX3` and `getX2AndX3_2` that are enough to get the concepts of named return values. Look at **line 17**. The return statement of `getX2AndX3` is: `return 2*input,3*input`. This is an example of the unnamed return values because we haven't named any value. Now look at **line 19**: `func getX2AndX3_2(input int) (x2 int, x3 int)`. Return

values (`x2` and `x3`) are explicitly named. Such naming allows you to use the

`return` statement simply (as we did at **line 23**). Or, you can use **line 22** instead of **line 23**; that is also a method to return named values.

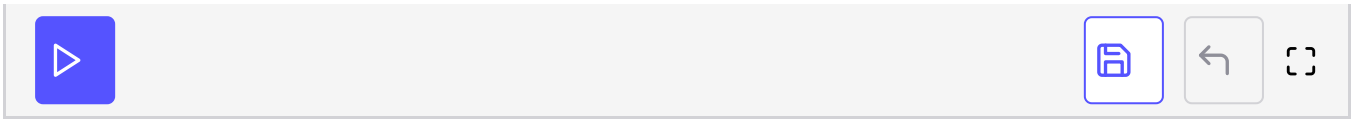
Even with named return variables, you can still ignore the names and return explicit values. When any of the result variables have been shadowed (not a good practice!), the return-statement must contain the result variable names. Use named return variables; they make the code clearer, shorter, and self-documenting.

Blank identifier

The *blank identifier* _ can be used to *discard* values, effectively assigning the right-hand-side value to nothing. Look at the following program to see how a blank identifier works.

```
package main
import "fmt"

func main() {
    var i1 int
    var f1 float32
    i1, _, f1 = ThreeValues() // blank identifier
    fmt.Printf("The int: %d, the float; %f\n", i1, f1)
}
func ThreeValues()(int, int, float32) {
    return 5, 6, 7.5
}
```



Blank Identifier

As you can see in the above code, there is a function at **line 10** `threeValues`, that takes three parameters and returns them. Now let's suppose we need two return variables, not three. What to do now? The answer lies in using the *blank identifier*. See **line** 7. We made three identifiers: `i1`, `_`, and `f1`. This means we need the first and last return value, and we want to ignore the second return value. You can't use `-` anywhere in the code. That is why we only print `i1` and `f1` at **line 8**.

Changing an outside variable

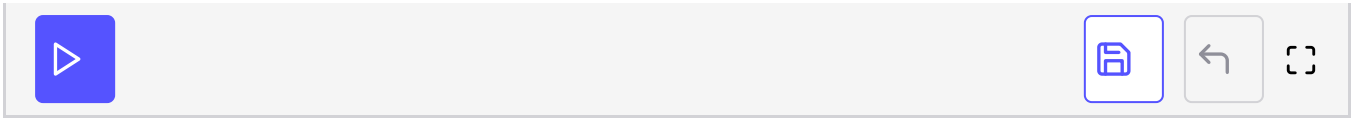
Passing a pointer to a function not only conserves memory because no copy of the value is made, but it also has as a side-effect. The variable or object can be changed inside the function so that the changed object doesn't have to be returned back from the function. See this in the following program. A pointer to an integer is being changed in the function.

```
package main
import (
    "fmt"
)

// this function changes reply:
func Multiply(a, b int, reply * int) {
    * reply = a * b // side-effect(changing n)
}

func main() {
    n := 0
    reply := & n
    fmt.Println("Before Multiplication:", n)
    fmt.Println("Before Multiplication:", * reply)

    Multiply(10, 5, reply)
    fmt.Println("Multiply:", * reply) // Multiply: 50
    fmt.Println("Multiply:", n) // Multiply: 50
}
```



Side-Effect

As you can see, at **line 12** we declare a variable `n` and initialize with value **0**. In the next line, we declare a new pointer variable `reply` and store the address of `n` in it. In the next two lines, we are printing the value of `n` using value type(**line 14**) and reference type(**line 15**). You notice that both lines will print **0**. Let's see side-effect.

At **line 17**, the `Multiply` function is called. Look at function header of `Multiply` at **line** 7. It has three parameters; `a`, `b` and `reply`. The type of `reply` is `int*`, which means it is a pointer variable. At **line 8**, we multiply `a` and `b` and change the value at the address stored in `reply`. This is the side-effect. Now, we have no need to return any variable from this function.

Control will transfer to **line 18**. In this line, we are printing value at the

address stored in `reply`. Similarly, at the next line, we are printing the value

`n`. Both the values are the same because `n` is the value that was stored at the address placed in `reply`. The value of `n` is changed as a side-effect.

This is only a didactic example, but it is much more useful to change a large object inside a function. However, this technique obscures a bit of what is going on. The programmer should be very much aware of this side-effect and if necessary, make it clear to users of the function through a comment.

Passing a variable number of parameters

If the *last* parameter of a function is followed by **...type**, this indicates that the function can deal with a variable number of parameters of that type, possibly also 0, a so-called **variadic function**:

```
func myFunc(a, b, arg ...int) {}
```

Consider the function:

```
func Greeting(prefix int, who ...string)
```

and the function call:

```
Greeting(4, "Joe", "Anna", "Eileen")
```

Here, `Greeting` takes an input of the integer 4 and a slice (which we'll cover in the next chapter) `who` that is equal to `[]string{"Joe", "Anna", "Eileen"}`.

The following is an example of a program implementing a function that takes a variable number of parameters.

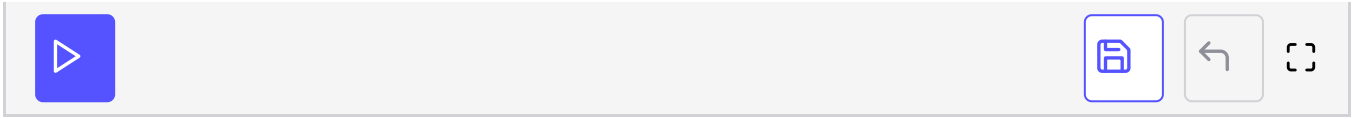
```
package main
import "fmt"

func main() {
    Print(1, 3, 2, 0)
}

func Print(a...int) { // variable number of parameters
    if len(a) == 0 {
        return
    }
```

```
    for _, v := range a {
        fmt.Printf("The number is: %d\n", v)
    }

    return
}
```



Variable Number of Parameters

As you can see, we implemented a function at **line 8** called `Print` that takes the variable number of parameters to get printed. Now, there can be a case where the user doesn't pass any parameter. For this, we simply return from the function at **line 10**. If the parameters list is not empty, we print the parameters using the `range` function at **line 14**.

Now, you are familiar with arguments and how functions return values. In the next lesson, you have to write a function to solve a problem.