CHAPTER 5

Functions

So far, your programs have been limited to a few lines in the `main` function. It's time to get bigger. In this chapter, you're going to learn how to write functions in Go and see all the interesting things you can do with them.

Declaring and Calling Functions

The basics of Go functions are familiar to anyone who has programmed in other languages with first-class functions, like C, Python, Ruby, or JavaScript. (Go also has methods, which I'll cover in Chapter 7.) Just as with control structures, Go adds its own twist on function features. Some are improvements, and others are experiments that should be avoided. I'll cover both in this chapter.

You've already seen functions being declared and used. Every Go program starts from a `main` function, and you've been calling the `fmt.Println` function to print to the screen. Since a `main` function doesn't take in parameters or return values, let's see what it looks like when a function does:

```
func div(num int, denom int) int {
    if denom == 0 {
        return 0
    }
    return num / denom
}
```

Let's look at all the new things in this code sample. A function declaration has four parts: the keyword `func`, the name of the function, the input parameters, and the return type. The input parameters are listed in parentheses, separated by commas, with the parameter name first and the type second. Go is a typed language, so you must specify the types of the parameters. The return type is written between the input parameter's closing parenthesis and the opening brace for the function body.

Just like other languages, Go has a `return` keyword for returning values from a function. If a function returns a value, you *must* supply a `return`. If a function returns nothing, a `return` statement is not needed at the end of the function. The `return` keyword is needed in a function that returns nothing only if you are exiting from the function before the last line.

The `main` function has no input parameters or return values. When a function has no input parameters, use empty parentheses (`()`). When a function returns nothing, you write nothing between the input parameter's closing parenthesis and the opening brace for the function body:

```
func main() {
    result := div(5, 2)
    fmt.Println(result)
}
```

The code for this simple program is in the *sample_code/simple_div* directory in the Chapter 5 repository.

Calling a function should be familiar to experienced developers. On the right side of the `:=`, you call the `div` function with the values 5 and 2. On the left side, you assign the returned value to the variable `result`.

When you have two or more consecutive input parameters of the same type, you can specify the type once for all of them like this:

```
func div(num, denom int) int {
```

Simulating Named and Optional Parameters

Before getting to the unique function features that Go has, I'll mention two that Go *doesn't* have: named and optional input parameters. With one exception that I will cover in the next section, you must supply all the parameters for a function. If you want to emulate named and optional parameters, define a struct that has fields that match the desired parameters, and pass the struct to your function. Example 5-1 shows a snippet of code that demonstrates this pattern.

Example 5-1. Using a struct to simulate named parameters

```
type MyFuncOpts struct {
    FirstName string
    LastName  string
    Age       int
}

func MyFunc(opts MyFuncOpts) error {
```

```
    // do something here
}

func main() {
    MyFunc(MyFuncOpts{
        LastName: "Patel",
        Age:      50,
    })
    MyFunc(MyFuncOpts{
        FirstName: "Joe",
        LastName:  "Smith",
    })
}
```

The code for this program is in the *sample_code/named_optional_parameters* directory in the Chapter 5 repository.

In practice, not having named and optional parameters isn't a limitation. A function shouldn't have more than a few parameters, and named and optional parameters are mostly useful when a function has many inputs. If you find yourself in that situation, your function is quite possibly too complicated.

Variadic Input Parameters and Slices

You've been using `fmt.Println` to print results to the screen and you've probably noticed that it allows any number of input parameters. How does it do that? Like many languages, Go supports *variadic parameters*. The variadic parameter *must* be the last (or only) parameter in the input parameter list. You indicate it with three dots (`...`) *before* the type. The variable that's created within the function is a slice of the specified type. You use it just like any other slice. Let's see how they work by writing a program that adds a base number to a variable number of parameters and returns the result as a slice of `int`. You can run this program on The Go Playground or in the *sample_code/variadic* directory in the Chapter 5 repository. First, write the variadic function:

```
func addTo(base int, vals ...int) []int {
    out := make([]int, 0, len(vals))
    for _, v := range vals {
        out = append(out, base+v)
    }
    return out
}
```

And now call it in a few ways:

```
func main() {
    fmt.Println(addTo(3))
    fmt.Println(addTo(3, 2))
    fmt.Println(addTo(3, 2, 4, 6, 8))
    a := []int{4, 3}
```

```
    fmt.Println(addTo(3, a...))
    fmt.Println(addTo(3, []int{1, 2, 3, 4, 5}...))
}
```

As you can see, you can supply however many values you want for the variadic parameter or no values at all. Since the variadic parameter is converted to a slice, you can supply a slice as the input. However, you must put three dots (`...`) *after* the variable or slice literal. If you do not, it is a compile-time error.

