Go IN ACTION

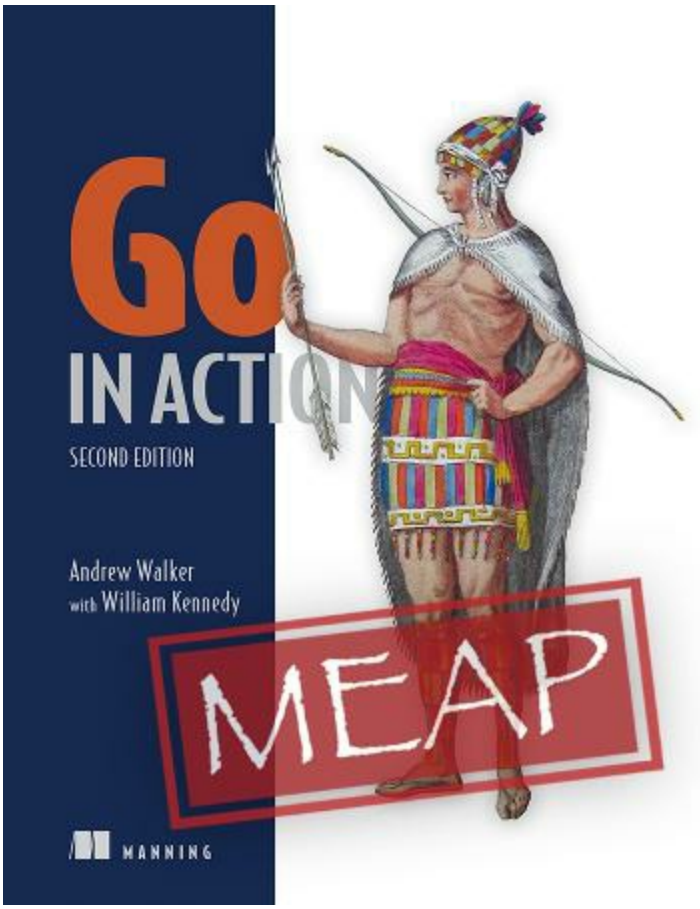
SECOND EDITION

Andrew Walker
with William Kennedy

MEAP

MANNING

Go in Action, Second Edition MEAP V03



MEAP VERSION 3

MANNING PUBLICATIONS

Welcome

Thanks for purchasing the MEAP for *Go In Action, Second Edition*! Go has come a long way since the first edition was published back in 2015, and it has never been a better time to be a Go programmer. Since Go In Action was published, Go has seen widespread adoption in the cloud, establishing itself as one of the most popular languages for backend software. Meanwhile, the ecosystem has continued to improve, with the addition of thousands of new libraries, tens of thousands of new developers and new features such as context, built-in fuzzing and, of course, the much-awaited generic programming provided by type parameters.

This new edition is more than just a couple new chapters to catch up the material. With the benefit of nearly 8 years of working with Go, we thought we would take the opportunity to not only introduce new language elements, but also to revisit the old material with the goal of laying better foundations for the new Go programmer, including answering some of the most important questions new users have, such as how to structure your Go projects and how to get the most out of Go's error handling paradigm. We have tried to keep the best practices that have evolved in the language since the last edition in mind, and have tried to make sure that potentially tricky concepts are explained thoroughly and intuitively, so that you can get the most out of starting your Go journey with us!

You don't need to know Go to read this book, however a background in another programming language will help you get the most out of it. This will also help you get a better idea of what makes Go special, and how it does things differently, and why.

We believe learning is done best by doing, and so we have tried to keep the dry, reference-style material to a minimum, with a greater focus on code examples, so that you can see how the concepts map to real Go, and almost every single line of code you see will be available in the book repository, ready for you to download, run and modify as you like.

We've tried to make this the best possible starting point for the new Go programmer, but we need your help to make sure we're on track for that, so please comment in the liveBook discussion forum. and let us know what you think. If something is confusing, or you think we should cover things in a different way, let us know. No feedback is bad feedback!

Thanks again for joining us in this process. We can't wait to hear from you!

— Andrew Walker & William Kennedy

In this book

1 Introducing Go

This chapter covers

- Solving modern computing challenges with Go
- Using the Go tools

Computers have evolved, and programming languages have needed to evolve with them. While processor speed continues to increase, the demands of modern computing are such that this alone has not been enough to keep up with the pace of technology, and many of the recent advances in CPU manufacturing have focused on ways of doing work in parallel. The mobile devices in our pockets might have more CPU cores than the high-end laptops we used just a few years ago, and high-powered servers now have 64, 128, or even more cores, yet programming languages have struggled to enable effective use of these advances.

In part, this is because programming languages had little cause to embrace parallelism during much of their evolution in the face of ever-increasing processing speed. The problem space of concurrent programming is much larger, with exponentially more interactions that can occur between parts of otherwise simple code, and so it was easier to defer the problem. Now that the needs are greater, modern programming languages need built in abstractions around concurrency, freeing the programmer to focus less on minutiae of coordinating concurrency, and more on the important work of solving problems.

The business of writing software itself has changed as well. Most programs are no longer written by a single developer, but by teams of people, often spread across several time zones and work schedules. Large projects are often broken up into units worked on in parallel by one or more developers who integrate their work into a larger whole, or provide reusable libraries that can be deployed across an entire suite of applications.

Recent decades have also seen the rise in popularity of the open source

software movement, allowing developers across the spectrum of technology to share in innovation, quality and collective oversight. Today's programmers and companies believe more than ever in the power of open source software, and the desire to participate in this movement only continues to grow.

Go has sought to meet all of these needs. As a language, it provides developers with refined abstractions around concurrency and a powerful runtime to make efficient use of it with the resources at hand. As a programming environment, it provides robust tooling to make inspection, observation, dependency resolution and code sharing sane and easy. As an enterprise language, it has adopted a simple, opinionated syntax and coding standards as first principles to enable efficient collaboraction across large teams. Finally, the project has embraced the open source ethos, with all work in the open for all to see, while also integrating with public open source repositories and revision control systems such as Github.

In this book, you will explore these features and gain a sense of the many decisions that have shaped the language, so that you can undersdand not only how the language works, but a bit more about *why* it works the way it does. Go was not developed in isolation, but as a response to the perceived deficits in programming environments that existed at the time of its creation. Its design is practical, and informed by many collective decades of programming experience, in a variety of environments and languages.

Much of what makes Go special is how it is deliberately different to that which came before it, and so, while the reader is not expected to have "decades" of programming experience, an intermediate-level developer with experience in at least one or two other languages will get the most from this text, and is our target audience.

The decision to pick up a new programming language is an inportant one, so the goal of this book is to help developers and other technical roles get a feel for what working in Go is like, and what its strenghts are, so you can make a more informed decision about whether or not to adopt Go into your tech stack. To this end, we won't dwell too much on introductory concepts, though we may revisit them from time to time where it makes sense to show how Go might do things differently to what you're used to.

What we *will* cover are as many of the practical aspects of using this language as a daily driver as we can, including syntax, concurrency primitives, and testing. For those of you with some prior experience in Go, we will try to hit the most important changes to the language since the first edition, such as modules, fuzzing, and the most recent major language addition: generic programming with type parameters.

We hope that by the time you've finished reading, you'll be just as hooked as we are, and on your way to writing better, faster, more reliable software!

The source code for the examples in the book is available at **https://github.com/flowchartsman/go-in-action**

1.1 Solving modern programming challenges with Go

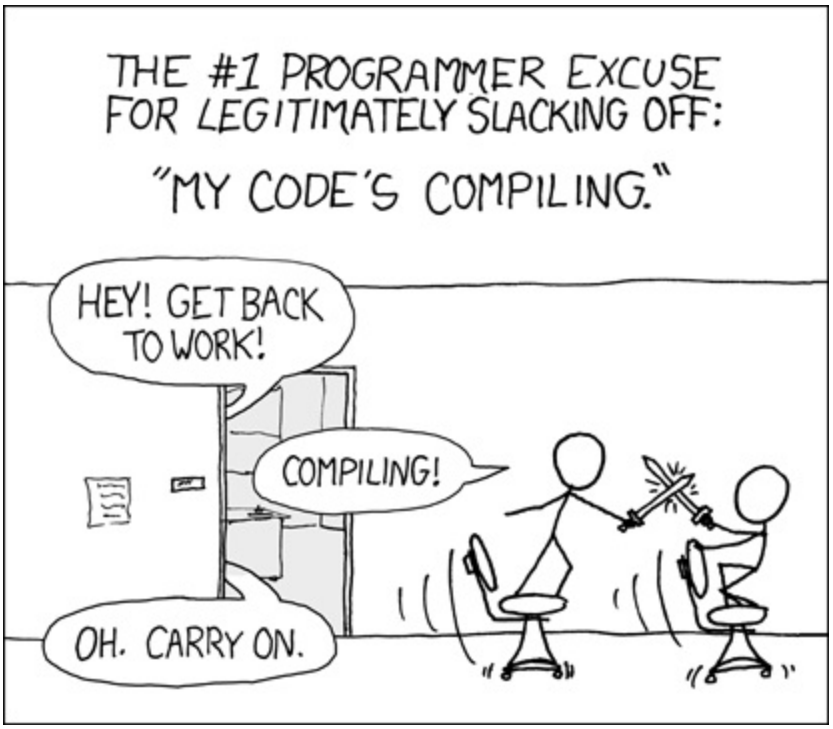
Developers today often have to make an uncomfortable choice between rapid development and performance when choosing a language for their projects. Languages like C and C++ offer fast execution, whereas languages like Ruby and Python offer rapid development. The Go designers went to great lengths to create a language that bridges these competing worlds and offers a high-performance language with features that make development fast, without unnecessary complexity.

As you explore Go, you'll find well-planned features and a concise syntax. As a language, Go is defined not only by what it includes, but by what it doesn't include. Go has few keywords to memorize and a type system which provides expressiveness, composition and code reuse without all of the cognitive overhead of a rigidly object-oriented paradigm. If you've been at the mercy of long compilation times, you will be refreshed to use a compiler that's so fast, sometimes you'll forget it's running. Because of Go's built-in concurrency features, your software will scale to use the resources available without forcing you to use special threading libraries. The Go runtime also has a highly-optimized, concurrent garbage collector, freeing you from the tedious and error-prone process of manual memory management. Let's look quickly at these key features.

1.1.1 Development speed

Compiling a large application in C or C++ takes more time than getting a cup of coffee. Figure 1.1 shows an XKCD classic excuse for messing around in the office.

Figure 1.1. Working hard? (via XKCD)



As a Go developer, you will need to find another excuse if you want to slack off. Go offers lightning-quick compiles by using a smart compiler and simplified dependency resolution algorithms, made reproducible and explicit by the recent addition of the Go Modules ecosystem. When you build a Go program, the compiler only needs to look at the libraries that you directly need, rather than traversing all the libraries that are included in the entire dependency chain like Java, C, and C++. Consequently, many Go applications compile in under a second. The entire Go source tree compiles in under 20 seconds on modern hardware.

This focus on quick compilation helps close the productivity gap with dynamic languages, which have traditionally enjoyed an advantage in productivity by eliminating the expensive compilation and linking steps. By

allowing you to minimize the time between writing your code and running it, Go provides much of the speed of writing dynamic languages.

1.1.2 Type Safety

If you've ever worked in a dynamic language before, you have likely seen an error message that looks something like this:

Figure 1.2. A Common Dynamic Typing Exception

If you're lucky, you ran into it during testing, however many of us have also been surprised and confused by a message very much like this one in our logs after something goes wrong.

Type errors like this occur when a weakly-typed variable is assigned a value that is incompabible with its usage elsewhere in the program. They can be notoriously difficult to debug, since they often sneak in as edge cases that crop up rarely or in unexpected circumstances, making it unclear where a problematic value came from. The addition of user input to the mix only serves to complicate this further, since you have less control over the values that are flowing through your program.

With a strongly-typed language like Go, every value has a type assigned to it at its creation that is tracked by the compiler, making this sort of bug impossible. This is appealing to developers, and for this reason we have seen a push for at least some level of type safety in popular dynamic languages. Typscript, for example, brings type safety to Javascript by converting (or "transpiling") strongly-typed Typescript code into the weakly-typed Javascript code that actually runs in the browser, while Python's type hinting

system has been growing in popularity. Both approaches have their tradeoffs, of course; transpiled languages can provide strong guarantees, but the transpilation steps mean that the code you write isn't the code that actually runs, and can come with some performance hit, while type hints serve primarily to inform, and are not enforced natively without special extensions.

Go, on the other hand, is strongly-typed by design, and simply won't let you use a value in an inconsistent way:

Listing 1.1. Attempting To Assign Values Of Different Types

```
var number int
var str string

number = 1
str = "one"
number = str
// Error: cannot use str (variable of type string) as type int in
```

This code will not cause an error when it runs, becuse it cannot be run in the first place—it simply won't build at all.

1.1.3 Concurrency

As we approach the limits of how small we can make transistors, chip manufacturers have been turning to multi-core processors and parallel computing techniques to speed up execution time. New techniques and novel architectures are constantly being developed to make the most of multi-threaded computing, however developments in programming languages have lagged behind somewhat.

In part, this is due to the inherent difficulty in programming for a multi-threaded environment using languages that were not designed with this in mind. Special care needs to be taken to synchronize access to shared memory to prevent simultaneous writes, which which can corrupt data. Doing this correctly can be difficult, and doing it efficiently is even more so.

Another difficulty is that it's not enough to just run multiple things at once, you also need to pay attention to when tasks are *blocked*, or waiting, on

something else, such as data from a file or a network connection. I/O can be extraordinarily slow compared to code, since it depends on things like storage hardware or packets coming over the network, each of which can be orders of magnitude slower than the code running on the processor. If every task stopped to wait for every I/O operation, your profgram would spend most of its time waiting or even grind to a halt when it could be doing other things! The business of tracking all of these moving pieces is called "scheduling", and it’s also tricky to get right.

Go addresses these problems by integrating concurrency directly into the language runtime itself, making all Go programs concurrent by default.[1] This allows you to concentrate on writing code that is easy to read and reason about without managing all of these complexities yourself. In fact, thanks to the concurrency features built into the standard library, a lot of the time you won’t even have to write a single line of concurrent code yourself! A great example of this is the `net/http` package, which allows you to write things like web APIs that can handle thousands of simultaneous requests with ease.

Goroutines

For those times when you need to write your own concurrent code, Go has a couple of killer features that help cut down on a lot of the headache of managing concurrency yourself: *goroutines* and *channels*.

Goroutines are the name given to functions run with the `go` keyword, which causes them to run concurrently with the rest of the code.

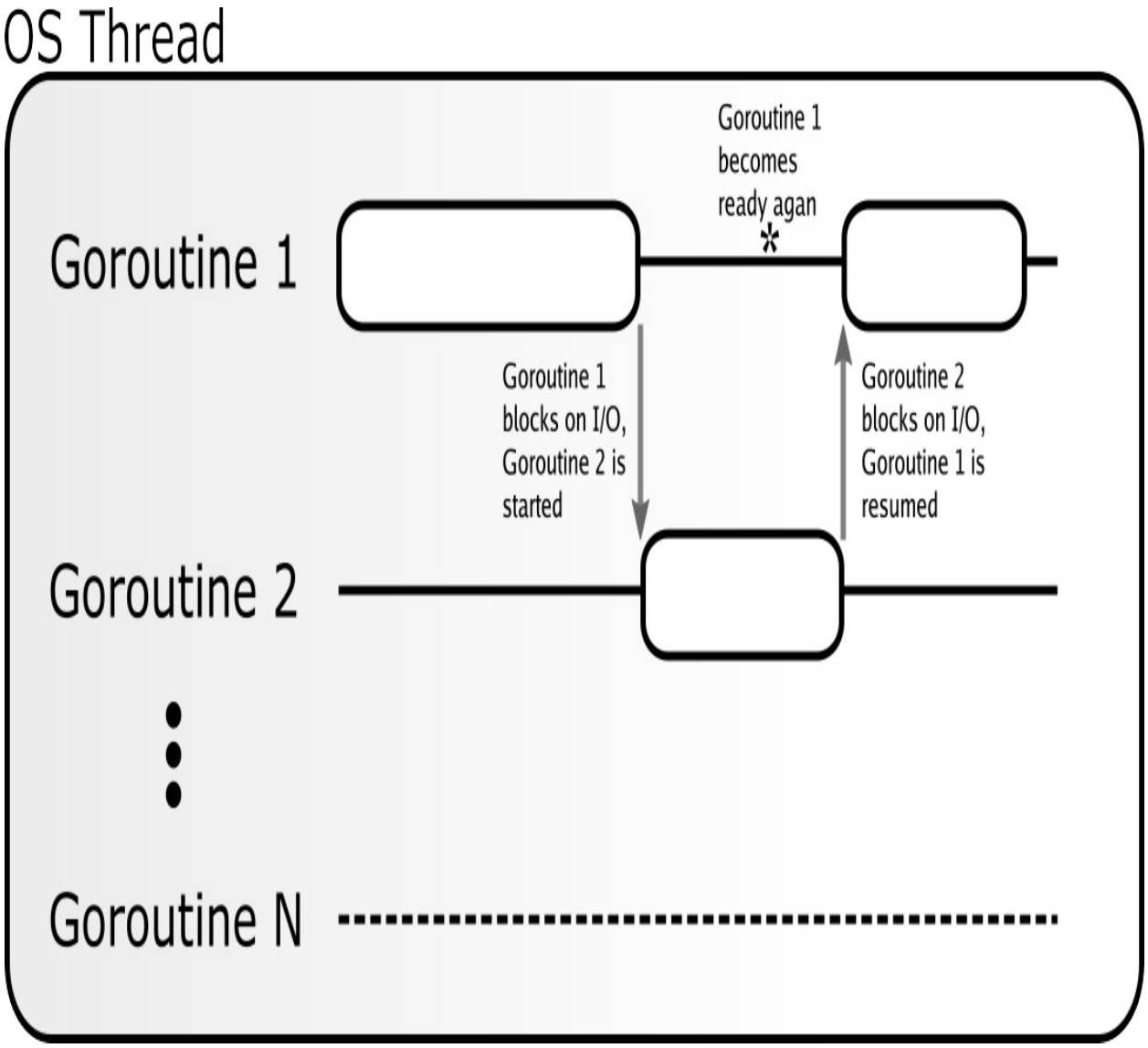
Listing 1.2. Running A Function As A Goroutine

```
func makeMeConcurrent() {
        // Do some concurrent work...
}

func main() {
        go makeMeConcurrent()
        // Continue with the main goroutine...
}
```

Goroutines are managed by the Go runtime, and you can think of them as a

bit like threads, only they are much more lightweight, and less expensive to create. In fact, many Goroutines can share a single OS thread, and can be started and stopped as needed to allow other goroutines to run.

Figure 1.3. Many Goroutines Execute On a Single OS Thread



Because they are so cheap, you don’t need to worry about using them whenever you need them, and it is not uncommon for a single Go program to have many hundreds or even thousands of them active at one time.

If you want to execute some code that might be slow while you move on to accomplish other things, a goroutine is the perfect tool for the job. Creating

one is as easy as calling a function with the go keyword. Here's a quick example:

Listing 1.3. Doing Something Slow

```
func DoSomethingSlow() {
        fmt.Println("SLOW: maybe I'm doing network stuff?")
        time.Sleep(1 * time.Second)
        fmt.Println("SLOW: Okay, finally finished!")
}

func main() {
        fmt.Println("Starting the main task...")

        // Create a goroutine to do something slow.
        go DoSomethingSlow()

        fmt.Println("Resuming the main task...")
        time.Sleep(500 * time.Millisecond)
        fmt.Println("Finished the main task!")
        time.Sleep(1 * time.Second)
}
```

Listing 1.4. Running A Slow Function Concurrently

```
Doing the main task...
Resuming the main task...
SLOW: maybe I'm doing network stuff?
Finished the main task!
SLOW: Okay, finally finished!
```

Calling `DoSomethingSlow` function with the `go` keyword is all you need to run the slow code while the rest of the `main` function does other work. This often results in greater thoroughput and thus better performance for your users. You will learn more about this in **Chapter 8**

Channels

For synchronization and shared data, Go provides _channel_s, which are typed, synchronizd buffers and combine access and locking with a single operation.

You can think of channels as a synchronized message box, where one or more "senders" place data that’s ready to be shared, and one or more "receivers" wait to take custody of it. Channels can be "buffered", meaning they can hold more than one message in the order they were sent, or "unbuffered", meaning they can hold only one message at a time, but the underlying mechanisms are the same.

In the case of an unbuffered channel (or a buffered channel which is full) the sending goroutine will wait until the receiving goroutine has received the message or the receiving goroutine will wait until the message is sent, depending on who accesses the channel first. You can see an example of this flow in figure 1.3, where a channel is used to send data between two running goroutines. Imagine an application where many different processes need to know about or modify data sequentially. Using goroutines and channels, you can model this process safely.

Figure 1.4. Channels Pass Data Between Goroutines

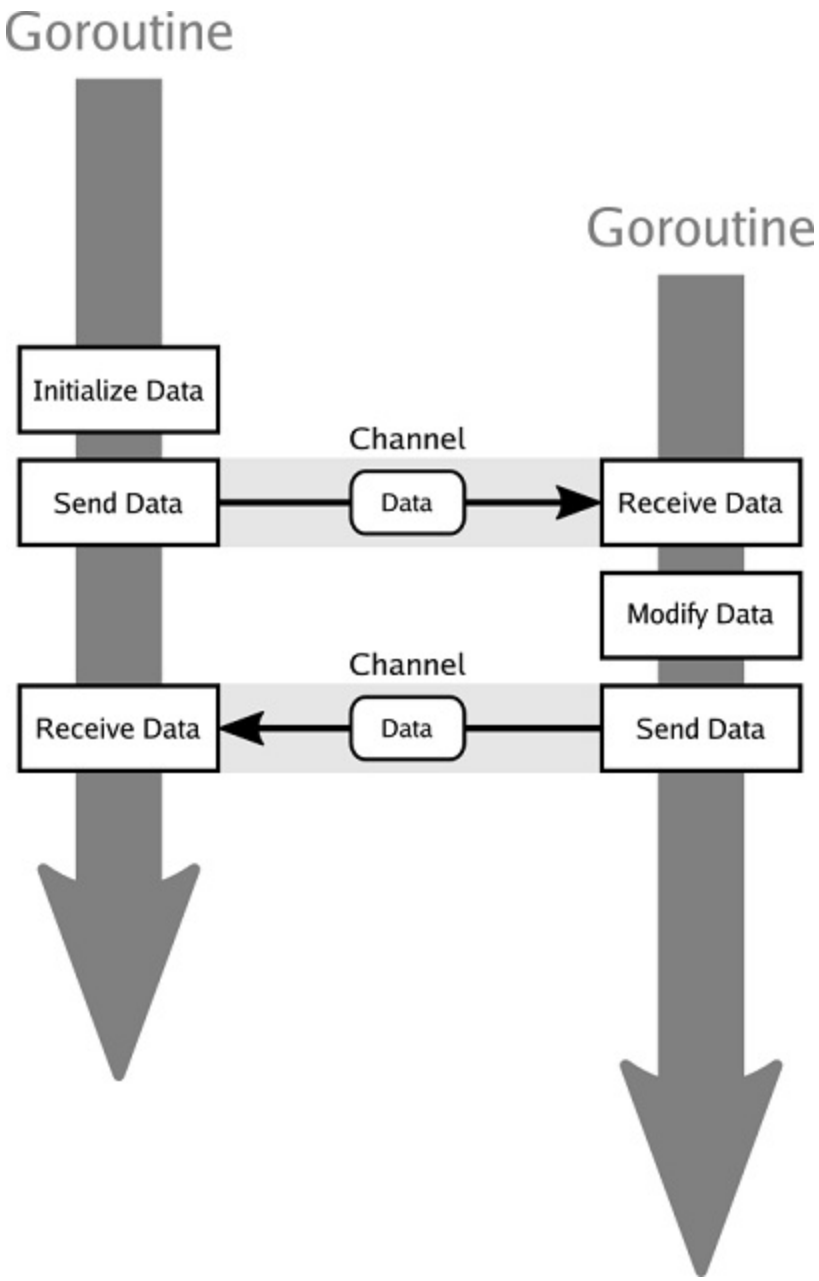


Figure 1.3 shows two goroutines and a shared channel. The first goroutine passes a data value through the channel to the second goroutine, which receives a copy of it. This exchange is synchronized, so both goroutines know the exchange took place, and can do other work before exchanging data again. After the second goroutine performs its tasks with the data, it then sends the data back to the first, waiting for acknowledgement. If the first goroutine is already ready to receive, the exchange will take place immediately, otherwise the second goroutine will wait patiently until the data is exchanged. This process requires no other locks or synchronization

mechanisms beyond the send and receive operations themselves. In code, that might look something like this, with the second goroutine sending the final message back to `main` on another channel.

Listing 1.5. Exchanging Data With A Channel

```
func main() {
        channel := make(chan string)
        output := make(chan string)

        go func() {
                // First goroutine waits for a message.
                s := <-channel
                s = s + "World!"
                output <- s
        }()

        go func(){
                // Second goroutine sends the initial message.
                s := "Hello, "
                channel <- s
        }()

        // Main goroutine waits for the final output.
        finalString :=  <- output
        fmt.Println("This string was built concurrently:", finalS
}
```

Listing 1.6. The Final Result On The `output` Channel

```
This string was built concurrently: Hello, World!
```

This is a simple and powerful abstraction, and, because the data is copied, it is generally a much safer way to share data between concurrent processes.

