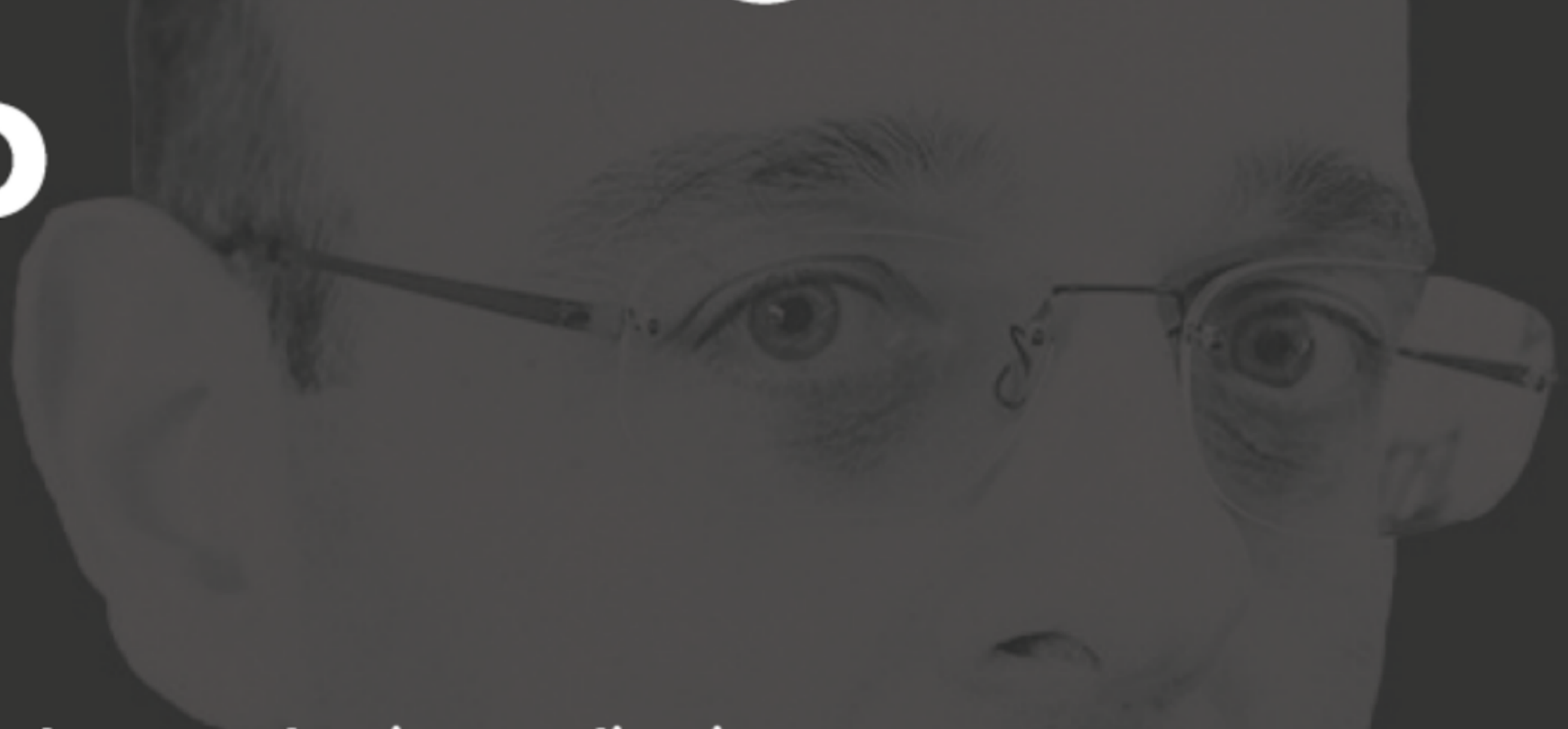
EXPERT INSIGHT

Mihalis Tsoukalos

Mastering Go

Create Golang production applications using network libraries, concurrency, and advanced Go data structures



Packt

Mastering Go

Acquisition Editors: Frank Pohlmann, Suresh Jain
Project Editor: Kishor Rit
Content Development Editor: Gary Schwarts
Technical Editors: Gaurav Gavas, Nidhisha Shetty
Proofreader: Tom Jacob
Indexer: Mariammal Chettiyar
Graphics: Tom Scaria
Production Coordinator: Shantanu Zagade

First published: April 2018

Production reference: 1270418

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham
B3 2PB, UK.

ISBN 978-1-78862-654-5

www.packtpub.com

Table of Contents

Preface

The book you are reading right now is called *Mastering Go* and is all about helping you become a better Go developer!

I tried to include the right amount of theory and hands on practice, but only you, the reader, can tell if I succeeded or not! Additionally, all presented examples are self-contained, which means that they can be used on their own or as templates for creating more complex applications.

Please try to do the exercises located at the end of each chapter and do not hesitate to contact me with ways to make any future editions of this book even better!

Who this book is for

This book is for amateur and intermediate Go programmers who want to take their Go knowledge to the next level as well as for experienced developers in other programming languages who want to learn Go without learning again how a `for` loop works.

Some of the information found in this book can be also found in my other book, *Go Systems Programming* by *Packt Publishing*. The main difference between these two books is that *Go Systems Programming* is about developing system tools using the capabilities of Go, whereas *Mastering Go* is about explaining the capabilities and the internals of Go in order to become a better Go developer. Both books can be used as a reference after reading them for the first or the second time.

What this book covers

`Chapter 1`, *Go and the Operating System*, begins by talking about the history of Go and the advantages of Go before describing the `godoc` utility and explaining how you can compile and execute Go programs. After that, it talks about printing the output and getting user input, working with the command-line arguments of a program, and using log files. The last topic of the first chapter is error handling, which plays a key role in Go.

`Chapter 2`, *Understanding Go Internals*, discusses the Go garbage collector and the way it operates. Then it talks about unsafe code and the unsafe package, how to call C code from a Go program, and how to call Go code from a C program. After that, it showcases the use of the `defer` keyword and presents the `strace(1)` and `dtrace(1)` utilities. In the remaining sections of the chapter, you will learn how to find information about your Go environment and the use of the Go assembler.

`Chapter 3`, *Working with Basic Go Data Types*, talks about the data types offered by Go, which includes arrays, slices, and maps as well as Go pointers, constants, loops, and working with dates and times. You would not want to miss this chapter!

`Chapter 4`, *The Uses of Composite Types*, begins by teaching you about Go structures and the `struct` keyword before discussing tuples, strings, runes, byte slices, and string literals. The rest of the chapter talks about regular expressions and pattern matching, the switch statement, the strings package, the `math/big` package, and about developing a key-value store in Go.

`Chapter 5`, *Enhancing Go Code with Data Structures*, is about developing your own data structures when the structures offered by Go do not fit a particular problem. This includes developing binary trees, linked lists, hash tables, stacks, and queues and learning about their advantages. This chapter also showcases the use of the structures found in the container standard Go package. The last topic of this chapter is random number generation.

`Chapter 6`, *What You Might Not Know About Go Packages*, is all about packages and functions, which also includes the use of the `init()` function, the `syscall` standard Go package, and the `text/template` and `html/template` packages. This chapter will definitely make you a better Go developer!

`Chapter 7`, *Reflection and Interfaces for All Seasons*, discusses three advanced Go concepts: reflection, interfaces, and type methods. The last part of the chapter is about object oriented programming in Go!

`Chapter 8`, *Telling a Unix System What to Do*, is about systems programming in Go, which includes subjects such as the `flag` package for working with command-line arguments, handling Unix signals, file input and output, the bytes package, and the `io.Reader` and `io.Writer` interfaces. As I told you before, if you are really into systems programming in Go, then getting *Go Systems Programming* after reading *Mastering Go* is highly recommended!

`Chapter 9`, *Concurrency in Go – Goroutines, Channels, and Pipelines*, discusses goroutines, channels and pipelines, which is the Go way of achieving concurrency. You will also learn about the differences between processes, threads, and goroutines, and the `sync` package and the way the Go scheduler operates.

`Chapter 10`, *Concurrency in Go – Advanced Topics*, will continue from the point where the previous chapter left off and make you a master of goroutines and channels! You will learn more about the Go scheduler, the use of the powerful `select` keyword and the various types of Go channels as well as shared memory, mutexes, the `sync.Mutex` type, and the `sync.RWMutex` type. The last part of the chapter will talk about the `context` package, worker pools, and how to detect race conditions.

`Chapter 11`, *Code Testing, Optimization, and Profiling*, discusses code testing, code optimization, and code profiling as well as about cross compilation, creating documentation, benchmarking Go code, creating example function, and finding unreachable Go code.

`Chapter 12`, *The Foundations of Network Programming in Go*, is all about the `net/http` package and how you can develop web clients and web servers in Go. This also includes the use of the `http.Response`, `http.Request` and `http.Transport` structures and the `http.NewServeMux` type. You will even learn how to develop an entire website in Go! Furthermore, in this chapter, you will learn how to read the configuration of your network interfaces and how to perform DNS lookups in Go.

`Chapter 13`, *Network Programming – Building Your Own Servers and Clients*, talks about creating UDP and TCP servers and clients in Go, using the functionality offered by the `net` package. Other topics included in this chapter are how to create RPC clients and servers as well as develop a concurrent TCP server in Go and read raw network packages!

To get the most out of this book

This book requires any Unix machine with a relatively recent Go version installed, which includes any machine running Mac OS X, macOS or Linux. Most of the presented code will also work on Microsoft Windows machines.

To get the most out of this book, you should try to apply the knowledge of each chapter in your own programs as soon as possible, and see what works and what does not! As I told you before, try to solve the exercises found at the end of each chapter, or create your own programming problems.

Download the example code files

You can download the example code files for this book from your account at `www.packtpub.com`. If you purchased this book elsewhere, you can visit `www.packtpub.com/support` and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at `www.packtpub.com`.
2. Select the **SUPPORT** tab.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at `https://github.com/PacktPublishing/Mastering-Go`. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at `https://github.com/PacktPublishing/`. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: `https://www.packtpub.com/sites/default/files/downloads/MasteringGo_ColorImages.pdf`.

Conventions used

There are a number of text conventions used throughout this book.

`CodeInText`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The first way is similar to using the `man(1)` command, but for Go functions and packages."

A block of code is set as follows:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("This is a sample Go program!")
}
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("This is a sample Go program!")
}
```

Any command-line input or output is written as follows:

```
$ date
Sat Oct 21 20:09:20 EEST 2017
$ go version
go version go1.9.1 darwin/amd64
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this. Here is an example: "Select **System info** from the **Administration** panel."

Warnings or important notes appear like this.

Tips and tricks appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: Email `feedback@packtpub.com` and mention the book title in the subject of your message. If you have questions about any aspect of this book, please email us at `questions@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit `www.packtpub.com/submit-errata`, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packtpub.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit `authors.packtpub.com`.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit `packtpub.com`.

1
Go and the Operating System

This chapter will serve as an introduction to various Go topics that may appear slightly ingenuous and naïve at first. The topics contained in this chapter, however, will be used throughout the entire book, so you'll need to make sure that you completely understand them. As happens with most practical subjects, the best way to understand something is to experiment with it. In this case, experimenting means writing Go code on your own, making your own mistakes, and learning from them! Just don't let the error messages discourage you.

In the first chapter, you will learn the following topics:

- The history of the Go programming language
- The reasons that Go is a good choice for developing your applications
- Compiling Go code
- Executing Go code
- Downloading and using external Go packages
- Unix standard input, output, and error
- Printing data on the screen
- Getting user input
- Printing data to standard error
- Working with log files
- Dealing with error handling in Go

The structure of the book

Mastering Go can be divided into three logical parts. The first part consists of four chapters, and it takes a sophisticated look at some important Go concepts, including user input and output, downloading external Go packages, compiling Go code, calling C code from Go, as well as using Go basic types and Go composite types.

The second part consists of three chapters that deal with Go code organization, the design of Go projects, and some advanced features of Go, respectively.

The third part includes the remaining six chapters and deals with the more practical Go topics, including systems programming in Go, concurrency in Go, code testing, optimization, and profiling. The last two chapters of this book will also talk about network programming in Go.

The book will present relatively small, yet complete Go programs that illustrate the concepts presented. This has two main advantages: first, you do not have to look at an endless code listing when trying to learn a single technique, and second, you can use this code as a starting point when creating your own applications and utilities.

Note that the focus of this book is machines that run a variant of the **Unix** operating system; this does not mean that the Go code presented will not run on Microsoft Windows machines—after all Go is portable! It just means that the included Go code has been tested on various Unix variants, mainly on **macOS** High Sierra and Debian **Linux**.

The history of Go

Go is a modern, generic purpose open-source programming language that was officially announced at the end of 2009. It began as an internal Google project, which means that it was started as an experiment, and it is inspired by many other programming languages, including **C**, **Pascal**, **Alef**, and **Oberon**. Its spiritual fathers are *Robert Griesemer*, *Ken Thomson*, and *Rob Pike*, who are professional programmers who designed Go as a language for professional programmers who want to build reliable, robust, and efficient software. Apart from its syntax and its standard functions, Go comes with a pretty rich standard library.

At the time of writing this chapter, the current stable Go version is 1.9.1, but version 1.9.2 is on its way:

```
$ date
Sat Oct 21 20:09:20 EEST 2017
$ go version
go version go1.9.1 darwin/amd64
```

I am pretty confident that by the time this book is published, the output of the `go version` command will be different! The good news is that due to the way that Go progresses, this book will remain relevant for many years!

If you are installing Go for the first time, you can start by visiting `https://golang.org/dl/`. However, there is a good chance that your Unix variant already has a ready-to-install package for the Go programming language, so you might want to get Go, using your favorite package manager.

Why learn Go?

Go is a modern programming language that allows you to write safe code without silly bugs—do not worry, you can still create complex bugs! Most of all, though, Go wants to have happy developers; therefore, by design, Go code looks attractive and familiar, and it is easy to write.

The next section talks more analytically about the advantages of Go.

Go advantages

Go has many advantages—some of them are unique to Go, while others are shared with other programming languages.

The most significant Go advantages and features are as follows:

- Go is a modern programming language that was created by experienced developers.
- Go release candidates are used first by Google staff for production use!
- Go code is easy to read and easy to understand.

- Go wants happy developers, because a happy developer writes better code!
- The Go compiler prints practical warning and error messages that help you solve the actual problem. Put simply, the Go compiler is here to help you, and not to make your life miserable by printing pointless output!
- Go code is portable, especially between Unix machines.
- Go has support for procedural, concurrent, and distributed programming.
- Go supports **Garbage Collection**, so you do not have to deal with memory allocation and deallocation.
- Go does not have a **preprocessor**. It does high-speed compilation. As a consequence, Go can also be used as a scripting language.
- Go can build web applications, and it provides a simple web server for testing purposes.
- The standard Go library offers many packages that simplify the work of the developer. Additionally, the functions found in the standard Go library are tested and debugged in advance by the people who develop Go, which means that most of the time these functions come without bugs.
- Go uses **static linking** by default, which means that the binary files produced can be easily transferred to other machines with the same OS. As a consequence, once a Go program is compiled successfully and an executable file is generated, the developer does not need to worry about libraries, dependencies, and different library versions anymore.
- You will not need a GUI for developing, debugging, and testing Go applications, as Go can be used from the command-line, which many Unix people prefer.
- Go supports **Unicode**, which means that you do not need any extra code for printing characters from multiple human languages.
- Go keeps concepts orthogonal, because a few orthogonal features work better than many overlapping ones.

Is Go perfect?

There is no such thing as the perfect programming language, and Go is no exception to this rule. However, some programming languages are better at some areas of programming, or we just like them more than other programming languages. Personally, I do not like Java, and while I used to like C++, I do not like it anymore. This is mainly because I find the look of Java and C++ code to be unpleasant.

Some of the disadvantages of Go are as follows:

- Go does not have direct support for **object-oriented programming** (**OOP**), which can be a problem for programmers who are used to writing code in an object-oriented manner. Nevertheless, you can use composition in Go to mimic inheritance.
- For some people who still prefer C, Go will never replace C!
- C is still faster than any other programming language for systems programming, mainly because Unix is written in C.

Nevertheless, Go is a pretty decent and modern programming language that will not disappoint if you find the time to learn it and program in it.

What is a preprocessor?

Earlier, I said that Go does not have a preprocessor, and that this is a good thing. A **preprocessor** is a program that processes your input data and generates output that will be used as the input to another program. In the context of programming languages, the input of a preprocessor is source code that will be processed by the preprocessor before given as input to the compiler of the programming language. The biggest disadvantage of a preprocessor is that it knows nothing about the underlying language or its syntax!

Put simply, this means that when a preprocessor is used, you cannot be certain that the final version of your code will do what you really want it to do because the preprocessor might alter the logic as well as the semantics of your original code!

The list of programming languages with a preprocessor includes, but is not limited to C, C++, Ada, and PL/SQL. The infamous C preprocessor processes lines that begin with # and are called **directives** or **pragmas**. Directives and pragmas are not part of the C programming language!

The godoc utility

The Go distribution comes with a plethora of tools that can simplify your life as a programmer. One of these tools is the `godoc` utility, which allows you to see the documentation of existing Go functions and packages without needing an internet connection.

The `godoc` utility can be executed either as a normal command-line application that displays its output on a Terminal window, or as a command-line application that starts a web server. In the latter case, you will need a web browser to look at the Go documentation.

If you type `godoc` without any command-line parameters, you will get the list of the command-line options supported by `godoc`.

The first way of executing godoc is similar to using the `man(1)` command, but for Go functions and packages. So, in order to find out information about the `Printf()` function of the `fmt` package, you should execute the following command:

```
$ godoc fmt Printf
```

Similarly, you can find out information about the entire `fmt` package by running the next command:

```
$ godoc cmd/fmt
```

The second way requires executing `godoc` with the `-http` parameter:

```
$ godoc -http=:8001
```

The numeric value used in the preceding command, `8001`, is the port number to which the HTTP server will listen. You can choose any port number that is available provided that you have the right privileges. However, note that port numbers `0-1023` are restricted and can only be used by the root user. Thus, it is better to avoid choosing one of them and to pick something else provided that it is not already in use by a different process.

You can omit the equal sign in the command presented and put a space character in its place. So, the next command is the complete equivalent of the previous one:

```
$ godoc -http :8001
```

After that, you should point your web browser to the `http://localhost:8001/pkg/` URL in order to get the list of the available Go packages and to browse their documentation.

Compiling Go code

In this section, you will learn how to compile Go code. The good news is that you can compile your Go code from the command-line without needing a graphical application.