When you build and run this program, you get this output:

```
[]
[5]
[5 7 9 11]
[7 6]
[4 5 6 7 8]
```

Multiple Return Values

The first difference that you'll see between Go and other languages is that Go allows for multiple return values. Let's add a small feature to the previous division program. You're going to return both the dividend and the remainder from the function. Here's the updated function:

```
func divAndRemainder(num, denom int) (int, int, error) {
    if denom == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
    return num / denom, num % denom, nil
}
```

There are a few changes to support multiple return values. When a Go function returns multiple values, the types of the return values are listed in parentheses, separated by commas. Also, if a function returns multiple values, you must return all of them, separated by commas. Don't put parentheses around the returned values; that's a compile-time error.

There's something else that you haven't seen yet: creating and returning an `error`. If you want to learn more about errors, skip ahead to Chapter 9. For now, you need to know only that you use Go's multiple return value support to return an `error` if something goes wrong in a function. If the function completes successfully, you return `nil` for the error's value. By convention, the `error` is always the last (or only) value returned from a function.

Calling the updated function looks like this:

```
func main() {
    result, remainder, err := divAndRemainder(5, 2)
    if err != nil {
        fmt.Println(err)
```

```
        os.Exit(1)
    }
    fmt.Println(result, remainder)
}
```

The code for this program is in the *sample_code/updated_div* directory in the Chapter 5 repository.

I talked about assigning multiple values at once in "var Versus :=" on page 28. Here you are using that feature to assign the results of a function call to three variables. On the right side of the `:=`, you call the `divAndRemainder` function with the values 5 and 2. On the left side, you assign the returned values to the variables `result`, `remainder`, and `err`. You check to see if there was an error by comparing `err` to `nil`.

Multiple Return Values Are Multiple Values

If you are familiar with Python, you might think that multiple return values are like Python functions returning a tuple that's optionally destructured if the tuple's values are assigned to multiple variables. Example 5-2 shows some sample code run in the Python interpreter.

Example 5-2. Multiple return values in Python are destructured tuples

```
>>> def div_and_remainder(n,d):
...   if d == 0:
...     raise Exception("cannot divide by zero")
...   return n / d, n % d
>>> v = div_and_remainder(5,2)
>>> v
(2.5, 1)
>>> result, remainder = div_and_remainder(5,2)
>>> result
2.5
>>> remainder
1
```

In Python, you can assign all the values returned to a single variable or to multiple variables. That's not how Go works. You must assign each value returned from a function. If you try to assign multiple return values to one variable, you get a compile-time error.

Ignoring Returned Values

What if you call a function and don't want to use all the returned values? As covered in "Unused Variables" on page 32, Go does not allow unused variables. If a function returns multiple values, but you don't need to read one or more of the values, assign the unused values to the name `_`. For example, if you weren't

going to read `remainder`, you would write the assignment as `result, _, err := divAndRemainder(5, 2)`.

Surprisingly, Go does let you implicitly ignore *all* of the return values for a function. You can write `divAndRemainder(5,2)`, and the returned values are dropped. You have actually been doing this since the earliest examples: `fmt.Println` returns two values, but it is idiomatic to ignore them. In almost all other cases, you should make it explicit that you are ignoring return values by using underscores.

Use _ whenever you don't need to read a value that's returned by a function.

Named Return Values

In addition to letting you return more than one value from a function, Go allows you to specify *names* for your return values. Rewrite the `divAndRemainder` function one more time, this time using named return values:

```
func divAndRemainder(num, denom int) (result int, remainder int, err error) {
    if denom == 0 {
        err = errors.New("cannot divide by zero")
        return result, remainder, err
    }
    result, remainder = num/denom, num%denom
    return result, remainder, err
}
```

When you supply names to your return values, what you are doing is predeclaring variables that you use within the function to hold the return values. They are written as a comma-separated list within parentheses. *You must surround named return values with parentheses, even if there is only a single return value*. Named return values are initialized to their zero values when created. This means that you can return them before any explicit use or assignment.

If you want to name only *some* of the return values, you can do so by using _ as the name for any return values you want to remain nameless.

One important thing to note: the name that's used for a named returned value is local to the function; it doesn't enforce any name outside of the function. It is perfectly legal to assign the return values to variables of different names:

```
func main() {
    x, y, z := divAndRemainder(5, 2)
    fmt.Println(x, y, z)
}
```

While named return values can sometimes help clarify your code, they do have some potential corner cases. First is the problem of shadowing. Just as with any other variable, you can shadow a named return value. Be sure that you are assigning to the return value and not to a shadow of it.