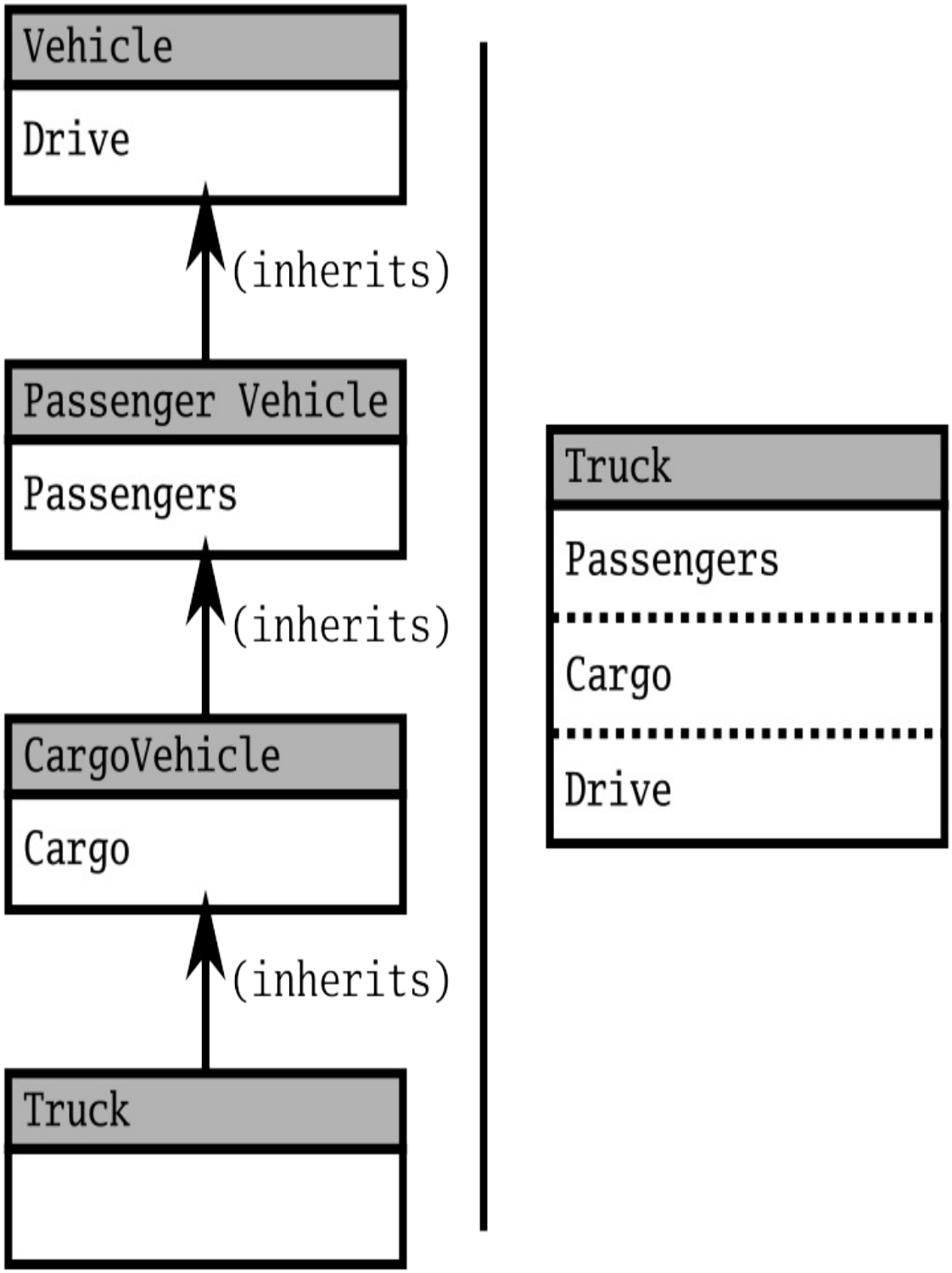
 **Note**

Special care must be taken if the data contains pointers, since only the pointer itself is copied (not the underlying data). This will be covered more in **Chapter 8**, but a good rule of thumb to remember for now is that when you pass data on a channel, you should be done with it.

1.1.4 Go’s Type System

Go provides a flexible type system that is similar to that of object-oriented langauges such as C++ in that it defines new types out of existing types, but where it differs is that it has no concept of a class hierarchy. There are no classes or inheritance in Go. Instead, Go types are *composed* of simpler types, which allows you, the programmer, to focus on your data first, rather than requiring you to come up with a class hierarchy first:

Figure 1.5. Inheritance Vs. Composition



Types Are Simple

Go types are made up of basic types such as **int** or **string** compoised together in **struct** types, which will be familiar to you if you have dealt with other C-like languages:

Listing 1.7. A Go Struct

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

Types can have methods, which help to define their behavior:

Listing 1.8. A Go Struct And Method

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

func (p Person) Name() string {
        return p.FirstName + " " + p.LastName
}

func (p Person) Describe() string {
        return fmt.Sprintf("%s is %d years old", p.Name(), p.Age)
}

func main(){
        andy := Person{
                FirstName: "Andy",
                LastName: "Walker",
                Age: 43,
        }
        fmt.Println(andy.Describe())
}
```

Listing 1.9. Output Of Calling The `Describe()` Method

```
Andy Walker is 43 years old.
```

Go encourages simple types that define their own behavior, rather than

inheriting it from somewhere else.

Interfaces Unify Behavior

Interfaces, in turn, allow related types to be used interchangably, provided they have the same behavior (as defined by their methors).[2] For example, you could make an interface, `Describer`, which represents anything that has a description.

Listing 1.10. An Interface And A Function That Uses It

```
type Describer interface {
        Describe() string
}

func PrintDescription(d Describer) {
        fmt.Println(d.Describe())
}

type Person struct {
        FirstName string
        LastName  string
        Age       int
}

func (p Person) Name() string {
        return p.FirstName + " " + p.LastName
}

func (p Person) Describe() string {
        return fmt.Sprintf("%s is a %d year-old human", p.Name(),
}

type Cat struct {
        Name     string
        Attitude string
        Color    string
}

func (c Cat) Describe() string {
        return fmt.Sprintf("%s is a %s cat (%s)", c.Name, c.Color
}

func main() {
```

```
        andy := Person{
                FirstName: "Andy",
                LastName:  "Walker",
                Age:       43,
        }
        barry := Cat{
                Name:     "Barry",
                Attitude: "the best lil' duder",
                Color:    "black and white",
        }
        benny := Cat{
                Name:     "Benny",
                Attitude: "kind of a jerk",
                Color:    "orange",
        }
        PrintDescription(andy)
        PrintDescription(barry)
        PrintDescription(benny)
}
```

Listing 1.11. Output Of Calling `PrintDescription` On Values Of Different Types

```
Andy Walker is a 43 year-old human
Barry is a black and white cat (the best lil' duder)
Benny is a orange cat (kind of a jerk)
```

It’s super easy to use interfaces in Go, since any type will "just work" as a n interface value, provided it matches the types. There’s no need to explicitly declare that a particular class implemented the `Describer` interface, the compiler can figure that out all on its own! As long as your type implements an interface, it’s good to go.

Interfaces are a powerful tool, and you will learn more about them in **Chapter 5**, which takes a deeper dive on the type system.

Generics Pick Up Where Interfaces Leave Off

Finally, as one of the more recent changes to the language, Go supports generic programming using type parameters, allowing you even greater flexibility in designing generic methods and data structures that work with a variety of different types directly:

Listing 1.12. Generic Sums

```
func SumNumbers[N int | float64](numberSlice ...N) N {
        var total N
        for i := range numberSlice {
                total += numberSlice[i]
        }
        return total
}

func main() {
        fmt.Println(SumNumbers(1, 2, 3))
        fmt.Println(SumNumbers(1.1, 2.2, 3.3))
}
//output:
// 6
// 6.6
```

This is just the tip of this iceberg! Generics are among the newest features to the Go language, and entire chapter of this book is dedicated to helping you get the most of them.

1.1.5 Garbage Collection

Like many of its peers, Go is a garbage-collected language, which means that you do not need to manage allocation of memory, since the runtime will take care of that for you.

Like the rest of the language, Go's memory managent is fully-concurrent, meaning it does not need to stop the entire program periodically to clean up memory, which is often a side-effect of this feature in languages that provide it.

Instead, Go's automatic memory management is highly optimized for intensive workloads common to the sorts of applications where it shines, and the performance and efficacy of Go memory management improves with each new release. Sub-millisecond pauses are now the norm, and soft memory limits and profile-guided optimizations can help ensure that your application is performing as well as it can, with in the bounds of its situation.

[1] "Runtime" here refers to the code that is part of every compiled Go

program. This is all of the low-level stuff for managing memory and concurrency, such as the garbage collector.

[2] This is known as *polymorphism*, specifically runtime polymorphism through structural typing for you compsci nerds.

1.2 Summary

- Go is a modern general-purpose language with powerful built-in concurrency features.
- Go is fast, and compiles quickly, leading to reduced development overhead.
- Go has a simple, easy-to-use type system with implicit interface implementation and generic programming features.
- Go is memory-safe and garbage collected, freeing the programmer from both the tedium and the worry of manual memory management.

Hopefully by this point you're as sold on Go as we were when we released the first edition of Go In Action, hot on the heels of Go Version 1.5. Now here we are nearly 8 years and 14 release cycles later, and there is more to be excited about than ever! Go has grown by leaps and bounds in popularity and features, including Generics, which are the largest addition to the language to date! It's never been a better time to be a Gopher, and we are happy to welcome you into the fold. Let's Go!

2 Diving Into Go

This chapter covers

- Building Your First Go Program
- Go Source Structure
- Go Language Fundamentals

You paid money for a book called Go In Action, so let’s see some code already! Rather than just throw a bunch of reference material at you (there are plenty of other great books for that!), we thought we might try something a little more hands-on instead. This chapter will walk you through the creation of a single piece of software that mirrors a real-world utility, touching on various language features and fundamentals along the way.

Go is more than just a language and a compiler, it’s an entire ecosystem of tools designed to help you format your code, manage dependencies, test and research documentation, and, rather than gloss over these tools until later, we believe you should get your hands dirty with them as soon as possible, so that you can have a more well-rounded foundation as a Go programmer.

So let’s get to it!

2.1 Goal: A Program To Count Words

If there’s one thing computers are good at, it’s counting things really, really quickly, which makes these tasks ripe for automation with the sort of elegant code programmers thirst for. A very common things for programmers to count are lines and words in text files, mainly because this is the quickest way to show off how many lines of code (or "loc" if you’re cool) you’ve written, so that other programmers will be jealous of how productive you are.

This is so common, in fact, that there is already a ubiquitous command-line called `wc` (short for "word count") which does this, and comes packaged with pretty much every Linux and macOS system. You could just use that, of

course, but then this chapter would be very short and you would leave very poor book reviews, and Manning would be very angry with us, so instead let's say you've decided that `wc` is too old and crusty for your refined tastes and you're going to rewrite it in Go![3]

[3] "11 stars!", "A++", "Changed my life, would buy this book for my family!", "Who says print media is dead?" These are just a few of the glowing reviews you could post, if you were so inclined.

2.2 What You Need First

You are encouraged to follow along with the code in this book to get the most out of it and get used to what it is like to write Go in an actual development environment. While you can do this with just a Go installation and your operating system's built-in terminal, you'll have a better experience with an IDE or an alternative terminal emulator. We'll discuss these in detail below, but you can also visit the book's website at https://goinaction.github.io where we keep an up-to-date list of recommended software and quick-start instructions for getting set up!

2.2.1 A Code Editor

You can use any editor you like to write Go, so use whatever is comfortable to you. If you're unsure where to start, the Go Wiki provides several popular suggestions, just check out the page above or head over to https://go.dev/doc/editors and look for links to the Wiki on IDEs and editor plugins.

The landscape of code editors is constantly shifting, so this book does not concentrate on any particular coding environment. Different editors have different levels of Go support, some of which might require extensions or plugins for things like autocompletion and documentation, so make sure you check the documentation for your particular editor if you don't find it in the link above and consider installing any support packages you need. Some editors might even install Go for you! If that's the case, just make sure you still have access to the Go tools themselves from the command line, since this book will be using those pretty regularly. If in doubt, you're probably safe to

follow the steps in the next section to install Go on your operating system of choice, just so you're guaranteed to be able to access the tools you need.

2.2.2 Git

Git (https://git-scm.com/) is a free and open-source revision control system, which is a piece of software used for managing software projects. It is one of the most popular tools for this purpose, and it is integrated into the Go toolchain as the default option to fetch dependencies from popular open software repositories, such as Github. If it is not installed by your editor integration of choice, you will want to download it from https://git-scm.com/downloads for your system and install it so that you can easily fetch 3rd-party libraries once you start to use dependencies beyond the Go Standard Library.

2.2.3 The Go Toolchain

Of course, you'll also need Go itself. Go comes packaged for most major operating systems, so the best way to kick this process off is to visit https://go.dev/doc/install and follow the instructions there. This will install the latest Go distribution on your system, which includes the extensive standard library as well as the compiler and various command-line tools that you'll be using to build and test your software.

2.2.4 Terminal Access

You will also want some interface to type these commands into, usually in the form of a *terminal emulator* application, sometimes also called a "command prompt" or simply a "terminal". If you are using an IDE, you may find that it has one built in, however every modern OS also has their own version of a built-in terminal. These can be a little bare-bones, however so you may want to try out something else. Check out https://goinaction.github.io/software/terminals/ for some possibilities!

2.3 Creating Your Project Root

The first step in every new Go project is creating a directory for your code to live. Whether this code is going to be a reusable library or a runnable program, this directory, called the *project root* is the home base for all of your code to live in so that Go knows where to find everything and how to build it. Each of the projects in this book will have their own root directory to keep them well contained. For this project, you'll be creating a directory called `wordcount` for this purpose. It might also help to create a single directory to hold all of the project root code for this book as well. Througout this book, we'll locate projects in the `gobook` directory, but feel free to omit the containing directory if you like.

Table 2.1. Creating The Project Directory

Command Description	In Linux/macOS	In Windows
Change to your home directory	`cd`	Default Terminal: `cd %HOMEPATH%` PowerShell: `cd~` PowerShell 7 `cd`
Create the book directory and `wordcount` project root	`mkdir -p gobook/wordcount`	`mkdir gobook\wordcount`
Switch to the project root directory	`cd gobook/wordcount`	`cd gobook\wordcount`

2.3.1 Initializing The Project Module

Go uses a system called *Go modules* to give names to individual software projects and manage their dependencies so that anyone who has access to the source code can be sure that they are working on the same program, with the same set of dependencies, at the same versions as everyone else. Modules will be covered in greater detail in the Chapter on Modules, but for now just issue the following command in the `wordcount` directory to get your project

ready for the code:

Listing 2.1. Initializing The `wordcount` Module

```
$ go mod init gobook/wordcount #1
go: creating new go.mod: module gobook/wordcount
```

You can use the module name `gobook/wordcount` whether or not you put your new project in a `gobook` directory. The only relevant portion of the name for now is the final path segment which *must* match the directory name you run the command in. You could, of course, simply have given it the name `wordcount`, and this will work fine, but for the purposes of convention, you might want to get used to prefixing the module name with something, since eventually this will be where repository information will go.

The `go` Command

This is the first of many appearances of your new best friend and constant companion throughout your journey as a Go programmer: the `go` command. You will be using it *a lot*.

This little Swiss Army Knife does everything from building your code to testing it, installing its dependencies, and everything in between. It is broken down into a series of subcommands, such as `mod` (which handles dependencies) being one. You'll only be touching on it briefly in this chapter, but, rest assured, there will be plenty to learn about this tool and more in the **Chapter 3**.

Once this command is run, you will be left with a directory containing one file: `go.mod`.This directory is now the *main module* for your project, and the Go toolchain will reference this `go.mod` when building any code in this directory (or in any directories beneath it). If you take a look inside this file, you will see something very similar to the following:

Listing 2.2. Your New `go.mod` File

```
module gobook/wordcount

go 1.18
```

There’s not a lot to see here yet, since this is a brand new project with no external dependencies (or code!). The first line is called the *module directive*, and it simply indicates that the name of this module is `gobook/wordcount`. The last line is called the *go directive*, and it says that this software was developed using Go version 1.18.

Now you have a shiny new module just brimming with possibilities for the future! Time to write some actual code!

`GOROOT` In Days Of Yore

History lesson: in the early days of Go, you needed to have a `GOPATH` environment variable that pointed to a directory where all of your source and dependencies lived. This directory needed a `src` subdirectory for all of your source, a `bin` subdirectory for tools and such that you built, and a `pkg` subdirectory for cached library objects. This was fine and all, but all of your code ended up in one place, which was a little weird. Oh, and it didn’t really handle versioning very well, so we had to make sites like `gopkg.in` to keep versions separate, otherwise you needed a bunch of separate GOPATHs which would each have their own copies of dependencies. Oh yeah, and also you had to check dependencies in sometimes to make sure they didn’t get deleted from public repos or to make sure they didn’t get updated on you. So, yeah, it was just a whole **thing**, and there were a bunch of third-party tools like `gb` and `glide` and `dep` that stepped up to fill in the gaps.

Eventually, however, we got Go Modules which does versioning and lets you build a project wherever, so long as you do a `go mod init` first, More importantly, we also got the Go package and checksum databases which act as public mirrors for most Go code checked in to the major public repositories, making it very unlikely that your dependencies will just disappear on you.

2.4 The First Iteration: Counting Spaces

Spaces are the natural division between words, so as a first pass, you’ll write a simple counter for spaces in a string. This will serve as the test case for the rest of the chapter, gaining improvements as you go.

2.4.1 Creating `main.go`

The first order of business is creating a file that contains some Go code to run. Open a file called `main.go` and put the following code in it.

Listing 2.3. Your First `main.go` File

```
package main

import "fmt"

func main() {
        // Use hard-coded text to start.
        text := "let's count some words!"

        var numSpaces int

        // Look at the text one byte at a time.
        for i := 0; i < len(text); i++ {
                if text[i] == ' ' {
                        numSpaces++
                }
        }

        fmt.Println("Found", numSpaces+1, "words")
}
```

Just like that, your first Go program on the books! This code reprsents a complete Go program that counts the number of spaces in the `text` variable by looking at each character and checking to see if it's a space. It's got a couple of rough edges, but you'll fix these soon enough. But first, try it out!

2.4.2 Building The Program With `go build`

In order to run your program for the first time, you can use the `go build` command from the directory where `main.go` resides. This will compile all of the .go files in that directory and, by default, it will create an executable file with the same name as the directory itself.

If you're following along, this should result in a binary file called `wordcount` (or `wordcount.exe` on Windows) that is ready to run, and you can simply call

it as a command:

Listing 2.4. Running the `wordcount` Executable From The Command Line

```
$ ./wordcount
Found 4 words
```

Congratulations! You are now a Go programmer! Welcome, fellow Gopher.

2.4.3 Runing The Program With `go run`

Alternatively, you can also type `go run` from inside the directory and run it without building first:

Listing 2.5. Running The Program

```
$ go run main.go
Found 4 words
```

`go run` is another handy command provided by the `go` tool for quick little tests like this (see Chapter 3 for more), and it will be helpful as you add to the program later in the chapter.

Before you go running off to monetize word-counting-as-a-service (WCAAS is a thing, right?), let's take a minute to check your your formatting.

2.4.4 Formatting Your Code With `gofmt`

If you're using a code editor or IDE, you may have noticed some small formatting changes to your code when you saved it. This is `gofmt` at work.

`gofmt` is another tool that comes bundled with the Go distrubution, and its job is simple: make all Go code look the same, everywhere.

Once upon a time, this might have seemed like a strange thing to add to a programming toolchain, but it is quickly becoming the norm, especially as public, open-source communities have become more popular. There was a time not so long ago when you'd see lengthy newsgroup diatribes on

syntactic style that could span weeks. It was all very fraught and dramatic and a lot of good development time was burnt debating the merits of variable grouping and cuddled else statements. Ain't no one got time for that.

With standard formatting, it doesn't matter what your preference might be. What style of bracketing should you use? Doesn't matter, it's gofmt's way. It's actually rather freeing, since now you just write your code and you don't worry about it. Moreover, standard formatting lowers the cognitive load for reading unfamiliar code since at the very least the shape of it will be the same. It's great.

If you're unsure about whether your editor is calling `gofmt` for you, just try the following command on your shiny new `main.go`:

Listing 2.6. Testing Your Formatting With `gofmt`

```
$ gofmt -d main.go
```

The `-d` ("for diff") flag tells `gofmt` that you want to see any differences your code has from the standard. If you're lucky, it will output nothing, meaning your editor has probably already taken care of that for you. If, on the other hand, you see something like this:

Listing 2.7. Output Of `gofmt -d` When Your Formatting Isn't Standard

```
--- main.go.orig
+++ main.go
@@ -3,13 +3,13 @@
 import "fmt"

 func main() {
-  // Use hard-coded text to start.
-  text := "let's count some words!"
+       // Use hard-coded text to start.
+       text := "let's count some words!"

        var numSpaces int

        // Look at the text one byte at a time.
-       for i := 0; i < len(text); i++{
+       for i := 0; i < len(text); i++ {
                if text[i] == ' ' {
```

```
                numSpaces++
        }
```

Then you probably need to check your editor settings or install some plugins. Check out the "editors" section of https://goinaction.github.io/software/editors/ for some guidance on this.

You can always use `gofmt -w` to modify your files in place, but who has time for that? The point of `gofmt` is not to care about it, and you shouldn't.

Okay, now that that's out of the way, let's break this down and explore how the different pieces fit together.

2.4.5 Package Declarations

Listing 2.8. Declaring Package `main`

```
package main
```

You may recall from 1 that Go programs are divided into "packages", which are the basic unit of code organization in Go. Packages define boundaries between the different concerns of your code, and which functions, types and values are visible in other parts of the program. Each module can contain one or more packages, each of which can be imported separately, if the programmer wishes. This line defines this file as belonging to the `main` package, which is special. This indicates to the Go compiler that you intend to produce an executable program that can be run all on its own. When Go sees a file or files in a directory that define the `main` package, it will search through them all looking for a function called `main` which is also special since it reprsents the starting point for the program.

If your program meets these two requirements, the compiler will output a single executable that can then be run from the command line.

2.4.6 Import Declarations

Listing 2.9. Importing The `fmt` Package

```
import "fmt"
```

This is what is known as an *import declaration*, which indicates that this package (`main`) is importing one or more external packages as a dependency. This helps the Go compiler know what additional software it needs to build to support your program.

In this case, there's only one dependency, but there can be any number of import declarations, and they can appear on a single line (as in this example) or in blocks of multiple imports, which you'll see shortly. This part of the code must come directly after the package declaration, so that the build toolchain can set about the business of gathering dependency information and building those dependencies if it needs to, before moving on to your code.

In this case the single dependency is the `fmt` package, which deals with formatted output and provides some helpful functions for printing output to the screen.

2.4.7 Blocks and Scope

The rest of the program is a single function `main`, which begins on line #5 and ends with the closing brace on line #19. The `main` takes no arguments and returns no values, and is the *entrypoint* into your program, meaning that when the program is actually run, code will start executing from this point. A `return` from `main` will terminate the program,

Listing 2.10. The `main` function

```
func main() {
```

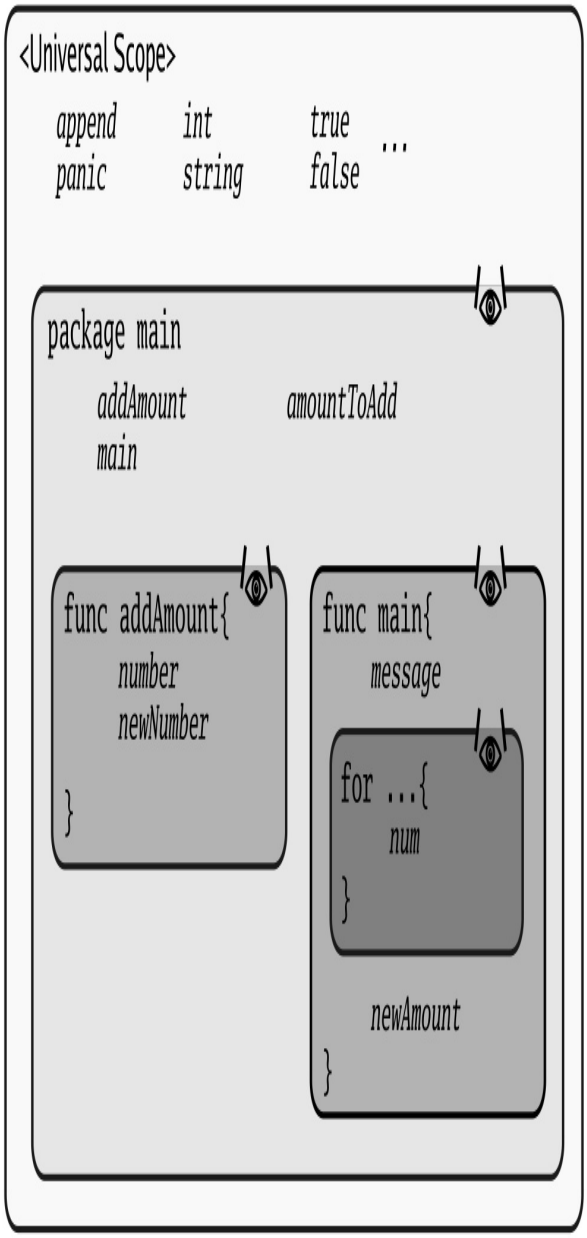
As you've might have noticed, Go code uses the C-style of defining *blocks* of code like functions and loops with curly braces `{}`, which mark the beginning and end of the block. Blocks determine the *scope*, or visibility, of identifiers such as functions names and variables. Names declared inside of a given block are visible only inside of that block after they are declared, and in any inner block declared after them.

In addition to the normal brace-delimited blocks around functions and loops,

there is also a single scope shared by each package, called the "package block" which includes all of the functions defined in that package as well as identifieres declared outside of functions, such as variables. Anything declared at this level is visible throughout the package. Above this is the "universe block", which contains built-in functions such as `append`, as well as all of the built-in Go types and constants:

Figure 2.1. Scopes And Visibility

```
package main

import "fmt"

var amountToAdd int = 1

func addAmount(number int) int {
    newNumber := number + amountToAdd
    return newNumber
}

func main() {
    message := "The new number is:"
    for num := 0; num < 5; num++ {
        fmt.Println(message, addAmountTo(num))
    }
    newAmount := 10
    fmt.Println("new amount is", newAmount
}
```



In the illustration above, each of the identifiers has been marked, and you can see four different levels of scope at work. The eye icons serve as indicators that each scope has visibility to the indentifiers of any of the scopes that surround it. This means that at the innermost level, the code in the `for` loop has access to any of the variables declared above it in the `main` function, as well as all of the items in the package scope. It cannot see `newAmount`, since this was declared later, nor does it have access to the identifiers in the `addAmount` function because they do not share a scope.

2.4.8 Variable Declaration

As you learned in 1, all variable in Go have a *type*, which defines what kind of values they can hold. Types can be assigned to variables explicitly by using the `var` keyword, or you can use the short declaration operator (`:=`) to do it automatically.

Listing 2.11. Explicitly Declaring the `numSpaces` Counter Variable With `var`

```
var numSpaces int
```

Here `numSpaces` is declared explicitly by using the `var` keyword, which defines the new variable and assigns it Go's integer type, `int` Because it is an numeric value, it will automatically be assigned a default value of 0, making it ready to use immediately. This is known as the *zero value* for that type.[4] Declarations using `var` can also specify an initial value in the form of `var name T = V`, where `T` is the type and `V` is the value, however the short variable form is usually the more convenient method.

Listing 2.12. Declaring And Assigning The `text` In A Single Step Using The Short Declaration Operator

```
text := "let's count some words!"
```

This line declares `text`, with the short variable operator `:=`, which acts exactly like an assignment, except that it also declares a new variable at the same time. `text` automatically receives the type of `string` through a mechanism is called *type inference*, which allows Go to infer the type of the

new variable from the value you are attempting to assign to it. This a very useful feature, since it saves you from needing to perform separate declaration and assignment steps every time. Instead, you can just create variables the first time you use them, and the type is taken care of for you.

Short Declaration Considerations

Short declaration is very useful, however it can have surprising behavior when changing scope. Consider the following example:

Listing 2.13. Short Declaration Shadowing A Variable

```
func main() {
        const defaultServer = "localhost"

        // getConfigValue returns false if the named value does n
        serverAddress, found := getConfigValue("server")

        if !found {
                serverAddress := defaultServer
                fmt.Println("default server set:", serverAddress)
        }
        fmt.Println("connecting to server:", serverAddress)
}
// Output:
// default server set: localhost
// connecting to server:
```

Inside of the `if` block, everything looks great, however once you exit, the value is empty again, what gives? This happens because the second assignment to `serverAddress` is actually not an assignment at all. Because it occurs within the `if` block, it's now happening inside of a new scope, and actually creates a *new* variable with the same name that only exists inside of the if block, meaning the value of the outer `serverAddress` never changes at all. This is called *shadowing*, and it can be easy to miss, but fortunately many editors will warn you about shadowing so that you can fix it.

2.4.9 Comments

Listing 2.14. Adding A Comment About The `text` variable

```
// Use hard-coded text to start.
text := "let's count some words!"
```

Comments are one of the most important parts of any software, and Go uses the C-style of syntax where line comments start with `//` and block comments are surrounded with `/*` and `*/`

Listing 2.15. Go Comment Styles

```
// A single line comment lasts until the end of the current line.