Furthermore, Go does not care about the name of the source file of an autonomous program as long as the package name is `main` and there is a single `main()` function in it, because the `main()` function is where the program execution begins. As a result, you cannot have multiple `main()` functions in the files of a single project.

We will start our first Go program compilation with a program named `aSourceFile.go` that contains the next Go code:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("This is a sample Go program!")
}
```

So, in order to compile `aSourceFile.go` and create a **statically linked** executable file, you will need to execute this command:

```
$ go build aSourceFile.go
```

After that, you will have a new executable file named `aSourceFile`:

```
$ file aSourceFile
aSourceFile: Mach-O 64-bit executable x86_64
$ ls -l aSourceFile
-rwxr-xr-x  1 mtsouk  staff  1933104 Oct 14 21:50 aSourceFile
$ ./aSourceFile
This is a sample Go program!
```

The main reason that `aSourceFile` is that big is because it is statically linked, which means that it does not require any external libraries in order to run.

Executing Go code

There is another way to execute your Go code that does not create any permanent executable files—it just generates some intermediate files that are automatically deleted afterwards.

The method presented in this chapter allows you to use Go as if it were a scripting language like **Python**, **Ruby**, and **Perl**.

In order to run `aSourceFile.go` without creating an executable file, you will need to execute the next command:

```
$ go run aSourceFile.go
This is a sample Go program!
```

As you can see, the output of the preceding command is exactly the same as before.

With `go run`, the Go compiler still needs to create an executable file. The fact that you will not see it, that it is automatically executed, and that it is automatically deleted after the program has finished might make you think that there is no need for an executable file!

This book mainly uses `go run` to execute the example code, primarily because it is simpler than running `go build` and then running the executable file. Additionally, `go run` does not leave any files on your hard drive after the program has finished its execution.

Two Go rules

Go has strict coding rules that are there to help you avoid silly errors and bugs in your code, as well as to make your code easier to read among the Go community. This section will present two such Go rules that you need to know.

Please remember that the Go compiler is here to help and not make your life miserable. As a result, the main purpose of the Go compiler is to compile and increase the quality of your Go code.

You either use a Go package or do not include it

Go has strict rules about package usage. Therefore, you cannot just include any package that you might think you will need and not use it afterwards. You will learn more about Go packages in `Chapter 6`, *What You Might Not Know About Go Packages*.

Look at the following naïve program, which is saved as `packageNotUsed.go`:

```
package main

import (
    "fmt"
    "os"
)

func main() {
    fmt.Println("Hello there!")
}
```

In this book, you are going to see lots of error messages, error situations, and warnings. I believe that examining code that fails to compile is useful and sometimes even more valuable than just looking at Go code that compiles without any errors. The Go compiler usually displays useful error messages and warnings that will most likely help you resolve an erroneous situation, so do not underestimate these error messages and warnings.

If you execute `packageNotUsed.go`, you will get the next error message from Go and the program will not be executed:

```
$ go run packageNotUsed.go
# command-line-arguments
./packageNotUsed.go:5:2: imported and not used: "os"
```

If you remove the `os` package from the import list of the program, `packageNotUsed.go` will compile just fine—try it on your own.

Although, this is not the perfect time to start talking about breaking Go rules, there is a way to bypass this restriction, which is showcased in the next Go code listing that is saved in the `packageNotUsedUnderscore.go` file:

```
package main

import (
    "fmt"
    _ "os"
)

func main() {
    fmt.Println("Hello there!")
}
```

Using an underscore character in front of a package name in the import list will not create an error message in the compilation process, even if that package is not used in the program:

```
$ go run packageNotUsedUnderscore.go
Hello there!
```

The reason that Go allows you to bypass this rule will become more evident in `Chapter 6`, *What You Might Not Know About Go Packages*.

There is only one way to format curly braces

Look at the next Go program, which is named `curly.go`:

```
package main

import (
    "fmt"
)

func main()
{
    fmt.Println("Go has strict rules for curly braces!")
}
```

Although it looks just fine, if you try to execute it, you will be fairly disappointed because you will get the next **syntax error** message, and the code will not compile and therefore not run:

```
$ go run curly.go
# command-line-arguments
./curly.go:7:6: missing function body for "main"
./curly.go:8:1: syntax error: unexpected semicolon or newline before
{
```

The official explanation for this error message is that Go requires the use of semicolons as statement terminators in many contexts, and the compiler automatically inserts the required semicolons when it thinks that they are necessary. Therefore, putting the opening brace (`{`) in its own line will make the Go compiler insert a semicolon at the end of the previous line (`func main()`), which produces the error message.

Downloading Go packages

Although the standard Go library is very rich, there are times when you will need to download external Go packages in order to use their functionality. This section will teach you how to download an external package and where it will be placed on your Unix machine.

Look at the next simple Go program that is saved as `getPackage.go`:

```
package main

import (
    "fmt"
    "github.com/mactsouk/go/simpleGitHub"
)

func main() {
    fmt.Println(simpleGitHub. AddTwo (5, 6))
}
```

This program uses an external package, because one of the import commands uses an internet address. In this case, the external package is called `simpleGitHub` and is located at `https://github.com/mactsouk/go`.

If you try to execute `getPackage.go` right away, you will be disappointed:

```
$ go run getPackage.go
getPackage.go:5:2: cannot find package
"github.com/mactsouk/go/simpleGitHub" in any of:
    /usr/local/Cellar/go/1.9.1/libexec/src/github.com/
mactsouk/go/simpleGitHub (from $GOROOT)
    /Users/mtsouk/go/src/github.com/mactsouk/go/
simpleGitHub (from $GOPATH)
```

The thing is that you will need to get the missing package onto your computer. In order to download it, you will need to execute the following command:

```
$ go get -v github.com/mactsouk/go/simpleGitHub
github.com/mactsouk/go (download)
github.com/mactsouk/go/simpleGitHub
```

After that, you can find the downloaded files at the following directory:

```
$ ls -l ~/go/src/github.com/mactsouk/go/simpleGitHub/
total 8
-rw-r--r--  1 mtsouk  staff  66 Oct 17 21:47 simpleGitHub.go
```

However, the `go get` command also compiles the package. The relevant files can be found at the following place:

```
$ ls -l ~/go/pkg/darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
-rw-r--r--  1 mtsouk  staff  1050 Oct 17 21:47 /Users/mtsouk/go/pkg/
darwin_amd64/github.com/mactsouk/go/simpleGitHub.a
```

You are now ready to execute `getPackage.go` without any problems:

```
$ go run getPackage.go
11
```

You can delete the intermediate files of a downloaded Go package as follows:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
cd /Users/mtsouk/go/src/github.com/mactsouk/go/simpleGitHub
rm -f simpleGitHub.test simpleGitHub.test.exe
rm -f /Users/mtsouk/go/pkg/darwin_amd64/github.com/mactsouk/
go/simpleGitHub.a
```

Similarly, you can delete an entire Go package that you have downloaded locally using the `rm(1)` Unix command to delete its Go source after using `go clean`:

```
$ go clean -i -v -x github.com/mactsouk/go/simpleGitHub
$ rm -rf ~/go/src/github.com/mactsouk/go/simpleGitHub
```

After executing the former commands, you will need to download the Go package again.

You will learn a lot more about Go packages in `Chapter 6`, *What You Might Not Know About Go Packages*.

Unix stdin, stdout, and stderr

Every Unix operating system has three files open all the time for its processes. As you know, Unix considers everything a file, even a printer or your mouse. Unix uses **file descriptors**, which are positive integer values, as an internal representation for accessing all of its open files, which is much more convenient than using long paths.

By default, all Unix systems support three special and standard filenames: `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`, which can also be accessed using file descriptors 0, 1, and 2, respectively. These three file descriptors are also called **standard input**, **standard output**, and **standard error**, respectively. Additionally, file descriptor 0 can be accessed as `/dev/fd/0` on a macOS machine and as both `/dev/fd/0` and `/dev/pts/0` on a Debian Linux machine.

Go uses `os.Stdin` for accessing standard input, `os.Stdout` for accessing standard output, and `os.Stderr` for accessing standard error. Although you can still use `/dev/stdin`, `/dev/stdout`, and `/dev/stderr`, or the related file descriptor values for accessing the same devices, it is better, safer, and more portable to stick with the `os.Stdin`, `os.Stdout`, and `os.Stderr` standard filenames that Go offers.

About printing output

As is the case with Unix and C, Go also offers a variety of ways for printing your output on the screen. All of the printing functions in this section require the use of the `fmt` Go standard package and are illustrated in the `printing.go` program, which will be presented in two parts.

The simplest way to print something in Go is by using the `fmt.Println()` and `fmt.Printf()` functions. The `fmt.Printf()` function has many similarities to the C `printf(3)` function. You can also use the `fmt.Print()` function instead of `fmt.Println()`. The main difference between `fmt.Print()` and `fmt.Println()` is that the latter automatically adds a newline character each time you call it, whereas the biggest difference between `fmt.Println()` and `fmt.Printf()` is that the latter requires a **format specifier** for each *thing* that you want to print, just like the C `printf(3)` function, which means that you have better control over what you are doing, though you have to write more code. Go calls these format specifiers **verbs**. You can find more information about verbs at `https://golang.org/pkg/fmt/`. If you have to perform any formatting before printing something, or you have to arrange multiple variables, then using `fmt.Printf()` might be a better choice. However, if you only have to print a single variable, then you might need to choose either `fmt.Print()` or `fmt.Println()`, depending on whether you need the newline character or not.

The first part of `printing.go` contains the next Go code:

```
package main

import (
    "fmt"
)

func main() {
    v1 := "123"
    v2 := 123
    v3 := "Have a nice day\n"
    v4 := "abc"
```

In this part, you can see the import of the `fmt` package and the definition of four Go variables. The `\n` used in `v3` is the line break character-if you just want to insert a line break in your output, however, you can call `fmt.Println()` without any arguments instead of using something like `fmt.Print("\n")`.

The second part follows:

```
    fmt.Print(v1, v2, v3, v4)
    fmt.Println()
    fmt.Println(v1, v2, v3, v4)
    fmt.Print(v1, " ", v2, " ", v3, " ", v4, "\n")
    fmt.Printf("%s%d %s %s\n", v1, v2, v3, v4)
}
```

In this part, you print the four variables using `fmt.Println()`, `fmt.Print()`, and `fmt.Printf()` in order to understand their differences better.

If you execute `printing.go`, you will get the following output:

```
$ go run printing.go
123123Have a nice day
abc
123 123 Have a nice day
abc
123 123 Have a nice day
abc
123123 Have a nice day
abc
```

As you can see in the preceding output, the `fmt.Println()` function also adds a space character between its parameters, which is not the case with `fmt.Print()`. As a result, a statement like `fmt.Println(v1, v2)` is equivalent to `fmt.Print(v1, " ", v2, "\n")`.

Apart from `fmt.Println()`, `fmt.Print()`, and `fmt.Printf()`, which are the simplest functions that can be used for generating output on the screen, there is also the `S` family of functions that includes `fmt.Sprintln()`, `fmt.Sprint()`, and `fmt.Sprintf()`, which are used for creating strings based on the given format and the `F` family of functions. This includes `fmt.Fprintln()`, `fmt.Fprint()` and `fmt.Fprintf()`, which are used for writing to files using an `io.Writer`.

You will learn more about the `io.Writer` and `io.Reader` interfaces in `Chapter 8`, *Telling a Unix System What to Do*.

The next section will teach you how to print your data using standard output, which is pretty common in the Unix world.

Using standard output

Standard output is more or less equivalent to printing on the screen. However, using **standard output** might require the use of functions that do not belong to the `fmt` package, which is why it is presented in its own section.

The relevant technique will be illustrated in `stdOUT.go`, which will be offered in three parts. The first part of the program follows:

```
package main

import (
    "io"
    "os"
)
```

Here `stdOUT.go` uses the `io` package instead of the `fmt` package. The `os` package is used for reading the command-line arguments of the program and for accessing `os.Stdout`.

The second portion of `stdOUT.go` contains the next Go code:

```
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
```

The `myString` variable holds the text that will be printed on the screen, which is either the first command-line argument of the program or, if the program was executed without any command-line arguments, a hard-coded text message.

The third part of the program is as follows:

```
    io.WriteString(os.Stdout, myString)
    io.WriteString(os.Stdout, "\n")
}
```

In this case, the `io.WriteString()` function works in the same way as the `fmt.Print()` function; however, it takes only two parameters. The first parameter is the file to which you want to write, which in this case is `os.Stdout`, and the second parameter is a `string` variable.

NOTE: Strictly speaking, the type of the first parameter of the `io.WriteString()` function should be an `io.Writer` interface, which requires a **slice** of bytes as the second parameter. However, in this case, a `string` does the job just fine. You will learn more about slices in `Chapter 3`, *Working with Basic Go Data Types*.

Executing `stdOUT.go` will produce the following output:

```
$ go run stdOUT.go
Please give me one argument!
$ go run stdOUT.go 123 12
123
```

The preceding output verifies that the `io.WriteString()` function sends the contents of its second parameter on the screen when its first parameter is `os.Stdout`.

Getting user input

There are three main ways to acquire user input:

1. By reading the command-line arguments of a program
2. By asking the user for input
3. By reading external files

This section will present the first two ways. Should you wish to learn how to read an external file, visit `Chapter 8`, *Telling a Unix System What to Do*.

About := and =

Before continuing, it would be very useful to talk about the use of `:=` and how it differs from `=`. The official name for `:=` is the **short assignment statement**. The short assignment statement can be used in place of a `var` declaration with an implicit type.

The `var` keyword is mostly used for declaring global variables in Go programs as well as for declaring variables without an initial value. The reason for the former is that every statement that exists outside of the code of a function must begin with a keyword, such as `func` or `var`. This means that the short assignment statement cannot be used outside of a function because it is not available there.

The `:=` operator works as follows:

```
m := 123
```

The result of the preceding statement is a new integer variable named `m` with a value of `123`.

However, if you try to use `:=` on an already declared variable, the compilation will fail with the next error message, which will make perfect sense:

```
$ go run test.go
# command-line-arguments
./test.go:5:4: no new variables on left side of :=
```

Now you might ask what will happen if you are expecting two or more values from a function, and you want to use an existing variable for one of them. Should you use `:=` or `=`? The answer is simple: you should use `:=` as shown in the next code example:

```
i, k := 3, 4
j, k := 1, 2
```

As the `j` variable is used for the first time in the second statement, you should use `:=` even though `k` has already been defined in the first statement.

Although it seems boring to talk about such insignificant things, knowing them will save you from various types of errors in the long run!

Reading from standard input

The reading of data from the standard input will be illustrated in `stdIN.go`, which you will see in two parts. The first part follows:

```
package main

import (
    "bufio"
    "fmt"
    "os"
)
```

In the preceding code, you see the use of the `bufio` package for the first time in this book.

You will learn more about the `bufio` package, which is related to file input and output, in `Chapter 8`, *Telling a Unix System What to Do*.

Although the `bufio` package is mostly used for file input and output, you will keep seeing the `os` package all of the time in this book because it contains many handy functions. Its most common functionality is that it provides you with a way to access the command-line arguments of a Go program (`os.Args`). The official description of the `os` package tells us that it offers functions that perform operating system operations. This includes functions for creating, deleting, and renaming files and directories, as well as functions for learning the Unix permissions and other characteristics of files and directories. The main advantage of the `os` package is that it is platform independent. Put simply, its functions will work on both Unix and Microsoft Windows machines!

The second part of `stdIN.go` contains the following Go code:

```
func main() {
    var f *os.File
    f = os.Stdin
    defer f.Close()

    scanner := bufio.NewScanner(f)
    for scanner.Scan() {
        fmt.Println(">", scanner.Text())
    }
}
```

Here there is a call to `bufio.NewScanner()` using standard input (`os.Stdin`) as its parameter. This call returns a `bufio.Scanner` variable, which is then used with the `Scan()` function for reading from `os.Stdin` line by line. Each line that is read is printed on the screen before getting the next one. Note that each line printed by the program begins with the > character.

The execution of `stdIN.go` will produce the following type of output:

```
$ go run stdIN.go
21
> 21
This is Mihalis!
> This is Mihalis!
```

In Unix, you can tell a program to stop reading data from standard input by pressing *Ctrl + D*.

The Go code of `stdIN.go` and `stdOUT.go` will be very useful when we talk about Unix **pipes** in `Chapter 8`, *Telling a Unix System What to Do*, so do not underestimate their simplicity.

Working with command-line arguments

The technique covered in this section will be illustrated by using the Go code of `cla.go`, which will be presented in three parts. The program will find the minimum and the maximum of its command-line arguments.

The first part of the program is as follows:

```
package main

import (
    "fmt"
    "os"
    "strconv"
)
```

What is important here is to realize that obtaining the command-line arguments requires the use of the `os` package. Additionally, you need another package, named `strconv`, in order to be able to convert a command-line argument, which is given as a string, into an arithmetical data type.