The other problem with named return values is that you don't have to return them. Let's take a look at another variation on `divAndRemainder`. You can run it on The Go Playground or in the `divAndRemainderConfusing` function in *main.go* in the *sample_code/named_div* directory in the Chapter 5 repository:

```
func divAndRemainder(num, denom int) (result int, remainder int, err error) {
    // assign some values
    result, remainder = 20, 30
    if denom == 0 {
        return 0, 0, errors.New("cannot divide by zero")
    }
    return num / denom, num % denom, nil
}
```

Notice that you assigned values to `result` and `remainder` and then returned different values directly. Before running this code, try to guess what happens when 5 and 2 are passed to this function. The result might surprise you:

```
2 1
```

The values from the `return` statement were returned even though they were never assigned to the named return parameters. That's because the Go compiler inserts code that assigns whatever is returned to the return parameters. The named return parameters give a way to declare an *intent* to use variables to hold the return values, but don't *require* you to use them.

Some developers like to use named return parameters because they provide additional documentation. However, I find them of limited value. Shadowing makes them confusing, as does simply ignoring them. Named return parameters are essential in one situation. I will talk about that when I cover `defer` later in the chapter.

Blank Returns—Never Use These!

If you use named return values, you need to be aware of one severe misfeature in Go: blank (sometimes called *naked*) returns. If you have named return values, you can just write `return` without specifying the values that are returned. This returns the last values assigned to the named return values. Rewrite the `divAndRemainder` function one last time, this time using blank returns:

```
func divAndRemainder(num, denom int) (result int, remainder int, err error) {
    if denom == 0 {
        err = errors.New("cannot divide by zero")
        return
    }
    result, remainder = num/denom, num%denom
    return
}
```

Using blank returns makes a few additional changes to the function. When there's invalid input, the function returns immediately. Since no values were assigned to `result` and `remainder`, their zero values are returned. If you are returning the zero values for your named return values, be sure they make sense. Also note that you still have to put a `return` at the end of the function. Even though the function is using blank returns, this function returns values. Leaving out `return` is a compile-time error.

At first, you might find blank returns handy since they allow you to avoid some typing. However, most experienced Go developers consider blank returns a bad idea because they make it harder to understand data flow. Good software is clear and readable; it's obvious what is happening. When you use a blank return, the reader of your code needs to scan back through the program to find the last value assigned to the return parameters to see what is actually being returned.

If your function returns values, *never* use a blank return. It can make it very confusing to figure out what value is actually returned.

Functions Are Values

Just as in many other languages, functions in Go are values. The type of a function is built out of the keyword `func` and the types of the parameters and return values. This combination is called the *signature* of the function. Any function that has the exact same number and types of parameters and return values meets the type signature.

Since functions are values, you can declare a function variable:

```
var myFuncVariable func(string) int
```

`myFuncVariable` can be assigned any function that has a single parameter of type `string` and returns a single value of type `int`. Here's a longer example:

```
func f1(a string) int {
    return len(a)
}

func f2(a string) int {
    total := 0
    for _, v := range a {
        total += int(v)
    }
    return total
}

func main() {
    var myFuncVariable func(string) int
    myFuncVariable = f1
    result := myFuncVariable("Hello")
    fmt.Println(result)

    myFuncVariable = f2
    result = myFuncVariable("Hello")
    fmt.Println(result)
}
```

You can run this program on The Go Playground or in the *sample_code/func_value* directory in the Chapter 5 repository. The output is:

```
5
500
```

The default zero value for a function variable is `nil`. Attempting to run a function variable with a `nil` value results in a panic (which is covered in "panic and recover" on page 218).

Having functions as values allows you to do some clever things, such as build a simple calculator using functions as values in a map. Let's see how this works. The following code is available on The Go Playground or in the *sample_code/calculator* directory in the Chapter 5 repository. First, create a set of functions that all have the same signature:

```
func add(i int, j int) int { return i + j }

func sub(i int, j int) int { return i - j }

func mul(i int, j int) int { return i * j }

func div(i int, j int) int { return i / j }
```

Next, create a map to associate a math operator with each function:

```
var opMap = map[string]func(int, int) int{
    "+": add,
    "-": sub,
    "*": mul,
    "/": div,
}
```