/*
        A multi-line comment
        can span multiple lines for longer
        documentation.
*/
```

 **Tip**

Go documentation is done using comments, so it helps to get in the habit of using complete sentences with proper punctuation.

2.4.10 Basic For Loops

Listing 2.16. Looping Over String Characters With A `for` Loop

```
for i := 0; i < len(text); i++ {
        if text[i] == ' ' {
                numSpaces++
        }
}
```

The next step in the process is to look at each character in the string to see if it’s a space or not. For this, you need a loop.

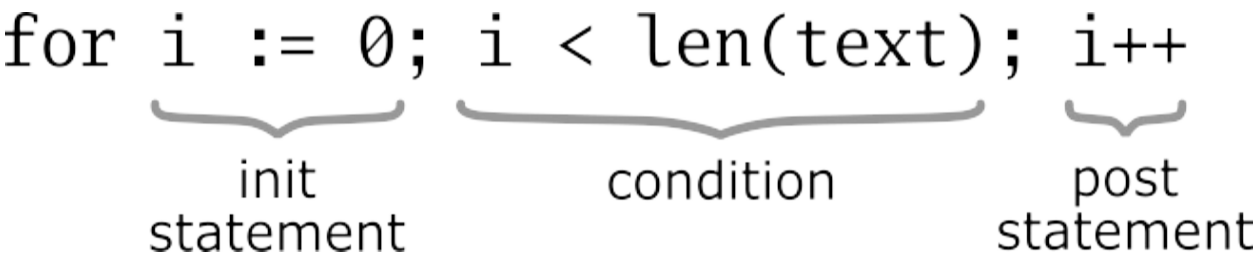
While other languages provide multiple looping constructs such as `while` and `do..while`, Go unifies all of these into the `for` statement.

The `for` loop in Go comes in three flavors, the first of which is the familiar three-part `for` loop, which begins on line #12 and ends on line #16. This will

be familiar if you have worked with c-like languages such as Javascript, C++ and Java.

It is comprised of three sections, followed by the block of code to execute:

Figure 2.2. Anatomy Of A `for` Statement



The *init statement* is run once at the start of the loop and usually initializes some important variable. It is optional.

The *condition* is a boolean expression evaluated before each loop iteration to see if the loop should run. Unlike C and Java, go requires a condition.

The *post statement* is run at the end of each loop iteration, including the last one. Usually it increments some kind of counter. It is optional.

In this code, the init statement creates a variable `i` used to track the character position in the string, this is then compared to a literal space character `' '`. If it matches, `numSpaces++` increments the counter, and the loop continues to the end.

2.4.11 Calling Functions In Other Packages

Listing 2.17. Using `fmt.Println` To Output The Results

```
fmt.Println("Found", numSpaces+1, "words")
```

Finally, after all of that counting and looping, it’s time to return your results! This is a great job for the `fmt` package, which provides methods for printing and formatting messages.

Go uses the familiar `<package>.<function>` syntax to call functions in other

packages, and so `fmt.Println` will call the `Println` function from the `fmt` packagg, which prints out its arguments in order with spaces between them to give you a nice tidy message.

 **Note**

In Go, when an identifier like a function name begins with an uppercase letter, this indicates that it is *exported*, which means that it can be accessed by anyone who imports the package. This is in contrast to other languages that require you to advertise the names you want to export and import using special keywords or lists. In Go, that first uppercase character is all you need. You'll see this throughout the rest of the chapter, when calling functions from the standard library..

[4] The zero values for boolean values, numeric values and strings are `false`, 0, and the empty string (""), respectively. All other types will get a value of `nil`. This is covered in greater detail in **Chapter 5**

2.5 A Better Way To Split The String

Counting spaces isn't the best algorithm, though. Sometimes people still use two spaces between words, and there's no guarantee that you won't have extra spaces elsewhere, either. Try adding spaces to the beginning and end of the `text` and see what happens:

Listing 2.18. Adding Some Spaces To The `text` Variable

```
text := " let's count some words! "
```

Now, suddenly the output is wrong:

Listing 2.19. Incorrect Output

```
Found 6 words
```

That's two words off! Not good! It would be better if, instead of counting the spaces, you could split the string into different words instead. Turns out

there’s a function that can do exactly this: `strings.Fields`

Listing 2.20. Altering `main.go` to use `strings.Fields And `len`

```
package main

import (
        "fmt"
        "strings"
)

func main() {
        text := "let's count some words!"
        words := strings.Fields(text)
        fmt.Println("Found", len(words), "words")
}
```

Much simpler! Only three lines of work to do all of that counting and looping.

2.5.1 Introducing Package `strings`

The `strings` package is another standard library package that, as you might imagine, deals with processing and manipulating strings. Unlike the `fmt` package, which deals primarily with printing strings and placing variables in them, `strings` provides functions to do all sorts of nifty stuff like converting strings into uppercase or lowercase, building them piece-by-piece and also splitting them up.

2.5.2 Using `go doc` To View Package Documentation

You can view the documentation for this package using another handy command, `go doc`, which parses the source for documentation comments and prints them out for you to read. By default, this command will print all package functions and types, however you can use the `-all` flag to make it print more detailed information, such as descriptions, methods or basic usage.

Listing 2.21. Using `go doc` To View The Documentation For Package `strings`

```
$ go doc -all strings
```

This will print out a listing of all of the relevant bits of package `strings` including the documentation for `strings.Fields`:

Listing 2.22. Documentaion For `strings.Fields`

```
func Fields(s string) []string
   Fields splits the string s around each instance of one or more
   consecutive white space characters, as defined by unicode.IsSp
   returning a slice of substrings of s or an empty slice if s co
   white space.
```

Viewing Documentation Online

`go doc` is a great tool for viewing code documentation, but you can also view documentation online at the go.dev website. This is a site maintained by the Go Project for viewing documentation and finding packages from across the Go ecosystem. Check out https://pkg.go.dev/strings to see the documetation online or to check out the documentation for any of the packages you see in this book.

Using standard library methods is a great convenience when you can, since many common use cases are already supported, which can save you time. In this case, you no longer have to worry whether there are extra spaces lurking around, since `strings.Fields` will simply ignore them for you.

2.5.3 Multiple Imports In An `import` Directive

In order to use `strings.Fields`, you will need to import the `strings` package by adding another package to the import directive. You can simply add another line (`import "strings"`), or you can import multiple packages in the same directive by using parenthesis.

 **Note**

Many code editors will also take care of this for you, and will arrange your imports in a consistent way. If you see your imports changing slightly when you save, this is what's going on.

Listing 2.23. Multiple Import Directive For Both `fmt` and `strings`

```
import (
        "fmt"
        "strings"
)
```

This is the way most Go code you run into will do it, though some authors will split it up. Usually you will see imports grouped with the standard library imports first, followed by a blank line and then other dependencies.

2.5.4 Slices And `len`

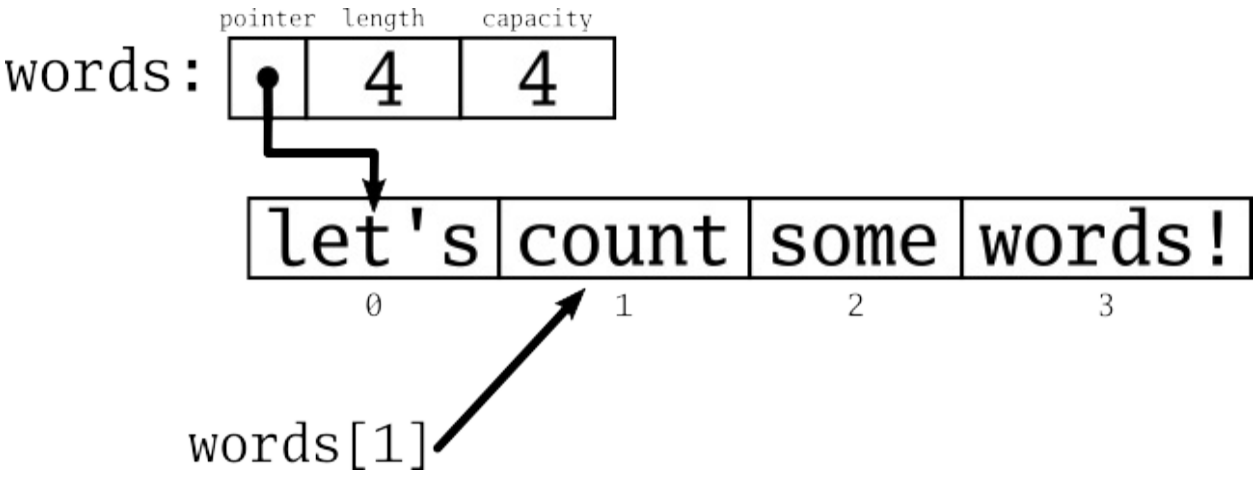
Once the `strings` package is imported, you can call `Fields` using dot-notation:

Listing 2.24. Calling `strings.Fields` And Getting The List Of Words

```
words := strings.Fields(text)
```

Looking at the documentation, you can see that `strings.Fields` takes a single string as an argument and returns a `[]string`, which is a *slice* of string values. You'll explore slices in the next chapter in more detail, but for now you can think of them as lists or arrays that can shrink and grow as you need them. Like arrays, they are *indexed* by number, and you can use the familiar bracket syntax to access individual elements in them.

Figure 2.3. The Slice Of Words



The built-in `len` method is provided so that you can get the length of things like strings, slices and maps, and so you can print it out directly:

Listing 2.25. Printing The Number Of Words Using `len`

```
fmt.Println("Found", len(words), "words")
```

 **Important**

`len` will always return the number of elements in a collection, however indices always start at 0, so remember to stop at `len(yourVariable) -1` if you're using a `for` loop.

2.6 Reading From A File

Now that you have a handy method to count words, it's time to think bigger. You're never going to grab market share from `wc` in the word-counting space if you have to edit your source code every time you need to check a new string!

For that, you need to be able to load from a file, and to do that, it's time to check out the `os` package.

2.6.1 Using Package `os` To Interact With The Filesystem

Package `os` exists to provide cross-platform access to things like environment variables, arguments, and filesystem access, which includes creating, removing and opening files on disk.

The quickest way to get the contents of the file is to just read it into memory, and if you look through the `os` package documentation, you'll find that there's a function called `os.ReadFile` that will do just that.

Listing 2.26. Altering `main.go` To Read The Contents Of A File Using `os.ReadFile`

```
package main
```

```
import (
        "fmt"
        "log"
        "os"
        "strings"
)

func main() {
        filename := "words.txt"

        fileContents, err := os.ReadFile(filename)
        if err != nil {
                log.Println(err)
                os.Exit(1)
        }

        words := strings.Fields(string(fileContents))

        fmt.Printf("found %d words", len(words))
}
```

Let's break it down:

2.6.2 Multiple Return Values

Listing 2.27. Getting Two Values From `os.ReadFile`

```
fileContents, err := os.ReadFile(filename)
```

`os.ReadFile` is a bit different from the functions you've seen so far, most notably in that it returns two values. If you look at the documentation, you can see what these are:

Listing 2.28. Documentation For `os.ReadFile`

```
func ReadFile(name string) ([]byte, error)
    ReadFile reads the named file and returns the contents.
```

Looking at the function definition, you can see that it returns two values: a slice of bytes representing the contents of the file, and a value of type `error`, indicating whether or not there were any problems opening the file.

Functions in Go can return any number of values, and a common pattern is to

return the the important values from a function first, followed by an `error` value at the end.

2.6.3 Basic Error Handling And The `log` Package

All error handling in Go eventually comes down to just two steps:

1. Check if an error occurred
2. If so, handle it as soon as possible

Error handling is important enough that we have dedicated **Chapter 7** to covering this topic in detail, but while developing small programs like this, it's often enough to just log the error and exit, returning to the error handling code later if you want to add special handling:

Listing 2.29. Logging An Error And Exiting

```
if err != nil {
        log.Println(err)
        os.Exit(1)
}
```

Error checking in Go is a simple matter of testing to see whether or not the error value is `nil`. If it is, you're good! If it's not, then an error has happened, and you need to handle it.

To separate error output from normal program output, you can swap out `fmt` for `log`, which provides many of the same functions, including `Println`, except that they will be printed to standard error and include a helpful timestamp, which makes them easy to spot.

This is followed by a call to another OS function, `os.Exit(1)`, which causes the program to exit immediately with an *exit code* of 1, indicating that the program terminated in an unexpected way.

2.6.4 Counting The Words In The Loaded File

Listing 2.30. Using `strings.Fields` On The File Contents

```
words := strings.Fields(string(fileContents))
```

After loading the file, and checking for an error, the technique is pretty similar to the last version of the program, except that `os.ReadFile` returns a slice of bytes instead of a string. This is because not all files are text, and Go doesn't want to assume anything about how you're going to handle the file. Instead it just returns the raw contents of the file that you can use however you want.

Luckily, strings are also just bytes, so if you're loading a text file, you can simply use a *type conversion* to convert the bytes directly to a string like so: `string(fileContents)`.[5]

2.6.5 Running The Program On A File

You can use `go run` to run this version as before, only this time you'll need to create a `words.txt` in the same directory as `main.go`. You can use your editor or just create the file directly from the command line:

Listing 2.31. Running The File On `words.txt`

```
$ echo this is four words > words.txt
$ go run main.go
found 4 words
```

You can also test your error handling by removing the file and seeing what happens:

Listing 2.32. Running The File With Nothing To Read

```
$ rm words.txt
$ go run main.go
2022/05/05 20:33:39 open words.txt: no such file or directory
```

Try out different contents and see how well it works!

[5] You can also use the `bytes` package which, among other things, provides many of the same methods as `strings`, but for slices of bytes instead: `words := bytes.Fields(fileContents)`

2.7 Specifying The File

Now that you can read the contents from a file, the next logical step is to make it work for *any* file, not just `words.txt`. To do this, you need to be able to process your program arguments. The `os` package has you covered here, too:

Listing 2.33. Altering `main.go` To Take The Filename As An Argument

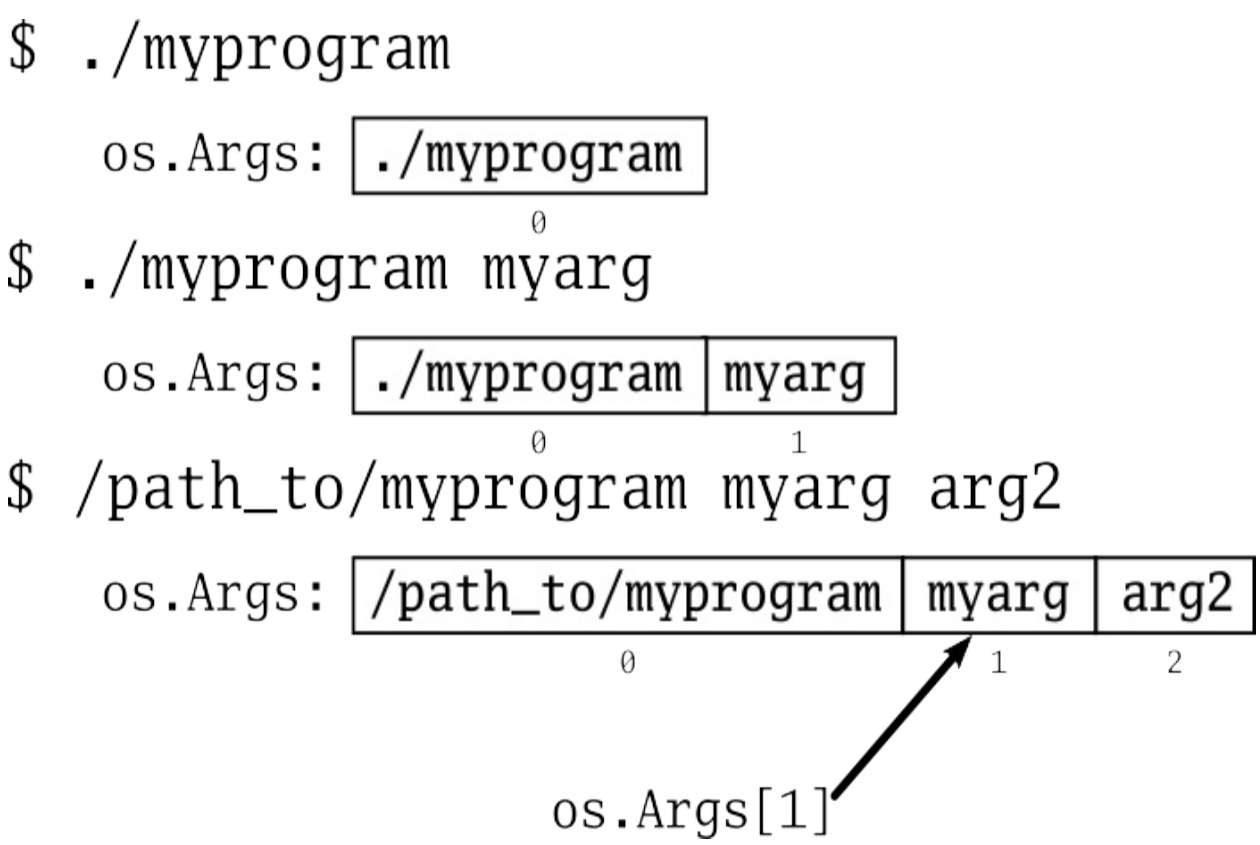
```
package main

import (
        "fmt"
        "log"
        "os"
        "strings"
)

func main() {
        if len(os.Args) < 2 {
                log.Println("need to provide filename!")
                os.Exit(1)
        }

        fileContents, err := os.ReadFile(os.Args[1])
        if err != nil {
                log.Println(err)
                os.Exit(1)
        }

        words := strings.Fields(string(fileContents))

        fmt.Println("Found", len(words), "words")
}
```

2.7.1 Getting Program Arguments From `os.Args`

Most languages will give you access to the arguments your program was called with, and Go is no exception. When your program is first started, the runtime will take stock of all of the arguments passed to your program and store them in memory. When you import the `os` package for the first time, it will place them into an exported variable named `Args`. By convention, this

list will start with the command itself, so it will always have a length of at least 1:

Figure 2.4. `os.Args` Illustrated



This allows your program to access the first argument at `os.Args[1]` and treat it as the filename to be loaded:

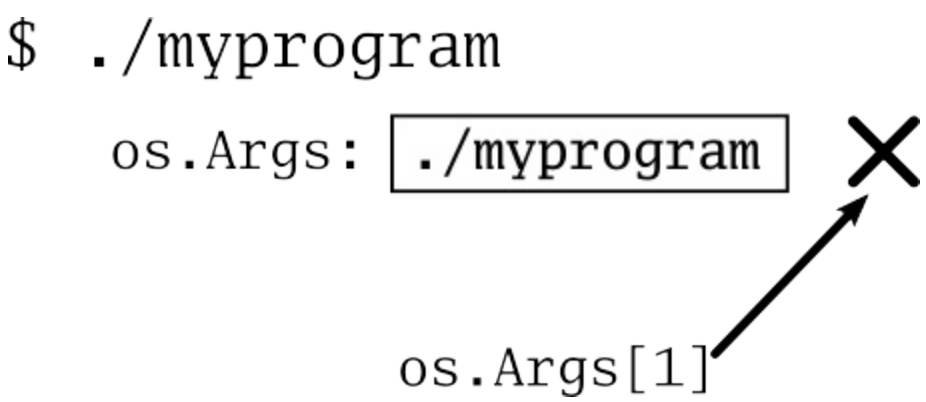
Listing 2.34. Opening The First Argument As A File

```
fileContents, err := os.ReadFile(os.Args[1])
```

2.7.2 Avoiding A Panic By Using `len`

Because `os.Args` will always contain the name of your program first, you need to look at `os.Args[1]` and above if you want to get at any of the arguments, but there's a catch: if your program isn't run with any arguments you could end up trying to access a value which doesn't exist!

Figure 2.5. Looking Where You Shouldn't



If you don't check the length of your arguments first and just blindly attempt to access the first argument, this will result in a *panic*, which is where the Go runtime will crash your program rather than trying to read a value that doesn't exist:

Listing 2.35. Panic Output On Missing Arguments

```
panic: runtime error: index out of range [1] with length 1
```

To avoid this, your best bet is to check the number of arguments first with `len`:

Listing 2.36. Checking The Argument Length With `len`

```
if len(os.Args) < 2 {
        log.Println("need to provide filename!")
        os.Exit(1)
}
```

This way, your program does the right thing if it's not given a filename to read from.

 **Important**

Always check the length of any slice with `len` before accessing an arbitrary element.

2.7.3 Running Your Program With Arguments

Now that you have a more flexible program, you can try it out on any text file you like, including its own source:

Listing 2.37. Running `wordcount` With Files As Arguments

```
$ go build
$ ./wordcount words.txt
Found 4 words
$ ./wordcount main.go
Found 41 words
```

If you have `wc` installed on your system, and you're feeling cheeky, you can even test it out to see how it stacks up:

Listing 2.38. Testing Against The Competition

```
$ wc -w words.txt
       4 words.txt
$ wc -w main.go
      41 main.go
```

Hah, take *that* `wc`!

2.8 A More Efficient Method

`os.ReadFile` has been working pretty well so far, but it has one small problem: it reads the whole file into memory each time. This is fine for smaller files, but you can make it work for any number of files of any size with just a little more work.

Reading and writing data in managable chunks is one of those tasks that's common enough that Go has an entire package dedicated to it. In this case, the package is called `bufio`, which buffers input and output so that your program doesn't hold on to more memory than it needs.

One of the types in this package that is particularly useful for reading bits text is the `Scanner` type, which can split data in a variety of ways and give it to

you a piece at a time. Using a package like this is handy, since it allows you to focus on the important parts of the task at hand, without requiring you to handle all of the implementation details yourself. In this case, it means you only have to worry about calling a couple of methods to get the latest bit of text and check for errors, without needing to write a bunch of extra code.

2.8.1 Using `go doc` To Learn More About Types

When you're looking to learn more about a package and its types, you can use `go doc` to get an overview by passing it the package and the type name:

Listing 2.39. Output Of `go doc bufio Scanner`

```
type Scanner struct {
        // Has unexported fields.
}
    Scanner provides a convenient interface for reading data such
    of newline-delimited lines of text. Successive calls to the S
    will step through the 'tokens' of a file, skipping the bytes
    tokens. The specification of a token is defined by a split fu
    type SplitFunc; the default split function breaks the input i
    with line termination stripped. Split functions are defined i
    package for scanning a file into lines, bytes, UTF-8-encoded
    space-delimited words. The client may instead provide a custo
    function.