The second part of the program is as follows:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    min, _ := strconv.ParseFloat(arguments[1], 64)
    max, _ := strconv.ParseFloat(arguments[1], 64)
```

Here, `cla.go` checks whether you have any command-line arguments or not by examining the length of `os.Args`, because the program needs at least one command-line argument to operate. Note that `os.Args` is a Go slice with `string` values. The first element in the slice is the name of the executable program. Therefore, in order to initialize the `min` and `max` variables, you will need to use the second element of the `os.Args` string slice that has an index value of `1`.

The important point here is that the fact that you are expecting one or more floats does not necessarily mean that the user will give you valid floats, either by accident or on purpose. However, as we have not talked about error handling in Go so far, `cla.go` assumes that all command-line arguments are in the right format and therefore will be acceptable. As a result, `cla.go` ignores the error value returned by the `strconv.ParseFloat()` function using the following statement:

```
n, _ := strconv.ParseFloat(arguments[i], 64)
```

The preceding statement tells Go that you only want to get the first value returned by `strconv.ParseFloat()` and that you are not interested in the second value, which in this case is an error variable by assigning it to the underscore character. The underscore character, which is called a **blank identifier**, is the Go way of discarding a value. If a Go function returns multiple values, you can use the blank identifier multiple times.

WARNING: Ignoring all or some of the return values of a Go function, especially the error values, is a very dangerous practice that should not be used in production code!

The third part comes with the following Go code:

```
for i := 2; i < len(arguments); i++ {
    n, _ := strconv.ParseFloat(arguments[i], 64)
    if n < min {
        min = n
    }
    if n > max {
        max = n
    }
}
    fmt.Println("Min:", min)
    fmt.Println("Max:", max)
}
```

Here you use a `for` loop that will help you visit all of the elements of the `os.Args` slice, which was previously assigned to the `arguments` variable.

Executing `cla.go` will create the next type of output:

```
$ go run cla.go -10 0 1
Min: -10
Max: 1
$ go run cla.go -10
Min: -10
Max: -10
```

As you might expect, the program does not behave well when it receives erroneous input. Worst of all, the program does not generate any warnings to inform the user that there were one or more errors while processing the command-line arguments:

```
$ go run cla.go a b c 10
Min: 0
Max: 10
```

About error output

This section presents a technique for sending data to **Unix standard error**, which is the Unix way of differentiating between actual values and error output.

The Go code for illustrating the use of standard error in Go is included in `stdERR.go` and will be presented in two parts. As writing to standard error requires the use of the file descriptor related to standard error, the Go code of `stdERR.go` will be based on the Go code of `stdOUT.go`.

The first part of the program follows:

```
package main
import (
    "io"
    "os"
)
func main() {
    myString := ""
    arguments := os.Args
    if len(arguments) == 1 {
        myString = "Please give me one argument!"
    } else {
        myString = arguments[1]
    }
```

So far, `stdERR.go` is almost identical to `stdOUT.go`.

The second part of the program, `stdERR.go`, is as follows:

```
io.WriteString(os.Stdout, "This is Standard output\n")
io.WriteString(os.Stderr, myString)
io.WriteString(os.Stderr, "\n")
```

Here you call `io.WriteString()` two times to write to standard error (`os.Stderr`) and one more time to write to standard output (`os.Stdout`).

Executing `stdERR.go` will create the following output:

```
$ go run stdERR.go
This is Standard output
Please give me one argument!
```

The preceding output cannot help you differentiate between data written to standard output and data written to standard error, which could be very useful at times. However, if you are using the `bash(1)` shell, there is a trick that you can use in order to distinguish between standard output data and standard error data. Almost all Unix shells offer this functionality in their own way.

Thus, when using `bash(1)`, you can redirect the standard error output to a file as follows:

```
$ go run stdERR.go 2>/tmp/stdError
This is Standard output
$ cat /tmp/stdError
Please give me one argument!
```

The number after the name of a Unix program or system call refers to the section of the manual to which its page belongs. Although most names can be found only once in the manual pages, which means that indicating the section number is not required, there are names that can be located in multiple sections because they have multiple meanings, such as `crontab(1)` and `crontab(5)`. Therefore, if you try to retrieve the manual page of a name with multiple meanings without stating its section number, you will get the entry that has the smallest section number.

Similarly, you can discard error output by redirecting it to the `/dev/null` device, which is like telling Unix to ignore it completely:

```
$ go run stdERR.go 2>/dev/null
This is Standard output
```

What we did in the two preceding examples is to redirect the file descriptor of standard error into a file and `/dev/null`, respectively. If you want to save both standard output and standard error to the same file, you can redirect the file descriptor of standard error (`2`) to the file descriptor of standard output (`1`)! The following command shows this technique, which is pretty common in Unix systems:

```
$ go run stdERR.go >/tmp/output 2>&1
$ cat /tmp/output
This is Standard output
Please give me one argument!
```

Last, you can send both standard output and standard error to `/dev/null` as follows:

```
$ go run stdERR.go >/dev/null 2>&1
```

Writing to log files

The `log` package allows you to send log messages to the system logging service of your Unix machine, whereas the `syslog` Go package, which is part of the `log` package, allows you to define the **logging level** and the **logging facility** that your Go program will use.

Usually, most system log files on a Unix operating system can be found under the `/var/log` directory. However, the log files of many popular services, such as Apache and Nginx, can be found elsewhere, depending on their configuration.

Generally speaking, using a log file to write some information is considered a better practice than writing the same output on the screen for two reasons:

1. The output does not get lost as it is stored in a file
2. You can search and process log files using Unix tools such as `grep(1)`, `awk(1)`, and `sed(1)`, which cannot be done when messages are printed on a Terminal window

The `log` package offers many functions for sending output to the syslog server of a Unix machine. The list of function includes `log.Printf()`, `log.Print()`, `log.Println()`, `log.Fatalf()`, `log.Fatalln()`, `log.Panic()`, `log.Panicln()`, and `log.Panicf()`.

Logging functions can be extremely handy for debugging your programs, especially server processes written in Go, so you should not undervalue their power.

Logging levels

The **logging level** is a value that specifies the severity of the log entry. Various logging levels exist including `debug`, `info`, `notice`, `warning`, `err`, `crit`, `alert`, and `emerg` (in reverse order of severity).

Logging facilities

A **logging facility** is a category used for logging information. The value of the logging facility part can be one of `auth`, `authpriv`, `cron`, `daemon`, `kern`, `lpr`, `mail`, `mark`, `news`, `syslog`, `user`, `UUCP`, `local0`, `local1`, `local2`, `local3`, `local4`, `local5`, `local6`, and `local7`. It is defined inside `/etc/syslog.conf`, `/etc/rsyslog.conf`, or another appropriate file depending on the server process used for system logging on your Unix machine. This means that if a logging facility is not defined and thus handled, the log messages you send to it might get ignored and therefore lost.

Log servers

All Unix machines have a separate server process that is responsible for receiving logging data and writing it to log files. Various log servers exist that work on Unix machines; however, only two of them are used on most Unix variants: `syslogd(8)` and `rsyslogd(8)`.

On macOS machines, the name of the process is `syslogd(8)`. On the other hard, most Linux machines use `rsyslogd(8)`, which is an improved and more reliable version of `syslogd(8)`, which was the original Unix system utility for message logging.

However, despite the Unix variant you are using or the name of the server process used for logging, logging works the same way on every Unix machine and therefore does not affect the Go code that you will write.

The configuration file of `rsyslogd(8)` is usually named `rsyslog.conf` and is located in `/etc`. The contents of a `rsyslog.conf` configuration file, without the lines with comments and lines starting with `$`, might look like the following:

```
$ grep -v '^#' /etc/rsyslog.conf | grep -v '^$' | grep -v '^\$'
auth,authpriv.*                 /var/log/auth.log
*.*;auth,authpriv.none          -/var/log/syslog
daemon.*                        -/var/log/daemon.log
kern.*                          -/var/log/kern.log
```

```
lpr.*                           -/var/log/lpr.log
mail.*                          -/var/log/mail.log
user.*                          -/var/log/user.log
mail.info                       -/var/log/mail.info
mail.warn                       -/var/log/mail.warn
mail.err                         /var/log/mail.err
news.crit                        /var/log/news/news.crit
news.err                         /var/log/news/news.err
news.notice                     -/var/log/news/news.notice
*.=debug;\
    auth,authpriv.none;\
    news.none;mail.none         -/var/log/debug
*.=info;*.=notice;*.=warn;\
    auth,authpriv.none;\
    cron,daemon.none;\
    mail,news.none              -/var/log/messages
*.emerg                         :omusrmsg:*
daemon.*;mail.*;\
    news.err;\
    *.=debug;*.=info;\
    *.=notice;*.=warn           |/dev/xconsole
local7.* /var/log/cisco.log
```

In order to send your logging information to `/var/log/cisco.log`, you will need to use the `local7` logging facility. The star character after the name of the facility tells the logging server to catch every logging level that goes to the `local7` logging facility and write it to `/var/log/cisco.log`.

The `syslogd(8)` server has a pretty similar configuration file that is usually `/etc/syslog.conf`. On macOS High Sierra, the `/etc/syslog.conf` file is almost empty and has been replaced by `/etc/asl.conf`. Nevertheless, the logic behind the configuration of `/etc/syslog.conf`, `/etc/rsyslog.conf`, and `/etc/asl.conf` is the same.

A Go program that sends information to log files

The Go code of `logFiles.go` will explain the use of the `log` and `log/syslog` packages.

The `log/syslog` package is not implemented on the Microsoft Windows version of Go.

The first part of `logFiles.go` follows:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
    "os"
    "path/filepath"
)

func main() {
    programName := filepath.Base(os.Args[0])
    sysLog, err := syslog.New(syslog.LOG_INFO|syslog.LOG_LOCAL7,
programName)
```

The first parameter of the `syslog.New()` function is the priority, which is a combination of the logging facility and the logging level. Therefore, a priority of `LOG_NOTICE | LOG_MAIL`, which is mentioned as an example, will send notice logging-level messages to the `MAIL` logging facility.

As a result, the preceding code sets the default logging to the `local7` logging facility using the `info` logging level. The second parameter of the `syslog.New()` function is the name of the process that will appear on the logs as the sender of the message. Generally speaking, it is considered a good practice to use the real name of the executable in order to be able to find the information you want easily in the log files at another time.

The second part of the program contains the following Go code:

```
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }
    log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")
```

After the call to `syslog.New()`, you will have to check the error variable that it returns so that you can make sure that everything is fine. If everything is OK, which means that the value of the error variable is equal to `nil`, you call the `log.SetOutput()` function. This sets the output destination of the default logger, which in this case is the logger you created earlier on (`sysLog`). Then you can use `log.Println()` to send information to the log server.

The third part of `logFiles.go` comes with the following code:

```
        sysLog, err = syslog.New(syslog.LOG_MAIL, "Some program!")
        if err != nil {
            log.Fatal(err)
        } else {
            log.SetOutput(sysLog)
        }

        log.Println("LOG_MAIL: Logging in Go!")
        fmt.Println("Will you see this?")
}
```

The last part shows that you can change the logging configuration in your programs as many times as you want, and that you can still use `fmt.Println()` for printing output on the screen.

The execution of `logFiles.go` will create the following output on the screen on a Debian Linux machine:

```
$ go run logFiles.go
Broadcast message from systemd-journald@mail (Tue 2017-10-17 20:06:08
EEST):
logFiles[23688]: Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL:
Logging in Go!
Message from syslogd@mail at Oct 17 20:06:08 ...
Some program![23688]: 2017/10/17 20:06:08 LOG_MAIL: Logging in Go!
Will you see this?
```

Executing the same Go code on a macOS High Sierra machine generated the following output:

```
$ go run logFiles.go
Will you see this?
```

Keep in mind that most Unix machines store logging information in more than one log file, which is also the case with the Debian Linux machine used in this section. As a result, `logFiles.go` sends its output to multiple log files, which can be verified by the output of the following shell commands:

```
$ grep LOG_MAIL /var/log/mail.log
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08
LOG_MAIL: Logging in Go!
$ grep LOG_LOCAL7 /var/log/cisco.log
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
$ grep LOG_ /var/log/syslog
```

```
Oct 17 20:06:08 mail logFiles[23688]: 2017/10/17 20:06:08 LOG_INFO +
LOG_LOCAL7: Logging in Go!
Oct 17 20:06:08 mail Some program![23688]: 2017/10/17 20:06:08
LOG_MAIL: Logging in Go!
```

The preceding output shows that the message of the `log.Println("LOG_INFO + LOG_LOCAL7: Logging in Go!")` statement was written on both `/var/log/cisco.log` and `/var/log/syslog`, whereas the message of the `log.Println("LOG_MAIL: Logging in Go!")` statement was written on both `/var/log/syslog` and `/var/log/mail.log`.

The important thing to remember from this section is that if the logging server of a Unix machine is not configured to catch all logging facilities, some of the log entries that you send to it might get discarded without any warnings.

About log.Fatal()

In this section, you will see the `log.Fatal()` function in action. The `log.Fatal()` function is used when something really bad has happened, and you just want to exit your program as fast as possible after reporting the bad situation. The use of `log.Fatal()` is illustrated in the `logFatal.go` program, which contains the following Go code:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    sysLog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL,
 "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Fatal(sysLog)
    fmt.Println("Will you see this?")
}
```

Executing `log.Fatal()` will create the following output:

```
$ go run logFatal.go
exit status 1
```

As you can easily understand, the use of `log.Fatal()` terminates a Go program at the point where `log.Fatal()` was called, which is the reason that you did not see the output from the `fmt.Println("Will you see this?")` statement.

However, because of the parameters of the `syslog.New()` call, a log entry has been added to the log file that is related to mail, which is `/var/log/mail.log`:

```
$ grep "Some program" /var/log/mail.log
Oct 17 20:20:29 iMac Some program![4663]: 2017/10/17 20:20:29 &{17
Some program! iMac.local   {0 0} 0xc42000c220}
```

About log.Panic()

There are situations where a program will fail for good, and you want to have as much information about the failure as possible. In such difficult circumstances, you might consider using `log.Panic()`, which is the logging function that is illustrated in this section using the Go code of `logPanic.go`.

The Go code of `logPanic.go` follows:

```
package main

import (
    "fmt"
    "log"
    "log/syslog"
)

func main() {
    sysLog, err := syslog.New(syslog.LOG_ALERT|syslog.LOG_MAIL,
 "Some program!")
    if err != nil {
        log.Fatal(err)
    } else {
        log.SetOutput(sysLog)
    }

    log.Panic(sysLog)
    fmt.Println("Will you see this?")
}
```

Executing `logPanic.go` on macOS High Sierra will produce the following output:

```
$ go run logPanic.go
panic: &{17 Some program! iMac.local   {0 0} 0xc42000c220}
goroutine 1 [running]:
log.Panic(0xc42004ff50, 0x1, 0x1)
    /usr/local/Cellar/go/1.9.1/libexec/src/log/log.go:330 +0xc0
main.main()
    /Users/mtsouk/Desktop/masterGo/ch/ch1/code/logPanic.go:17 +0xea
exit status 2
```

Running the same program on a Debian Linux machine with Go version 1.3.3 will generate the following output:

```
$ go run logPanic.go
panic: &{17 Some program! mail   {0 0} 0xc2080400e0}
goroutine 16 [running]:
runtime.panic(0x4ec360, 0xc208000320)
    /usr/lib/go/src/pkg/runtime/panic.c:279 +0xf5
log.Panic(0xc208055f20, 0x1, 0x1)
    /usr/lib/go/src/pkg/log/log.go:307 +0xb6
main.main()
    /home/mtsouk/Desktop/masterGo/ch/ch1/code/logPanic.go:17 +0x169
goroutine 17 [runnable]:
runtime.MHeap_Scavenger()
    /usr/lib/go/src/pkg/runtime/mheap.c:507
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 18 [runnable]:
bgsweep()
    /usr/lib/go/src/pkg/runtime/mgc0.c:1976
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
goroutine 19 [runnable]:
runfinq()
    /usr/lib/go/src/pkg/runtime/mgc0.c:2606
runtime.goexit()
    /usr/lib/go/src/pkg/runtime/proc.c:1445
exit status 2
```

The output of `log.Panic()` includes additional low-level information that will hopefully help you resolve difficult and rare situations that happen in your Go code.

Analogous to the `log.Fatal()` function, the use of the `log.Panic()` function will add an entry to the proper log file and will immediately terminate the Go program.

Error handling in Go

Errors and error handling are two very important Go topics. Go likes error messages so much that it has a separate data type for errors, named `error`! This also means that you can easily create your own **error messages** if you find that what Go gives you is inadequate. You will most likely need to create and handle your own errors when you are developing your own Go packages.

Note that having an error condition is one thing, while deciding how to react to an error condition is a totally different thing. Put simply, not all error conditions are created equal, which means that some error conditions might require that you immediately stop the execution of a program, whereas other error situations might require printing a warning message for the user and continuing with the execution of the program. It is up to the developer to use common sense and decide what to do with each `error` value that the program might get.

The error data type

Many occasions exist where you might end up having to deal with a new error case while developing your own Go application. The `error` data type is here to help you define your own errors.

This subsection will teach you how to create your own `error` variables. As you will see in a while, in order to create a new `error` variable, you will need to call the `New()` function of the `errors` standard Go package.

The example Go code illustrating this process can be found in `newError.go`, and it will be presented in two parts. The first part of the program follows next:

```
package main

import (
    "errors"
    "fmt"
)

func returnError(a, b int) error {
```

```
        if a == b {
            err := errors.New("Error in returnError() function!")
            return err
        } else {
            return nil
        }
    }
```

There are many interesting things happening here. First of all, you can see the definition of a Go function other than `main()` for the first time in this book. The name of this new unsophisticated function is `returnError()`. Additionally, you can see the `errors.New()` function in action, which takes a `string` value as its parameter. Last, if a function should return an `error` variable but there is no error to report, it returns `nil` instead.

You will learn more about the various types of Go functions in `Chapter 6`, *What You Might Not Know About Go Packages*.

The second part of `newError.go` is as follows:

```
func main() {
    err := returnError(1, 2)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    err = returnError(10, 10)
    if err == nil {
        fmt.Println("returnError() ended normally!")
    } else {
        fmt.Println(err)
    }

    if err.Error() == "Error in returnError() function!" {
        fmt.Println("!!")
    }
}
```

As the code illustrates, most of the time, you need to check whether an `error` variable is equal to `nil` or not and then act accordingly. Also presented here is the use of the `err.Error()` method, which allows you to convert an `error` variable into a `string` variable. This function lets you compare an `error` variable with a `string`.

Sending your error messages to the logging service of your Unix machine, especially when a Go program is a server or some other critical application. However, the code presented in this book will not follow this principle everywhere in order to avoid filling your log files with unnecessary data.

Executing `newError.go` will produce the following output:

```
$ go run newError.go
returnError() ended normally!
Error in returnError() function!
!!
```

If you try to compare an `error` variable with a `string` variable without converting the `error` variable to a `string` first, the Go compiler will create the following error message:

```
# command-line-arguments
./newError.go:33:9: invalid operation: err == "Error in returnError()
function!" (mismatched types error and string)
```

Error handling

Error handling is a very important feature of Go because almost all Go functions return an error message or `nil`, which is the Go way of saying whether there was an error condition while executing a function or not. You will most likely get tired of seeing the following Go code, not only in this book but also in every other Go program you can find on the Internet:

```
if err != nil {
    fmt.Println(err)
    os.Exit(10)
}
```

Do not confuse error handling with printing to error output, because they are two totally different things. The former has to do with Go code that handles error conditions, whereas the latter has to do with writing something to the standard error file descriptor.

The preceding code prints the error message on the screen and exits using `os.Exit()`. Should you wish to send the error message to the logging service instead of the screen, use the following variation of the preceding Go code:

```
if err != nil {
    log.Println(err)
    os.Exit(10)
}
```

Last, there is another variation of the preceding code that is used when something really bad has happened and you want to terminate the program:

```
if err != nil {
    panic(err)
    os.Exit(10)
}
```

The `panic` function is a built-in Go function that stops the execution of a program and starts panicking! If you find yourself using `panic` too often, you might want to reconsider your Go implementation. As you will see in `Chapter 2`, *Understanding Go Internals*, Go also offers the `recover` function, which might be able to save you when you're in some bad situations. For now, you will need to wait for the next chapter to learn more about the power of the `panic` and `recover` function pair.

It's now time to see a Go program that not only handles error messages generated by standard Go functions, but one that also defines its own error message. The name of the program is `errors.go`, and it will be presented to you in five parts. As you will see, the `errors.go` utility tries to improve the functionality of the `cla.go` program that you saw earlier in this chapter by examining whether its command-line arguments are acceptable floats or not.

The first part of the program follows:

```
package main

import (
    "errors"
    "fmt"
    "os"
    "strconv"
)
```

This part of `errors.go` contains the expected `import` statements.

The second portion of `errors.go` comes with the following Go code:

```
func main() {
    if len(os.Args) == 1 {
        fmt.Println("Please give one or more floats.")
        os.Exit(1)
    }

    arguments := os.Args
    var err error = errors.New("An error")
    k := 1
    var n float64
```

Here you create a new `error` variable named `err` in order to initialize it with your own value.

The third part of the program comes next:

```
    for err != nil {
        if k >= len(arguments) {
            fmt.Println("None of the arguments is a float!")
            return
        }
        n, err = strconv.ParseFloat(arguments[k], 64)
        k++
    }

    min, max := n, n
```

This is the trickiest part of the program because, if the first command-line argument is not a proper float, you will need to check the next one and keep checking until you find a suitable command-line argument. If none of the command-line arguments are in the correct format, `errors.go` will terminate and print a message on the screen. All this checking happens by examining the `error` value that is returned by `strconv.ParseFloat()`. All of this code is there just for the accurate initialization of the `min` and `max` variables.

The fourth part of the program comes with the following Go code:

```
    for i := 2; i < len(arguments); i++ {
        n, err := strconv.ParseFloat(arguments[i], 64)
        if err == nil {
            if n < min {
                min = n
            }
            if n > max {
                max = n
            }
```

```
        }
    }
```

Here you just process all of the right command-line arguments in order to find the minimum and maximum floats among them.

Finally, the last code portion of the program just deals with printing out the current values of the `min` and `max` variables:

```
    fmt.Println("Min:", min)
    fmt.Println("Max:", max)
}
```

As you can see from the Go code in `errors.go`, the biggest part of the code is about error handling than about the actual functionality of the program. Unfortunately, this is the case for software developed in most modern programming languages, and Go is no exception.

If you execute `errors.go`, you will get the following output:

```
$ go run errors.go a b c
None of the arguments is a float!
$ go run errors.go b c 1 2 3 c -1 100 -200 a
Min: -200
Max: 100
```

Additional resources

Have a look at the following resources:

- Visit the Go website at `https://golang.org/`
- Browse the Go documentation site at `https://golang.org/doc/`
- Visit the documentation page of the `log` package at `https://golang.org/pkg/log/`
- Visit the documentation of the `log/syslog` package at `https://golang.org/pkg/log/syslog/`
- Visit the documentation page of the `os` package at `https://golang.org/pkg/os/`
- Have a look at `https://golang.org/cmd/gofmt/`, which is the documentation page of the `gofmt` tool that is used for formatting Go code

- If you are working on a Mac, check the **TextMate** editor at `http://macromates.com/` as well as **BBEdit** at `https://www.barebones.com/products/bbedit/`
- Visit the documentation page of the `fmt` package at `https://golang.org/pkg/fmt/` to learn more about Go verbs and the available functions

Exercises

- Write a Go program that finds the sum of all of its numeric command-line arguments
- Write a Go program that finds the average value of all of its float command-line arguments
- Write a Go program that keeps reading integers until it gets the word STOP as input

Summary

This chapter addressed many interesting Go topics including compiling Go code, working with standard input, accessing standard output and standard error in Go, processing command-line arguments, printing on the screen, and using the logging service of a Unix system as well as error handling and some general information about Go. You should consider all of these topics as foundational information about Go.

The next chapter is all about the internals of Go, which includes learning about **Garbage Collection**, working with the Go compiler, calling C code from Go, using the `defer` keyword, and working with the Go assembler as well as the `panic` and `recover` function pair.

2
Understanding Go Internals

All of the Go features that you learned in the previous chapter are extremely handy, and you will be using them all the time. However, there is nothing more rewarding than being able to see and understand what is going on in the background and how Go operates behind the scenes.

In this chapter, you will learn about the Go garbage collector and how it works. Additionally, you will find out how to call C code from your Go programs, which you might find indispensable in certain situations. However, you will not need to use this capability too often, because Go is a very capable programming language. Likewise, you will also understand how to call Go code from your C programs and how to use the `panic()` and `recover()` functions and the `defer` keyword.

In this chapter of Mastering Go, you will learn the following topics:

- The Go compiler
- How garbage collection works in Go
- How to check the operation of the garbage collector
- Calling C code from your Go programs
- Calling Go code from a C program
- The `panic()` and `recover()` functions
- The `unsafe` package
- The handy, yet tricky `defer` keyword
- The `strace(1)` Linux utility
- The `dtrace(1)` utility that can be found in FreeBSD systems including macOS High Sierra
- Finding out information about your Go environment
- Node trees
- The Go assembler

The Go compiler

The **Go compiler** is executed with the help of the `go` tool. This tool does many more things than just generating executable files.

The `unsafe.go` file used in this section does not contain any special code-the commands presented will work on every valid Go source file. You will see the contents of `unsafe.go` in a short while.

You can compile a Go source file using the `go tool compile` command. What you will get is an **object file**, which is a file with the `.o` file extension. This is illustrated in the output of the following commands, which were executed on a macOS High Sierra machine:

```
$ go tool compile unsafe.go
$ ls -l unsafe.o
-rw-r--r--  1 mtsouk  staff  5495 Oct 30 19:51 unsafe.o
$ file unsafe.o
unsafe.o: data
```

An object file is a binary file that contains **object code**, which is machine code in a relocatable format that, most of the time is not directly executable. The biggest advantage of the relocatable format is that it requires as little memory as possible during the linking phase.

If you use the `-pack` command-line flag when executing `go tool compile`, you will get an **archive file** instead of an object file:

```
$ go tool compile -pack unsafe.go
$ ls -l unsafe.a
-rw-r--r--  1 mtsouk  staff  5680 Oct 30 19:52 unsafe.a
$ file unsafe.a
unsafe.a: current ar archive
```

An archive file is a binary file that contains one or more files that is primarily used for grouping multiple files into a single file. The archive format used by Go is called **ar**.

You can list the contents of an `.a` archive file as follows:

```
$ ar t unsafe.a
__.PKGDEF
_go_.o
```

Another truly valuable command-line flag of the `go tool compile` command is `-race`, which allows you to detect **race conditions**. You will learn more about race conditions and why you want to avoid them in `Chapter 10`, *Go Concurrency – Advanced Topics*.

You will learn the additional uses of the `go tool compile` command toward the end of this chapter, when we talk about assembly language and node trees. However, in order to tease you a little, try executing the following command:

```
$ go tool compile -S unsafe.go
```

The preceding command generates lots of output that you might find difficult to understand, which means that Go does a pretty good job on hiding any unnecessary complexities, unless you ask for them!

Garbage Collection

Garbage Collection is the process of freeing memory space that is not being used. In other words, the garbage collector sees which objects are out of scope and can no longer be referenced, and it frees the memory space they consume. This process happens in a concurrent manner while a Go program is running, not before or after the execution of a Go program. The documentation of the Go garbage collector implementation states the following:

The GC runs concurrently with mutator threads, is type accurate (aka precise), allows multiple GC threads to run in parallel. It is a concurrent mark and sweep that uses a write barrier. It is non-generational and non-compacting. Allocation is done using size segregated per P allocation areas to minimize fragmentation while eliminating locks in the common case.

There is a lot of terminology here that I will explain in a while. First, however, I will show you a way to look at some parameters of the garbage collection process. Fortunately, the Go standard library offers functions that allow you to study the operation of the garbage collector and learn more about what the garbage collector does secretly. The relevant code is saved as `gColl.go`, and it will be presented in three parts.

The first code segment of `gColl.go` is as follows:

```
package main

import (
    "fmt"
    "runtime"
    "time"
```

```
)

func printStats(mem runtime.MemStats) {
    runtime.ReadMemStats(&mem)
    fmt.Println("mem.Alloc:", mem.Alloc)
    fmt.Println("mem.TotalAlloc:", mem.TotalAlloc)
    fmt.Println("mem.HeapAlloc:", mem.HeapAlloc)
    fmt.Println("mem.NumGC:", mem.NumGC)
    fmt.Println("-----")
}
```

Note that each time that you need to retrieve the more recent garbage collections statistics, you will need to call the `runtime.ReadMemStats()` function. The purpose of the `printStats()` function is to avoid writing the same Go code all of the time.

The second part of the program is as follows:

```
func main() {
    var mem runtime.MemStats
    printStats(mem)

    for i := 0; i < 10; i++ {
          s := make([]byte, 50000000)
          if s == nil {
                fmt.Println("Operation failed!")
          }
    }
    printStats(mem)
```

The `for` loop creates multiple, big Go **slices** in order to allocate large amounts of memory and trigger the garbage collector.

The last part of `gColl.go` has the following Go code, which does more memory allocations using Go slices:

```
    for i := 0; i < 10; i++ {
          s := make([]byte, 100000000)
          if s == nil {
                fmt.Println("Operation failed!")
          }
          time.Sleep(5 * time.Second)
    }
    printStats(mem)
}
```

The output of `gColl.go` on a macOS High Sierra machine is as follows:

```
$ go run gColl.go
mem.Alloc: 66024
mem.TotalAlloc: 66024
mem.HeapAlloc: 66024
mem.NumGC: 0
-----
mem.Alloc: 50078496
mem.TotalAlloc: 500117056
mem.HeapAlloc: 50078496
mem.NumGC: 10
-----
mem.Alloc: 76712
mem.TotalAlloc: 1500199904
mem.HeapAlloc: 76712
mem.NumGC: 20
-----
```

Although you will not examine the operation of the Go garbage collector all of the time, being able to watch the way the Go garbage collector operates on a slow application can save you a lot of time in the long run—I can assure you that you will not regret the time you spend learning about garbage collection in general and, more specifically, about the way the Go garbage collector works.

There is a trick that allows you to get even more detailed output about the way the Go garbage collector operates, which is illustrated by the next command:

```
$ GODEBUG=gctrace=1 go run gColl.go
```

So, if you put `GODEBUG=gctrace=1` in front of any `go run` command, Go will print analytical data about the operation of the garbage collector. The generated data will have the following format:

```
gc 4 @0.025s 0%: 0.002+0.065+0.018 ms clock, 0.021+0.040/0.057/0.003+0.14
ms cpu, 47->47->0 MB, 48 MB goal, 8 P
gc 17 @30.103s 0%: 0.004+0.080+0.019 ms clock, 0.033+0/0.076/0.071+0.15 ms
cpu, 95->95->0 MB, 96 MB goal, 8 P
```

The preceding output gives you more information about the heap sizes during the garbage collection process. Let's take the `47->47->0 MB` trinity of values as an example. The first number is the heap size when the garbage collector is about to run. The second value is the heap size when the garbage collector ends its operation. The last value is the size of the live heap.

The Tricolor algorithm

The operation of the Go garbage collector is based on the **tricolor algorithm**, which is the subject of this subsection.

The tricolor algorithm is not unique to Go, and it can be used in other programming languages as well.

Strictly speaking, the official name for the algorithm used in Go is the **tricolor mark-and-sweep algorithm**. It can work concurrently with the program and uses a **write barrier**. This means that when a Go program runs, the Go scheduler is responsible for the scheduling of the application and the garbage collector as if the Go scheduler had to deal with a regular application with multiple **goroutines**! You will learn more about goroutines and the Go scheduler in `Chapter 9`, *Go Concurrency – Goroutines, Channels, and Pipelines*.

The core idea behind this algorithm is that of **Edsger W. Dijkstra**, **Leslie Lamport**, **A. J. Martin**, **C. S. Scholten**, and **E. F. M. Steffens**. It was first illustrated on a paper, *On-the-fly Garbage Collection: An Exercise in Cooperation*. The primary principle behind the tricolor mark-and-sweep algorithm is that it divides the objects of the heap into three different sets according to their color, which is assigned by the algorithm. I will address the mark-and-sweep algorithm further in the *More about the operation of the Go Garbage Collector* section of this chapter.

Now let's talk about the meaning of each color set. The objects of the **black set** are guaranteed to have no pointers to any object of the white set. However, an object in the **white set** can have a pointer to an object of the black set, because this has no effect on the operation of the garbage collector! The objects of the **grey set** might have pointers to some objects of the white set. Also, the objects of the white set are candidates for garbage collection.

Note that no object can go directly from the black set to the white set, which allows the algorithm to operate and be able to clear the objects in the white set. Additionally, no object of the black set can directly point to an object of the white set.

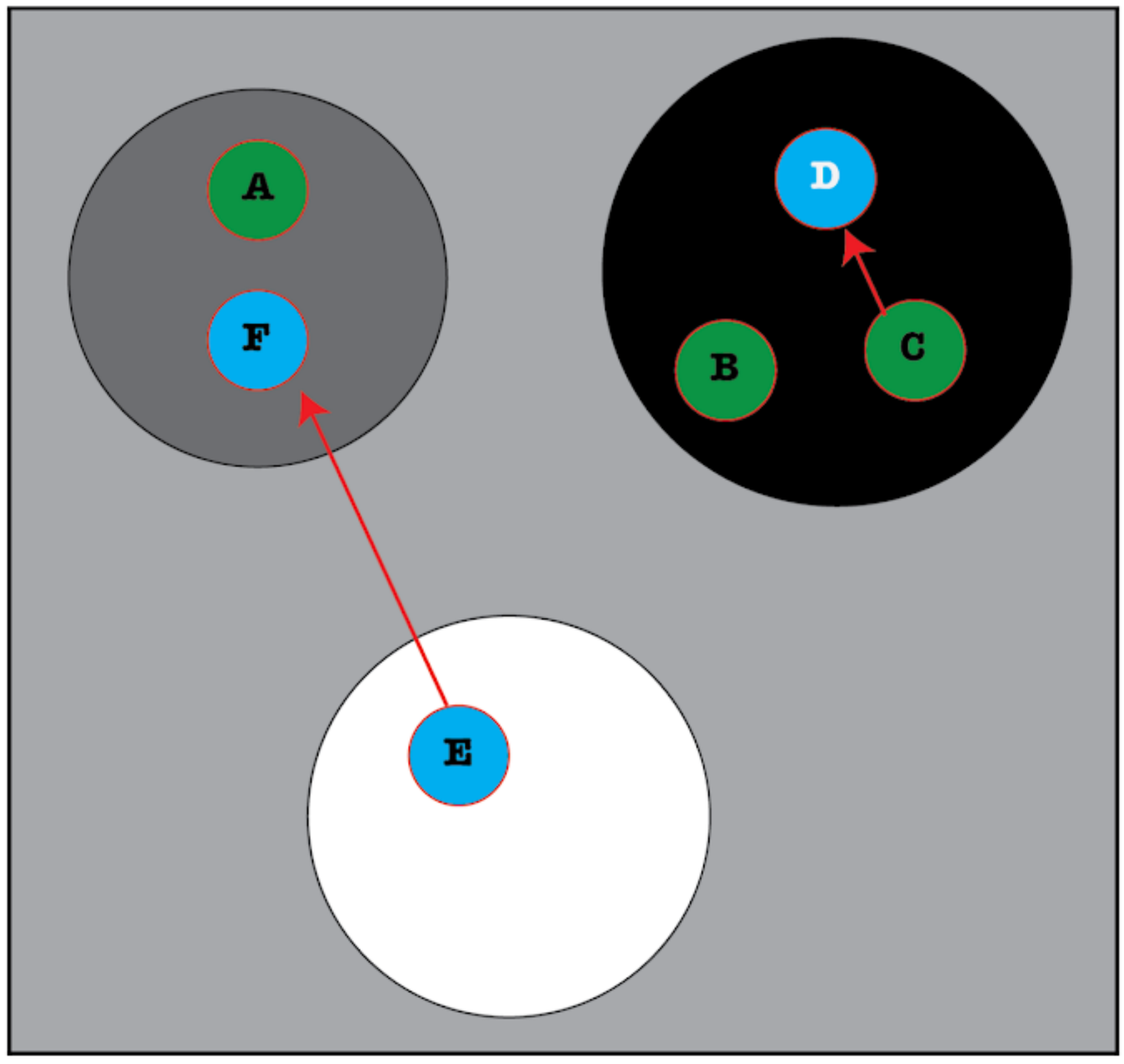
When the garbage collection begins, all objects are white and the garbage collector visits all of the root objects and colors them grey. The **roots** are the objects that can be directly accessed by the application, which includes global variables and other things on the stack. These objects mostly depend on the Go code of a particular program. After this, the garbage collector picks a grey object, makes it black, and starts searching to determine if that object has pointers to other objects of the white set. This means that when a grey object is being scanned for pointers to other objects, it is colored black. If that scan discovers that this particular object has one or more pointers to a white object, it puts that white object in the grey set. This process keeps going for as long as objects exist in the grey set. After that, the objects in the white set are unreachable and their memory space can be reused. Therefore, at this point, the elements of the white set are said to be garbage collected.

If an object of the grey set becomes unreachable at some point in a garbage collection cycle, it will not be collected in that garbage collection cycle but rather in the next one! Although this is not an optimal situation, it is not that bad.

During this process, the running application is called the **mutator**. The mutator runs a small function named **write barrier** that is executed each time a pointer in the heap is modified. If the pointer of an object in the heap is modified, which means that this object is now reachable, the write barrier colors it grey and puts it in the grey set.

The mutator is responsible for the invariant that no element of the black set has a pointer to an element of the white set. This is accomplished with the help of the write barrier function. Failing to accomplish this invariant will ruin the garbage collection process, and it will most likely crash your program in an ugly and undesired way.

As a result, the heap can be viewed as a graph of connected objects, as shown in the following diagram, which demonstrates a single phase of a garbage collection cycle:



The Go garbage collector represents the heap of a program as a graph

Thus, there are three different colors: black, white, and grey. When the algorithm begins, all objects are colored white. As the algorithm continues, white objects are moved into one of the other two sets. The objects that are left in the white set are the ones that will be cleared at some point.

In the preceding graph, you can see that while object **E**, which is in the white set, can access object **F**, it cannot be accessed by any other object because no other object points to object **E**, which makes it a perfect candidate for garbage collection! Additionally, objects **A**, **B**, and **C** are root objects and are always reachable and therefore cannot be garbage collected.

Can you guess what will happen next in that graph? Well, it is not that difficult to fathom that the algorithm will have to process the remaining elements of the grey set, which means that objects **A** and **F** will go into the black set. Object **A** will go into the black set because it is a root element, and **F** will go into the black set because it does not point to any other object while it is in the grey set. After object **A** is garbage collected, object **F** will become unreachable and will be garbage collected in the next cycle of the garbage collector, as an unreachable object cannot magically become reachable in the next iteration of the garbage collection cycle.

The Go garbage collection can also be applied to variables such as **channels**! When the garbage collector finds out that a channel is unreachable and that the channel variable can no longer be accessed, it will free its resources even if the channel has not been closed! You will learn more about channels in `Chapter 9`, *Go Concurrency – Goroutines, Channels and Pipelines*.

Go allows you to initiate a garbage collection manually by putting a `runtime.GC()` statement in your Go code. However, keep in mind that `runtime.GC()` will block the caller, and it might block the entire program, especially if you are running a very busy Go program with many objects. This happens mainly because you cannot perform garbage collections while everything else is rapidly changing, as this will not give the garbage collector the opportunity to identify clearly the members of the white, black, and grey sets! This garbage collection status is also called the **garbage collection safe-point**.

You can find the long and relatively advanced Go code of the garbage collector at `https://github.com/golang/go/blob/master/src/runtime/mgc.go`. You can study this if you want to learn even more about the garbage collection operation. You can even make changes to that code if you are brave enough!

The Go garbage collector is always being improved by the Go team, mainly by trying to make it faster by lowering the number of scans it needs to perform over the data of the three sets . However, despite the various optimizations, the general idea behind the algorithm remains the same.

More about the operation of the Go Garbage Collector

This section will explore the Go garbage collector further and present additional information about its activities.

The main concern of the Go garbage collector is low latency, which basically means short pauses in its operation in order to have a real-time operation. On the other hand, what a program does all the time is to create new objects and manipulate existing objects with pointers. This process can end up creating objects that cannot be accessed any longer because no pointers exist that point to these objects. Such objects are now garbage waiting for the garbage collector to clean them up and free their memory space. After that, the memory space that has been freed is ready to be used again.

The classic algorithm for garbage collection is called mark-and-sweep, and it is the simplest algorithm in use. The way that the **mark-and-sweep algorithm** works is pretty simple and easy to understand: the algorithm stops the program execution (**stop-the-world garbage collector**) in order to visit all of the accessible objects of the heap of a program and *marks* them. After that, it *sweeps* the inaccessible objects. During the mark phase of the algorithm, each object is marked as white, grey, or black. The children of a grey object are colored grey, whereas the original grey object is now colored black. The sweep phase begins when there are no more grey objects to examine. This technique works because there are no pointers from the black set to the white set, which is a fundamental invariant of the algorithm.

Although the mark-and-sweep algorithm is simple, it suspends the execution of the program while it is running, which means that it adds **latency** to the actual process. Go tries to lower that particular latency by running the garbage collector as a concurrent process and using the tricolor algorithm described in the previous section. However, other processes can move pointers or create new objects while the garbage collector runs concurrently. This fact can make things pretty difficult for the garbage collector. As a result, the principal point that allows the tricolor algorithm to run concurrently is to be able to maintain the fundamental invariant of the mark-and-sweep algorithm-no object of the black set can point to an object of the white set.

The solution to this problem is to fix all of the cases that can cause a problem for the algorithm! Therefore, new objects must go to the grey set, because this way the fundamental invariant of the mark-and-sweep algorithm cannot be altered. Additionally, when a pointer of the program is moved, you color the object to which the pointer points as grey. You can say that the grey set acts like a barrier between the white set and the black set. Last, each time a pointer is moved, some Go code gets automatically executed, which is the **write barrier** mentioned earlier that does some recoloring.

The **latency** introduced by the execution of the write barrier code is the price you have to pay for being able to run the garbage collector concurrently.

Note that the **Java** programming language has many garbage collectors that are highly configurable with the help of multiple parameters. One of these Java garbage collectors is called G1, and it is recommended for low-latency applications.

It is really important to remember that the Go garbage collector is a real-time garbage collector, which runs concurrently with the other **goroutines** of a Go program and only optimizes for low latency.

In `Chapter 11`, *Code Testing, Optimization and Profiling*, you will learn how to represent graphically the performance of a program. This chapter also includes information about the operations of the Go garbage collector.

That's enough about garbage collection. The next topic covered will be **unsafe code** and the `unsafe` standard Go package.

Unsafe code

Unsafe code is Go code that bypasses the type safety and the memory security of Go. Most of the time, unsafe code is related to pointers. However, keep in mind that using unsafe code can be dangerous for your programs, so if you are not completely sure that you need to use unsafe code in one of your programs, do not use it!

The use of unsafe code will be illustrated in the `unsafe.go` program, which is presented in three parts.

The first part of `unsafe.go` is as follows:

```
package main
import (
    "fmt"
    "unsafe"
)
```

Note that in order to use unsafe code, you will need to import the `unsafe` standard Go package.

The second part of the program occurs with the following Go code:

```
func main() {
   var value int64 = 5
   var p1 = &value
   var p2 = (*int32)(unsafe.Pointer(p1))
```

Note the use of the `unsafe.Pointer()` function here which allows you, at your own risk, to create a `int32` pointer named `p2` that points to a `int64` variable named `value`, which is accessed using the `p1` pointer. Any Go pointer can be converted to `unsafe.Pointer`.

A pointer of the `unsafe.Pointer` type can override the type system of Go. This is unquestionably fast, but it can also be dangerous if used incorrectly or carelessly. Additionally, it gives developers more control over data.

The last part of `unsafe.go` contains the following Go code:

```
   fmt.Println("*p1: ", *p1)
   fmt.Println("*p2: ", *p2)
   *p1 = 54341234123124312l2
   fmt.Println(value)
   fmt.Println("*p2: ", *p2)
   *p1 = 54341234
   fmt.Println(value)
   fmt.Println("*p2: ", *p2)
}
```

You can **dereference a pointer** and get, use, or set its value using the star character (`*`).

If you execute `unsafe.go`, you will get the following output:

```
$ go run unsafe.go
*p1:  5
*p2:  5
54341234123124312l2
*p2:  -930866580
54341234
*p2:  54341234
```

What does this output tell us? It tells us that a 32-bit pointer cannot store a 64-bit integer!

As you will see in the next section, the functions of the `unsafe` package can do many more interesting things with memory.

About the unsafe package

Now that you have seen the `unsafe` package in action, it's a good time to talk about what makes it a special kind of a package!

First of all, if you look at the source code of the `unsafe` package, you might be a little surprised. On a macOS High Sierra system with Go version 1.9.1 that is installed using Homebrew (`https://brew.sh/`), the source code of the `unsafe` package is located at `/usr/local/Cellar/go/1.9.1/libexec/src/unsafe/unsafe.go` and its contents without the comments are as follows:

```
$ cd /usr/local/Cellar/go/1.9.1/libexec/src/unsafe/
$ grep  -v '^//' unsafe.go | grep -v '^$'
package unsafe
type ArbitraryType int
type Pointer *ArbitraryType
func Sizeof(x ArbitraryType) uintptr
func Offsetof(x ArbitraryType) uintptr
func Alignof(x ArbitraryType) uintptr
```

OK. Where is the rest of the Go code of the `unsafe` package? The answer to that question is relatively simple: the Go compiler implements the `unsafe` package when you import it into your programs.

Many low-level packages such as `runtime`, `syscall`, and `os` constantly use the `unsafe` package.

Another example of the unsafe package

In this subsection, you will learn more about the `unsafe` package and its capabilities with the help of another small Go program named `moreUnsafe.go`. This program will be presented in three parts. What `moreUnsafe.go` does is to access all the elements of an array using pointers.

The first part of the program is as follows:

```
package main

import (
    "fmt"
    "unsafe"
)
```

The second part of `moreUnsafe.go` comes with the following Go code:

```
func main() {
    array := [...]int{0, 1, -2, 3, 4}
    pointer := &array[0]
    fmt.Print(*pointer, " ")
    memoryAddress := uintptr(unsafe.Pointer(pointer)) +
unsafe.Sizeof(array[0])

    for i := 0; i < len(array)-1; i++ {
          pointer = (*int)(unsafe.Pointer(memoryAddress))
          fmt.Print(*pointer, " ")
          memoryAddress = uintptr(unsafe.Pointer(pointer)) +
unsafe.Sizeof(array[0])
    }
```

At first, the `pointer` variable points to the memory address of `array[0]`, which is the first element of the array of integers. Next, the `pointer` variable that points to an integer value is converted to an `unsafe.Pointer()` function and then to `uintptr`. The result is stored in `memoryAddress`.

The value of `unsafe.Sizeof(array[0])` is what gets you to the next element of the array, because this is the memory occupied by each array element. This value is added to the `memoryAddress` variable in each iteration of the `for` loop, which allows you to get the memory address of the next array element. The `*pointer` notation dereferences the pointer and returns the stored integer value.

The third part is as follows:

```
    fmt.Println()
    pointer = (*int)(unsafe.Pointer(memoryAddress))
    fmt.Print("One more: ", *pointer, " ")
    memoryAddress = uintptr(unsafe.Pointer(pointer)) +
unsafe.Sizeof(array[0])
    fmt.Println()
}
```

In the last part, you are trying to access an element of the array that does not exist using pointers and memory addresses. The Go compiler cannot catch such a logical error due to the use of the `unsafe` package and therefore will return something inaccurate.

Executing `moreUnsafe.go` will create the following output:

```
$ go run moreUnsafe.go
0 1 -2 3 4
One more: 842350722816
```

You have now accessed all of the elements of a Go array using pointers! However, the real problem here is that when you tried to access an invalid array element, the program did not complain and returned a random number instead.

Calling C code from Go

Although the intention of Go is to make your programming experience better and save you from having to deal with the quirks of C, C remains a very capable programming language that is still useful. This means that there are situations, such as using a database or a device driver written in C, which still require the use of C. This means that you will need to work with C code in your Go projects.

If you find yourselves using this capability several times in the same project, you might need to reconsider your approach or your choice of programming language!

Calling C code from Go using the same file

The simplest way to call C code from a Go program is to include the C code in your Go source file. This requires a special treatment, but it is pretty fast and not that difficult to do.

The name of the Go source file that contains both C and Go code will be `cGo.go`, and it will be presented in three parts.

The first part of the Go source file is as follows:

```
package main

//#include <stdio.h>
//void callC() {
//    printf("Calling C code!\n");
//}
import "C"
```

As you can see, the C code is included in the comments of the Go program. However, the `go` tool knows what to do with these kinds of comments because of the use of the `C` Go package.

The second part of the program contains the following Go code:

```
import "fmt"

func main() {
```

All of the other packages should be imported separately.

The last part of `cGo.go` contains the following code:

```
    fmt.Println("A Go statement!")
    C.callC()
    fmt.Println("Another Go statement!")
}
```

Thus, in order to execute the `callC()` C function, you will need to call it as `C.callC()`.

Executing `cGo.go` will create the following output:

```
$ go run cGo.go
A Go statement!
Calling C code!
Another Go statement!
```

Calling C code from Go using separate files

Now let's continue learning how to call C code from a Go program when the C code is located in a separate file.

First, let me explain the imaginary problem that we will solve with our program. We will need to use two C functions that we have implemented in the past and that we do not want or cannot rewrite in Go.

The C code

This subsection will present the C code for the example which comes in two files: `callC.h` and `callC.c`. The include file (`callC.h`) contains the following code:

```
#ifndef CALLC_H
#define CALLC_H

void cHello();
void printMessage(char* message);

#endif
```

The C source file (`callC.c`) contains the following C code:

```
#include <stdio.h>
#include "callC.h"

void cHello() {
    printf("Hello from C!\n");
}

void printMessage(char* message) {
    printf("Go send me %s\n", message);
}
```

Both the `callC.c` and `callC.h` files are stored in a separate directory, which in this case is `callClib`. You can use any directory name you want, however.

The actual C code is not important as long as you call the right C functions with the correct type and number of parameters. There is nothing in the C code that tells you that it will be used from a Go program. You should look at the Go code for the juicy part.

The Go code

This subsection will address the Go source code of the example, which will be called `callC.go` and will be presented in three parts.

The first part of `callC.go` includes the following Go code:

```
package main

// #cgo CFLAGS: -I${SRCDIR}/callClib
// #cgo LDFLAGS: ${SRCDIR}/callC.a
// #include <stdlib.h>
// #include <callC.h>
import "C"
```

The single most important Go statement of the entire Go source file is the inclusion of the `C` package using a separate `import` statement. However, `C` is a virtual Go package that just tells `go build` to preprocess its input file using the `cgo` tool before the Go compiler processes the file! You can still see that you need to use comments to inform the Go program about the C code. In this case, you tell `callC.go` where to find the `callC.h` file as well as where to find the `callC.a` library file that we will create in a little while—such lines begin with `#cgo`.

The second part of the program is as follows:

```
import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println("Going to call a C function!")
    C.cHello()
```

The last part of `callC.go` follows:

```
    fmt.Println("Going to call another C function!")
    myMessage := C.CString("This is Mihalis!")
    defer C.free(unsafe.Pointer(myMessage))
    C.printMessage(myMessage)

    fmt.Println("All perfectly done!")
}
```

In order to pass a string to a C function from Go, you will need to create a C string using `C.CString()`. Additionally, you will need a `defer` statement in order to free the memory space of the C string when it is no longer needed. The `defer` statement includes a call to `C.free()` and another one to `unsafe.Pointer()`.

In the next section, you will see how to compile and execute `callC.go`.

Mixing Go and C code

Now that you have the C and the Go code, it is time to learn what to do next in order to execute the Go file that calls the C code.

The good news is that you do not need to do anything particularly difficult because all of the critical information is contained in the Go file. The only critical thing that you will need to do is to compile the C code in order to create a library, which requires the execution of the following commands:

```
$ ls -l callClib/
total 16
-rw-r--r--@ 1 mtsouk  staff  162 Oct 31 18:14 callC.c
-rw-r--r--@ 1 mtsouk  staff   89 Oct 31 18:14 callC.h
$ gcc -c callClib/*.c
$ ls -l callC.o
-rw-r--r--  1 mtsouk  staff  944 Oct 31 18:14 callC.o
$ file callC.o
callC.o: Mach-O 64-bit object x86_64
$ ar rs callC.a *.o
ar: creating archive callC.a
$ ls -l callC.a
-rw-r--r--  1 mtsouk  staff  1152 Oct 31 18:15 callC.a
$ file callC.a
callC.a: current ar archive random library
$ rm callC.o
```

After that, you will have a file named `callC.a` located in the same directory as the `callC.go` file. The `gcc` executable is the name of the **C compiler**.

Now you are ready to compile the file with the Go code and create a new executable file:

```
$ go build callC.go
$ ls -l callC
-rwxr-xr-x  1 mtsouk  staff  2322000 Oct 31 18:16 callC
$ file callC
callC: Mach-O 64-bit executable x86_64
```

Executing the `callC` executable file will create the following output:

```
$ ./callC
Going to call a C function!
Hello from C!
Going to call another C function!
Go send me This is Mihalis!
All perfectly done!
```

If you will call a small amount of C code, then using a single Go file for both the C and Go code is highly recommended because of its simplicity. However, if you will do something more complex and advanced, creating a static C library should be your preferred method.

Calling Go functions from C code

It is also possible to call a Go function from your C code. Therefore, this section will present a small example where two Go functions will be called from a C program. The Go package will be converted into a **C shared library** that will be used in the C program.

The Go package

This subsection will present you with the code of the Go package that will be used in a C program. The name of the Go package needs to be `main`, but its filename can be anything you want. In this case, the filename will be `usedByC.go`, and it will be presented in three parts.

You will learn more about Go packages in `Chapter 6`, *What You Might Not Know About Go Packages*.

The first part of the code of the Go package is as follows:

```
package main

import "C"

import (
    "fmt"
)
```

As I mentioned before, it is mandatory that you name the Go package as `main`. You will also need to import the `C` package in your Go code.

The second part of the program contains the following Go code:

```
//export PrintMessage
func PrintMessage() {
   fmt.Println("A Go function!")
}
```

Each Go function that will be called by the C code needs to be exported first. This means that you should put a comment line starting with `//export` before its implementation. After `//export`, you will need to put the name of the function because this is what the C code will use.

The last part of `usedByC.go` is as follows:

```
//export Multiply
func Multiply(a, b int) int {
   return a * b
}

func main() {
}
```

The `main()` function of `usedByC.go` needs no code because it will not be exported and therefore used by the C program. Additionally, as you also want to export the `Multiply()` function, you will need to put `//export Multiply` before its implementation.

After this, you will need to generate a C shared library from the Go code by executing the following command:

```
$ go build -o usedByC.o -buildmode=c-shared usedByC.go
```

The preceding command will generate two files named `usedByC.h` and `usedByC.o`:

```
$ ls -l usedByC.*
-rw-r--r--@ 1 mtsouk  staff      204 Oct 31 20:37 usedByC.go
-rw-r--r--  1 mtsouk  staff     1365 Oct 31 20:40 usedByC.h
-rw-r--r--  1 mtsouk  staff  2329472 Oct 31 20:40 usedByC.o
$ file usedByC.o
usedByC.o: Mach-O 64-bit dynamically linked shared library x86_64
```

You should not make any changes to `usedByC.h`.

The C code

The relevant C code can be found in the `willUseGo.c` source file, which will be presented in two parts. The first part of `willUseGo.c` is as follows:

```
#include <stdio.h>
#include "usedByC.h"

int main(int argc, char **argv) {
    GoInt x = 12;
    GoInt y = 23;

    printf("About to call a Go function!\n");
    PrintMessage();
```

If you already know C, you should understand why you need to include `usedByC.h`. This is the way that the C code knows about the available functions of a library.

The second part of the C program follows next:

```
    GoInt p = Multiply(x,y);
    printf("Product: %d\n",(int)p);
    printf("It worked!\n");
    return 0;
}
```

The `GoInt p` variable is needed for getting an integer value from a Go function, which is converted to a C integer using the `(int) p` notation.

Compiling and executing `willUseGo.c` on a macOS High Sierra machine will create the following output:

```
$ gcc -o willUseGo willUseGo.c ./usedByC.o
$ ./willUseGo
About to call a Go function!
A Go function!
Product: 276
It worked!
```

The defer keyword

The `defer` keyword postpones the execution of a function until the surrounding function returns. It is widely used in file input and output operations because it saves you from having to remember when to close an opened file: the `defer` keyword allows you to put the function call that closes an opened file near to the function call that opened it. As you will learn about the use of `defer` in file-related operations in `Chapter 8`, *Telling a Unix System What to Do*, this section will present a different usage of `defer`. You will also see `defer` in action in the section that talks about the `panic()` and `recover()` built-in Go functions.

It is very important to remember that **deferred functions** are executed in **Last In First Out** (**LIFO**) order after the return of the surrounding function. Put simply, this means that if you `defer` function `f1()` first, function `f2()` second, and function `f3()` third in the same surrounding function, when the surrounding function is about to return, function `f3()` will be executed first, function `f2()` will be executed second, and function `f1()` will be the last one to get executed.

As this definition of `defer` is a little unclear, I think that you will understand the use of `defer` a little better by looking at the Go code and the output of the `defer.go` program, which will be presented in three parts.

The first part of the program is as follows:

```
package main

import (
    "fmt"
)

func d1() {
    for i := 3; i > 0; i-- {
            defer fmt.Print(i, " ")
    }
}
```

Apart from the `import` block, the preceding Go code implements a function named `d1()` with a `for` loop and a `defer` statement that will be executed three times.

The second part of `defer.go` contains the following Go code:

```
func d2() {
    for i := 3; i > 0; i-- {
            defer func() {
                    fmt.Print(i, " ")
```

```
            }()
        }
        fmt.Println()
    }
```

In this part of the code, you can see the implementation of another function that is named `d2()`. The `d2()` function also contains a `for` loop and a `defer` statement that will be also executed three times. However, this time the `defer` keyword is applied to an **anonymous function** instead of a single `fmt.Print()` statement. Additionally, the anonymous function takes no parameters.

The last part of the Go code is as follows:

```
func d3() {
    for i := 3; i > 0; i-- {
            defer func(n int) {
                fmt.Print(n, " ")
            }(i)
    }
}

func main() {
    d1()
    d2()
    fmt.Println()
    d3()
    fmt.Println()
}
```

Apart from the `main()` function that calls the `d1()`, `d2()`, and `d3()` functions, you can also see the implementation of the `d3()` function, which has a `for` loop that uses the `defer` keyword on an anonymous function. However, this time the anonymous function requires one integer parameter named `n`. The Go code tells us that the `n` parameter takes its value from the `i` variable used in the `for` loop.

Executing `defer.go` will create the following output:

```
$ go run defer.go
1 2 3
0 0 0
1 2 3
```

You will most likely find the generated output complicated and challenging to understand. This underscores the fact that the operation and the results of the use of `defer` can be tricky if your code is not clear and unambiguous.

Let's examine the results in order to get a better idea of how tricky `defer` can be if you do not pay close attention to your code. We will start with the first line of the output (`1 2 3`), which is generated by the `d1()` function. The values of `i` in `d1()` are `3`, `2`, and `1` in that order. The function that is deferred in `d1()` is the `fmt.Print()` statement. As a result, when the `d1()` function is about to return, you get the three values of the `i` variable of the `for` loop in reverse order because deferred functions are executed in LIFO order.

Now, let's inspect the second line of the output that is produced by the `d2()` function. It is really strange that we got three zeros instead of `1 2 3` in the output. The reason for this, however, is relatively simple. After the `for` loop has ended, the value of `i` is `0`, because it is that value of `i` that made the `for` loop terminate. However, the tricky part here is that the deferred anonymous function is evaluated after the `for` loop ends, because it has no parameters. This means that is evaluated three times for an `i` value of `0`, hence the generated output! This kind of confusing code is what might lead to the creation of nasty bugs in your projects, so try to avoid it!

Also, we will talk about the third line of the output, which is generated by the `d3()` function. Due to the parameter of the anonymous function, each time the anonymous function is deferred, it gets and uses the current value of `i`. As a result, each execution of the anonymous function has a different value to process, thus the generated output.

After this, it should be clear that the best approach to the use of `defer` is the third one, which is exhibited in the `d3()` function. This is so because you intentionally pass the desired variable in the anonymous function in an easy to understand way.

Panic and Recover

This section will present you with a tricky technique that was first mentioned in `Chapter 1`, *Go and the Operating System*. This technique involves the use of the `panic()` and `recover()` functions, and it will be presented in `panicRecover.go`, which you will review in three parts.

Strictly speaking, `panic()` is a built-in Go function that terminates the current flow of a Go program and starts panicking! On the other hand, the `recover()` function, which is also a built-in Go function, allows you to take back the control of a **goroutine** that just panicked using `panic()`.

The first part of the program follows:

```
package main

import (
    "fmt"
)

func a() {
    fmt.Println("Inside a()")
    defer func() {
          if c := recover(); c != nil {
                fmt.Println("Recover inside a()!")
          }
    }()
    fmt.Println("About to call b()")
    b()
    fmt.Println("b() exited!")
    fmt.Println("Exiting a()")
}
```

Apart from the `import` block, this part includes the implementation of the `a()` function. The most important part of the `a()` function is the `defer` block of code, which implements an anonymous function that will be called when there is a call to `panic()`.

The second code segment of `panicRecover.go` follows next:

```
func b() {
    fmt.Println("Inside b()")
    panic("Panic in b()!")
    fmt.Println("Exiting b()")
}
```

The last part of the program, which illustrates the `panic()` and `recover()` functions, is as follows:

```
func main() {
    a()
    fmt.Println("main() ended!")
}
```

Executing `panicRecover.go` will create the following output:

```
$ go run panicRecover.go
Inside a()
About to call b()
Inside b()
Recover inside a()!
main() ended!
```

What just happened here is really impressive! However, as you can see from the output, the `a()` function did not end normally, because its last two statements did not get executed:

```
    fmt.Println("b() exited!")
    fmt.Println("Exiting a()")
```

Nevertheless, the good thing is that `panicRecover.go` ended according to our will without panicking because the anonymous function used in `defer` took control of the situation! Also note that function `b()` knows nothing about function `a()`. However, function `a()` contains Go code that handles the panic condition of the `b()` function!

Using the panic function on its own

You can also use the `panic()` function on its own without any attempt to **recover**, and this subsection will show you its results using the Go code of `justPanic.go`, which will be presented in two parts.

The first part of `justPanic.go` follows next:

```
package main

import (
    "fmt"
    "os"
)
```

As you can see, the use of `panic()` does not require any extra Go packages.

The second part of `justPanic.go` is shown in the following Go code:

```
func main() {
   if len(os.Args) == 1 {
         panic("Not enough arguments!")
   }

   fmt.Println("Thanks for the argument(s)!")
}
```

If your Go program does not have at least one command-line argument, it will call the `panic()` function. The `panic()` function takes one parameter, which is the error message that you want to print on the screen.

Executing `justPanic.go` on a macOS High Sierra machine will create the following output:

```
$ go run justPanic.go
panic: Not enough arguments!
goroutine 1 [running]:
main.main()
      /Users/mtsouk/ch2/code/justPanic.go:10 +0x9e
exit status 2
```

Thus, using the `panic()` function on its own will terminate the Go program without giving you the opportunity to recover! Therefore use of the `panic()` and `recover()` pair is much more practical and professional than just using `panic()` alone.

The output of the `panic()` function looks like the output of the `Panic()` function from the `log` package. However, the `panic()` function sends nothing to the logging service of your Unix machine.

Two handy Unix utilities

There are times that a Unix program fails for some unknown reason or does not perform well, and you want to find out why without having to rewrite your code and add a plethora of debugging statements.

Consequently, this section presents two command-line utilities that allow you to see the C system calls executed by an executable file. The names of the two tools are `strace(1)` and `dtrace(1)`, and they allow you to inspect the operation of a program.

Remember that, at the end of the day, all programs that work on Unix machines end up using C system calls to communicate with the Unix kernel and perform most of their tasks.

Although both tools can work with the `go run` command, you will get less unrelated output if you first create an executable file using `go build` and use that file. This occurs mainly because, as you already know, `go run` creates various temporary files before actually running your Go code, and both tools will see that and try to display information about the temporary files, which is not what you want.

The strace tool

The `strace(1)` command-line utility allows you to trace system calls and signals. As `strace(1)` is not available on macOS, this section will use a Debian Linux machine to showcase `strace(1)`.

The `strace(1)` tool only works on Linux machines.

The output that `strace(1)` generates looks like the following:

```
$ strace ls
execve("/bin/ls", ["ls"], [/* 15 vars */]) = 0
brk(0)                                  = 0x186c000
fstat(3, {st_mode=S_IFREG|0644, st_size=35288, ...}) = 0
```

The `strace(1)` output displays each system call with its parameters as well as its return value. Note that in the Unix world, a return value of 0 is a good thing!

In order to process a binary file, you will need to put the `strace(1)` command in front of the executable that you want to process. However, you will need to interpret the output on your own in order to use it to make useful conclusions. The good thing is that tools like `grep(1)` can get you the output that you are actually seeking:

```
$ strace find /usr 2>&1 | grep ioctl
ioctl(0, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or TCGETS,
0x7ffe3bc59c50) = -1 ENOTTY (Inappropriate ioctl for device)
ioctl(1, SNDCTL_TMR_TIMEBASE or SNDRV_TIMER_IOCTL_NEXT_DEVICE or TCGETS,
0x7ffe3bc59be0) = -1 ENOTTY (Inappropriate ioctl for device)
```

The `strace(1)` tool can count time, calls, and errors for each system call when used with the `-c` command-line option:

```
$ strace -c find /usr 1>/dev/null
% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ------------
 82.88    0.063223           2     39228           getdents
 16.60    0.012664           1     19587           newfstatat
  0.16    0.000119           0     19618        13 open
```

As the normal program output is printed in standard output form, and the output of `strace(1)` is printed in standard error form, the previous command discards the output of the command that is examined and shows the output of `strace(1)`. As you can see from the last line of the output, the `open(2)` system call was called 19,618 times, generated 13 errors, and took about 0.16% percent of the execution time of the entire command, or about 0.000119 seconds.

The dtrace tool

Although debugging utilities such as `strace(1)` and `truss(1)` can trace system calls produced by a process, they can be slow and therefore not appropriate for solving performance problems on busy Unix systems. Another tool named **DTrace** allows you to see what happens behind the scenes on a system-wide basis without the need to modify or recompile anything. It also allows you to work on production systems and watch running programs or server processes dynamically without introducing a big overhead.

Although there is a version of `dtrace(1)` that works on Linux, the `dtrace(1)` tool works best on macOS and the other FreeBSD variants.

This subsection will use the `dtruss(1)` command-line utility that comes with macOS, which is just a `dtrace(1)` script that shows the system calls of a process and saves you from having to write `dtrace(1)` code. Note that both `dtrace(1)` and `dtruss(1)` need root privileges to run.

The output that `dtruss(1)` generates looks like the following:

```
$ sudo dtruss godoc
ioctl(0x3, 0x80086804, 0x7FFEEFBFEC20)            = 0 0
close(0x3)          = 0 0
access("/AppleInternal/XBS/.isChrooted\0", 0x0, 0x0)    = -1 Err#2
thread_selfid(0x0, 0x0, 0x0)          = 1895378 0
geteuid(0x0, 0x0, 0x0)          = 0 0
getegid(0x0, 0x0, 0x0)          = 0 0
```

So, `dtruss(1)` works in the same way as the `strace(1)` utility. Analogously to `strace(1)`, `dtruss(1)` will print system call counts when used with the `-c` parameter:

```
$ sudo dtruss -c go run unsafe.go 2>&1
CALL                                        COUNT
access                                          1
bsdthread_register                              1
getuid                                          1
ioctl                                           1
issetugid                                       1
kqueue                                          1
write                                           1
read                                          244
kevent                                        474
fcntl                                         479
lstat64                                       553
```

The preceding output will quickly inform you about potential bottlenecks in your Go code or allow you to compare the performance of two different command-line programs.

You need to get used to using utilities such as `strace(1)`, `dtrace(1)`, and `dtruss(1)`, but such tools can make your lives so much easier and better. I strongly suggest that you start learning at least one such tool right now!

You can learn more about the `dtrace(1)` utility by reading *DTrace: Dynamic Tracing in Oracle Solaris, Mac OS X and FreeBSD* by **Brendan Gregg** and **Jim Mauro** (Prentice Hall, 2011) and by visiting `http://dtrace.org/`. Please bear in mind that `dtrace(1)` is much more powerful than `strace(1)` because it has its own programming language. However, `strace(1)` is more versatile when all you want to do is to watch the system calls of an executable file.

Your Go environment

This section will help you find out information about your current Go environment using the functions and properties of the `runtime` package. The name of the program that will be developed in this section is `goEnv.go`, and it will be presented in two parts.

The first part of `goEnv.go` follows:

```
package main

import (
    "fmt"
    "runtime"
)
```

As you will see shortly, the `runtime` package contains functions and properties that will reveal the desired information. The second portion of the code of `goEnv.go` contains the implementation of the `main()` function:

```
func main() {
    fmt.Print("You are using ", runtime.Compiler, " ")
    fmt.Println("on a", runtime.GOARCH, "machine")
    fmt.Println("Using Go version", runtime.Version())
    fmt.Println("Number of CPUs:", runtime.NumCPU())
    fmt.Println("Number of Goroutines:", runtime.NumGoroutine())
}
```

Executing `goEnv.go` on a macOS High Sierra machine with Go version 1.9.2 will create the following output:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.9.2
Number of CPUs: 8
Number of Goroutines: 1
```

The same program generates the following output on a Debian Linux machine with Go version 1.3.3:

```
$ go run goEnv.go
You are using gc on a amd64 machine
Using Go version go1.3.3
Number of CPUs: 1
Number of Goroutines: 4
```

The real benefit you can get from being able to find information about your Go environment, however, is illustrated in the next program, named `requiredVersion.go`, which tells you if you are using Go version 1.8 or higher:

```
package main

import (
    "fmt"
    "runtime"
    "strconv"
    "strings"
)

func main() {
    myVersion := runtime.Version()
    major := strings.Split(myVersion, ".")[0][2]
    minor := strings.Split(myVersion, ".")[1]
    m1, _ := strconv.Atoi(string(major))
    m2, _ := strconv.Atoi(minor)

    if m1 == 1 && m2 < 8 {
          fmt.Println("Need Go version 1.8 or higher!")
          return
    }
    fmt.Println("You are using Go version 1.8 or higher!")
}
```

The `strings` Go standard package is used for splitting the Go version string you get from `runtime.Version()` in order to get its first two parts, whereas the `strconv.Atoi()` function is used for converting a string to an integer.

Executing `requiredVersion.go` on the macOS High Sierra machine will create the following output:

```
$ go run requiredVersion.go
You are using Go version 1.8 or higher!
```

If you run `requiredVersion.go` on the Debian Linux machine, however, it will generate the following output:

```
$ go run requiredVersion.go
Need Go version 1.8 or higher!
```

Thus using the Go code of `requiredVersion.go`, you will be able to identify whether your Unix machine has the required Go version or not.

The Go Assembler

This section will briefly talk about the assembly language and the **Go assembler**, which is a Go tool that allows you to see the assembly language used by the Go compiler.

As an example, you can see the assembly language of the `goEnv.go` program from the previous section by executing the following command:

```
$ GOOS=darwin GOARCH=amd64 go tool compile -S goEnv.go
```

The value of the `GOOS` variable defines the name of the target operating system, whereas the value of the `GOARCH` variable defines the compilation architecture. The preceding command was executed on a macOS High Sierra machine, hence the use of the `darwin` value for the `GOOS` variable.

The output of the preceding command is pretty large, even for a small program such as `goEnv.go`. Some of the output is shown next:

```
"".main STEXT size=859 args=0x0 locals=0x118
   0x0000 00000 (goEnv.go:8)       TEXT    "".main(SB), $280-0
   0x00be 00190 (goEnv.go:9)       PCDATA  $0, $1
   0x0308 00776 (goEnv.go:13)      PCDATA  $0, $5
   0x0308 00776 (goEnv.go:13)      CALL    runtime.convT2E64(SB)
"".init STEXT size=96 args=0x0 locals=0x8
   0x0000 00000 (<autogenerated>:1) TEXT    "".init(SB), $8-0
   0x0000 00000 (<autogenerated>:1) MOVQ    (TLS), CX
   0x001d 00029 (<autogenerated>:1) FUNCDATA        $0, gclocals
d4dc2f11db048877dbc0f60a22b4adb3(SB)
```

```
    0x001d 00029 (<autogenerated>:1) FUNCDATA        $1, gclocals
33cdeccccebe80329f1fdbee7f5874cb(SB)
```

The lines that contain the `FUNCDATA` and `PCDATA` directives are read and used by the Go **garbage collector** and are automatically generated by the Go compiler.

An equivalent variant of the preceding command follows:

```
$ GOOS=darwin GOARCH=amd64 go build -gcflags -S goEnv.go
```

The list of valid `GOOS` values includes `android`, `darwin`, `dragonfly`, `freebsd`, `linux`, `nacl`, `netbsd`, `openbsd`, `plan9`, `solaris`, `windows`, and `zos`. On the other hand, the list of valid `GOARCH` values includes `386`, `amd64`, `amd64p32`, `arm`, `armbe`, `arm64`, `arm64be`, `ppc64`, `ppc64le`, `mips`, `mipsle`, `mips64`, `mips64le`, `mips64p32`, `mips64p32le`, `ppc`, `s390`, `s390x`, `sparc`, and `sparc64`.

If you are really interested in the Go assembler and you want to learn more, visit `https://golang.org/doc/asm`.

Node Trees

Now it's time for something completely different and low level. You are free to skip this section if you like, but I bet that you are not reading this book for its easy topics but for the difficult ones, such as **node trees**!

The `go tool 6g -W test.go` command does not work on newer Go versions. You should use `go tool compile -W test.go` instead.

A Go node is a `struct` with a large number of properties. You will learn more about defining and using Go **structures** in `Chapter 4`, *The Uses of Composite Types*. Everything in a Go program is being parsed and analyzed by the modules of the Go compiler according to the grammar of the Go programming language. The final product of this analysis is a **tree** that is specific to the provided Go code, and it represents the program in a different way that is suited for the compiler rather than for the developer.