Finally, try out the calculator with a few expressions:

```
func main() {
    expressions := [][]string{
        {"2", "+", "3"},
        {"2", "-", "3"},
        {"2", "*", "3"},
        {"2", "/", "3"},
        {"2", "%", "3"},
        {"two", "+", "three"},
        {"5"},
    }
    for _, expression := range expressions {
        if len(expression) != 3 {
            fmt.Println("invalid expression:", expression)
            continue
        }
        p1, err := strconv.Atoi(expression[0])
        if err != nil {
            fmt.Println(err)
            continue
        }
        op := expression[1]
        opFunc, ok := opMap[op]
        if !ok {
            fmt.Println("unsupported operator:", op)
            continue
        }
        p2, err := strconv.Atoi(expression[2])
        if err != nil {
            fmt.Println(err)
            continue
        }
        result := opFunc(p1, p2)
        fmt.Println(result)
    }
}
```

You're using the `strconv.Atoi` function in the standard library to convert a `string` to an `int`. The second value returned by this function is an `error`. Just as before, you check for errors that are returned by functions and handle error conditions properly.

You use `op` as the key to the `opMap` map, and assign the value associated with the key to the variable `opFunc`. The type of `opFunc` is `func(int, int) int`. If there wasn't a

function in the map associated with the provided key, you print an error message and skip the rest of the loop. You then call the function assigned to the `opFunc` variable with the `p1` and `p2` variables that you decoded earlier. Calling a function in a variable looks just like calling a function directly.

When you run this program, you can see the simple calculator at work:

```
5
-1
6
0
unsupported operator: %
strconv.Atoi: parsing "two": invalid syntax
invalid expression: [5]
```

Don't write fragile programs. The core logic for this example is relatively short. Of the 22 lines inside the `for` loop, 6 of them implement the actual algorithm, and the other 16 are for error checking and data validation. You might be tempted to not validate incoming data or check errors, but doing so produces unstable, unmaintainable code. Error handling is what separates the professionals from the amateurs.

Function Type Declarations

Just as you can use the `type` keyword to define a `struct`, you can use it to define a function type too (I'll go into more details on type declarations in Chapter 7):

```
type opFuncType func(int,int) int
```

You can then rewrite the `opMap` declaration to look like this:

```
var opMap = map[string]opFuncType {
    // same as before
}
```

You don't have to modify the functions at all. Any function that has two input parameters of type `int` and a single return value of type `int` automatically meets the type and can be assigned as a value in the map.

What's the advantage of declaring a function type? One use is documentation. It's useful to give something a name if you are going to refer to it multiple times. You will see another use in "Function Types Are a Bridge to Interfaces" on page 173.

Anonymous Functions

You can not only assign functions to variables, but also define new functions within a function and assign them to variables. Here's a program to demonstrate this, which

you can run on The Go Playground. The code is also available in the *sample_code/anon_func* directory in the Chapter 5 repository:

```
func main() {
    f := func(j int) {
        fmt.Println("printing", j, "from inside of an anonymous function")
    }
    for i := 0; i < 5; i++ {
        f(i)
    }
}
```

Inner functions are *anonymous*; they don't have a name. You declare an anonymous function with the keyword `func` immediately followed by the input parameters, the return values, and the opening brace. It is a compile-time error to try to put a function name between `func` and the input parameters.

Just like any other function, an anonymous function is called by using parentheses. In this example, you are passing the `i` variable from the `for` loop in here. It is assigned to the `j` input parameter of the anonymous function.

The program gives the following output:

```
printing 0 from inside of an anonymous function
printing 1 from inside of an anonymous function
printing 2 from inside of an anonymous function
printing 3 from inside of an anonymous function
printing 4 from inside of an anonymous function
```

You don't have to assign an anonymous function to a variable. You can write them inline and call them immediately. The previous program can be rewritten into this:

```
func main() {
    for i := 0; i < 5; i++ {
        func(j int) {
            fmt.Println("printing", j, "from inside of an anonymous function")
        }(i)
    }
}
```

You can run this example on The Go Playground or in the *sample_code/anon_func* directory in the Chapter 5 repository.

Now, this is not something that you would normally do. If you are declaring and executing an anonymous function immediately, you might as well get rid of the anonymous function and just call the code. However, declaring anonymous functions without assigning them to variables is useful in two situations: `defer` statements and launching goroutines. I'll talk about `defer` statements in a bit. Goroutines are covered in Chapter 12.