    Scanning stops unrecoverably at EOF, the first I/O error, or
    large to fit in the buffer. When a scan stops, the reader may
    advanced arbitrarily far past the last token. Programs that n
    control over error handling or large tokens, or must run sequ
    scans on a reader, should use bufio.Reader instead.

func NewScanner(r io.Reader) *Scanner
func (s *Scanner) Buffer(buf []byte, max int)
func (s *Scanner) Bytes() []byte
func (s *Scanner) Err() error
func (s *Scanner) Scan() bool
func (s *Scanner) Split(split SplitFunc)
func (s *Scanner) Text() string
```

This is a typical document listing for a package type, and a good example of how `go doc` presents documentation at a higher level. The first thing you'll see is the definition of the type itself, as well as any fields it might export

inside of the curly braces. In this case, `Scanner` doesn't expose any fields, so all you see is a comment that it contains unexported fields, just in case you care about that sort of thing.

After the type definition, `go doc` will add the type documentation, which is usally a description of the type and its purpose, often with info about how it works. This can be more or less detailed, depending on the package, but standard library packages like this one tend to be fairly detailed to give you as much information as possible so that you don't miss anything that might be important (even if it might not concern your use-case). In this example, the `Scanner` type documentation has a lot to say about how the type is used, as well as some more technical details about how it's implemented. There's a lot going on here, but what it breaks down to is:

1. `Scanner` splits up data into smaller pieces, called "tokens".
2. The default mode is to split on newlines, meaning each "token" will be a single line of your file, with the trailing newline chopped off. You can also change this behavior if you want, using one of several pre-defined `SplitFunc` functions, or you write your own.
3. `Scanner` will keep returning tokens until you reach the end of the file (EOF) or an error occurs.

After the description, but before any methods, `go doc` will also list any functions in the same package that return a value (or pointer) of the type you are asking about. The reason for this is because some types are designed to be initialized with before they are used, so `go doc` will list anything that looks like it might be a constructor function for that type first, so that you're aware of them. In this case there is a function called `NewScanner` that plugs in the data source you'll be reading from and returns a new scanner for that data, ready to use.

Constructors In Go

While many types in Go are designed to be useful without needing initialization, others require some configuration first, and will provide constructor functions that do the setup for you. The convention for this is to have a function of the form `NewT(<arguments>) *T` where T is the type it returns, and the arguments areany requirements or options are passed as

arguments to this function.

In this case, `Scanner` needs a source to read from, so `NewScanner` takes an `io.Reader` for the data source and initializes a new scanner to read from it. We'll get to `io.Reader` in just a moment, but first, you can examine the methods in greater detail to see if you can get an idea of how `bufio.Scanner` is supposed to be used. This can be done one method at a time by using `go doc <package name> <type name>.<method name>`, or by using `-all` with type to list them all at once.

In particular, there are three important methods for the word-counting program: `Scan`, `Text`, and `Err`.

Listing 2.40. `go doc` Output For The `Scan`, `Text` and `Err` Methods

```
func (s *Scanner) Scan() bool
    Scan advances the Scanner to the next token, which will then
    available through the Bytes or Text method. It returns false
    scan stops, either by reaching the end of the input or an err
    Scan returns false, the Err method will return any error that
    during scanning, except that if it was io.EOF, Err will retur
    panics if the split function returns too many empty tokens wi
    advancing the input. This is a common error mode for scanners
func (s *Scanner) Text() string
    Text returns the most recent token generated by a call to Sca
    newly allocated string holding its bytes.
func (s *Scanner) Err() error
    Err returns the first non-EOF error that was encountered by t
```

So it looks like the general flow is to call `Scan` repeatedly, while getting new tokens with `Text`, and finally calling `Err` to see if anything went wrong. This is starting to make sense, but you'd be forgiven for thinking that the documentation alone doesn't give you the whole picture. Documentation is great for understanding the particulars, but somtimes it's far more helpful to see an example. To fill this gap many Go packages include example snippets as a part of their test suite, and you can view these using `go doc -ex`.

Listing 2.41. Example Usage From `go doc -ex bufio.NewScanner`

```
scanner := bufio.NewScanner(os.Stdin)
for scanner.Scan() {
```

```
        fmt.Println(scanner.Text())
}
if err := scanner.Err(); err != nil {
        fmt.Fprintln(os.Stderr, "reading standard input:", err)
}
```

IMPORTANT: Dear MEAP readers: For the longest time, it was actually **impossible** to list examples using `go doc`, and you had to visit https://pkg.go.dev (or its predecessor godoc.org) to find them, since the indexing process there includes special processing for example functions so that they can be presented alongside the rest of the documentatioin. This totally ruined my `go doc` self-guided-learning zen thing, and I was kind of bummed that I had to tell you to crack open a web browser just to view the examples you already had on disk. However, as luck would have it, a proposal to add this functionality (https://github.com/golang/go/issues/26715) has seen some recent activity, and seems set to land before publication of this book. Once the flags and output are finalized, this section will be updated with the proper command, though it will likely be something like `go doc -ex bufio.NewScanner`. Don't get me wrong, pkg.go.dev is still awesome, and is totally worth your time, since it provides ecosystem information `go doc` cannot, and it's very well-organized, but it's so much faster to use `go doc` locally, and knowing you can use it for whatever you need is pretty darn cool.

This makes the usage much more clear! Thanks to the example code, you can see that `Scan` is designed to be used with a `for` loop, which automatically takes care of getting new data on each iteration while at the same time making sure it's okay to proceed. Since `Scan` will return `false` when it's done, this means that the loop will keep running as long as there is more data to get and nothing has gone wrong. This is a clever pattern, since it avoids the need to check the `Scan` and `Err` methods repeatedly.

Examples like this can be very valuable when you are evaluating packages for use in your own code and can help jump-start development, since they provide a fully-working example as a starting point.

 **Tip**

Whenever you're in doubt about how you're supposed to use a package, try

using `go doc -ex` to look for example code.

Adapthing the example to the word-counting code you have so far, you can just replace the code in the loop with the code you want to call for each line, and just check `Err` when you're done. This means you can use the same `strings.Fields` approach as before, only this time you're only using it on one line at a time and keeping a running total outside of the loop:

Listing 2.42. Using `strings.Fields` Inside Of The `Scan` Loop

```
// Counter to track the running total.
var wordCount int

for scanner.Scan() {
        // This is done for each line of the file.
        words := strings.Fields(scanner.Text())
        wordCount += len(words)
}
```

This is clean and simple, and, thanks to the buffering provided by the `Scanner` type, it is much more efficient. The only outstanding task is to open the file in such a way that it can be read by the scanner as needed, instead of all at once with `os.ReadFile`.

2.8.2 Files And `io` Interfaces

Earlier, you learned that `NewScanner` takes something called an `io.Reader` as its only input, and you need some way connect a file to this type. If you check out the documentation, you'll find out that `Reader` is acually an interface, with just a single method, `Read`.

Listing 2.43. Excerpt from `go doc io.Reader`

```
type Reader interface {
        Read(p []byte) (n int, err error)
}
    Reader is the interface that wraps the basic Read method.

    Read reads up to len(p) bytes into p. It returns the number o
    read (0 <= n <= len(p)) and any error encountered. Even if Re
    n < len(p), it may use all of p as scratch space during the c
```

```
    some data is available but not len(p) bytes, Read conventiona
    what is available instead of waiting for more.
```

As we mentioned in 1, interfaces values can accept any type, so long as that type has all of the methods defined in the interface. With `Reader`, this is an easy requirement to meet, since all you need is a type that has a `Read` method that takes a byte slice and returns an integer and an error. So what you really need is a type that represents a file, and has a `Read` method.

As luck would have it, you don't have to look very far to find what you need, since right alongside `Readfile` in the `os` package lives the `os.File` type, which is the standard Go type for representing any file on disk. A quick glance at the documentation shows that the `os.File` does indeed have `Read` method, and all you need to do is call `os.Open` with a filename, and you'll get a freshly opened `*File` object, ready to be accessed.

Excerpt from `go doc os.File`

```
type File struct {
        // Has unexported fields.
}
    File represents an open file descriptor.
//...
func Open(name string) (*File, error)
//...
func (f *File) Read(b []byte) (n int, err error)
```

The Function Of The `io` Interfaces

One of the important functions of the `io`, package is that it defines a number of simple interface types that model important behavior in interacting with input/output sources in Go. In addition to `Reader`, you will find `Writer`, which defines sources that can be written to, `Seeker`, which defines sources that allow you to jump around in the data, and `Closer`, for types that have a `Close` method for indicating that the program is done with them. Some, such as `ReadCloser` or `ReadSeekCloser` are combinations of two or more of the others.

By themselves, these interfaces don't perform any function, but they are important because they establish conventions for any Go code that needs to

perform input and output. When you see a function that takes one of these interfaces as an argument, you will know right away how it intends to use the value, and what it can and cannot do with the underlying type. Further, by implementing one of these interfaces in your own code, you allow any types you might create to be used in any one of the many different locations that follow these conventions.

The `io` interfaces are one of the most common standard interfaces you will run across looking at Go code in the standard library and elsewhere, and you are encouraged to read the documentation for the `io` package to learn more about them.

2.8.3 Integrating The Scanner Loop

With the final piece in place, all you have to do is replace the `os.Readfile` call with `os.Open`, pass the file to the scanner, and start your loop:

Listing 2.44. Altering `main.go` To Use `bufio.Scanner`

```
func main() {
        if len(os.Args) < 2 {
                log.Println("need to provide filename!")
                os.Exit(1)
        }

        // Open the first argument as a file.
        file, err := os.Open(os.Args[1])
        if err != nil {
                log.Println(err)
                os.Exit(1)
        }

        // Create a scanner to read from the file.
        scanner := bufio.NewScanner(file)

        // Counter to track the running total.
        var wordCount int

        // For each line of the file, get the number of words, an
        // the total.
        for scanner.Scan() {
                words := strings.Fields(scanner.Text())
```

```
                wordCount += len(words)
        }

        if scanner.Err() != nil {
                log.Println(scanner.Err())
                os.Exit(1)
        }

        fmt.Println("Found", wordCount, "words")
}
```

This functions exactly as before, only now it will work for any file, no matter how large it is. Which is pretty cool.

2.8.4 Function Types For Modifying Behavior

There's one other interesting Go feature lurking in the type description for `bufio.Scanner` that can make things even easier still. The docs mention that you can change the method used to split the input into tokens using something called a `SplitFunc`. Looking at the godoc for this type reveals something interesting: the type is neither a struct nor an interface, but a *function*.

Listing 2.45. Excerpt From `go doc bufio SplitFunc`

```
type SplitFunc func(data []byte, atEOF bool)
        (advance int, token []byte, err error)

        SplitFunc is the signature of the split function used to
    input. The arguments are an initial substring of the remainin
    unprocessed data and a flag, atEOF, that reports whether the
    no more data to give. The return values are the number of byt
    advance the input and the next token to return to the user, i
    an error, if any.
```

This is a *function type*, which means that variables of this type can be called just like any other function. The interesting bit is that these variables can hold *any* function that function has the same inputs and outputs. This is an extremely powerful language feature that we'll expand on this more in **Chapter 5**, but the upshot here is that you can plug in a variety of different functions into a scanner to get different behavior, while still getting all of the

benefits of the `Scanner` type.

The `bufio` package even provides a few of these for you, including one called `ScanWords`, which changes the tokens from individual lines, to individual words instead.

Listing 2.46. `go doc bufio ScanWords`

```
func ScanWords(data []byte, atEOF bool)
        (advance int, token []byte, err error)

    ScanWords is a split function for a Scanner that returns each
    space-separated word of text, with surrounding spaces deleted
    never return an empty string. The definition of space is set
    unicode.IsSpace.
```

Now the task becomes almost trivially easy, since all you really need to do is add 1 to the running total each time through the loop.

Listing 2.47. Using The `ScanWords` Function Instead Of `Strings.Fields`

```
// Create a scanner to read from the file.
scanner := bufio.NewScanner(file)

// Change the split function to split on words instead of lines.
scanner.Split(bufio.ScanWords)

// Counter to track the running total.
var wordCount int

// Add 1 to the count for each word.
for scanner.Scan() {
        wordCount++
}
```

This is a great example of why it pays to check out the documentation when working with new packages: often you will find something that can make the task at hand easier or more flexible. Sometimes, you'll even learn a new trick, or another way to do things entirely, such as the `Scanner` loop technique. Documentation is a programmer's best friend and, whether it is on your local machine with `go doc` or online at https://pkg.go.dev/, you should familiarize yourself with the tools to read Go documentation, and use them often.

2.9 Reading Multiple Files

Now that you don’t have to worry about how much to read into memory, you can extend the program to read from any number of files in a single run. The basic operation remains the same only, instead of reading only one program argument, you read one *or more* of them instead.

To accomplish this, you can create a loop over all of the program arguments, starting from index 1:

Listing 2.48. Using A Range Loop To Visit Multiple Items In A List

```
func main() {
        if len(os.Args) < 2 {
                log.Println("need to provide at least one filenam
                os.Exit(1)
        }

        // Loop over all arguments from index 1 onwards.
        for _, filename := range os.Args[1:] {
                file, err := os.Open(filename)

                // If there's an error, print it and skip to the
                if err != nil {
                        log.Printf("%s: %v", filename, err)
                        continue
                }
                scanner := bufio.NewScanner(file)
                scanner.Split(bufio.ScanWords)

                var wordCount int

                for scanner.Scan() {
                        wordCount++
                }

                if scanner.Err() != nil {
                        log.Printf("scan error %s: %v", filename,
                        file.Close()
                        continue
                }

                file.Close()
                fmt.Printf("%s: %d\n", filename, wordCount)
```

```
        }
}
```

This combines a couple of useful features. `os.Args[1:]` is a *slice expression* (see Chapter 4 that lets you start from a partiular index, while the `range` loop allows you to visit each of these items in turn, assigning the value to `filename` for each pass through the loop.

`range` are a flexible and convenient syntax that lets you avoid the need to check the limits of your collections, since they will automatically terminate after viewing every item in the list. They are also one of the few places in the language where the behavior changes, depending on how many values you assign:

Listing 2.49. The Two Different `range` Loop Forms For Slices

```
stringList := []string{"A", "B", "C"}

// Single-value form gets just the index.
for index := range stringList {
        fmt.Println(index)
}
// Output:
// 0
// 1
// 2

// Two-value form gets the index and a copy of the value.
for index, value := range stringList {
        fmt.Println(index, value)
}
// Output:
// 0 A
// 1 B
// 2 C

// Discarding the index.
for _, value := range stringList {
        fmt.Println(value)
}
// Output:
// A
// B
// C
```

If you assign just one value, you can get the index for each new item, while using two values will get you a copy of the value at that index as well. Optionally, you can just get the value by assigning the index to the special blank identifier _. Range loops can be used on strings, slices and arrays, maps and channels, and we will touch on these more in the chapters on collections and concurrency.

In the meantime, let's examine the fruits of your labors. If everything has gone well, you should be able to create as many new files as you want and call the program on all of them at once. Try throwing in a non-existint file for good measure to see how well it holds up:

Listing 2.50. Running The New Code On Multiple Files

```
$ cat words2.txt
This file has a few more words in it, but it's still pretty tame
the collected works of Leo Tolstoy.
But, hey, now that you're scanning the file instead of reading it
memory, you can scan whatever you want, without fear. wc is for c
re-write everything in Go, this is the way.

$ go run main.go words.txt words2.txt not_there.txt
words.txt: 14
words2.txt: 57
not_there.txt: open not_there.txt: no such file or directory
```

Working great! Now that you have your first working program, try coming up with some other useful things you can do with it. Maybe you want to take more of the market share from WC and count lines in addition to words with a combination of the two scanning approaches. Or maybe you want to count word frequency or look for particular words. It's not too much of a stretch, so long as you handle case and punctuation (hint: check out the godoc for `strings ToLower` and `regexp ReplaceAllString`). Or just keep reading and come back to it later. The world of words is your oyster.

2.10 Summary

- Go is a compiled, statically-typed language with a simple syntax and rich feature-set.

- Go has great documetation support, and once you know how to look things up, you can find information on types and methods as well as examples to help you get started.
- Go has an extensive standard library that can provide a variety of useful features and help you simplify your code.

3 Primitive Types And Operators

This chapter covers

- Numeric Types and numeric operations
- The `bool` type
- Structs
- Pointers
- Strings, runes and string manipulation

Go comes with a number of types built in to the language for some of the most common computing tasks, such as working with numbers, strings and boolean types. In a statically-typed language like Go, these built-in types are often called *primitive types*, since they are the building blocks for all of the other types in the language.

Primitive types are the most fundamental tools in your toolbelt when building software with Go, and you will use them in every single program you write. If you think of writing software like building a house, the types in this chapter are like the nails and the screws that hold everything together.

In this chapter you'll learn more about these fundamental types, as well as some tips on using them most effectively.

3.1 Integer Types

Integer types store whole numbers of a certain range, depending on their size. In Go we have two varieties of integer—signed and unsigned—along with several different size variations for each, along with a couple of alias types which declare alternate names for certain circumstances. Integer types are very common in Go, and are used for basic math, array and slice indices, and as configuration arguments to many of the standard library functions.

Table 3.1. Predeclared Integer Types

Type Name	Description	Value Range
`int`	signed integers	*architecture-dependent*[1]
`int8`	signed 8-bit integers	-128 to 127
`int16`	signed 16-bit integers	-32768 to 32767
`int32`	signed 32-bit integers	-2147483648 to 2147483647
`int64`	signed 64-bit integers	-9223372036854775808 to 9223372036854775807
`uint`	unsigned integers	*architecture-dependent*[1]
`uint8`	unsigned 8-bit integers	0 to 255
`uint16`	unsigned 16-bit integers	0 to 65535
`uint32`	unsigned 32-bit integers	0 to 4294967295
`uint64`	unsigned 64-bit integers	0 to 18446744073709551615
`byte`	*an alias for uint8*	

`rune`	*an alias for int32*	

[1]*either 32 or 64 bits, depending on your architecture*

The types that start with `int` are the *signed integers*, which are stored in two's complement format, allowing them to represent positive and negative values. The types that start with `uint` are *unsigned integers*, and these types can only store positive values starting from 0. You might notice that the range of positive values for unsigned integers is higher than in the corresponding signed equivalent. This is because signed integers reserve one of their bits for indicating whether the value is negative or not. Unsigned integers have a few quirks that make them useful for a more narrow range of purposes, which we will discuss later in the chapter.

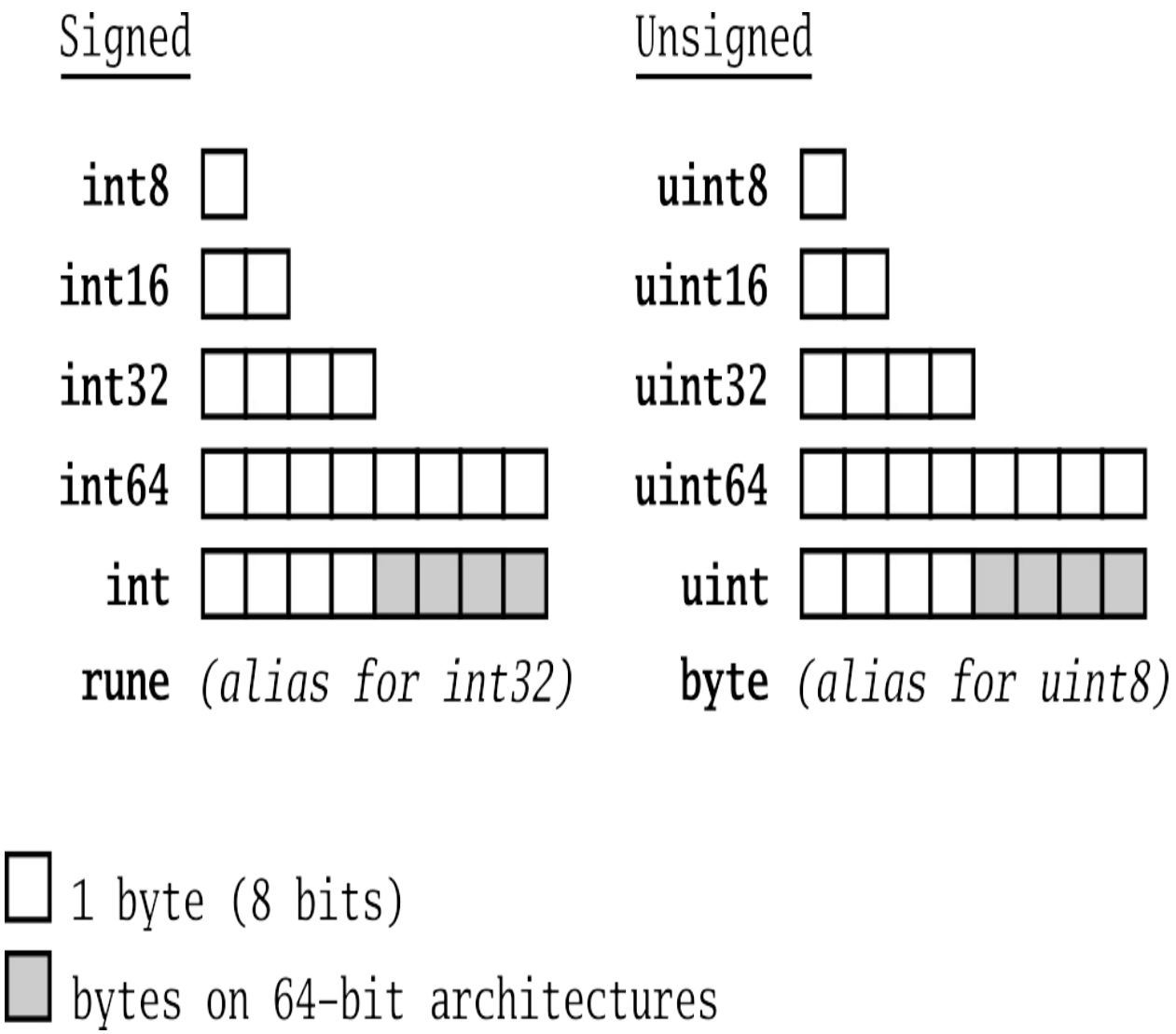
Types with a numeric suffix such as `int64` or `uint8` are *constant-sized* or *architecture-independent* types, and the number indicates the size of each type in bits. These types will always consume the same amount of memory, and have the same range of values whether they are being run on a 32-bit or 64-bit system.

By contrast, the `int`, and `uint` types are *architecture-dependent*, which means they will be either 32 or 64 bits in size, depending on the architecture of the system they are compiled for. These types are provided to give you general-purpose default integer types that will perform optimally, no matter which architecuture they are compiled for.

Rounding out the list, `byte` and `rune` are aliases that exist to give more meaningful names to integer values in certain contexts. `byte` is an alias for `uint8`, because it is extremely common to refer to data in terms of 8-bit chunks, or "bytes." The alias was created to distinguish between when code is dealing with data and when it is dealing with a numeric value. This convention is ubiquitous in the Go ecosystem, and you'll run across many functions in the standard library and elsewhere that either accept or return data as a sequence of bytes (`[]byte`). Similarly, `rune` is exactly the same as `int32`, except that it is used when talking about unicode characters in a string.

This will be covered in greater detail when we explore the `string` type, later in the chapter.

Figure 3.1. Integer Type Sizes In Memory



As you learned in Chapter 2, variables can be declared using the `var` keyword with an optional value, or with the short declaration operator `:=`, which declares the variable and sets the value at the same time. Integers declared with `var` will be of whatever type you choose, and will receive the zero value of `0` if you don't assign one. Integers declared with short declaration will receive the type of `int` along with whatever value you assign.

Listing 3.1. Declaring And Initializing Integer Values

```
// Declare a variable with a type of int (and a default value of
var myInt int

// Declare variable with type int64
var largeInt int64

// Use short declaration to create variable with type int and a v
// of 3.
i := 3

// Use a type conversion if you want to use short declaration wit
// integer types.
u := uint64(4)
```

The values `3` and `4` in the example above are known as *integer literals*, which is another term for an integer value appearing directly in the code. Integer literals can be positive or negative, and can be written in base-10, hexadecimal, octal or binary notation, depending on the prefix:

Listing 3.2. Integer Declaration With Various Different Notations

```
// These are different ways of writing the same value.
decInt := 1000         // Base 10 Notation          (no prefix)
hexInt := 0x3E8        // Hexadecimal Notation      (Prefix: "0x"
octInt := 01750           // Octal Notation            (Prefix:
octInt = 0o1750        // Alternate Octal Notation  (Prefix: "0o"
binInt := 0b1111101000 // Binary Notation           (Prefix: "0b"

// Optional underscores can be used to separate digits for readab
withSep := 1_000
hexWithSep := 0x3_E_8

// Notation is purely cosmetic, and does not affect value:
fmt.Println(decInt, hexInt, octInt, binInt, withSep, hexWithSep)
// Output: 1000 1000 1000 1000 1000 1000

// Negative integer literals:
negativeInt := -10
negativeHexInt := -0xa
```

What Are Literals?

A *literal* or *constant value* is an explicit, fixed value appearing in your source code. They are called "literals", to indicate that the value being assigned is

specified in the source, as opposed to coming from somewhere else, like a variable or function call. When you make an assignment such as `myInt := 5` or `myString = "hello"`, the `5` and the `"hello"` are literals.

3.1.1 Choosing An Integer Type

With so many integer types to choose from, it helps to know which types you might want to use, and for what purpose. For most tasks, you will probably want to reach for `int` first, since it can represent both positive and negative values, and because it will be the most efficient integer type for your architecture. It will also be less work for you in terms of conversions because it is:

1. The default type for integer literals
2. The most common integer type in standard library functions and most 3rd party packages
3. The type returned by built-in functions such as `len()` and `cap()`
4. The type used in indices in for-range loops

`int64` is the largest native integer type in the language, and you'll find it used in the standard library for values that need the greatest range, such as timestamps, file offsets and size limites. If you need the full range of values provided by `int64`, or if you will be making a lot of calls to functions that take an `int64` as an argument, it can sometimes be helpful to have a value of this type for compatibility. Otherwise, you can just convert from `int` as needed.

As for the rest of the signed types (`int32`, `int16`, and `int8`), consider them on an as-needed basis if required for compatibility or if there are measurable performance reasons for using them. If you don't *know* you need them, you probably don't. Try to avoid using another type just because it "takes less memory", since this is a form a premature optimization, and will come with extra overhead in type conversions in order to interoperate with more common types. When in doubt, run benchmarks and profile your programs to ensure that your choice in types is really the problem (usually it's not).

Unsigned integers can be a bit trickier to use correctly since they can easily

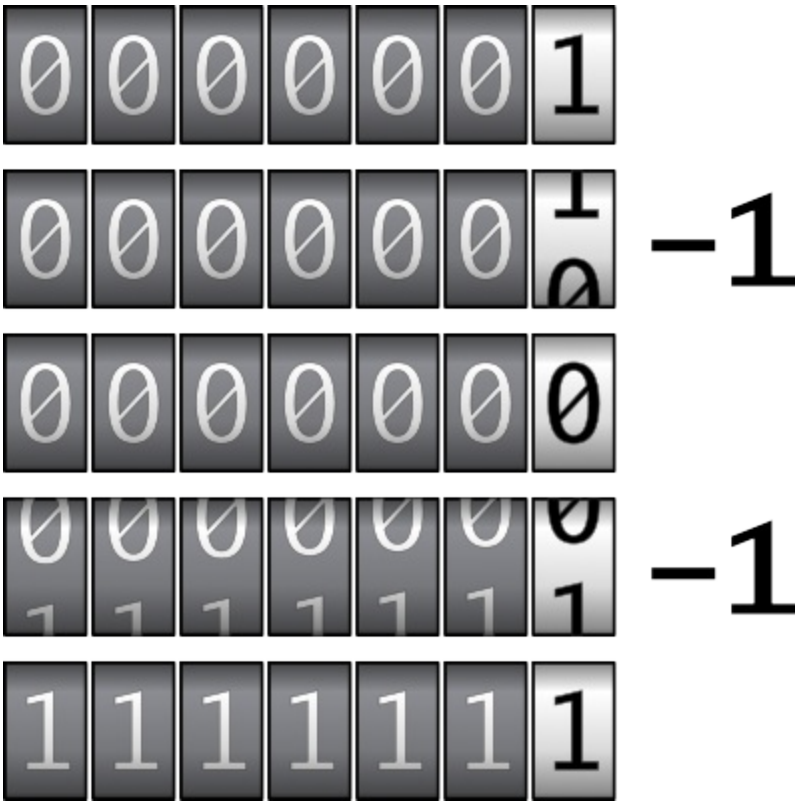
"wrap around", especially when values are small. You can see this in action by subtracting from an unsigned integer to create a value that would otherwise be negative:

Listing 3.3. Subtracting From An Unsigned Integer

```
var u uint64 // 0
u = u - 1

fmt.Println(u) // output: 18446744073709551615
```

All of the sudden, the result is much larger! The reason this happens is because the value of `u` is already as low as it can go and so subtracting causes the bits to "wrap around" from 0 to the maximum value the type can hold. You can think of this a bit like an odometer in reverse: once the value bottoms out at zero, the bits roll back around again.

Figure 3.2. Unsigned Integer Wraparound



This behavior is common to all integer types when they reach the limits of

the values they can contain, however it's much more likely with unsigned integers, since they start at the edge of their range.

A similar condition can happen when exposing unsigned integers as function arguments. It can be tempting to declare functions that take unsigned integers to indicate that only positive values are accepted, but this can have unintended consequences.

Consider the following function which takes a `uint` in an attempt to assert that it only takes positive values:

Listing 3.4. Function Taking An Unsigned Int