This section will start with the following Go code, which is saved as `nodeTree.go`. It is presented here as an example of the kind of low-level information that the `go` tool can provide:

```
package main

import (
    "fmt"
)

func main() {
    fmt.Println("Hello there!")
}
```

The Go code of `nodeTree.go` is pretty easy to understand, so you will not be surprised by its output, which comes next:

```
$ go run nodeTree.go
Hello there!
```

Now it's time to see some of the internal workings of Go by executing the following command:

```
$ go tool compile -W nodeTree.go
before main
.   CALLFUNC l(8) tc(1) STRUCT-(int, error)
.   .   NAME-fmt.Println a(true) l(4) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
.   .   DDDARG l(8) esc(no) PTR64-*[1]interface {}
.   CALLFUNC-list
.   .   CONVIFACE l(8) esc(h) tc(1) implicit(true) INTER-interface {}
.   .   .   NAME-main.statictmp_0 a(true) l(8) x(0) class(PEXTERN) f(1)
tc(1) used string
.   VARKILL l(8) tc(1)
.   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) used
ARRAY-[1]interface {}
after walk main
.   CALLFUNC-init
.   .   AS l(8) tc(1)
.   .   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
.   .   AS l(8) tc(1)
.   .   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used PTR64-*[1]interface {}
.   .   .   ADDR l(8) tc(1) PTR64-*[1]interface {}
.   .   .   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) addrtaken assigned used ARRAY-[1]interface {}
```

```
.   .   BLOCK l(8)
.   .   BLOCK-list
.   .   .   AS l(8) tc(1) hascall
.   .   .   .   INDEX l(8) tc(1) assigned bounded hascall INTER-interface
{}
.   .   .   .   .   IND l(8) tc(1) implicit(true) assigned hascall ARRAY-
[1]interface {}
.   .   .   .   .   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO)
esc(N) tc(1) assigned used PTR64-*[1]interface {}
.   .   .   .   .   LITERAL-0 a(true) l(8) tc(1) int
.   .   .   .   EFACE l(8) tc(1) INTER-interface {}
.   .   .   .   .   ADDR a(true) l(8) tc(1) PTR64-*uint8
.   .   .   .   .   .   NAME-type.string a(true) x(0) class(PEXTERN) tc(1)
uint8
.   .   .   .   .   ADDR l(8) tc(1) PTR64-*string
.   .   .   .   .   .   NAME-main.statictmp_0 a(true) l(8) x(0)
class(PEXTERN) f(1) tc(1) addrtaken used string
.   .   BLOCK l(8)
.   .   BLOCK-list
.   .   .   AS l(8) tc(1) hascall
.   .   .   .   NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
.   .   .   .   SLICEARR l(8) tc(1) hascall SLICE-[]interface {}
.   .   .   .   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO)
esc(N) tc(1) assigned used PTR64-*[1]interface {}
.   CALLFUNC l(8) tc(1) hascall STRUCT-(int, error)
.   .   NAME-fmt.Println a(true) l(4) x(0) class(PFUNC) tc(1) used FUNC-
func(...interface {}) (int, error)
.   .   DDDARG l(8) esc(no) PTR64-*[1]interface {}
.   CALLFUNC-list
.   .   AS l(8) tc(1)
.   .   .   INDREGSP-SP a(true) l(8) x(0) tc(1) addrtaken main.__ SLICE-
[]interface {}
.   .   .   NAME-main..autotmp_1 a(true) l(8) x(0) class(PAUTO) esc(N)
tc(1) assigned used SLICE-[]interface {}
.   VARKILL l(8) tc(1)
.   .   NAME-main..autotmp_0 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1)
addrtaken assigned used ARRAY-[1]interface {}
before init
.   IF l(1) tc(1)
.   .   GT l(1) tc(1) bool
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 a(true) l(1) tc(1) uint8
.   IF-body
.   .   RETURN l(1) tc(1)
.   IF l(1) tc(1)
.   .   EQ l(1) tc(1) bool
```

```
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 a(true) l(1) tc(1) uint8
.   IF-body
.   .   CALLFUNC l(1) tc(1)
.   .   .   NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-1 a(true) l(1) tc(1) uint8
.   CALLFUNC l(1) tc(1)
.   .   NAME-fmt.init a(true) l(4) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-2 a(true) l(1) tc(1) uint8
.   RETURN l(1) tc(1)
after walk init
.   IF l(1) tc(1)
.   .   GT l(1) tc(1) bool
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 a(true) l(1) tc(1) uint8
.   IF-body
.   .   RETURN l(1) tc(1)
.   IF l(1) tc(1)
.   .   EQ l(1) tc(1) bool
.   .   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1)
assigned used uint8
.   .   .   LITERAL-1 a(true) l(1) tc(1) uint8
.   IF-body
.   .   CALLFUNC l(1) tc(1) hascall
.   .   .   NAME-runtime.throwinit a(true) x(0) class(PFUNC) tc(1) used
FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-1 a(true) l(1) tc(1) uint8
.   CALLFUNC l(1) tc(1) hascall
.   .   NAME-fmt.init a(true) l(4) x(0) class(PFUNC) tc(1) used FUNC-func()
.   AS l(1) tc(1)
.   .   NAME-main.initdone  a(true) l(1) x(0) class(PEXTERN) tc(1) assigned
used uint8
.   .   LITERAL-2 a(true) l(1) tc(1) uint8
.   RETURN l(1) tc(1)
```

As you can appreciate, the Go compiler and its tools do many things behind the scenes, even for a small program such as `nodeTree.go`.

The `-W` parameter tells the `go tool compile` command to print the **debug parse tree** after the type checking.

Look at the output of the next two commands:

```
$ go tool compile -W nodeTree.go | grep before
before main
before init
$ go tool compile -W nodeTree.go | grep after
after walk main
after walk init
```

As you can see, the `before` keyword is about the beginning of the execution of a function. If your program had more functions, you would have gotten more output. This is illustrated in the following example:

```
$ go tool compile -W defer.go | grep before
before d1
before d2
before d3
before main
before d2.func1
before d3.func1
before init
before type..hash.[2]interface {}
before type..eq.[2]interface {}
```

The preceding example uses the Go code of `defer.go`, which is much more complicated than `nodeTree.go`. However, it should be obvious than the `init()` function is automatically created by Go as it exists in both examples (`nodeTree.go` and `defer.go`).

I will now present you with a juicier version of `nodeTree.go`, named `nodeTreeMore.go`:

```
package main

import (
    "fmt"
)

func functionOne(x int) {
    fmt.Println(x)
```

```
}

func main() {
   varOne := 1
   varTwo := 2
   fmt.Println("Hello there!")
   functionOne(varOne)
   functionOne(varTwo)
}
```

The `nodeTreeMore.go` program has two variables, named `varOne` and `varTwo`, and one additional function named `functionOne`. Searching the output of `go tool compile -W` for `varOne`, `varTwo`, and `functionOne` reveals the following information:

```
$ go tool compile -W nodeTreeMore.go  | grep functionOne | uniq
before functionOne
after walk functionOne
.   .   NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used
FUNC-func(int)
$ go tool compile -W nodeTreeMore.go  | grep varTwo | uniq
.   .   NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) f(1) tc(1)
used int
.   .   .   NAME-main.varTwo a(true) g(2) l(13) x(0) class(PAUTO) f(1)
tc(1) used int
$ go tool compile -W nodeTreeMore.go  | grep varOne | uniq
.   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) f(1) tc(1)
used int
.   .   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) f(1)
tc(1) used int
```

Thus, `varOne` is represented as `NAME-main.varOne` while `varTwo` is indicated by `NAME-main.varTwo`. The `functionOne()` function is referenced as `NAME-main.functionOne`. Consequently, the `main()` function is referenced as `NAME-main`.

Now let's examine the following code of the debug parse tree of `nodeTreeMore.go`:

```
before functionOne
.   AS l(8) tc(1)
.   .   NAME-main..autotmp_2 a(true) l(8) x(0) class(PAUTO) esc(N) tc(1)
assigned used int
.   .   NAME-main.x a(true) g(1) l(7) x(0) class(PPARAM) f(1) tc(1) used
int
```

This data is related to the definition of `functionOne()`. The `l(8)` string tells us that the definition of this node can be found in line 8; that is, after reading line 7. The `NAME-main..autotmp_2` integer variable is automatically generated by the compiler.

The next part of the debug parse tree output that I will explain below follows:

```
.   CALLFUNC l(15) tc(1)
.   .   NAME-main.functionOne a(true) l(7) x(0) class(PFUNC) tc(1) used
FUNC-func(int)
.   CALLFUNC-list
.   .   NAME-main.varOne a(true) g(1) l(12) x(0) class(PAUTO) f(1) tc(1)
used int
```

The first line says that in line 15 of the program, which is specified by `l(15)`, you will call `NAME-main.functionOne`, which is defined in line 7 of the program, as specified by `l(7)`, which is a function that requires a single integer parameter, as specified by `FUNC-func(int)`. The function list of parameters, which is specified after `CALLFUNC-list`, includes the `NAME-main.varOne` variable that is defined in line 12 of the program as `l(12)` shows.

Learning more about go build

If you want to learn more about what is happening behind the scenes when you execute a `go build` command, you should add the `-x` flag to it, as in the next example:

```
$ go build -x defer.go
WORK=/var/folders/sk/ltk8cnw50lzdtr2hxcj5sv2m0000gn/T/go-build563076777
mkdir -p $WORK/command-line-arguments/_obj/
mkdir -p $WORK/command-line-arguments/_obj/exe/
cd /Users/mtsouk/Desktop/masterGo/ch/ch2/code
/usr/local/Cellar/go/1.9.2/libexec/pkg/tool/darwin_amd64/compile -o
$WORK/command-line-arguments.a -trimpath $WORK -goversion go1.9.2 -p main -
complete -buildid 594c089b099f8acfb529046cd09a886b401d57c8 -D
_/Users/mtsouk/Desktop/masterGo/ch/ch2/code -I $WORK -pack ./defer.go
cd .
/usr/local/Cellar/go/1.9.2/libexec/pkg/tool/darwin_amd64/link -o
$WORK/command-line-arguments/_obj/exe/a.out -L $WORK -extld=clang -
buildmode=exe -buildid=594c089b099f8acfb529046cd09a886b401d57c8
$WORK/command-line-arguments.a
mv $WORK/command-line-arguments/_obj/exe/a.out defer
```

Once again, there are many things happening in the background, and it is important to be aware of them. However, most of the time, you will not have to deal with the commands of the compilation process.

General Go coding advices

The following list offers practical advices that will help you write better Go code:

- If you have an error in a Go function, either log it or return it, do not do both unless you have a really good reason for doing so!
- Go **interfaces** define behaviors, not data and data structures.
- Use the `io.Reader` and `io.Writer` interfaces because they make your code more extensible.
- Make sure that you pass a pointer to a variable of a function only when needed. The rest of the time, just pass the value of the variable.
- Error variables are not `string` variables; they are `error` variables!
- Do not test your Go code on production machines!
- If you don't really know a Go feature, test it before using it for the first time, especially if you are developing an application or a utility that will be used by a large number of users.
- If you are afraid of making mistakes, you will most likely end up doing nothing really useful! So, experiment as much as you can!

Additional Resources

Have a look at the following resources:

- Learn more about the `unsafe` standard Go package by visiting its documentation page at `https://golang.org/pkg/unsafe/`.
- Visit the website of DTrace at `http://dtrace.org/`.
- You can find out more information about the functions of the `runtime` package by visiting `https://golang.org/pkg/runtime/`.

- Reading research papers might be difficult, but it is very rewarding. Be sure to download the *On-the-fly Garbage Collection: An Exercise in Cooperation* paper and read it. The paper can be found in many places, including `https://dl.acm.org/citation.cfm?id=359655`.
- Visit `https://github.com/gasche/gc-latency-experiment` in order to find benchmarking code for the garbage collector of various programming languages.
- Should you wish to learn more about garbage collection, definitely visit `http://gchandbook.org/`.
- Visit the documentation page of `cgo` at `https://golang.org/cmd/cgo/`.

Exercises

- Write an example where you use your own C code from a Go program.
- Use `strace(1)` on your Linux machine to inspect the operation of some standard Unix utilities such as `cp(1)` and `ls(1)`. What do you see?
- If you are using a macOS machine, use `dtruss(1)` to see how the `sync(8)` utility works.
- Write a Go function, and use it in a C program.

Summary

This chapter covered many interesting Go topics including theoretical and practical information about the Go **garbage collector**; how to call C code from your Go programs; the handy, yet sometimes tricky, `defer` keyword; the `panic()` and `recover()` functions; the `strace(1)`, `dtrace(1)`, and `dtruss(1)` Unix tools; and the use of the `unsafe` standard Go package. The chapter also presented some assembly language generated by Go.

Toward the end of the chapter, it told you how to find out information about your Go environment using the `runtime` package and how to reveal and explain the node tree of a Go program before giving you some handy Go coding advice.

At the end of the day, what you should remember from this chapter is that tools such as the `unsafe` Go package and the ability to call C code from Go are usually used for three occasions: first, when you want the best performance and you are willing to sacrifice some Go safety for it; second, when you want to communicate with another programming language; and third, when you want to implement something that cannot be implemented in Go.

In the next chapter, you will start to learn about the basic data types that come with Go including **arrays**, **slices**, and **maps**. Despite their simplicity, those data types are the building blocks of almost every Go application because they are the basis of more complex data structures that allow you to store your data and move information inside your Go projects. Additionally, you will learn about **pointers**, which are also found in other programming languages, Go **loops**, and the unique way that Go works with dates and times.

3
Working with Basic Go Data Types

In the previous chapter, we covered many fascinating topics including the way the Go garbage collector works, the `panic()` and `recover()` functions, the `unsafe` package, how to call C code from a Go program, and how to call Go code from a C program. We also addressed the node tree created by the Go compiler when compiling a Go program.

The core subject matter of this chapter is the basic data types of Go. This list includes **arrays**, **slices**, and **maps**. Despite their simplicity, these data types can help you store, retrieve, and alter the data in your programs in a very convenient and quick way. Moreover, this chapter will talk about **pointers**, **constants**, **loops**, and working with dates and times, which is a very interesting subject.

In this chapter, you will learn the following topics:

- Go arrays
- Go slices and why slices are much better than arrays
- Go maps
- Pointers in Go
- Looping in Go
- Constants in Go
- Working with times
- Operating with dates

Go loops

Every programming language has a way of looping, and Go is no exception. Go offers the `for` loop, which allows you to iterate over many kinds of data types.

Go does not offer support for the `while` keyword. However, the `for` loops in Go can replace `while` loops!

The for loop

The most common programming loop type is the `for` loop, which allows you to iterate for a predefined number of times or for as long as a condition is valid or according to a value that is calculated at the beginning of the `for` loop. Such values include the size of a slice or an array and the number of the keys on a map. This means that one of the most common ways for accessing all of the elements of an array, a slice, or a map is the `for` loop. The other way is with the use of the `range` keyword.

The following code shows the simplest form of a `for` loop, where a given variable takes a range of predefined values:

```
for i := 0; i < 100; i++ {
}
```

In the preceding loop, the values that `i` will take are from `0` to `99`. As soon as `i` reaches `100`, the execution of the `for` loop will stop. In this case, `i` is a local and temporary variable, which means that after the termination of the `for` loop, `i` will be garbage collected at some point and disappear. However, if `i` was defined outside the `for` loop, it would have kept its value after the termination of the `for` loop. In that case, the value of `i` after the termination of the `for` loop would have been `100`.

You can completely exit a `for` loop using the `break` keyword. The `break` keyword also allows you to create a `for` loop without an exit condition, such as `i < 100` used in the preceding example, because the exit condition can be included in the code block of the `for` loop. You are also allowed to have multiple exit conditions in a `for` loop.

Additionally, you can skip a single iteration of a `for` loop using the `continue` keyword. The `continue` keyword stops running the block, jumps back up to the top of the `for` loop, and carries on for the next item.

The while loop

As discussed earlier, Go does not offer the `while` keyword for writing `while` loops, but it allows you to use a `for` loop instead of a `while` loop. This section will present two examples where a `for` loop does the job of a `while` loop.

The following is the typical case where you want to write something like `while(condition)`:

```
for {
}
```

It is the job of the developer to use the `break` keyword to exit this `for` loop!

However, the `for` loop can also emulate a `do...while` loop that are found in other programming languages. As an example, the following Go code is equivalent to a `do...while(anExpression)` loop:

```
for ok := true; ok; ok = anExpression {
}
```

As soon as the `ok` variable reaches the `false` value, the `for` loop will terminate.

The range keyword

Go also offers the `range` keyword, which is used in `for` loops and allows you to write easy to understand code for iterating over Go data types.

The main advantage of the `range` keyword is that you do not need to know the **cardinality** of a slice or a map in order to process its elements one by one. You will see `range` in action in a while.

Examples of Go for loops

This section will display multiple examples of `for` loops. The name of the file is `loops.go`, and it will be presented in four parts. The first code segment of `loops.go` is as follows:

```
package main

import (
    "fmt"
)
```

```
func main() {

    for i := 0; i < 100; i++ {
        if i%20 == 0 {
            continue
        }

        if i == 95 {
            break
        }

        fmt.Print(i, " ")
    }
```

The preceding code shows a typical `for` loop as well as the use of the `continue` and `break` keywords.

The next code segment is as follows:

```
fmt.Println()
i := 10
for {
    if i < 0 {
        break
    }
    fmt.Print(i, " ")
    i--
}
fmt.Println()
```

The preceding code emulates a typical `while` loop. Note the use of the `break` keyword to exit the `for` loop.

The third part of `loops.go` follows next:

```
i = 0
anExpression := true
for ok := true; ok; ok = anExpression {
    if i > 10 {
        anExpression = false
    }

    fmt.Print(i, " ")
    i++
}
fmt.Println()
```

In this part, you see the use of a `for` loop that does the job of a `do...while` loop as discussed earlier in this chapter.

The last part of `loops.go` is shown in the following Go code:

```
        anArray := [5]int{0, 1, -1, 2, -2}
        for i, value := range anArray {
            fmt.Println("index:", i, "value: ", value)
        }
    }
```

Applying the `range` keyword to an array variable returns two values: an array index and the value of the element at that index, respectively. You can use both of them, one of them, or none of them; that is, in case you just want to count the elements of the array.

Executing `loops.go` will produce the following output:

```
$ go run loops.go
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55
56 57 58 59 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 81 82
83 84 85 86 87 88 89 90 91 92 93 94
10 9 8 7 6 5 4 3 2 1 0
0 1 2 3 4 5 6 7 8 9 10 11
index:  0 value:   0
index:  1 value:   1
index:  2 value:   -1
index:  3 value:   2
index:  4 value:   -2
```

Go arrays

Arrays are one of the most popular data structures for two reasons: arrays are simple and easy to understand and they are very versatile and can store many different kinds of data.

You can declare an array that stores four integers as follows:

```
anArray := [4]int{1, 2, 4, -4}
```

The size of the array is stated before its type, which is defined before its elements. You can find the length of an array with the help of the `len()` function: `len(anArray)`.