Since you can declare variables at the package scope, you can also declare package scope variables that are assigned anonymous functions:

```
var (
    add = func(i, j int) int { return i + j }
    sub = func(i, j int) int { return i - j }
    mul = func(i, j int) int { return i * j }
    div = func(i, j int) int { return i / j }
)

func main() {
    x := add(2, 3)
    fmt.Println(x)
}
```

Unlike a normal function definition, you can assign a new value to a package-level anonymous function:

```
func main() {
    x := add(2, 3)
    fmt.Println(x)
    changeAdd()
    y := add(2, 3)
    fmt.Println(y)
}

func changeAdd() {
    add = func(i, j int) int { return i + j + j }
}
```

Running this code gives the following output:

```
5
8
```

You can try this sample on The Go Playground. The code is also in the *sample_code/package_level_anon* directory in the Chapter 5 repository.

Before using a package-level anonymous function, be very sure you need this capability. Package-level state should be immutable to make data flow easier to understand. If a function's meaning changes while a program is running, it becomes difficult to understand not just how data flows, but how it is processed.

Closures

Functions declared inside functions are special; they are *closures*. This is a computer science word that means that functions declared inside functions are able to access and modify variables declared in the outer function. Let's look at a quick example to see how this works. You can find the code on The Go Playground. The code is also in the *sample_code/closure* directory in the Chapter 5 repository:

```
func main() {
    a := 20
    f := func() {
        fmt.Println(a)
        a = 30
    }
    f()
    fmt.Println(a)
}
```

Running this program gives the following output:

```
20
30
```

The anonymous function assigned to `f` can read and write `a`, even though `a` is not passed in to the function.

Just as with any inner scope, you can shadow a variable inside a closure. If you change the code to:

```
func main() {
    a := 20
    f := func() {
        fmt.Println(a)
        a := 30
        fmt.Println(a)
    }
    f()
    fmt.Println(a)
}
```

this will print out the following:

```
20
30
20
```

Using `:=` instead of `=` inside the closure creates a new `a` that ceases to exist when the closure exits. When working with inner functions, be careful to use the correct assignment operator, especially when multiple variables are on the lefthand side. You can try this code on The Go Playground. The code is also in the *sample_code/closure_shadow* directory in the Chapter 5 repository.

This inner function and closure stuff might not seem all that useful at first. What benefit do you get from making mini-functions within a larger function? Why does Go have this feature?

One thing that closures allow you to do is to limit a function's scope. If a function is going to be called from only one other function, but it's called multiple times, you can use an inner function to "hide" the called function. This reduces the number of declarations at the package level, which can make it easier to find an unused name.

If you have a piece of logic that is repeated multiple times within a function, a closure can be used to remove that repetition. I wrote a simple Lisp interpreter with a `Scan` function that processes an input string to find the parts of a Lisp program. It relies on two closures, `buildCurToken` and `update`, to make the code shorter and easier to understand. You can see it on GitHub.

Closures really become interesting when they are passed to other functions or returned from a function. They allow you to take the variables within your function and use those values *outside* of your function.

Passing Functions as Parameters

Since functions are values and you can specify the type of a function using its parameter and return types, you can pass functions as parameters into functions. If you aren't used to treating functions like data, you might need a moment to think about the implications of creating a closure that references local variables and then passing that closure to another function. It's a very useful pattern that appears several times in the standard library.

One example is sorting slices. The `sort` package in the standard library has a function called `sort.Slice`. It takes in any slice and a function that is used to sort the slice that's passed in. Let's see how it works by sorting a slice of a struct using two different fields.

The `sort.Slice` function predates the addition of generics to Go, so it does some internal magic to make it work with any kind of slice. I'll talk about this magic more in Chapter 16.

Let's see how to use closures to sort the same data different ways. You can run this code on The Go Playground. The code is also in the *sample_code/sort_sample* directory in the Chapter 5 repository. First, define a simple type, a slice of values of that type, and print out the slice:

```
type Person struct {
    FirstName string
    LastName  string
    Age       int
}

people := []Person{
    {"Pat", "Patterson", 37},
    {"Tracy", "Bobdaughter", 23},
    {"Fred", "Fredson", 18},
}
fmt.Println(people)
```

Next, sort the slice by last name and print out the results:

```
// sort by last name
sort.Slice(people, func(i, j int) bool {
    return people[i].LastName < people[j].LastName
})
fmt.Println(people)
```

The closure that's passed to `sort.Slice` has two parameters, `i` and `j`, but within the closure, `people` is used, so you can sort it by the `LastName` field. In computer science terms, `people` is *captured* by the closure. Next you do the same, sorting by the `Age` field:

```
// sort by age
sort.Slice(people, func(i, j int) bool {
    return people[i].Age < people[j].Age
})
fmt.Println(people)
```

Running this code gives the following output:

```
[{Pat Patterson 37} {Tracy Bobdaughter 23} {Fred Fredson 18}]
[{Tracy Bobdaughter 23} {Fred Fredson 18} {Pat Patterson 37}]
[{Fred Fredson 18} {Tracy Bobdaughter 23} {Pat Patterson 37}]
```

The `people` slice is changed by the call to `sort.Slice`. I talk about this briefly in "Go Is Call by Value" on page 114 and in more detail in the next chapter.