```
// OnlyPositive only accepts positive values.
func OnlyPositive(u uint) {
        fmt.Println("positive value is", u)
}
```

If you call the function with a negative literal value, this will appear to work as intended, since the compiler can recognize that a negative value cannot go inside of an unsigned variable:

Listing 3.5. Calling An Unsigned Int Function With An Explicitly Negative Value

```
func main() {
        OnlyPositive(-1)
}
// error: cannot use -1 (untyped int constant) as uint value in a
// OnlyPositive (overflows)
```

However, functions are not always called with literal value. You may end up in a situation where a user has an `int` value they wish to pass to your function, which means they will need to convert their `int` to a `uint` first. This is where it can go horribly wrong:

Listing 3.6. Call A Function That Takes A `uint` With A Converted Negative Value

```
func main() {
        // User has a signed integer value from somewhere else th
        // unintentionally negative.
        configVal := GetConfigValue() // returns int(-1)
```

```
        // They need to convert it to call your function, but wai
        OnlyPositive(uint(configVal))
}
// output: positive value is 18446744073709551615
```

All of the sudden it's the maximum value again! This time it's because the signed value passed is negative, and it just so happens that negative signed values contain a lot of 1s, thanks to two's complement, so you're back in a similar situation, only it's much harder to spot ahead of time.[6] Bugs like this can be tough to find once they bite you.

To avoid this pitfall, it's better to take an `int` and handle negative values yourself. The simplest option is just to return an error if a negative value is unacceptable along with some helpful documentation on the function so that the user knows the score:

Listing 3.7. Checking For A Negative Value And Returning An Error

```
// BetterOnlyPositive only accepts positive values. An error will
// returned if a negative value is passed.
func BetterOnlyPositive(i int) error {
        if i < 0 {
                return fmt.Errorf("value must be positive!")
        }
        fmt.Println("positive value is", i)
        return nil
}

func main() {
        configVal := GetConfigValue() // returns int(-1)

        if err := BetterOnlyPositive(configVal); err != nil {
                // Now you can detect the error!
                log.Fatal(err)
        }
        // Output (failure):
        // value must be positive!
}
```

Another technique is to give negative numbers a special meaning, if it applies to your use case. This is a technique you can find in the standard library for functions that perform an action some number of times, with negative

numbers meaning "unlimited" or "as many times as necessary". An example of this is `strings.Replace`, which uses negative numbers to mean "replace every occurrence":

Listing 3.8. `go doc` for `strings.Replace`

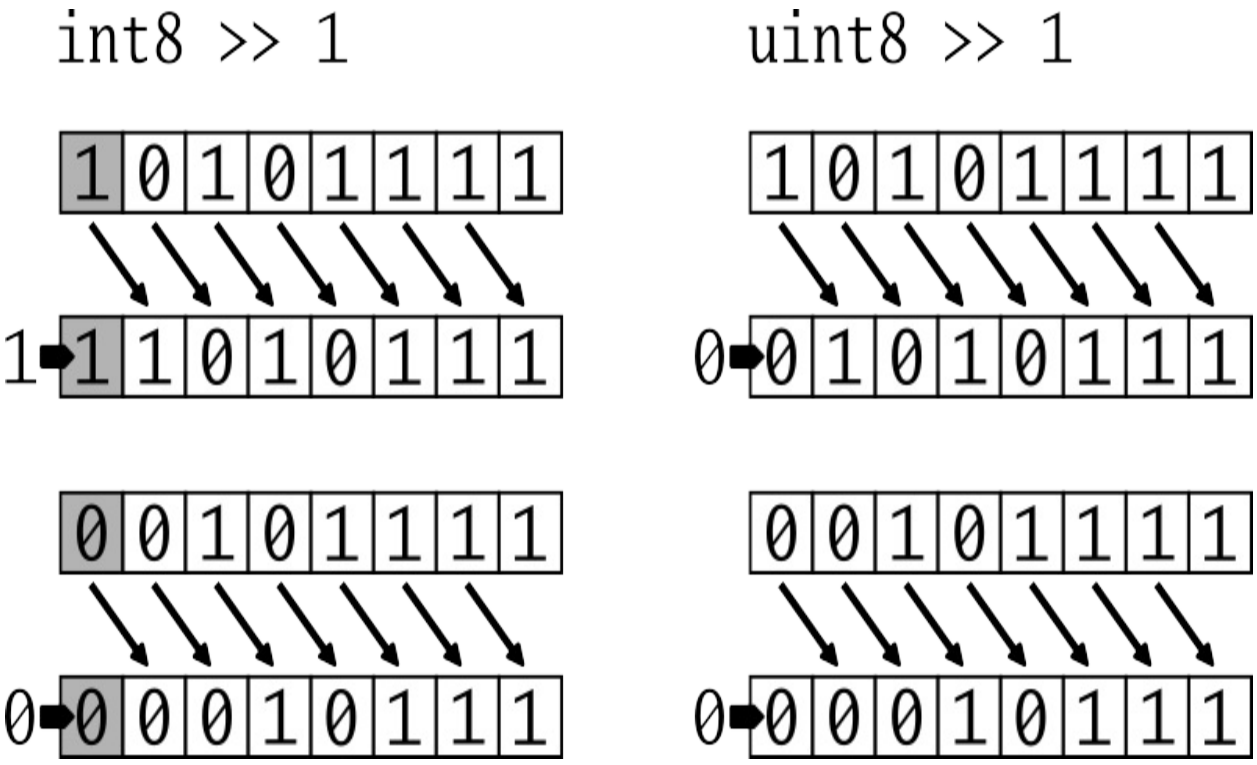
```
func Replace(s, old, new string, n int) string
    Replace returns a copy of the string s with the first n non-o
    instances of old replaced by new. If old is empty, it matches
    beginning of the string and after each UTF-8 sequence, yieldi
        k+1 replacements for a k-rune string. If n < 0, there is
        number of replacements.
```

Tip

Do not use unsigned integers to mean "I don't accept positive values". Instead, just use a signed integer and either return an error if it's negative or, alternatively use negative numbers as a "special" value, if your use case supports it.

This doesn't mean you shouldn't use unsigned integers, though! One place unsigned integers are actually very useful is in bitwise operations, since they are treated as large bit fields and you don't have to worry about the sign bit. Unsigned integers are much more predictable when shifting bits around, since they will always shift in a `0`, while signed integegers will have special behavior shifting from the right, and will use whatever their sign bit happens to be.[7]

Figure 3.3. Right Shifting In Signed And Unsigned Integer



When doing bitwise operations on unsigned values, the opposite advice applies to picking a signed type: you will probably want to stick to sized variants for these kinds of operations. This is because they will be exact sizes, and this almost always what you want when doing bit manipulation, since it will model the data in memory more precisely. The `uint` type doesn't provide this, because its length will change depending on the architecture, so you won't have as much use for it.

[6] The full particulars of two's complement are beyond the scope of this book, but you're encouraged to look it up if you're curious. Go is far from the only language to use this technique, and this potential pitfall is shared by all languages that represent numbers in this way.

[7] This is another two's complement behavior.

3.2 Floating-point Number Types

Floating-point types are used for those situations you need a number with a frational (or decimal) part, such as when calculating percentages or doing

scientific calculations. These numbers can represent a greater range of values than integers, at the cost of some precision.

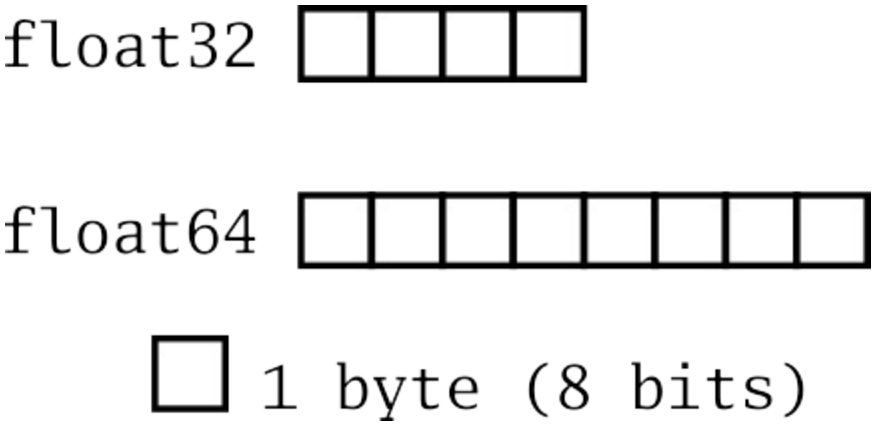
Go provides two types that implement the IEEE-754 standard for binary floating-point number representation:

Table 3.2. Predeclared Floating-point Types

Type Name	Description
`float32`	Single-precision floating-point numbers
`float64`	Double-precision floating-point numbers

`float32` takes up 32 bits of memory, while `float64` takes up 64 bits, which gives it greater precision. If you're coming from C or Java, these are equivalent to the `float` and `double` types, respectively.

Figure 3.4. Floating-point Type Sizes In Memory



Similar to integer declaration, floating-point variables declared with `var` will get a value of `0` if a value is not specified, while floating-point numbers declared with short declaration will get a type of `float64` by default:

Listing 3.9. Declaring And Initializing Floating-point Types

```
// Declare a variable doubleFloat with type float64 (and a defaul
// value of 0).
var doubleFloat float64

// Declare a variable with type float32.
var singleFloat float32

// Use short declaration to create a variable f with type float64
// value of 12.1.
f := 12.1

// Use a type conversion if you want to use short declaration wit
g := float32(12.1)
```

Why Isn't There A Plain `float` Type?

You may have noticed that, while Go provides architecture-dependent `int` and `uint` types, there is no such corresponding type for floating-point numbers. This is because the bit size of floating-point numbers is very important for precision, so it was decided early on in the evolution of the language to force an explicit choice. This is also why there is no `complex` type, since complex numbers are made up of a combination of two floating-point values.

To assign floating-point values using literals, you can add a decimal point `.` to tell the compiler that you want a floating-point value instead of an integer. You can leave out either the integer or fractional parts if you want, so `1.0`, `1.`, `0.1` and `.1` will all work. Another option is to use scientific notation by adding the `e` character to your number, followed by a positive or negative integer exponent:

Listing 3.10. Floating Point Literal Notations

```
// The following are different ways to write the same value.
floatVal = 12.          // fractional part is optional
floatVal = 12e0         // decimal point not required with exponen
floatVal = 012.         // leading zero is fine
floatVal = .12e+2       // integer part is optional
floatVal = 1_2.         // underscores allowed in floats, too
floatVal = float64(12) // explicit type for otherwise integer lit

fmt.Println(floatVal, myFloat)
```

```
// Output: 12 12

// Floats can be negative.
negativeFloat := -12.0

fmt.Println(negativeFloat)
// Output: -12
```

 **Note**

The easiest way to use an integer value as a floating point is to simply add a decimal point on the end (`12.` instead of `12`), but you can also wrap the literal with a floating point type if you want to be more explicit like so: `x:=float64(12).`

3.2.1 Special Floating-point Values

The standard Go compiler does not do any special checks for floating-point math in cases of overflow, division by zero or operations which are not mathematically valid, such as taking the natural logarithm of a negative number. In these situations the value will be set to one of the following values:

Table 3.3. Special Floating-point Values

Value	Description
+Inf	Positive Infinity
-Inf	Negative Infinity
NaN	Not a Number
These values are defined by the IEEE-754 standard	

```
These values can be checked with the `math.IsInf` and `math.IsNaN
```

Listing 3.11. Checking For Special Floating-point Values

```
f := 2.0
// Make an "infinitely huge" number.
posInf := math.Pow(f, 10_000)
fmt.Println(posInf) // output: +Inf

// Make an "infinitely small" number.
negInf := f - math.Pow(f, 10_000)
fmt.Println(negInf) // output: -Inf

// The second argument to math.IsInf lets you check for a specifi
// <0: negative infinity
// >0: positive infinity
// 0: either infinity
fmt.Println(math.IsInf(posInf, 0))  // true
fmt.Println(math.IsInf(negInf, 1))  // false
fmt.Println(math.IsInf(negInf, -1)) // true

// Make an invalid number.
notANumber := math.Log(-f)
fmt.Println(notANumber) // output: NaN

fmt.Println(math.IsNaN(notANumber)) // true
```

 **Tip**

Make sure to use use the checks in the `math` package if there's a possibility you could generate floating-point numbers that are invalid or out of range.

3.2.2 Choosing A Floating-point Type

For floating-point operations, you should pick `float64` unless you have a compelling performance reason to do otherwiwse. Most modern hardware, including 32-bit architectures, has special support for 64-bit floating-point numbers, so there little reason to sacrifice the greater precision by using `float32`.

Similar to the advice we gave in choosing integer types, you should only use

`float32` if you are trying to decrease memory consumption or improve performance, and you have determined that using `float64` is a significant bottleneck.

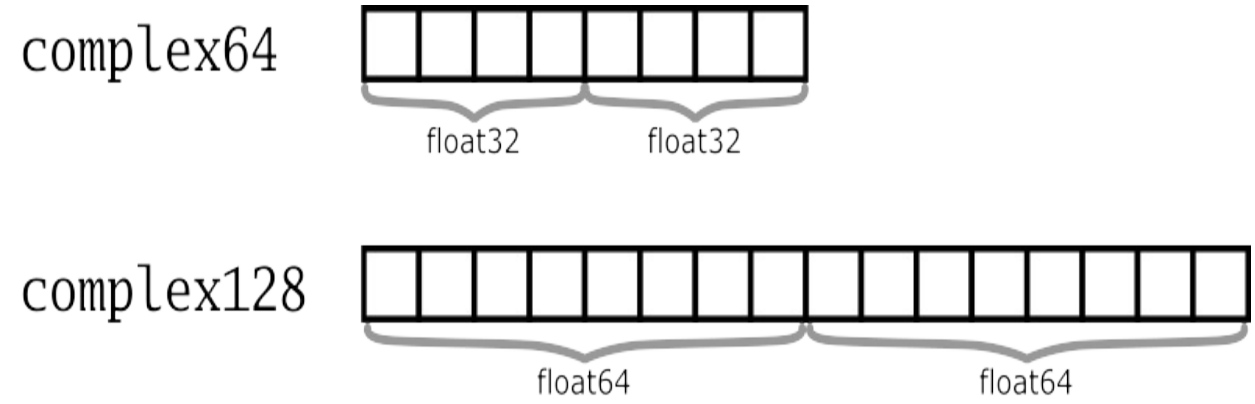
3.3 Complex Number Types

Complex numbers are a type of value that has both real and imaginary parts, and are useful in calculations for physics and signal analysis, among others. Many languages implement complex numbers in special libraries, however Go has them as native types. Complex types in Go are made up of two floating-point values representing the real and imaginary parts of the number, which makes them double the size of the float type used to implement them:

Table 3.4. Predeclared Complex Types

Type Name	Usage
`complex64`	complex numbers with `float32` real and imaginary parts
`complex128`	complex numbers with `float64` real and imaginary parts

Figure 3.5. Complex Type Sizes In Memory



Complex numbers are a bit different to the other primitive types, since they need to be declared using either the special built-in `complex()` function, or with special complex expressions.

The `complex()` function takes two arguments, which are taken to be float values, and will determine the type of the resulting complex value. If the values are `float64` or literals, they will result in a value that has the type `complex128`. If the values are `float32`, the result will be a `complex64`. Types cannot be mixed.

Listing 3.12. Creating Complex Values Using The Built-in `complex()` Function

```
cmplxValue1 := complex(1.1, 2.2)                    // complex128
cmplxValue2 := complex(float32(1.1), float32(2.2)) // complex64

cmplxValue3 := complex(1, 2)       // complex128, equivalent to 1
cmplxValue4 := complex(0, -3)      // complex128, equivalent to 0
cmplxValue5 := complex(-3.3, -4.4) // complex128, equivalent to -
```

Complex expression declarations take the form of an optional numeric value, which can be a float or an integer literal, followed by a `+` or `-`, and then an imaginary literal, which is a numeric value followed by the `i` character.

Listing 3.13. Creating Complex Values Using Complex Expressions

```
cmplxValue1 = 1.1 + 2.2i  // equivalent to complex(1.1, 2.2)
cmplxValue3 = 1 + 2i      // equivalent to complex(1,2)
cmplxValue4 = -3i         // equivalent to complex(0, -3)
cmplxValue5 = -3.3 - 4.4i // equivalent to complex(-3.3,-4.4)
```

Go also has special built-in functions `real` and `imag` that can be used to extract the float components that make up the complex number. In essence, these are the inverse of the `complex` function, and will return `float32` or `float64` values, depending on the type.

Listing 3.14. Using The `real()` And `imag()` Functions

```
cmplxValue1 := complex(1.1, 2.2)
fmt.Println(real(c1), imag(c1)) // "1.1 2.2"
```

Complex types in Go are rarely used on their own, since most of the useful functions to manipulate them reside in the `math` and `math/cmplx` packages. Additionally, most uses of complex numbers are relatively niche and… well, honestly they're pretty complex. Most programmers could go years without using them even once, but it's important to know that they exist.[8]

[8] And we would have felt bad if we left them out and then people went online and left bad book reviews because we didn't mention them.

3.4 Mathematical Operators

Numeric types are kind of boring if you can't do math and stuff with them, so Go defines all of the basic arithmetic opetations you'd expect, along with bitwise operations so you can manipulate the individual bits of integer values.

Table 3.5. Binary Arithmetic Operators

Operator	Description	For Types
+	sum	integers, floating-point, complex, strings
-	difference	integers, floating-point, complex

*	product	integers, floating-point, complex
/	quotient	integers, floating-point, complex
%	remainder	integers

With the exception of the remainder (or modulo) operatior %, binary arithmetic operators apply to all numeric types, and return the corresponding types of their operands. Importantly, operations must be performed on values of the same type, or on literals that can represent that type.

Listing 3.15. Examples Of Arithmetic In Go

```
a := 7
    b := 3

    var i int
    i = a + b // 10
    i = a - b // 4
    i = b - a // -4
    i = a * b // 21
    i = a % b // 1

    // Divide by zero can be caught by the compiler if the 0
    //c = a / 0  // error: invalid operation: division by zer

    // Otherwise, divide by zero is a runtime panic.
    zero := 0
    c = a / zero // panic: runtime error: integer divide by z

    // Float divide by zero does not panic and will result in
    // value.
    f := 1.0
    f = f / float64(zero) // +Inf

    u := uint64(1)
    u = u + i // error: mismatched types uint64 and int
    // Type conversion is needed to allow different types to
```

```
u = u + uint64(i) // 2
```

Bitwise operations only operate on integers, and will be familiar if you have worked with them in other languages. The exception is the bit clear operator `&^`, which is unique to Go, and can be thought of as the opposite of bitwise-OR (`|`), where all of the `1` bits in the second operand will be set to `0` in the result.

Table 3.6. Bitwise Operators (Integers Only)

Operator	Description
&	AND
|	OR
^	XOR
&^	bit clear
<<	shift left
>>	shift right

3.5 The `bool` Type

Go has a dedicated `bool` type for representing boolean (true/false) values. Values of this type can have one of two predeclared values, `true` and `false`, and have a zero value of `false`. `bool` is also the default type for boolean expressions, so short variable declarations such as `value := x ⇐ y` will create variables of this type.

Listing 3.16. Examples Of Boolean Values

```
var boolVar bool // Defaults to false

// Boolean variables can be used for conditionals, either
// on their own, or as part of a boolean expression. In this case
// boolVar has not been explicitly set to anything yet, so its va
// false, and the if block will not run.
if boolVar {
        fmt.Println("boolVal is true.")
}

// Once boolVar is set to true, the same if condition will pass a
// block will run.
boolVar = true
if boolVar {
        fmt.Println("NOW boolVal is true!")
}

// Short declaration variables assigned to boolean expressions wi
// given the bool type.
x := 0
y := 1
v := x < y // Type: bool, value: true.
```

3.5.1 Boolean Functions

Booleans are used primarily in conditional logic such as `if` statements, often as the return value for a helper function intended to give you a yes or no answer about a particular condition or a piece of data with some meaningful name. For example, if you wanted to print all of the even items from a list of integers, you might write a helper function called `IsEven` that would take an integer value and return a boolean indicating whether or not the number was evenly divisible by two.

Listing 3.17. A Boolean Helper Function For Use In Conditional Logic

```
func IsEven(i int) bool {
        return i%2 == 0
}

func main() {
        numbers := []int{1, 2, 3, 4}
```

```
        for _, n := range numbers {
                if IsEven(n) {
                        fmt.Printf("%d is even.\n", n)
                }
        }
}
```

While you could easily replace `IsEven` with a simple boolean expression (`if n%2 == 0 {}`) and not lose too much clarity, more complex tests or those that might depend on several pieces of information will benefit greatly from having a dedicated function, since it allows you to keep all of the logic in one place without repeating it and allows you to add additional logic to that function later if necessary without changing the calling code.

For example, you might want to write a function to tell you whether or not a particular time is during normal working hours or not. This would depend on a couple of pieces of information, such as the day of the week, and the time of day. By putting this logic into a function called `IsWorkHours()` which returns a `bool` value, you could then make determinations about things like sending a push notification, versus an email:

Listing 3.18. Using A `bool` As A Return Value For A More Complicated Query

```
// IsWorkHours returns true if the given time falls within standa
// hours.
func IsWorkHours(t time.Time) bool {
        switch t.Weekday() {
        case time.Saturday, time.Sunday:
                return false
        }
        if t.Hour() < 9 || t.Hour() > 17 {
                return false
        }
        // TODO: Add holiday tracking
        return true
}

func main() {
        if IsWorkHours(time.Now()) {
                fmt.Println("During working hours - sending a pus
                // notification sending code
        } else {
                fmt.Println("Outside of working hours - sending a
```

```
                // email sending code
        }
}
```

By putting all of this logic in `IsWorkHours`, you can use the funtion in a conditional for any actions you only want to take during working hours. It can also be extended later to take holidays into account, as marked by the TODO.

3.6 Struct Types

Struct types are a useful way to bundle several related values together into a single unit. Each struct is defined as a set of named values, called *fields*, each of which can be accessed separately.

Structs are declared using the `struct` keyword, followed by a list of fields enclosed by brackets.

Listing 3.19. An Example Struct Type

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

Fields can then be assigned and retrieved by name using dot-notation:

Listing 3.20. Setting And Retrieving Struct Values With Dot Notation

```
andy.name = "Andy"
andy.age = 42
fmt.Println(andy.name, andy.age) // output: Andy 42
```

Struct variables can also be initialized with short declaration using *struct literals*, which are a list of the names values for each field:

Listing 3.21. Declaring A Struct With Struct Literals

```
james := struct {
        name string
```

```
        age  int
}{
        name: "James",
        age:  25,
}
fmt.Println(james.name, james.age) // output: James 26
```

There are a couple of things to note here, however. For one, the trailing comma after the value for `age` is required. This is different to what you might expect if you have experience with Javascript. In Go, it is always required, which makes actually makes it easier to rearrange and add lines without worrying about removing or adding commas, and has the nice side-effect of reducing the noise in diffs!

The second thing to note is that this is *ugly* and tedious. While it might be useful for declaring one-off structs, no one wants to clutter up their code with a bunch of random struct defintions if they can help it!

For this reason, structs will often take the form of types, which allow them to be reused and passed around.

3.6.1 Structs As Types

Listing 3.22. A Simple Struct Type In Go

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

Using a type like this lets you give a struct definition a meaningful name, and use it multiple times. With `person` declared as a type, variable declaration is as simple as providing the type and the values for each field, which is much nicer:

Listing 3.23. Using A Type In Declartion

```
andy := person{
        name: "Andy",
        age:  42,
}
```

```
james := person{
        name: "James",
        age:  26,
}
fmt.Println(andy.name, andy.age)   // output: Andy 42
fmt.Println(james.name, james.age) // output: James 26
```

Structs and types form the basis of user-defined types in Go, and this example is just the very beginning of what's possible. Check out Chapter 5 for more information about declaring types in Go.

3.7 Pointer Types

Pointers are special types that represent the *location* of data, rather than the data itself. They provide a way to share memory across different parts of a program without needing to copy it. This can be very useful for passing around large or complex data structures such as those representing a connection to a database or collections of data. Pointers play a large role in Go since they are the basis for the map and slice types as well as a fundamental part of the type system.

To define a pointer variable with `var`, you can add a `*` in front of the type, which tells the compiler which type your variable will point to:

Listing 3.24. Pointer Variable Declaration

```
var intPtr *int //intPtr is a pointer(*) to an integer value(int)
```

As with all declarations, pointer values will start out with the a zero value when they are initialized. The zero value for pointers is the special variable `nil`, indicating that the pointer does not yet point to a value.

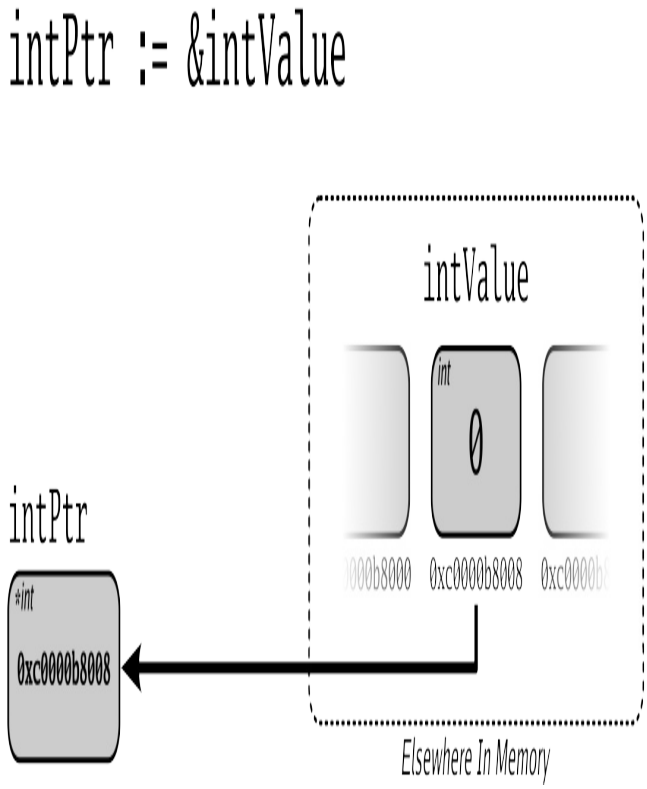
Figure 3.6. An Uninitialized Pointer Variable

In order to get an address to store in a pointer, you need to use the *address operator*, `&`, which will return the memory address of the value it's in front of. This address value will be of the type `*T`, where `T` is the type of the object.

Listing 3.25. Assigning The Address Of a Value To A Pointer

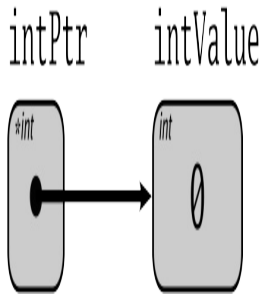
```
// declare an integer variable
intValue := 0

var intPtr *int

// get the address (&) of the variable to store in the pointer
intPtr = &intValue