The index of the first element of any dimension of an array is 0, the index of the second element of any array dimension is 1, and so on. This means that for an array with one dimension named `a`, the valid indexes are from `0` to `len(a)-1`.

Although you might be familiar with accessing the elements of an array in other programming languages using a `for` loop and a numeric variable, there are cooler ways to visit all of the elements of an array in Go that involve the use of the `range` keyword and that allow you to bypass the use of the `len()` function in the `for` loop. Look to the Go code of `loops.go` for such an example.

Multi-dimensional arrays

Arrays can have more than one dimension. However, using more than three dimensions without a serious reason can make your program difficult to read and might create bugs.

Arrays can store all the types of elements. We are just using integers here because they are easier to understand and type.

The following Go code shows how you can create an array with two dimensions (`twoD`) and another one with three dimensions (`threeD`):

```
twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12},
{13, 14, 15, 16}}
threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Accessing, assigning, or printing a single element from one of the previous two arrays can be done easily. As an example, the first element of the `twoD` array is `twoD[0][0]` and its value is `1`.

Therefore, accessing all of the elements of the `twoD` array with the help of multiple `for` loops can be done as follows:

```
for i := 0; i < len(threeD); i++ {
    for j := 0; j < len(v); j++ {
        for k := 0; k < len(m); k++ {
        }
    }
}
```

As you can see, you need as many `for` loops as the dimensions of the array in order to access all of its elements. The same rules apply to slices, which will be presented in the next section.

The code of `usingArrays.go`, which will be presented in three parts, presents a complete example of how to deal with arrays in Go.

The first part of the code is as follows:

```
package main

import (
    "fmt"
)

func main() {
    anArray := [4]int{1, 2, 4, -4}
    twoD := [4][4]int{{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12},
 {13, 14, 15, 16}}
    threeD := [2][2][2]int{{{1, 0}, {-2, 4}}, {{5, -1}, {7, 0}}}
```

Here, you define three array variables, named `anArray`, `twoD`, and `threeD`, respectively.

The second part of `usingArrays.go` is shown here:

```
    fmt.Println("The length of", anArray, "is", len(anArray))
    fmt.Println("The first element of", twoD, "is", twoD[0][0])
    fmt.Println("The length of", threeD, "is", len(threeD))

    for i := 0; i < len(threeD); i++ {
        v := threeD[i]
        for j := 0; j < len(v); j++ {
            m := v[j]
            for k := 0; k < len(m); k++ {
                fmt.Print(m[k], " ")
            }
        }
        fmt.Println()
    }
```

What you get from the first `for` loop is a two-dimensional array (`threeD[i]`), whereas what you get from the second `for` loop is an array with one dimension (`v[j]`). The last `for` loop iterates over the elements of the array with one dimension.

The last code part is shown in the following Go code:

```
        for _, v := range threeD {
            for _, m := range v {
                for _, s := range m {
                    fmt.Print(s, " ")
                }
            }
            fmt.Println()
        }
    }
```

The `range` keyword does exactly the same job as the iteration variables used in the `for` loops of the preceding code segment, but it does so in a more elegant and clearer way.

The `range` keyword also works with Go maps, which makes it pretty handy and my preferred way of iteration.

Executing `usingArrays.go` generates the following output:

```
$ go run usingArrays.go
The length of [1 2 4 -4] is 4
The first element of [[1 2 3 4] [5 6 7 8] [9 10 11 12] [13 14 15 16]] is 1
The length of [[[1 0] [-2 4]] [[5 -1] [7 0]]] is 2
1 0 -2 4
5 -1 7 0
1 0 -2 4
5 -1 7 0
```

One of the biggest problems with arrays is out-of-bounds errors, which means trying to access an element that does not exist. It's like trying to access the sixth element of an array with only five elements. The Go compiler considers issues that can be detected as compiler errors because this helps the development workflow. Therefore, the Go compiler can detect out-of-bounds array access errors as follows:

```
./a.go:10: invalid array index -1 (index must be non-negative)
./a.go:10: invalid array index 20 (out of bounds for 2-element array)
```

The shortcomings of Go arrays

Go arrays have many disadvantages that will make you reconsider using them in your Go projects. First of all, once you define an array, you cannot change its size, which means that Go arrays are not dynamic. Putting it simply, if you need to add an element to an existing array that has no space left, you will need to create a bigger array and copy all of the elements of the old array to the new one. Second, when you pass an array to a function as a parameter, you actually pass a copy of the array, which means that any changes you make to an array inside a function will be lost after the function exits. Last, passing a large array to a function can be pretty slow, mostly because Go has to create a copy of the array. The solution to all of these problems is to use Go slices, which will be presented in the next section.

WARNING: Because of their disadvantages, arrays are rarely used in Go!

Go slices

Go **slices** are very powerful, and it would not be an exaggeration to say that slices could totally replace the use of arrays in Go. There are only a few occasions where you will need to use an array instead of a slice. The most obvious one is when you are absolutely sure that you will need to store a fixed number of elements.

Slices are implemented using arrays internally, which means that Go uses an underlying array for each slice.

As slices are **passed by reference** to functions, which means that what is actually passed is the memory address of the slice variable, any modifications that you make to a slice inside a function will not get lost after the function exits. Additionally, passing a big slice to a function is significantly faster than passing an array with the same number of elements because Go will not have to make a copy of the slice—it will just pass the memory address of the slice variable.

Performing basic operations on slices

You can create a new slice literal as follows:

```
aSliceLiteral := []int{1, 2, 3, 4, 5}
```

This means that **slice literals** are defined just like arrays but without the element count. If you put an element count in a definition, you will get an array instead!

However, there is also the `make()` function that allows you to create empty slices with the desired **length** and **capacity** as the parameters passed to `make()`. The capacity parameter can be omitted. In that case, the capacity of the slice will be the same as its length.

You can define a new empty slice with 20 places that can be automatically expanded when needed as follows:

```
integer := make([]int, 20)
```

Please note that Go automatically initializes the elements of an empty slice to the zero value of its type, which means that the value of the initialization depends on the type of the object stored in the slice. Although it is good to know that Go initializes the elements of every slice created with `make`, you should not make any assumptions based on this fact because that behavior might change in the future.

After that, you can access all of the elements of a slice in the following way:

```
for i := 0; i < len(integer); i++ {
    fmt.Println(integer[i])
}
```

If you want to empty an existing slice, the zero value for a slice variable is `nil`.

```
aSliceLiteral = nil
```

You can add an element to the slice, which will automatically increase its size, using the `append()` function:

```
integer = append(integer, -5000)
```

You can access the first element of the `integer` slice as `integer[0]`, whereas you can access the last element of the `integer` slice as `integer[len(integer)-1]`.

Also, you can access multiple continuous slice elements using the `[:]` notation. The next statement selects the second and the third elements of a slice:

```
integer[1:3]
```

Additionally, you can use `[:]` notation for creating a new slice from an existing slice or array:

```
s2 := integer[1:3]
```

Please note that this process is called **re-slicing**, and it can cause problems in some cases. Examine the following program:

```
package main

import "fmt"

func main() {

    s1 := make([]int, 5)
    reSlice := s1[1:3]
    fmt.Println(s1)
    fmt.Println(reSlice)

    reSlice[0] = -100
    reSlice[1] = 123456
    fmt.Println(s1)
    fmt.Println(reSlice)

}
```

First, note that in order to select the second and third elements of a slice using the `[:]` notation, you should use `[1:3]`, which means starting with index number 1 and going up to index number 3, without including index number 3.

Given an array `a1`, you can create a slice `s1` that references that array by executing `s1 := a1[:]`.

Executing the preceding code, which is saved as `reslice.go`, will create the next output:

```
$ go run reslice.go
[0 0 0 0 0]
[0 0]
[0 -100 123456 0 0]
[-100 123456]
```

At the end of the program, the contents of the `s1` slice will be `[0 -100 123456 0 0]`, even though we did not change them directly. This means that altering the elements of a re-slice modifies the element of the original slice because both slices point to the same memory address! Put simply, the re-slice process does not make a copy of the original slice.

The second problem of re-slicing is that, even if you re-slice a slice in order to use a small part of the original slice, the underlying array from the original slice will be kept in memory for as long as the smaller re-slice exists because the original slice is being referenced by the smaller re-slice. Although this is not truly important for small slices, it can cause problems when you are reading big files into slices and you only want to use a small part of them.

Slices are being expanded automatically

Slices have two main properties: **capacity** and **length**. The tricky part is that usually these two properties have different values. The length of a slice is the same as the length of an array with the same number of elements and can be found using the `len()` function. The capacity of a slice is the current room that has been allocated for this particular slice, and it can be found with the `cap()` function. As slices are dynamic in size, if a slice runs out of room, Go automatically doubles its current length to make room for more elements.

Put simply, if the length and the capacity of a slice have the same values and you try to add another element to the slice, the capacity of the slice will be doubled, whereas its length will be increased by one. Although this might work well for small slices, adding a single element to a really huge slice might take more memory than expected.

The code of `lenCap.go` illustrates the concepts of capacity and length in more detail, and it will be presented in three parts. The first part of the program follows:

```
package main

import (
    "fmt"
)

func printSlice(x []int) {
```

```
        for _, number := range x {
            fmt.Print(number, " ")
        }
        fmt.Println()
}
```

The `printSlice()` function helps you print a one-dimensional slice without having to repeat the same Go code all of the time.

The second part of `lenCap.go` contains the next piece of Go code:

```
func main() {
    aSlice := []int{-1, 0, 4}
    fmt.Printf("aSlice: ")
    printSlice(aSlice)

    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
    aSlice = append(aSlice, -100)
    fmt.Printf("aSlice: ")
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
```

In this part, as well as the next one, you will add some elements to the `aSlice` slice to alter its length and its capacity.

The last portion of Go code is as follows:

```
    aSlice = append(aSlice, -2)
    aSlice = append(aSlice, -3)
    aSlice = append(aSlice, -4)
    printSlice(aSlice)
    fmt.Printf("Cap: %d, Length: %d\n", cap(aSlice), len(aSlice))
}
```

The execution of `lenCap.go` will create the following output:

```
$ go run lenCap.go
aSlice: -1 0 4
Cap: 3, Length: 3
aSlice: -1 0 4 -100
Cap: 6, Length: 4
-1 0 4 -100 -2 -3 -4
Cap: 12, Length: 7
```

As you can see, the initial size of the slice was 3. As a result, the initial value of its capacity was also 3. After adding one element to the slice, its size became 4, whereas its capacity became 6. After adding three more elements to the slice, its size became 7, whereas its capacity was doubled one more time and became 12.

Byte slices

A **byte slice** is a slice where its type is `byte`. You can create a new byte slice named `s` as follows:

```
s := make([]byte, 5)
```

There is nothing special in the way that you can access a byte slice compared to the other types of slices. It is just that byte slices are used in file input and output operations. You will see byte slices in action in `Chapter 8`, *Telling a Unix System What to Do*.

The copy() function

You can create a slice from the elements of an existing array, and you can copy an existing slice to another one using the `copy()` function. However, as the use of `copy()` can be very tricky, this subsection will try to clarify its usage with the help of the Go code of `copySlice.go`, which will be presented in four parts.

You should be very careful when using the `copy()` function on slices because the built-in `copy(dst, src)` copies the minimum of the `len(dst)` and `len(src)` elements.

The first part of the program is shown in the following Go code:

```
package main

import (
    "fmt"
)

func main() {

    a6 := []int{-10, 1, 2, 3, 4, 5}
    a4 := []int{-1, -2, -3, -4}
    fmt.Println("a6:", a6)
    fmt.Println("a4:", a4)
```

```
copy(a6, a4)
fmt.Println("a6:", a6)
fmt.Println("a4:", a4)
fmt.Println()
```

In the preceding code, we define two slices named `a6` and `a4`, print them, and then try to copy `a4` to `a6`. As `a6` has more elements than `a4`, all of the elements of `a4` will be copied to `a6`. However, as `a4` has only four elements and `a6` has six elements, the last two elements of `a6` will remain the same.

The second part of `copySlice.go` is as follows:

```
b6 := []int{-10, 1, 2, 3, 4, 5}
b4 := []int{-1, -2, -3, -4}
fmt.Println("b6:", b6)
fmt.Println("b4:", b4)
copy(b4, b6)
fmt.Println("b6:", b6)
fmt.Println("b4:", b4)
```

In this case, only the first four elements of `b6` will be copied to `b4` because `b4` has only four elements.

The third code segment of `copySlice.go` is shown in the following Go code:

```
fmt.Println()
array4 := [4]int{4, -4, 4, -4}
s6 := []int{1, 1, -1, -1, 5, -5}
copy(s6, array4[0:])
fmt.Println("array4:", array4[0:])
fmt.Println("s6:", s6)
fmt.Println()
```

Here you try to copy an array with four elements to a slice with six elements. Note that the array is converted to a slice with the help of the `[:]` notation (`array4[0:]`).

The last code portion of `copySlice.go` is as follows:

```
        array5 := [5]int{5, -5, 5, -5, 5}
        s7 := []int{7, 7, -7, -7, 7, -7, 7}
        copy(array5[0:], s7)
        fmt.Println("array5:", array5)
        fmt.Println("s7:", s7)
}
```

Here you can see how to copy a slice to an array that has places for five elements. As `copy()` only accepts slice arguments, you should also use the `[:]` notation to convert the array into a slice.

If you try to copy an array into a slice or vice versa without using the `[:]` notation, the program will fail to compile and will display one of the following error messages:

```
# command-line-arguments
./a.go:42:6: first argument to copy should be slice; have [5]int
./a.go:43:6: second argument to copy should be slice or string; have
[5]int
./a.go:44:6: arguments to copy must be slices; have [5]int, [5]int
```

Executing `copySlice.go` will create the following output:

```
$ go run copySlice.go
a6: [-10 1 2 3 4 5]
a4: [-1 -2 -3 -4]
a6: [-1 -2 -3 -4 4 5]
a4: [-1 -2 -3 -4]

b6: [-10 1 2 3 4 5]
b4: [-1 -2 -3 -4]
b6: [-10 1 2 3 4 5]
b4: [-10 1 2 3]
array4: [4 -4 4 -4]
s6: [4 -4 4 -4 5 -5]
array5: [7 7 -7 -7 7]
s7: [7 7 -7 -7 7 -7 7]
```

Multidimensional slices

Slices can have many dimensions as is also the case with arrays. The next statement creates a slice with two dimensions:

```
s1 := make([][]int, 4)
```

If you find yourselves using slices with many dimensions all of the time, you might need to reconsider your approach and choose a simpler design that does not require multidimensional slices.

You will find a code example with a **multidimensional slice** in the next section.

Another example of slices

The Go code of the `slices.go` program will hopefully clarify many things about slices for you, and it will be presented in five parts.

The first part of the program contains the expected preamble as well as the definition of two slices:

```
package main

import (
    "fmt"
)

func main() {
    aSlice := []int{1, 2, 3, 4, 5}
    fmt.Println(aSlice)
    integer := make([]int, 2)
    fmt.Println(integer)
    integer = nil
    fmt.Println(integer)
```

The second part of the program shows how to use the `[:]` notation to create a new slice that references an existing array. Remember that you are not creating a copy of the array, just a reference to it, which will be verified in the output of the program:

```
anArray := [5]int{-1, -2, -3, -4, -5}
refAnArray := anArray[:]

fmt.Println(anArray)
fmt.Println(refAnArray)
anArray[4] = -100
fmt.Println(refAnArray)
```

The third code segment defines a slice with one dimension and another one with two dimensions using the `make()` function:

```
s := make([]byte, 5)
fmt.Println(s)
twoD := make([][]int, 3)
fmt.Println(twoD)
fmt.Println()
```

As slices are automatically initialized by Go, all of the elements of the two preceding slices will have the zero value of the slice type, which for integers is `0` and for slices is `nil`. Keep in mind that the elements of a multidimensional slice are slices!

In the fourth part of `slices.go` that is shown in the next piece of Go code, you will learn how to initialize all the elements of a slice with two dimensions manually:

```
for i := 0; i < len(twoD); i++ {
    for j := 0; j < 2; j++ {
        twoD[i] = append(twoD[i], i*j)
    }
}
```

The preceding Go code shows that in order to expand an existing slice and make it grow, you will need to use the `append()` function and not reference an index that does not exist! The latter would create a `panic: runtime error: index out of range` error message! Note that the values of the slice elements have been chosen arbitrarily.

The last part of the program shows you how to use the `range` keyword to visit and print all of the elements of a slice with two dimensions:

```
        for _, x := range twoD {
            for i, y := range x {
                fmt.Println("i:", i, "value:", y)
            }
            fmt.Println()
        }
    }
```

If you execute `slices.go`, you will get the following output:

```
$ go run slices.go
[1 2 3 4 5]
[0 0]
[]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -5]
[-1 -2 -3 -4 -100]
[0 0 0 0 0]
[[] [] []]
i: 0 value: 0
i: 1 value: 0
i: 0 value: 0
i: 1 value: 1
i: 0 value: 0
i: 1 value: 2
```

It should not come as a surprise to you that the objects of the slice with the two dimensions are initialized to `nil` and therefore printed as empty. This happens because the zero value for the slice type is `nil`.

Sorting slices using sort.slice()

This subsection will illustrate the use of the `sort.Slice()` function that was first introduced in Go version 1.8. This means that the code presented, which is saved in `sortSlice.go`, will not run on older Go versions. The program will be presented in three parts. The first part of the program is as follows:

```
package main

import (
    "fmt"
    "sort"
)

type aStructure struct {
    person string
    height int
    weight int
}
```

Apart from the expected preamble, you can also see the definition of a **Go structure** for the first time in this book. `Chapter 4`, *The Uses of Composite Types*, will totally explore Go structures. For now, however, keep in mind that structures are data types with multiple variables of various types.

The second part of `sortSlice.go` is shown in the following Go code:

```
func main() {

    mySlice := make([]aStructure, 0)
    mySlice = append(mySlice, aStructure{"Mihalis", 180, 90})
    mySlice = append(mySlice, aStructure{"Bill", 134, 45})
    mySlice = append(mySlice, aStructure{"Marietta", 155, 45})
    mySlice = append(mySlice, aStructure{"Epifanios", 144, 50})
    mySlice = append(mySlice, aStructure{"Athina", 134, 40})

    fmt.Println("0:", mySlice)
```

Here you create a new slice named `mySlice` with elements from the `aStructure` structure created earlier.

The final part of the program is as follows:

```
    sort.Slice(mySlice, func(i, j int) bool {
        return mySlice[i].height < mySlice[j].height
    })
```