Passing functions as parameters to other functions is often useful for performing different operations on the same kind of data.

Returning Functions from Functions

In addition to using a closure to pass some function state to another function, you can also return a closure from a function. Let's demonstrate this by writing a function that returns a multiplier function. You can run this program on The Go Playground. The code is also in the *sample_code/makeMult* directory in the Chapter 5 repository. Here is a function that returns a closure:

```
func makeMult(base int) func(int) int {
    return func(factor int) int {
        return base * factor
    }
}
```

And here is how the function is used:

```
func main() {
    twoBase := makeMult(2)
    threeBase := makeMult(3)
    for i := 0; i < 3; i++ {
        fmt.Println(twoBase(i), threeBase(i))
    }
}
```

Running this program gives the following output:

```
0 0
2 3
4 6
```

Now that you've seen closures in action, you might wonder how often they are used by Go developers. It turns out that they are surprisingly useful. You saw how they are used to sort slices. A closure is also used to efficiently search a sorted slice with `sort.Search`. As for returning closures, you will see this pattern used when you build middleware for a web server in "Middleware" on page 340. Go also uses closures to implement resource cleanup, via the `defer` keyword.

If you spend any time with programmers who use functional programming languages like Haskell, you might hear the term *higher-order functions*. That's a fancy way to say that a function has a function for an input parameter or a return value. As a Go developer, you are as cool as they are!

defer

Programs often create temporary resources, like files or network connections, that need to be cleaned up. This cleanup has to happen, no matter how many exit points a function has, or whether a function completed successfully or not. In Go, the cleanup code is attached to the function with the `defer` keyword.

Let's take a look at how to use `defer` to release resources. You'll do this by writing a simple version of `cat`, the Unix utility for printing the contents of a file. You can't pass in arguments on The Go Playground, but you can find the code for this example in the *sample_code/simple_cat* directory in the Chapter 5 repository:

```
func main() {
    if len(os.Args) < 2 {
```

```
        log.Fatal("no file specified")
    }
    f, err := os.Open(os.Args[1])
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()
    data := make([]byte, 2048)
    for {
        count, err := f.Read(data)
        os.Stdout.Write(data[:count])
        if err != nil {
            if err != io.EOF {
                log.Fatal(err)
            }
            break
        }
    }
}
```

This example introduces a few new features that I cover in more detail in later chapters. Feel free to read ahead to learn more.

First, you make sure that a filename was specified on the command line by checking the length of `os.Args`, a slice in the `os` package. The first value in `os.Args` is the name of the program. The remaining values are the arguments passed to the program. You check that the length of `os.Args` is at least 2 to determine whether the argument to the program was provided. If it wasn't, use the `Fatal` function in the `log` package to print a message and exit the program. Next, you acquire a read-only file handle with the `Open` function in the `os` package. The second value that's returned by `Open` is an error. If there's a problem opening the file, you print the error message and exit the program. As mentioned earlier, I'll talk about errors in Chapter 9.

Once you know there is a valid file handle, you need to close it after you use it, no matter how you exit the function. To ensure that the cleanup code runs, you use the `defer` keyword, followed by a function or method call. In this case, you use the `Close` method on the file variable. (I cover `at` methods in Go in Chapter 7.) Normally, a function call runs immediately, but `defer` delays the invocation until the surrounding function exits.

You read from a file handle by passing a slice of bytes into the `Read` method on a file variable. I'll cover how to use this method in detail in "io and Friends" on page 319, but `Read` returns the number of bytes that were read into the slice and an error. If an error occurs, you check whether it's an end-of-file marker. If you are at the end of the file, you use `break` to exit the `for` loop. For all other errors, you report it and exit immediately, using `log.Fatal`. I'll talk a little more about slices and function

parameters in "Go Is Call by Value" on page 114 and go into details on this pattern when I discuss pointers in the next chapter.