// when the address operator is used with short declaration,
// the new variable will automatically be the appropriate pointer
intPtr2 := &intValue //intPtr2 is type *int
```

Figure 3.7. The Address Operator Returns The Memory Address Of A Variable.



By convention, we say that the pointer variable, "points to" the variable it references, and we will represent this with an arrow throughout the book for simplicity (and to avoid drawing a bunch of variables with random addresses in them):

Figure 3.8. `intPtr` **"Points To"** `intVar`.



The address operator is used whenever you need to get the location of a value in memory somewhere, and, with a few exceptions, you can get a pointer to almost anything. We'll cover these exceptions in the section on addressability in Chapter 5.

3.7.1 Dereferencing Pointers

Once assigned, the value of a pointer is just the address that it points to, which you can see if you try and print out the value of `intPtr`:

Listing 3.26. A Pointer Variable Just Holds An Address

```
fmt.Println(intPtr) //output: 0x14000122008#1
```

To get at the actual value being pointed to, you need to prefix the pointer variable with a `*`, which is known as *dereferencing*. This allows you to access the value at that location in memory and even modify it, if you wish.

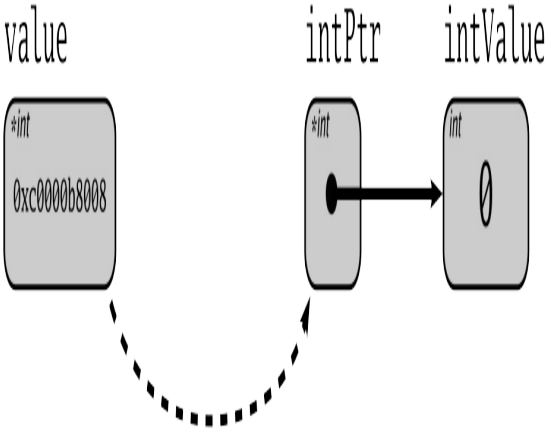
Listing 3.27. Dereferencing A Pointer To Access Or Modify Its Contents

```
fmt.Println(*intPtr) //output: 0
// You can also use dereferencing to assign to the value.
*intPtr = 1
fmt.Println(intValue) //output: 1
```

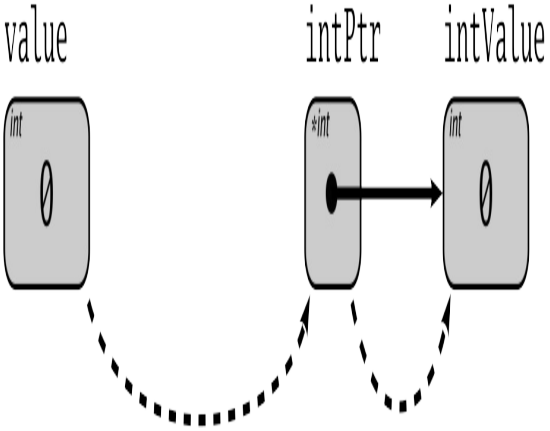
Dereferencing, in effect, follows the reference to get the actual value behind it.

Figure 3.9. Pointer Dereference Vs. Pointer Value

value := intPtr



value := *intPtr



REMEMBER: To refer to the actual value that's being pointed to, use `*variableName` (with a `*`). To get the pointer address value, use `variableName`.

3.7.2 Creating Pointers With `new` And Literals

Pointers can also be created without an intermediate value by using the address operator and a literal directly or with the built-in function `new`.

Listing 3.28. Creating Pointers Directly

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

// These are the same.
newPerson := new(person)
newPerson = &person{}
```

These are functionally identical, however the second form is preferred when working with struct types, since it also allows you to initialize struct fields while creating the pointer.

Listing 3.29. Creating A Pointer With Struct Initialization

```
newPerson = &person{
        name: "Andy",
        age: 42,
}
```

This is the more flexible style of pointer initialization for most types, and the one you'll be using most of the time, instead of `new`. The main reason to still use `new` is that it is the only way to initialize pointers to built-in types such as `int`:

Listing 3.30. Using `new` To Create A Reference To A Fundamental type

```
newIntPtr := &int    // This is not allowed
newIntPtr := &int(1) // Neither is this

newIntPtr := new(int) // New is required for fundamental types.
```

3.8 The `string` Type

Textual data is an important thing for programming languages in order to allow them to display and process messages in a variety of different

langauges. Go supports this through a combination of robust unicode support and a dedicated `string` type.

`string` a relatively simple data type in Go, which is essentially a read-only array of bytes designed for holding text. What makes it different to the other types you've learned about in this chapter is how those bytes are interpreted for display, and what happens when you modify these values.

3.8.1 Interpreted Strings

There are two different types of string literals in Go, both of which can be assigned to `string` variables. The first is the *interpreted string*, which uses the double quote (`""`) style common to most other languages:

Listing 3.31. Declaring A String In Go

```
basicStr := "Hello, Gophers!"
```

Because Go source code supports UTF-8, this means that any printable character is valid in a string:

Listing 3.32. Declaring A String With Unicode Characters

```
unicodeStr := "你好，地鼠！"
```

Interpreted strings also support a variety of escape codes, allowing you to represent non-printable characters such as tabs and newlines or to reference unicode characters by their codepoint number.

Listing 3.33. Strings With Escape Sequences

```
strWithEscapes := "Hello\n\u5730\u9F20"
fmt.Println(strWithEscapes) // output: Hello<newline>地鼠
```

3.8.2 Raw String Values

The second type of string is enclosed with backtick characters (``` `` ```), and is known as a *raw string*. Raw strings may span multiple lines, and are

interpreted literally, without any escape codes whatsoever. They also support raw unicode characters:

Listing 3.34. Declaring A Raw String

```
rawString := `Hello\n\u5730\u9F20`
fmt.Println(rawString)

rawStrWithNewlines := `I can
span विभिन्न lines`
fmt.Println(rawStrWithNewlines)
```

Listing 3.35. Raw String Output

```
Hello\n\u5730\u9F20
I can
span विभिन्न lines
```

3.8.3 String Operations

Strings can be compared using the standard comparison operators, with "less than" and "greater than" being calculated from the byte values of the string.

Listing 3.36. String Operations

```
fmt.Println("hello" == "hello") // true

fmt.Println("地鼠" == "\u5730\u9F20") // true, these escape codes
// same characters

fmt.Println("地鼠" == `\u5730\u9F20`) // false, escape codes do nc
// in raw strings

fmt.Println("abc" < "cba")   // true
fmt.Println("andy" > "bill") // false
```

Strings can also be concatenated to with the + operator:

Listing 3.37. Basic String Contcatenation

```
strHello := "hello"
strGophers := " Gophers!"
```

```
strHello += strGophers
fmt.Println(strHello) // output: "hello Gopher!"
```

3.8.4 `len()` And Unicode Characters

As we touched on in the last chapter, a `string` variable is essentially just a length value, coupled with a pointer to a backing array. You can use the built-in `len` function to get the length of the string in bytes, and, if your string is entirely ASCII characters this works exactly as you'd expect:

Listing 3.38. Using `len` On An ASCII String

```
asciiCharStr := "easy, right?"
fmt.Println(len(asciiCharStr)) // output: 12
```

Things get a little different once Unicode characters enter the mix, however:

Listing 3.39. Using `len` On A Unicode String

```
unicodeCharStr := "地鼠"
fmt.Println(len(unicodeCharStr)) // output: 6
```

All of the sudden, the length doesn't match up! What looks like two characters actually has a length that is three times that.

This is one of the most common head-scratchers people have when first experimenting with strings in Go, and the answer involves a surprising truth: strings are not actually arrays of *characters*, but arrays of the bytes that *represent* those characters in UTF-8.

Without getting too far into the weeds on UTF-8 encoding in Go (see Appendix A for that), the simple answer is that not all characters use the same number of bytes for representation. When you create a string in Go that contains characters beyond the basic ASCII characters (i.e. basic English characters and numbers), Go transparently encodes that string into UTF-8 and stuffs those bytes into the backing array for your string.

You can get an idea for the size of different characters by using `fmt.Printf` along with the special `%x` (hex) verb to visualize the bytes that make up your

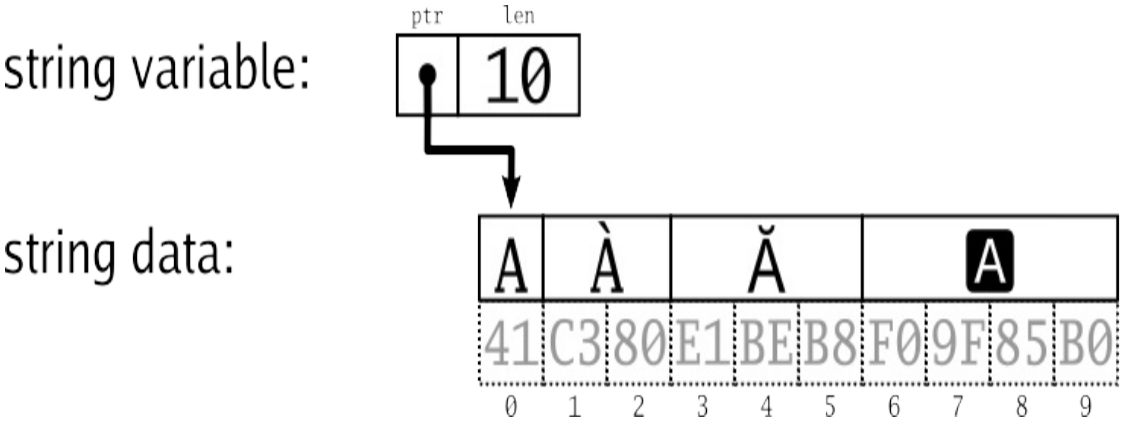
string:

Listing 3.40. Print Several Single-Character Strings Of Different Lengths

```
fmt.Printf("%x\n", "A") // output: 41
fmt.Printf("%x\n", "À") // output: c380
fmt.Printf("%x\n", "Ă") // output: e1beb8
fmt.Printf("%x\n", "🅐") // output: f09f85b0
```

The output of these lines shows that characters can be anywhere from 1-4 bytes in total length, depending on the character.

Figure 3.10. Visualizing Character Size In Bytes



Note that `"A"` is just one byte in length, exactly as it would be in ASCII. This is no coincidence, since UTF-8 is backwards-compatible with ASCII, which is the reason you can iterate a basic ASCII string with a traditional `for` loop and things work as you'd expect:

Listing 3.41. Getting ASCII Bytes As Strings With A For Loop

```
asciiCharStr := "easy, right?"
for i := 0; i < len(asciiCharStr); i++ {
        fmt.Print(string(asciiCharStr[i]) + " ")
}
fmt.Println()
// output: e a s y ,   r i g h t ?
```

But, try that with a string containing unicode, and you're gonna have a bad time:

Listing 3.42. Attempting To Get Unicode Bytes As Strings With A For Loop

```
unicodeCharStr := "地鼠"
for i := 0; i < len(unicodeCharStr); i++ {
        fmt.Print(string(unicodeCharStr[i]) + " ")
}
fmt.Println()
// output:  å  ° é ¼
```

To get at the individual characters, another approach is called for.

3.8.5 Iterating Unicode Strings Using A `range` Loop

As a special case for strings, you can use a `range` loop instead, to get a list of characters one at a time:

Listing 3.43. `range` loop On A `string`

```
unicodeCharStr := "地鼠"
for i, r := range unicodeCharStr {
        fmt.Printf("%d:%s ", i, string(r))
}
fmt.Println()
//output: 0:地 3:鼠
```

This breaks the string up into its consituent characters, regardless of their size. You still get the index where each character starts, but instead of individual bytes, you get one individual character at a time as a `rune` value.

3.8.6 The `rune` Type

Earlier in the chapter, we mentioned that the `rune` type was an alias for `int32` for string characters. This is so that you can have a type that is large enough to represent any single character that you might find in a string, and a way to compare them to each other.

Runes also separate the characters from their encoding, since they represent the value of the unicode character itself, and, once you have a character's rune, you can get more information about it, using the `unicode` and `unicode/utf8` packages:

Listing 3.44. Using The `unicode` and `unicode/utf8` Packages To Inspect Characters

```
interestingCharacters := "À🅰"
for _, r := range interestingCharacters {
        fmt.Printf("rune: %s byte length: %d\n", string(r), utf8.
        if unicode.IsLetter(r) && unicode.IsUpper(r) {
                fmt.Println("rune is an uppercase letter")
                fmt.Println("lowercase is:", string(unicode.ToLow
        }
        if unicode.IsSymbol(r) {
                fmt.Println("rune is a symbolic character")
        }
}
```

This can provide some interesting insight into the characters of a string:

Listing 3.45. Output Of Unicode Character Information

```
rune: À byte length: 2
rune is an uppercase letter
lowercase is: à
rune: 🅰 byte length: 4
rune is a symbolic character
```

Check out the documentation for these packages for more useful info!

3.8.7 Converting A `string` To A Slice Of `rune`

The `rune` type also gives you another way to get at the characters of a string, which is to convert the string directly into a slice of runes. This will give you similar results, except that now the index represents the character position in the printed string.

Listing 3.46. Converting A `string` To A `[]rune`

```
unicodeCharStr := "地鼠"
characters := []rune(unicodeCharStr)
for i, r := range characters {
        fmt.Printf("%d:%s ", i, string(r))
}
fmt.Println()
//output: 0:地 1:鼠
```

3.9 Summary

- Go provides a number of built-in primitive types for common computing tasks.
- There are several different integer types, however `int` is the most flexible, and the one you should use for most tasks.
- `bool` types are useful in constructing conditional logic, and can make complex checks easier when used as return values.
- Struct types group related values together, and are used in defining new types.
- Go supports pointer types, allowing you to pass references to data without copying it.
- Strings are read-only arrays of bytes that can contain UTF-8-encoded characters from any language.

4 Collection Types

This chapter covers

- Ordered Collections Of Data With Arrays
- Dynamic Ordered Collections Using Slices
- Slice Manipulation And Slice Expressions
- Key-value Data With Maps

Programming would be very difficult if you could only ever deal with one piece of data at a time. This is the reason we have collection types, which help us gather items together.

Go has three different collection types: arrays, slices and maps. Arrays represent fixed-sized lists like you might find in C or C++, while slices take this a step further and provide an efficient dynamic array type. If you've ever wanted a flexible list of items that can expand as needed to suit dynamic data at runtime, slices have your back.

If you need more flexibility, the native map type allows you to store any type of data using a key of any type. If you want to store items by name, size or any other index you might want, maps are a great solution.

The best part is that these data structures are baked into the language, so that they can be used anywhere in your program at any time, without needing to import a special package.

Between slices and maps, you can solve a whole host of interesting problems and, once you learn how these data structures work, programming in Go will become fun, fast, and flexible.

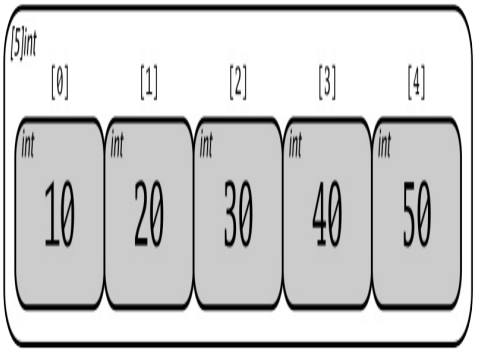
4.1 Arrays

An array in Go is a fixed-length collection that contains a contiguous block of elements of any type.

Listing 4.1. An Example Array

```
var intArray [5]int

intArray[1] = 20
intArray[2] = 30
intArray[3] = 40
intArray[4] = 50
```

Figure 4.1. Illustation Of `intArray` In Listing 4.1 After Declaration And Assignment



The fiture above illustrates the data structure `intArray` as it exists in memory after its declaration. The elements of the array are marked a grey boxes and are connected in series to each other. Each element contains the same type and can be accessed through a unique index position.

Note

Indexes start at `0`.

Arrays are valuable data structures because the memory is allocated sequentially. Having memory in a contiguous form can help keep data loaded within CPU caches longer. Using index arithmetic, you can iterate through all the elements of an array quickly.[9]

4.1.1 Declaring Arrays

Arrays in Go require a size, which defines the number of elements they can hold, as well as the type of elements. Together, the size and the element type

form the type of the array. Array declaration can take several different forms, depending on the size of the array you want and whether or not you want to give a starting set of values

Table 4.1. Different Types Of Array Declarations

Format	Description	Example	Result
`var A [SIZE]TYPE`	Array Variable Declaration	`var intArray[5]int`	`[0 0 0 0 0]`
`A := [SIZE]TYPE{VALUES}`	Array literal with optional starting values	`intArray := [5]int{10,20,30}`	`[10 20 30 0 0]`
`A := [SIZE]TYPE{INDEX: VALUE}`	Array literal with sparse values	`intArray := [5]int{1: 20, 3: 40}`	`[0 20 0 40 0]`
`A := […]TYPE{VALUES}`	Array literal with automatic size	`intArray := […] {10,20,30}`	`[10 30 30]`

Arrays are always initialized with zero values when created, depending on the type of their elements. For example, arrays of numbers will start out with `0` for all unspecified values, while arrays of strings will have the empty string (`""`), and so on.

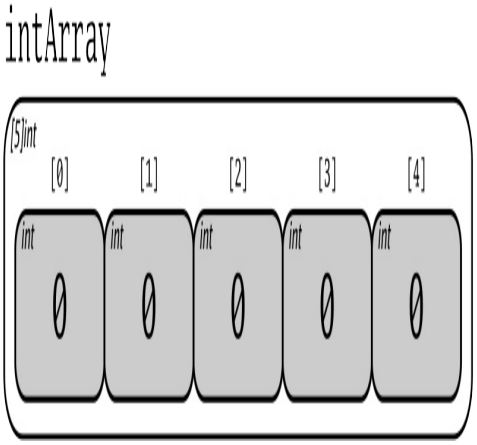
Once an array is declared, neither the type of data being stored nor its length can be changed. If you need more elements, you need to create a new array with the length needed and then copy the values from one array to the other.

Declaring an array with `var` will give you an array of the specified size, with all values pre-set to the zero value:

Listing 4.2. Declaring An Array With `var`

```
var intArray [5]int
```

Figure 4.2. The `intArray` Value, After Declaration

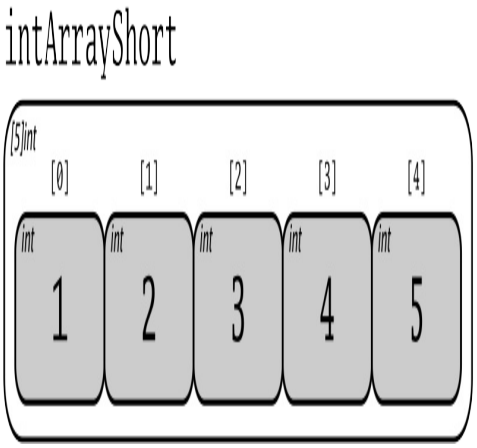


You can also use short variable declaration with an *array literal* to declare and initialize an array at the same time. Array literals take the form of two curly brackets (`{}`) with a comma-separated list of values inside:

Listing 4.3. Initializing An Array With Values Using Short Variable Declaration

```
// Declare and initialize an array with values.
intArrayShort := [5]int{1, 2, 3, 4, 5}
```

Figure 4.3. The `intArrayShort` Value, After Declaration

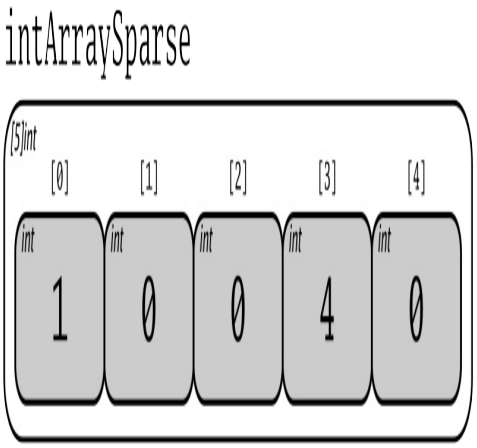


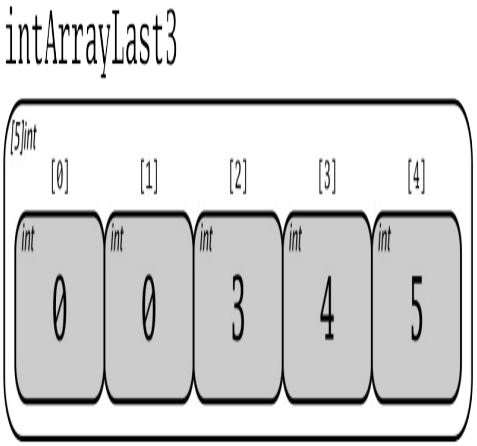
Optionally, you can specify only certain indices allowing you to set specific values or start from another location.

Listing 4.4. Initializing Only Some Elements Of An Array

```
// Declare and initialize an array with only certain values.
intArraySparse := [5]int{1: 2, 3: 4}
// Declare and initialize an array starting from index 2.
intArrayLast3 := [5]int{2: 3, 4, 5}
```

Figure 4.4. Array Declared With Only Certain Values



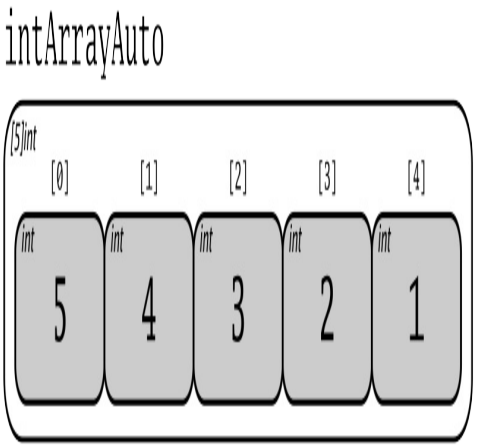


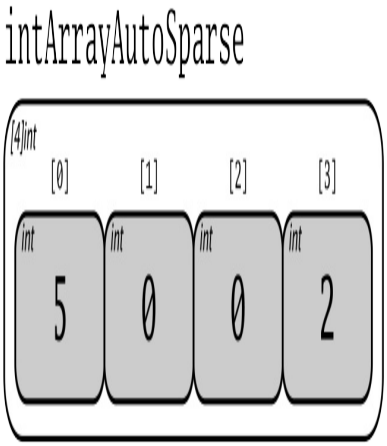
As a further convenience, you can also replace the length with ..., and the compiler will automatically create an array type of the approproate length.

Listing 4.5. Initializing An Array With Automatic Length

```
// Declare and initialize an array with automatic length.
intArrayAuto := [...]int{5, 4, 3, 2, 1}

// If only using certain values, the largest value will be taken
// size.
intArrayAutoSparse := [...]int{0: 5, 3: 2}
```

Figure 4.5. Arrays Declared With Lengths Automatically Calculated





Important

While automatic array length is a useful feature in fixtures such as tests, it should be used with care, since you may end up with lengths you did not intend. It is better to be explicit.

4.1.2 Array Type And Length

In Go, the length of the array is part of its type, which is the reason you cannot change an array's size, since that would make it something else!

This also means that the type `[5]int` is completely different to `[4]int`, and so they are not interchangable, even if you attempt to assign a smaller array to a larger one.

Listing 4.6. Attempting To Assign Array Values Of Different Lengths

```
var (
        fourArray [4]int
        fiveArray [5]int
)
fiveArray = fourArray
// error: cannot use fourArray (variable of type [4]int) as type
// in assignment
```

To get the size of an array, you can use the builtin `len()` function, which will return an `int` value representing the number of elements the array can hold. This number will always be the full size of the array, whether you've chosen to initialize any of its values or not:

Listing 4.7. Getting The Size Of An Array With `len()`

```
var intArray [3]int
myArray := [3]int{1, 2, 3}

fmt.Println(len(intArray)) //output: 3
fmt.Println(len(myArray))  //output: 3
```

4.1.3 Working Array Elements

Array elements can be set and retrieved using the familiar square bracket notation, where the index is an integer literal or a variable of any integer type.

Listing 4.8. Accessing Array Elements With Bracket Notation

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

firstValue := intArray[0] // 1

// Access the item using a variable value as the index.
index := 0
integerValue := intArray[index] // 1

// Other integer types work as well.
index32 := int32(1)
index64 := int64(2)
indexUint := uint64(3)
```

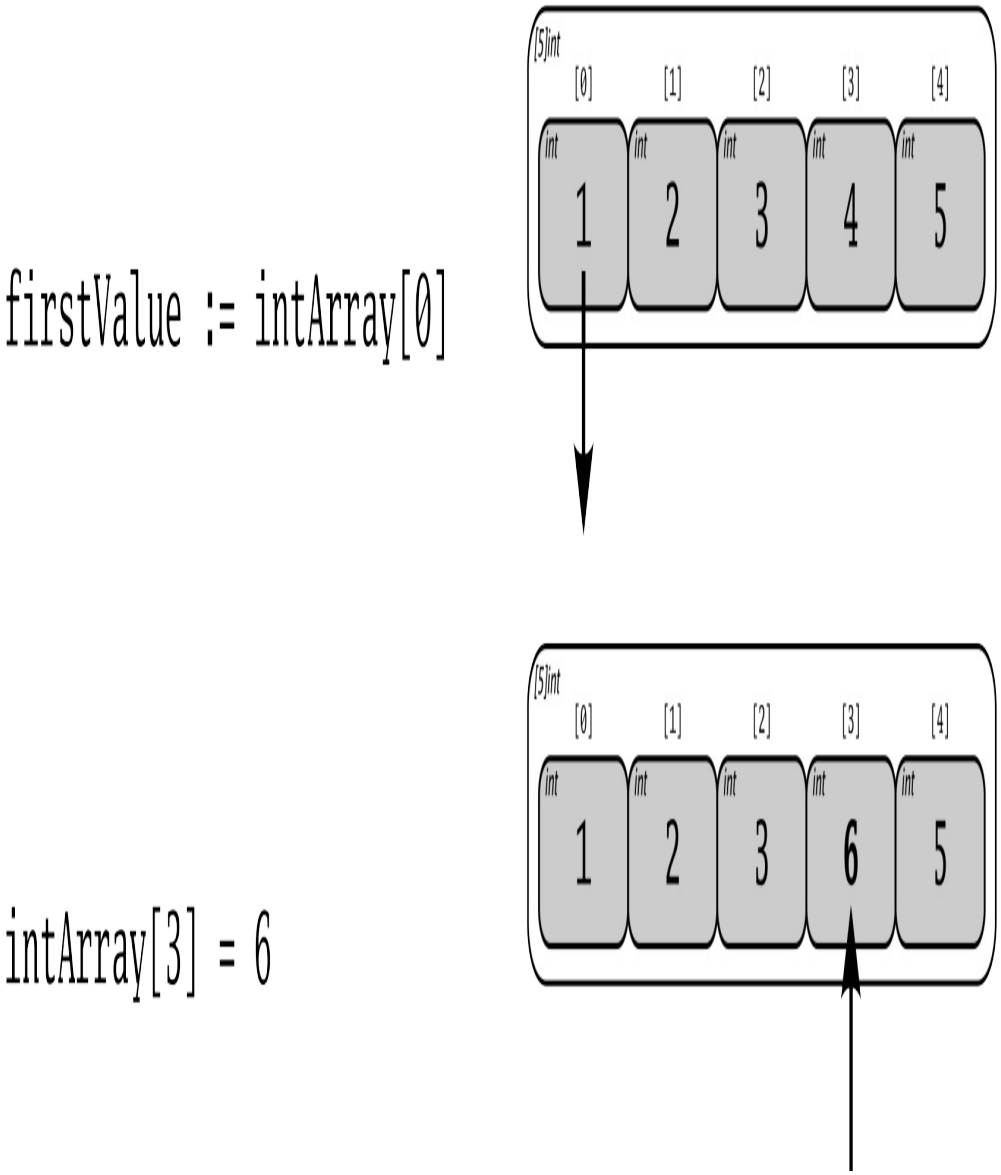
```
integerValue = intArray[index32] // 2
integerValue = intArray[index64] // 3
integerValue = intArray[indexUint] // 4
```

Changing the value of an array value is done in the same way:

Listing 4.9. Changing An Array Element

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

intArray[3] = 6
```

Figure 4.6. Retrieving And Setting Array Values



If the array value is a pointer, the value of that pointer can be accessed using the indirection operator, like you learned in Chapter 3.

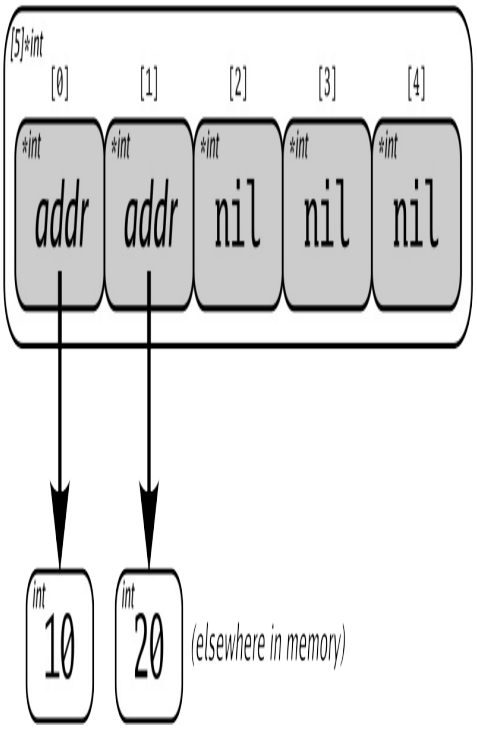
Listing 4.10. Accessing Array Pointer Elements

```
// Declare an integer pointer array of five elements.
// Initialize index 0 and 1 of the array with integer pointers.
ptrArray := [5]*int{0: new(int), 1: new(int)}
// Assign values to index 0 and 1.
*ptrArray[0] = 10
*ptrArray[1] = 20
```

```
valOfPtr := *ptrArray[0] // 10
```

This will result in the following, after the array operations are complete:

Figure 4.7. An Array Of Pointers That Point To Integers



4.1.4 Iterating Arrays With `for`

Iterating arrays in Go is done with a `for` loop, using either a `range` clause, or the more traditional way using `len` and indices.

Listing 4.11. Example Loops

```
for i := range items {
        // do something with items[i]
}

for i, item := range items {
        // do something with item or items[i]
}

for i := 0; i < len(items); i++{
```

```
        // do something with items[i]
}
```

`range` loops are the easiest and most flexible way to loop through items in Go, since they take care of providing sequential indices and items for you, and will and will stop at the end of the list automatically. In the case of arrays (and slices, which we will explore shortly), range loops can be used with either 1 or 2 values. When used with a single value, you will get an integer value starting at `0`, representing the starting index, and this value will increase by one for each iteration of the loop, until it has returned the final index.

Listing 4.12. Iterating An Array Using `range` To Get Indices

```
array := [...]int{1, 2, 3, 4}

// Use a range statement to provide indices in order.
fmt.Println("Starting range loop:")
for i := range array {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
fmt.Println("Range loop complete!")
// output:
// Starting range loop:
// index 0 is: 1
// index 1 is: 2
// index 2 is: 3
// index 3 is: 4
// Range loop complete!
```

If you add a second value to the `range` expression, you will receive both the current index as well as a copy of the value at that index:

Listing 4.13. Iterating An Array Using `range` To Get Indices And Values

```
array := [...]int{1, 2, 3, 4}

// Use a range statement to provide indices and copies of each va
for i, item := range array {
        fmt.Printf("index %d is: %d\n", i, item)
}
// output:
// index 0 is: 1
// index 1 is: 2
```

```
// index 2 is: 3
// index 3 is: 4
```

You can also ignore the index by assigning it to the blank identifier (_), leaving you with just a copy of each value. This can be useful for performing actions on every item in a list without needing to handle the index:

Listing 4.14. Iterating An Array Using `range` To Get Indices And Values

```
array := [...]int{1, 2, 3, 4}

// Use a range statement to operate on values only.
for _, item := range array {
        fmt.Println(item)
}

// output:
// 1
// 2
// 3
// 4
```

Range Loops And Value Copies

As mentioned above, the values you get from a two-value range loop will be *copies* of the values in the array, which can result in unexpected behavior if you attempt to modify them:

Listing 4.15. Attempting To Modify Items In A Range Loop

```
strings := [...]string{"hello", "gophers"}
// Get each element of the array as variable 's'.
for _, s := range strings {
        // Add an exclamation point.
        s = s + "!"
}
fmt.Println(strings)
// output: [hello gophers]
```

This example does not do what you might want, since each value is a copy. To actually modify the elements, you could either make the array a list of pointers, or simply just use the index to change them directly.

Listing 4.16. Modifying Items In A Range Loop Using Their Index

```
for i := range strings {
        strings[i] = strings[i] + "!"
}
fmt.Println(strings)
// output: [hello! gophers!]
```

Iterating with a for loop and indices is fairly straightforward, and will be familiar to you if you have used any C-like language before:

Listing 4.17. Iterating An Array Using `for` and indices

```
array := [...]int{1, 2, 3, 4}

for i := 0; i < len(array); i++ {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
```

This style of iteration is useful if you need to go through the elements in another order, such as in reverse or skipping every other element:

Listing 4.18. Iterating An Array Using `for` to traverse the list in a different order.

```
array := [...]int{1, 2, 3, 4}

// Iterate the array backwards.
for i := len(array) -1; i >=0 ; i-- {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
// output:
// index 3 is: 4
// index 2 is: 3
// index 1 is: 2
// index 0 is: 1

// Skip every other element.
for i := 0; i<len(array); i+=2 {
        fmt.Printf("index %d is: %d\n", i, array[i])
}
// output:
// index 0 is: 1
// index 2 is: 3
```

4.1.5 Arrays As Values

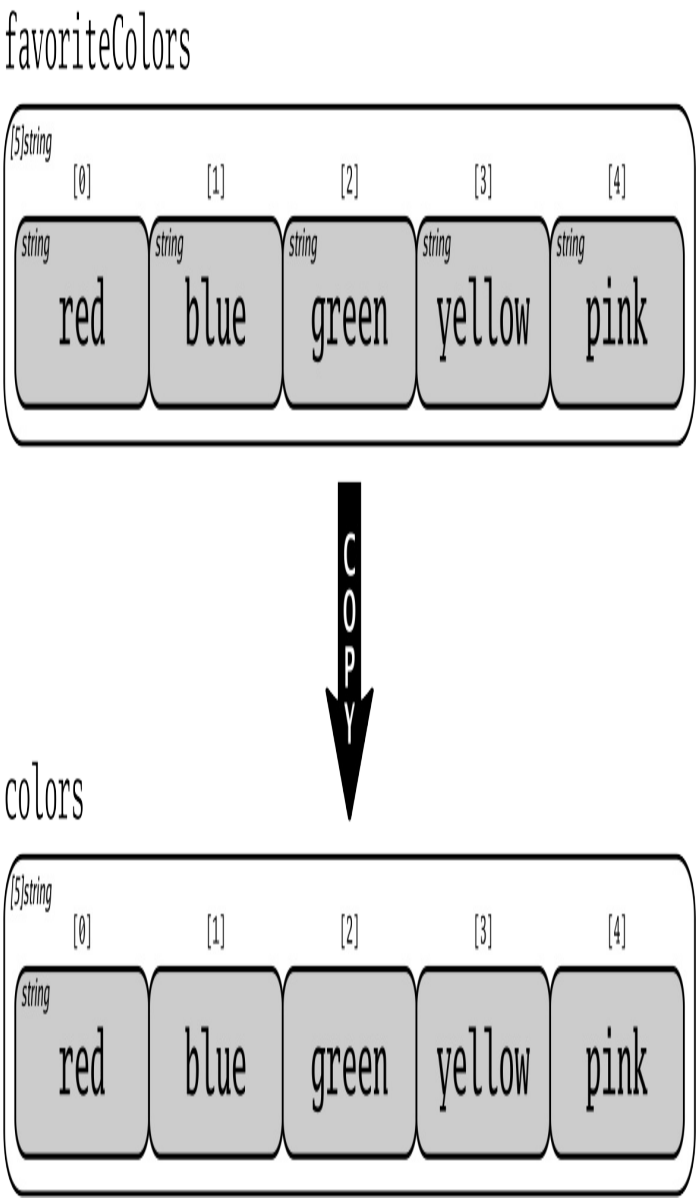
An array is a value in Go, which means you can assign array values of the same type to each other. The variable name denotes the entire array and, therefore, an array can be assigned to other arrays of the same type.

Listing 4.19. Assigning One Array To Another Of The Same Type

```
// Declare a string array of length five.
var colors [5]string

// Initialize another five element array with values.
favoriteColors := [5]string{"Red", "Blue", "Green", "Yellow", "Pi

// Assignment will copy the contents of the array.
colors = favoriteColors
```

It’s important to note that copying an array duplicates all of the contents and places them into the new array. After the line `colors = favoriteColors` you will have two separate arrays with two sets of identical values.

Figure 4.8. Array Assignment



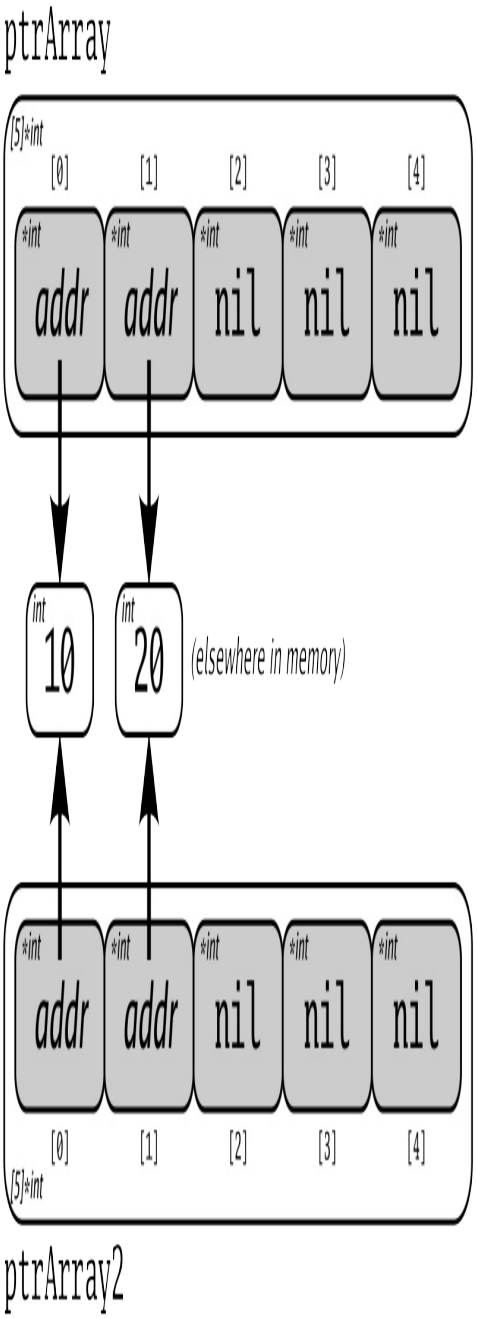
This applies to all arrays, regardless of the type. Even if the array contains pointer types like the one you saw earler, copies of those pointers will be created. Try adding an additional assignment to that example.

Listing 4.20. Listing 4.x With An Additional Assignment

```
ptrArray := [5]*int{0: new(int), 1: new(int)}
*ptrArray[0] = 10
*ptrArray[1] = 20

ptrArray2 := ptrArray
```

What you'll get in `ptrArray2` is still a separate array with separate pointer values. They just happen to point to the same memory.

Figure 4.9. Two Arrays Of Pointers Referencing The Same Objects



4.1.6 Multidimensional Arrays

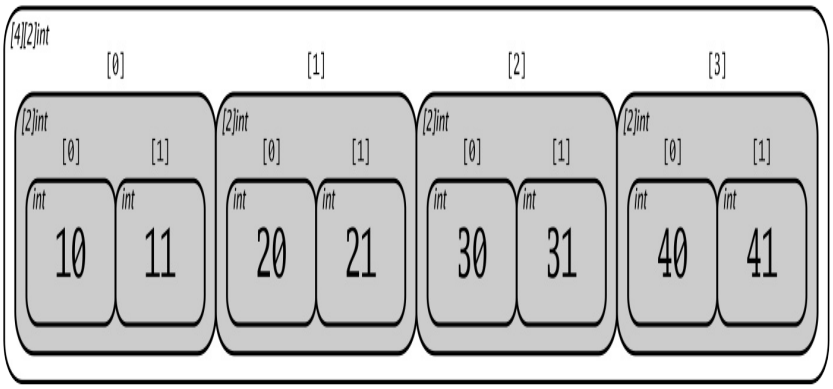
While Go does not have true multidimensional array support in the form of concise declaration or matrix math, you can simulate the access semantics by composing arrays together. This can be useful for representing tabular data or things like cordinate systems.

Listing 4.21. Declaring Two-dimensional Arrays

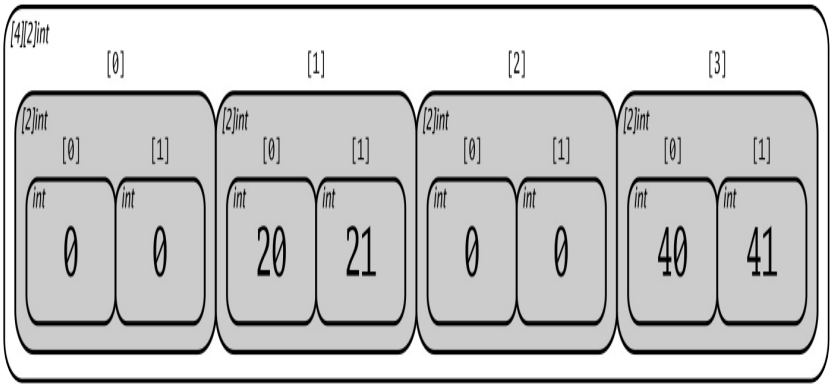
```
// Declare a two dimensional integer array of four elements by
//two elements.
var array [4][2]int

// Use an array literal to nitialize a two dimensional integer ar
array = [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}

// Declare and initialize index 1 and 3 of the outer array.
array = [4][2]int{1: {20, 21}, 3: {40, 41}}

// Declare and initialize individual elements of the outer
// and inner array.
array = [4][2]int{1: {0: 20}, 3: {1: 41}}
```

These declarations will result in the following data structures:

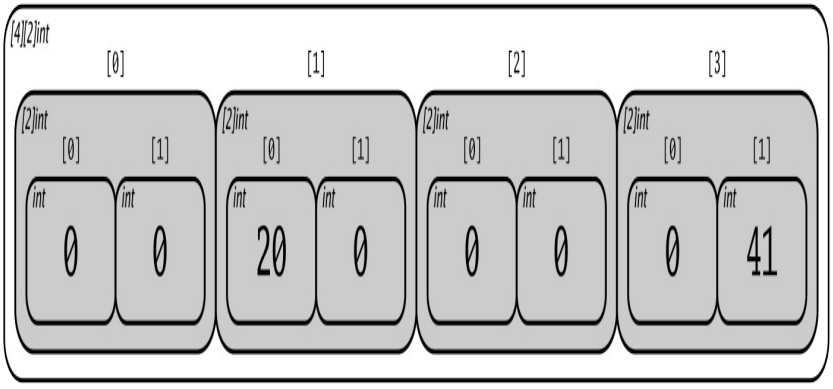
Figure 4.10. Two-dimensional Arrays And Their Outer and Inner Values




```
array = [4][2]int{{10, 11}, {20, 21}, {30, 31}, {40, 41}}
```




```
array = [4][2]int{1: {20, 21}, 3: {40, 41}}
```



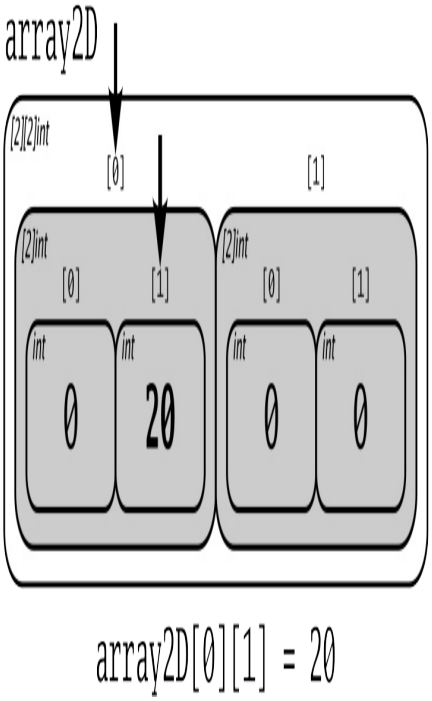

```
array = [4][2]int{1: {0: 20}, 3: {1: 41}}
```

Because these are just arrays of arrays, you can simply add another set of brackets to access the inner values:

Listing 4.22. Accessing Elements Of A Two-Dimensional Array

```
// Declare a two dimensional integer array of two elements.
var array2D [2][2]int
// Set integer values to each individual element.
array2D[0][0] = 10
array2D[0][1] = 20
array2D[1][0] = 30
array2D[1][1] = 40
```

Figure 4.11. Assigning An Element Of A Two-Dimensional Array



As with other arrays, you can copy multimiensional arrays into each other, so long as they have the same element type and the lengths are the same all the way down.

Listing 4.23. Assigning Multidimensional Arrays Of The Same Type

```
// Declare two different two dimensional integer arrays.
```

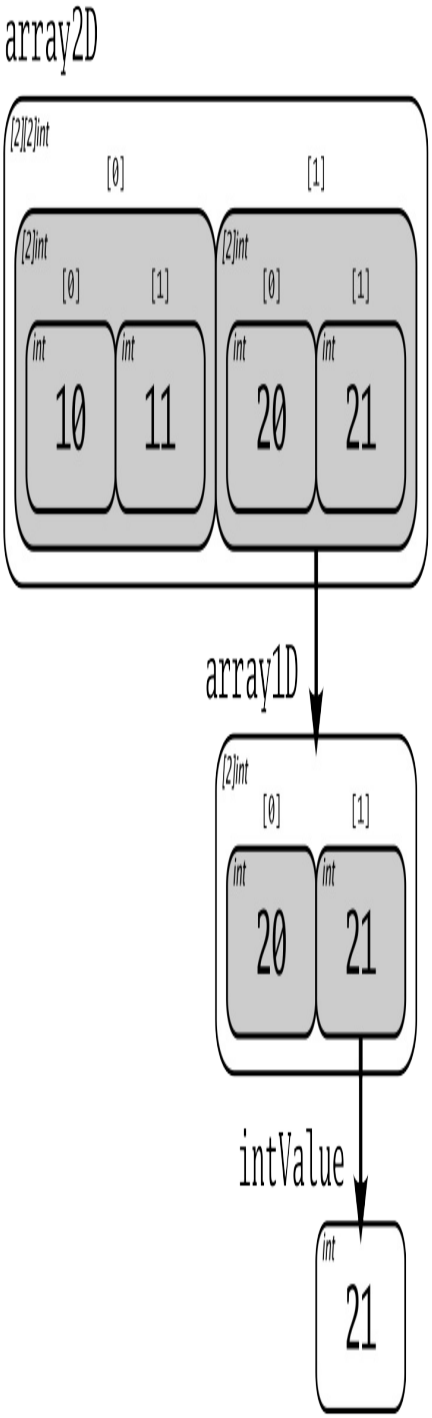
```
var array1 [2][2]int
var array2 [2][2]int
// Add integer values to each individual element.
array2[0][0] = 10
array2[0][1] = 20
array2[1][0] = 30
array2[1][1] = 40
// Copy the values from array2 into array1.
array1 = array2
```

Because each array is a value, you extract individual sub-arrays or values, depending on how many levels you access:

Listing 4.24. Copying A Sub-array And Value

```
array2D := [2][2]int{{10, 11}, {20, 21}}

array1D := array2D[1] // [20 21]

intValue := array1D[0] // 20
```

Figure 4.12. Values At Different Levels



What Is The Type Of A Collection?

Every variable in Go has one --and only one-- type. This is easy to see with simple variables like `var count int`, since it reads naturally as "variable count is an integer", but it can be harder to intuit with collections like `var`

`coordinates [4][2]int`, since it is temping to think of collections only in terms of the items they contain. A helpful trick can be to use the `%T` print verb with `fmt.Printf`. This will tell you the Go type of a variable:

Listing 4.25. Printing The Type Of A Variable

```
var array3D [4][3][2]string
fmt.Printf("array3D is a %T\n", array3D)
```

This will output `array3D is a [4][3][2]string`, which you can read as "array3D is a length-four array of length-three arrays of length-two arrays of strings".

4.1.7 Passing Arrays To Functions

Passing an array value to a function can be a an expensive operation in terms of performance and memory usage if you're not careful. This is because all values in Go are passed by value, so every time you pass an array to a function, your program will copy the entire array and pass it to the function, regardless of how large it is.

For example, let's say you have a large array representing a 4K image, and you want to pass those to a function for processing. That's 3840 x 2160, which is around 8.2 million pixels.

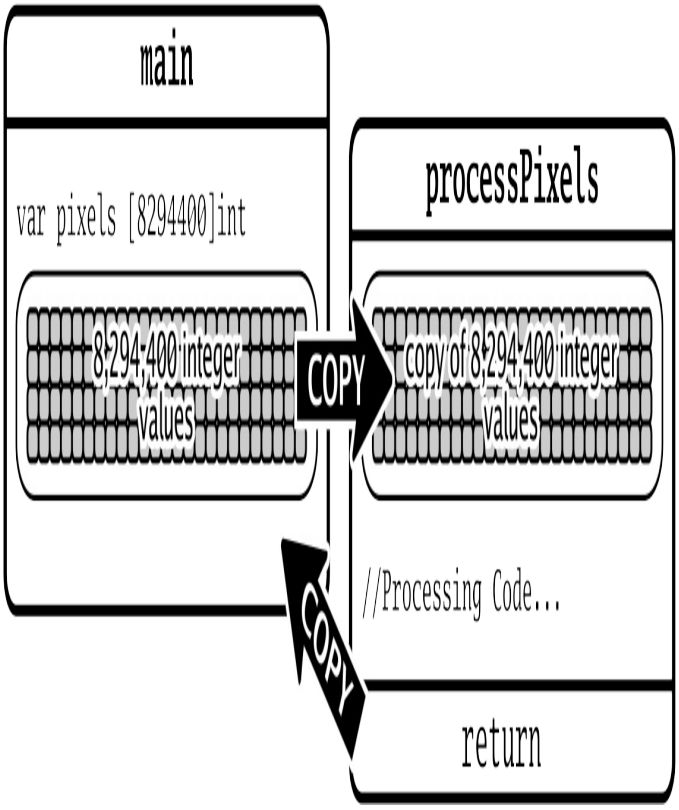
Listing 4.26. Passing A Large Array By Value To A Function

```
// image4K is a large array representing a 4K image #1
type image4K [829400]int // 3840 * 2160

func processPixels(pixels image4K) image4K {
        // Image Processing Code Here.
        return pixels
}

func main() {
        var pixels image4K
        pixels = processPixels(pixels)
}
```

Every time `processPixels` is called, 8.2 million new integer values will be

be allocated, and all of the values in the `pixels` array will be copied into that allocation. On return, the values will be copied again.

Figure 4.13. Passing A Large Array By Value



Go can handle this copy operation for you, but there's a better and more efficient way of doing this, which is to pass a pointer.

Listing 4.27. Passing A Large Array By Pointer To A Function

```
func processPixels(pixels *image4K) {
        // Processing code...
}

func main() {
        var pixels image4K

        processPixels(&pixels)
}
```

This time `processPixels` takes a pointer to an array, which only uses about 8 bytes of memory, as opposed to the roughly 66 megabytes of it would take to copy 8.2 million integers.

This operation is much more efficient with memory and will yield better performance, though you need to keep in mind that any changes to the array will be reflected in the original value.

4.1.8 The Problem With Arrays

While they are useful for certain applications requiring a fixed number of items, arrays are actually a very poor general collection type.

Because the size is fixed, there is no way for them to grow or shrink in response to different circmustances, which makes them difficult to use in real-world situations where the number of items in a list is not always known ahead of time. Arrays will always take up the same amount of memory which is determined when the program is compiled, which means that if you use less than the full amount you will be wasting memory, and if you need more space, you will need to rebuild your program with a larger array size.

Arrays also take extra care to use when sharing their contents, since, as you've just learned, they are copied every time they pass from function to function.

Slices address both of these problems in an elegant way, and allow you to work with lists much more effectively.

[9] Note that while index arithmetic works in Go, pointer arithmetic is not allowed, so indices are the only way to access elements in an array or slice.