Building and running the program from within the *simple_cat* directory produces the following result:

```
$ go build
$ ./simple_cat simple_cat.go
package main

import (
    "io"
    "log"
    "os"
)
...
```

You should know a few more things about `defer`. First, you can use a function, method, or closure with `defer`. (When I mention functions with `defer`, mentally expand that to "functions, methods, or closures.")

You can `defer` multiple functions in a Go function. They run in last-in, first-out (LIFO) order; the last `defer` registered runs first.

The code within `defer` functions runs *after* the return statement. As I mentioned, you can supply a function with input parameters to a `defer`. The input parameters are evaluated immediately and their values are stored until the function runs.

Here's a quick example that demonstrates both of these features of `defer`:

```
func deferExample() int {
    a := 10
    defer func(val int) {
        fmt.Println("first:", val)
    }(a)
    a = 20
    defer func(val int) {
        fmt.Println("second:", val)
    }(a)
    a = 30
    fmt.Println("exiting:", a)
    return a
}
```

Running this code gives the following output:

```
exiting: 30
second: 20
first: 10
```

You can run this on The Go Playground. It is also available in the *sample_code/defer_example* directory in the Chapter 5 repository.

You can supply a function that returns values to a `defer`, but there's no way to read those values:

```
func example() {
    defer func() int {
        return 2 // there's no way to read this value
    }()
}
```

You might be wondering whether there's a way for a deferred function to examine or modify the return values of its surrounding function. There is, and it's the best reason to use named return values. It allows your code to take actions based on an error. When I talk about errors in Chapter 9, I will discuss a pattern that uses `defer` to add contextual information to an error returned from a function. Let's look at a way to handle database transaction cleanup using named return values and `defer`:

```
func DoSomeInserts(ctx context.Context, db *sql.DB, value1, value2 string)
                  (err error) {
    tx, err := db.BeginTx(ctx, nil)
    if err != nil {
        return err
    }
    defer func() {
        if err == nil {
            err = tx.Commit()
        }
        if err != nil {
            tx.Rollback()
        }
    }()
    _, err = tx.ExecContext(ctx, "INSERT INTO FOO (val) values $1", value1)
    if err != nil {
        return err
    }
    // use tx to do more database inserts here
    return nil
}
```

I'm not going to cover Go's database support in this book, but the standard library includes extensive support for databases in the `database/sql` package. In the example function, you create a transaction to do a series of database inserts. If any of them fails, you want to roll back (not modify the database). If all of them succeed, you want to commit (store the database changes). You use a closure with `defer` to check whether `err` has been assigned a value. If it hasn't, you run `tx.Commit`, which could also return an error. If it does, the value `err` is modified. If any database interaction returned an error, you call `tx.Rollback`.

New Go developers tend to forget the set of parentheses after the braces when specifying a function for `defer`. It is a compile-time error to leave them out, and eventually the habit sets in. It helps to remember that supplying parentheses allows you to specify values that will be passed into the function when it runs.

A common pattern in Go is for a function that allocates a resource to also return a closure that cleans up the resource. In the *sample_code/simple_cat_cancel* directory in the Chapter 5 repository, there is a rewrite of the simple `cat` program that does this. First you write a helper function that opens a file and returns a closure:

```
func getFile(name string) (*os.File, func(), error) {
    file, err := os.Open(name)
    if err != nil {
        return nil, nil, err
    }
    return file, func() {
        file.Close()
    }, nil
}
```

The helper function returns a file, a function, and an error. That `*` means that a file reference in Go is a pointer. I'll talk more about that in the next chapter.

Now in `main`, you use the `getFile` function:

```
f, closer, err := getFile(os.Args[1])
if err != nil {
    log.Fatal(err)
}
defer closer()
```

Because Go doesn't allow unused variables, returning the `closer` from the function means that the program will not compile if the function is not called. That reminds the user to use `defer`. As covered earlier, you put parentheses after `closer` when you `defer` it.

Using `defer` can feel strange if you are used to a language that uses a block within a function to control when a resource is cleaned up, like the `try/catch/finally` blocks in Java, JavaScript, and Python or the `begin/rescue/ensure` blocks in Ruby.

The downside to these resource-cleanup blocks is that they create another level of indentation in your function, and that makes the code harder to read. It's not just my opinion that nested code is harder to follow. In research described in a 2017 paper in *Empirical Software Engineering*, Vard Antinyan et al. discovered that "Of… eleven proposed code characteristics, only two markedly influence complexity growth: the nesting depth and the lack of structure."

Research on what makes a program easier to read and understand isn't new. You can find papers that are many decades old, including a paper from 1983 by Richard Miara et al. that tries to figure out the right amount of indentation to use (according to their results, two to four spaces).

Go Is Call by Value

You might hear people say that Go is a *call-by-value* language and wonder what that means. It means that when you supply a variable for a parameter to a function, Go *always* makes a copy of the value of the variable. Let's take a look. You can run this code on The Go Playground. It is also in the *sample_code/pass_value_type* directory in the Chapter 5 repository. First, you define a simple struct:

```
type person struct {
    age  int
    name string
}
```

Next, you write a function that takes in an `int`, a `string`, and a `person`, and modifies their values:

```
func modifyFails(i int, s string, p person) {
    i = i * 2
    s = "Goodbye"
    p.name = "Bob"
}
```

You then call this function from `main` and see whether the modifications stick:

```
func main() {
    p := person{}
    i := 2
    s := "Hello"
    modifyFails(i, s, p)
    fmt.Println(i, s, p)
}
```

As the name of the function indicates, running this code shows that a function won't change the values of the parameters passed into it:

```
2 Hello {0 }
```

I included the `person` struct to show that this isn't true just for primitive types. If you have programming experience in Java, JavaScript, Python, or Ruby, you might find the struct behavior strange. After all, those languages let you modify the fields in an object when you pass an object as a parameter to a function. The reason for the difference is something I will cover when I talk about pointers.

The behavior is a little different for maps and slices. Let's see what happens when you try to modify them within a function. You can run this code on The Go Playground. It is also in the *sample_code/pass_map_slice* directory in the Chapter 5 repository. You're going to write a function to modify a map parameter and a function to modify a slice parameter:

```
func modMap(m map[int]string) {
    m[2] = "hello"
    m[3] = "goodbye"
    delete(m, 1)
}

func modSlice(s []int) {
    for k, v := range s {
        s[k] = v * 2
    }
    s = append(s, 10)
}
```

You then call these functions from `main`:

```
func main() {
    m := map[int]string{
        1: "first",
        2: "second",
    }
    modMap(m)
    fmt.Println(m)

    s := []int{1, 2, 3}
    modSlice(s)
    fmt.Println(s)
}
```

When you run this code, you'll see something interesting:

```
map[2:hello 3:goodbye]
[2 4 6]
```

For the map, it's easy to explain what happens: any changes made to a map parameter are reflected in the variable passed into the function. For a slice, it's more

complicated. You can modify any element in the slice, but you can't lengthen the slice. This is true for maps and slices that are passed directly into functions as well as map and slice fields in structs.

This program leads to the question: why do maps and slices behave differently than other types? It's because maps and slices are both implemented with pointers. I'll go into more detail in the next chapter.

Every type in Go is a value type. It's just that sometimes the value is a pointer.

Call by value is one reason that Go's limited support for constants is only a minor handicap. Since variables are passed by value, you can be sure that calling a function doesn't modify the variable whose value was passed in (unless the variable is a slice or map). In general, this is a good thing. It makes it easier to understand the flow of data through your program when functions don't modify their input parameters and instead return newly computed values.

While this approach is easy to understand, at times you need to pass something mutable to a function. What do you do then? That's when you need a pointer.

Exercises

These exercises test your knowledge of functions in Go and how to use them. Solutions are available in the *exercise_solutions* directory in the Chapter 5 repository.

1. The simple calculator program doesn't handle one error case: division by zero. Change the function signature for the math operations to return both an `int` and an `error`. In the `div` function, if the divisor is `0`, return `errors.New("division by zero")` for the error. In all other cases, return `nil`. Adjust the `main` function to check for this error.
2. Write a function called `fileLen` that has an input parameter of type `string` and returns an `int` and an `error`. The function takes in a filename and returns the number of bytes in the file. If there is an error reading the file, return the error. Use `defer` to make sure the file is closed properly.
3. Write a function called `prefixer` that has an input parameter of type `string` and returns a function that has an input parameter of type `string` and returns a `string`. The returned function should prefix its input with the string passed into `prefixer`. Use the following `main` function to test `prefixer`:

```
func main() {
    helloPrefix := prefixer("Hello")
    fmt.Println(helloPrefix("Bob")) // should print Hello Bob
    fmt.Println(helloPrefix("Maria")) // should print Hello Maria
}
```

Wrapping Up

In this chapter, you've looked at functions in Go, how they are similar to functions in other languages, and their unique features. In the next chapter, you're going to look at pointers, find out that they aren't nearly as scary as many new Go developers expect them to be, and learn how to take advantage of them to write efficient programs.