4.2 Slices

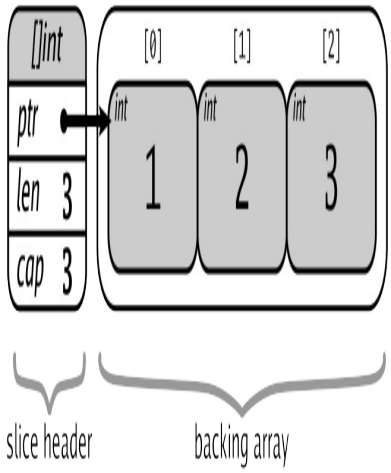
Slices are Go's primary ordered data type, and can be used in all of the same ways you would use an array, with additional dynamic features that make them much more flexible and memory efficient. On first glance, slices a lot like arrays, except that they do not specify a size:

Listing 4.28. An Example Slice

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

value := slice[0] // 1

for i := range slice {
        fmt.Println(slice[i])
}
// output:
// 1
// 2
// 3
// 4
```

However, if you pull back the curtain and look at their representation in memory, slices start to get a bit more interesting:

Figure 4.14. Slice Internals



From this diagram, you can see that slices are comprised of two parts: an array used for storage (called the *backing array*), and the slice data structure itself, also called the *slice header*. The slice header is the data that is actually stored in the slice variable, and represents a dynamic view, or "slice", of an array, which is where the data structure gets its name.

The slice header contains just three values: a pointer to the memory location where the slice begins, an `int` representing the number of elements in the

slice (the length), and an `int` that specifies the total number of items available for storage or expansion (the capacity). This simple structure is what gives slices their flexibility and what makes them so powerful. By separating the storage of an array from its representation, slices provide a collection type that works like an array, but can grow or shrink as needed, and be efficiently copied without needing to copy all of the values in the array. It's a really neat trick!

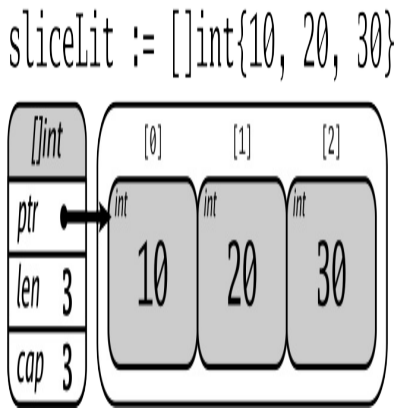
4.2.1 Declaring Slices With Slice Literals

Using slice literals is the most straightforward way to create a new slice, and looks exactly like using an array literal without a size:

Listing 4.29. Declaring And Initializing A Slice With A Literal

```
// Declare and initialize an integer slice with a literal.
sliceLit := []int{10, 20, 30}
```

Figure 4.15. Slice Variable Declared And Initialized With A Literal



Slices declared in this way will be allocated with a new backing array containing the items provided, and will point to the first element of that array. The length and capacity will also be set to the number of values provided, which you can inspect using the built-in `len` and `cap` functions:

Listing 4.30. Getting The Length And Capacity Of A Slice

```
// Get the length and capacity of newly-created slice.
```

```
sliceLitLen := len(sliceLit) // int(3)
sliceLitCap := cap(sliceLit) // int(3)
```

4.2.2 Declaring Slices With `make`

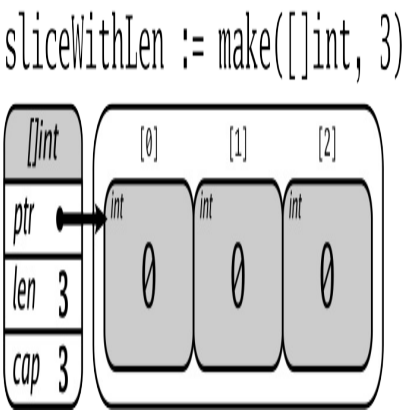
You can also create new slices using the built-in `make` function, which returns a new slice with all of the values set to the zero value for its element type. Calls to `make` take the form of `make([]T, length)` or `make([]T, length, capacity)`, where `T` is the element type of the slice and `length` and `capacity` are integers defining the number of values in your slice and extra capacity for expansion. Specifying a capacity allows you to pre-allocate storage for a slice ahead of time which allows it to grow without needing to create a new backing array. This is be helpful in cases where you will be expanding your slice with the `append` call, which is how slices grow in Go.

If `make` is called with only a length, it will return a slice pointing to a backing array of exactly that size, with all values zeroed-out.

Listing 4.31. Declaring A Slice With `make` And A Length

```
// Declare and initialize an integer slice with length 3
sliceWithLen := make([]int, 3) // identical to []int{0, 0, 0}
fmt.Println(len(sliceWithLen)) // 3
```

Figure 4.16. Slice Initialized With `make` And A Length

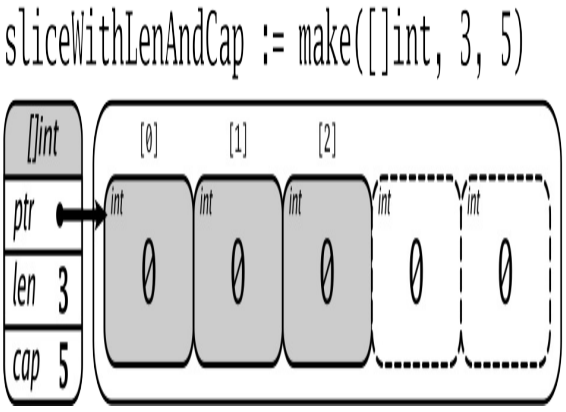


If `make` is given a capacity, it will allocate extra storage beyond the length value for use in further expansion.

Listing 4.32. Declaring A Slice With `make` And Both Length And Capacity

```
// Declare and initialize an integer slice with length 3, and cap
sliceWithLenAndCap := make([]int, 3, 5)
fmt.Println(len(sliceWithLenAndCap)) // 3
```

Figure 4.17. Slice Initialized With A Length And A Larger Capacity



Here the dashed values in white represent items in the backing array that have been allocated, but are currently inaccessible from within the slice, since they are reserved for expansion.

 **Important**

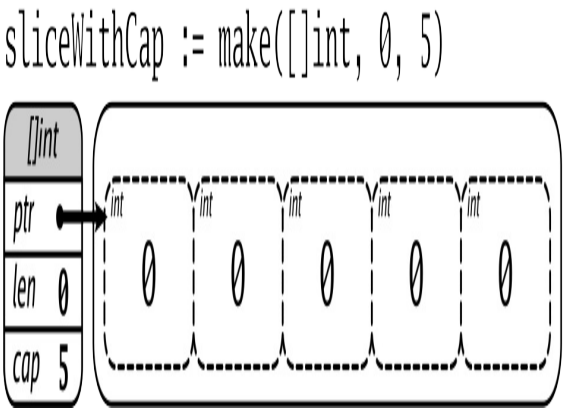
Capacity must be greater than or equal to length. If the compiler can catch this for you, it will throw an error, otherwise, you will get a runtime panic.

If the slice is declared with a capacity greater than zero but a length of 0, the slice will appear to be empty, however it will be backed by an array of the specified capacity size:

Listing 4.33. Declaring A Slice With `make` And Capacity, But Zero Length

```
// Declare and initialize an integer slice with length 0, and cap
sliceWithCap := make([]int, 0, 5)
fmt.Println(len(sliceWithCap)) // 0
```

Figure 4.18. Slice Initialized With Length 0 And A Larger Capacity



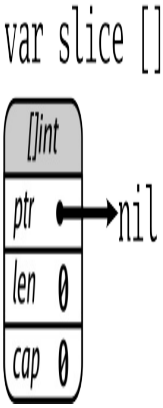
4.2.3 Nil Slices

Slices declared with `var` are uninitialized, and are known as *nil slices*, because they point to `nil` instead of a backing array. They will still return values for `len` and `cap` (both `0`), but are considered to be equal to `nil`, and attempts to access their values will cause a runtime panic, since they don't point to anything yet:

Listing 4.34. Slice Variable Declaration with `var`

```
// Declare an integer slice variable.
var sliceVar []int

fmt.Println(len(sliceVar), cap(sliceVar)) // 0 0
fmt.Println(sliceVar == nil) // output: true

value := sliceVar[0] // panic: index out of range [0] with length
```

Figure 4.19. A Nil-slice



Nil slices can be useful as a placeholder if the slice is optional (for example as part of a struct), or you plan to assign a value later through other means, however it's important to ensure that they are allocated before you attempt to access values from them. This can be done with literals or `make`, as above, or with an assignment using `append` or slice expressions.

Important

Slices declared with `var` are not usable until they have a value assigned.

4.2.4 Growing Slices With `append`

One of the benefits to the way slices are implemented is that they can grow by increasing their length if capacity is available or allocate new storage if it is not. This is done with the built-in `append` function.

Listing 4.35. The Append Function

```
func append(slice []Type, elems ...Type) []Type
```

The `append` function works by taking an existing slice and one or more new values as arguments, and returning a modified slice with the new values added. It examines the input slice and will allocate storage as necessary to grow the slice. This makes it safe to call on any slice, even those that have not been initialized yet:

Listing 4.36. Using Append To Add An Element To A Slice

```
// Initialize a slice of integers with three values.
intSlice := []int{10, 20, 30}
// Assign the slice to the appended version of itself.
intSlice = append(intSlice, 40)

fmt.Println(intSlice)
// output: [10 20 30 40]

var sliceVar []string // nil slice
sliceVar := append(sliceVar, "This works!")
fmt.Println(sliceVar)
```

```
// output: ["This works!"]
```

Using `append` is easy and fairly straightforward, but it helps to know a bit more about how it works to understand how it can change the underlying array. Let's take a look at the `intSlice` example again, only this time with some extra print statements to see what happens to the length and capacity:

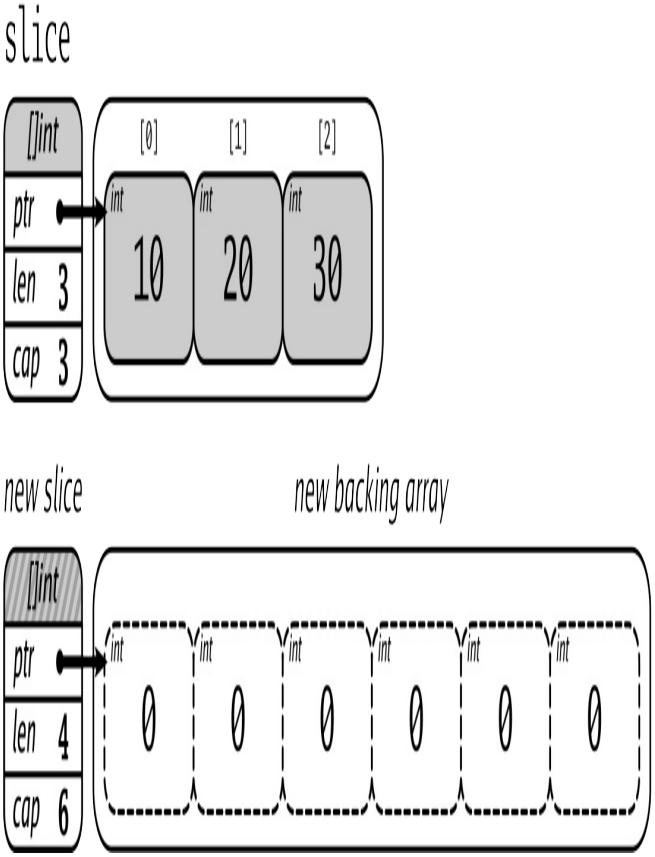
Listing 4.37. The Affect Of `append` On Length And Capacity

```
intSlice := []int{10, 20, 30}
fmt.Printf("len: %d, cap: %d\n", len(intSlice), cap(intSlice))
// output: len 3, cap: 3

intSlice = append(intSlice, 40)
fmt.Printf("len: %d, cap: %d\n", len(intSlice), cap(intSlice))
// output: len: 4, cap: 6
```

From the output you can see that the capacity of the slice starts out at `3`, which means that the underlying array only has the space to represent three values, and will need to grow in order to accomodate more. After the `append` the length of the slice is `4`, which makes sense since one element has been added, but the capacity is now `6`, which is twice that of the original slice. This hints that the underlying array has increased in size. To see how this works, let's take a look at what happens during the call to `append`:

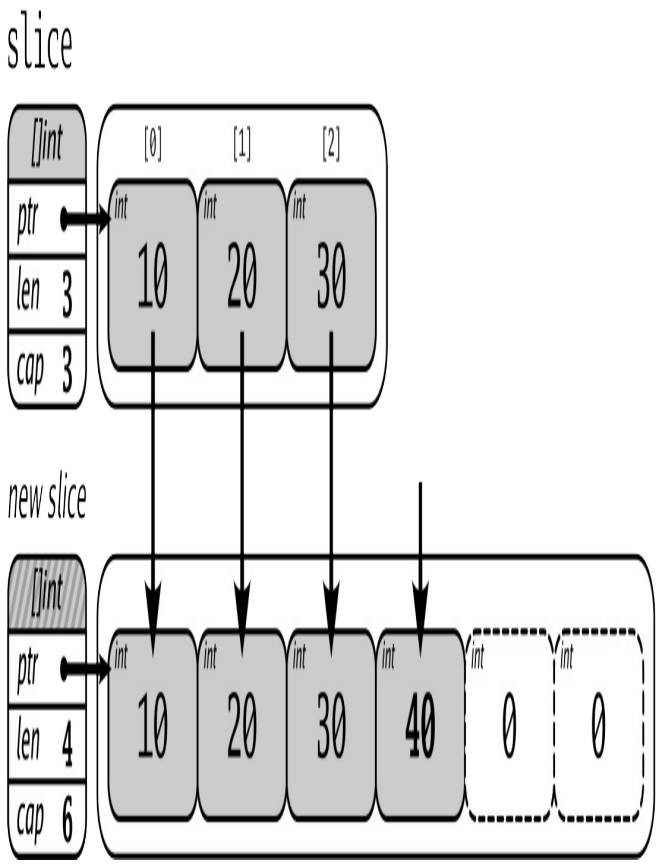
First, a new, larger backing array is allocated to hold the new slice values.

Figure 4.20. Append With Allocation Step 1: Allocate New Slice With Storage



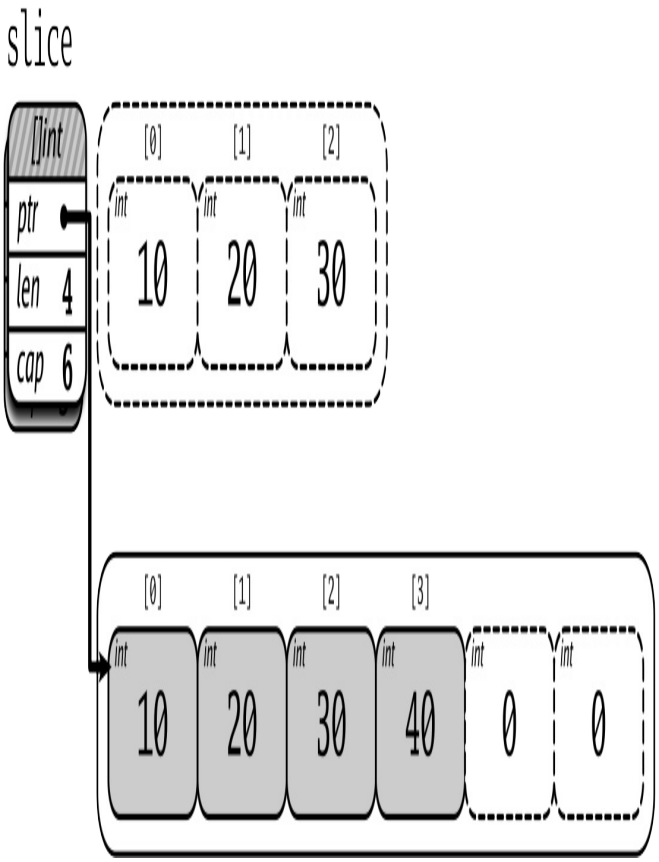
Next, the old values are copied into the new array, along with any new values to be appended:

Figure 4.21. Append With Allocation Step 2: Copy Old Values And Assigne New Ones



Finally, the new slice header is returned, and the assignment overwrites the old one, releasing the memory held by the old array.

Figure 4.22. Append With Allocation Step 3: Return The New Slice To Overwrite The Old One



The reason the new slice has a larger capacity than it needs is because copying values is an expensive operation, and doing it for every append would be wasteful, so Go pre-allocates extra memory every time a slice needs to allocate new storage, so that, over time, the cost incurred by copying doesn't happen as much.[10]

Keeping this behavior in mind, what do you think will happen to the capacity if you append to this same slice again?

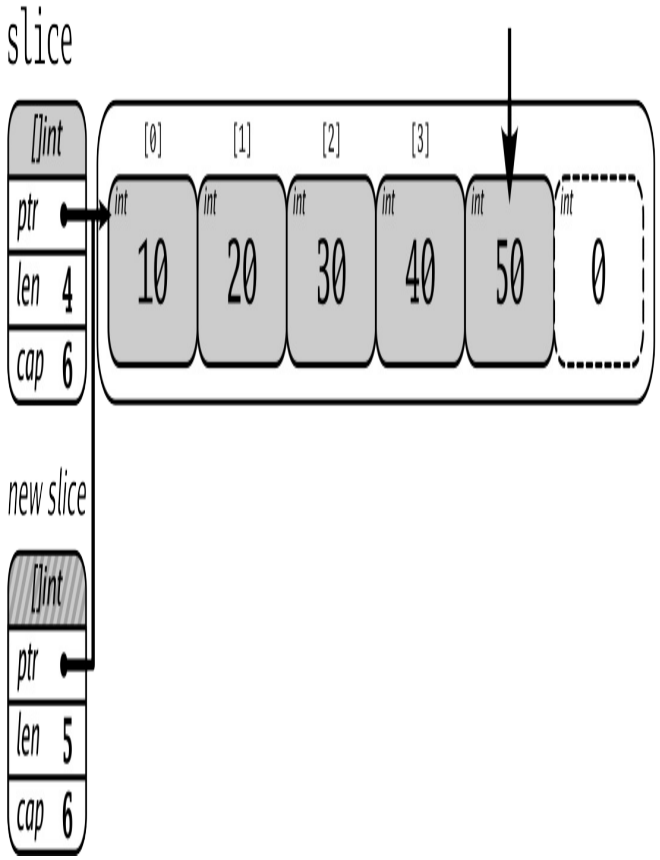
Listing 4.38. Appending To A Slice That Has Capacity

```
slice = append(slice, 50)
fmt.Println(slice)
// output: [10 20 30 40 50]
fmt.Printf("len: %d, cap: %d\n", len(slice), cap(slice))
// output: len: 5, cap: 6
```

The length has now increased, but the capacity has stayed the same. This is because the previous allocation left enough extra capacity to hold the new

value without a reallocation.

Figure 4.23. Append Without Allocation



This approach makes appending much less expensive over time, since the extra capacity means that the slice won't need to be copied again until the new storage is used up.

How Much Will append Grow The Capacity?

The append operation is clever when growing the capacity of the backing array. As of this printing, the algoritm will always double the existing capacity if the total number of elements less than 1,024. Past this number, the capacity is grown by a factor of .25, or 25%, though this could change in future versions of the language as more optimizations are added.

Multiple values can be appended by simply adding more arguements or, if you are appending one slice to another, you can unpack the second slice with

the special … operator, which will transform a slice into a list of arguments.

Listing 4.39. Using Append To Add Multiple Values To A Slice

```
slice := []int{10, 20, 30}
slice := append(slice, 40, 50)
fmt.Println(slice)
// output: [10 20 30 40 50]

slice2 := []int{60, 70}
slice := append(slice, slice2...) // The same as append(slice, 60
fmt.Println(slice)
// output: [10 20 30 40 50 60 70]
```

Why Does Append Return A New Slice?

It might look a little strange to append to slices with an assignment instead of a built-in method, but this approach is actually more flexible, since it allows you to choose whether or not you want to modify a slice by inserting new values or simply create a new slice with extra elements.

4.2.5 Avoiding Append Surprises

Because `append` will sometimes allocate new storage, it can sometimes be a source of confusion to new Go programmers who expect to be able to see changes to their slice after every append, but things can get a little tricky when slices cross function boundaries. Normally, values in a slice can be modified by functions, like the following, which adds 1 to every slice element.

Listing 4.40. Modifying A Slice With A Function

```
// addOneToAll adds 1 to every slice value.
func addOneToAll(s []int) {
        for i := range s{
                s[i]+=1
        }
}

func main(){
        slice := []int{1,2,3}
```

```
        addOneToAll(slice)
        fmt.Println(slice)
}
```

Listing 4.41. Output After `addOneToAll`

```
[2 3 4]
```

This function works as intended, but if an append is added the results are different:

Listing 4.42. Adding An Append To `addOneToAll`

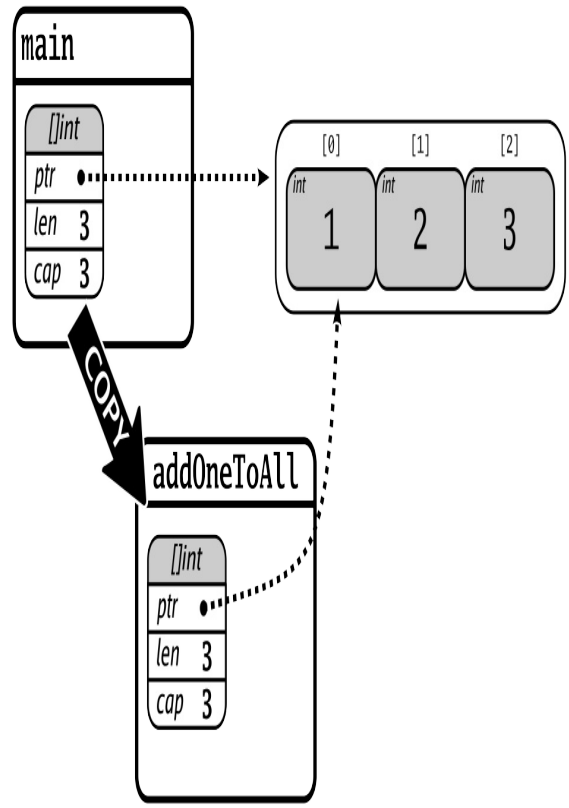
```
// addOneToAll grows the slice and adds 1 to every slice
// value.
func addOneToAll(s []int) {
        s = append(s, 0)
        for i := range s {
                s[i] += 1
        }
}

func main(){
        slice := []int{1,2,3}
        addOneToAll(slice)
        fmt.Println(slice)
}
```

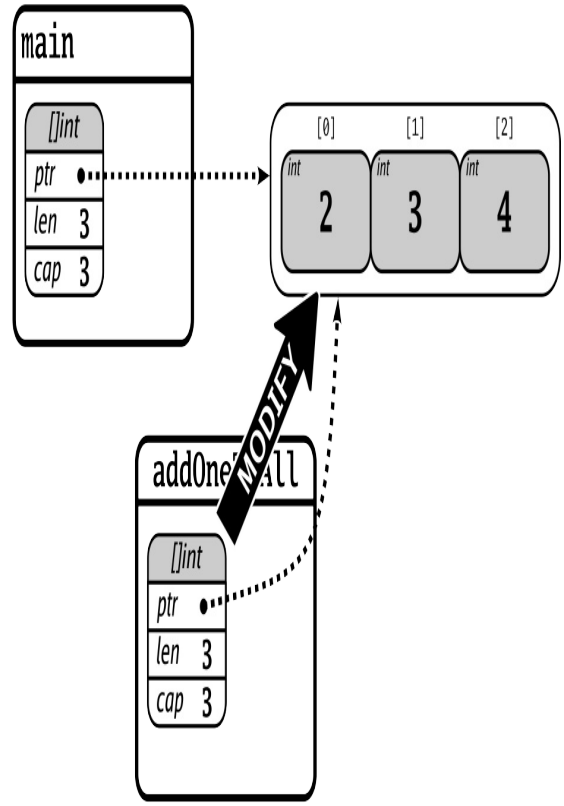
Listing 4.43. Output After `addOneToAll` With An Append

```
[1 2 3]
```

What happened to the changes? In fact they were made, only it was to a completely different slice! In the original, both slices pointed to the same memory, so after the function is complete, the slice in `main` points to the new values.

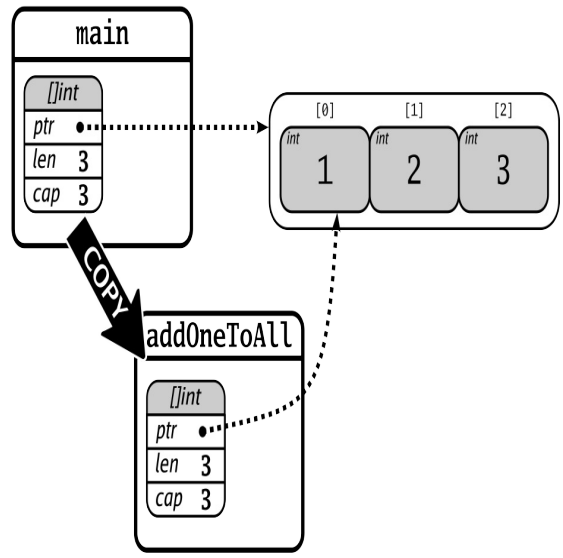
Figure 4.24. `addOneToAll` Modifying The Same Values In Memory

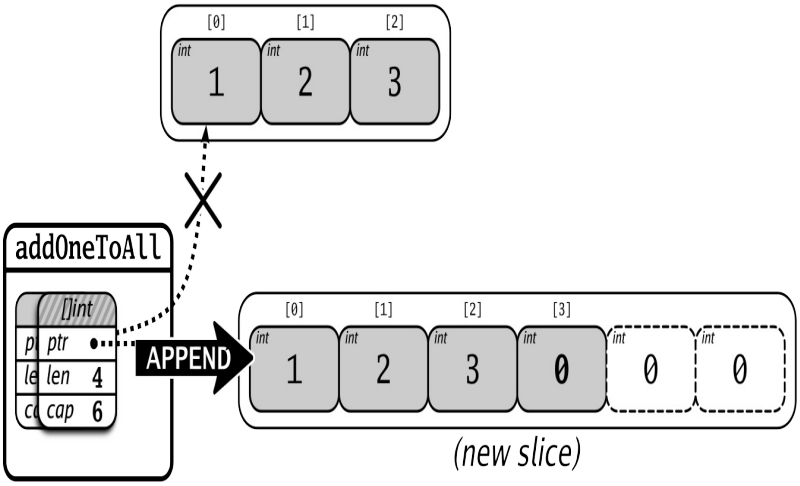

main
[]int
ptr
len 3
cap 3
[0]
[1]
[2]
int
1
int
2
int
3
COPY
addOneToAll
[]int
ptr
len 3
cap 3

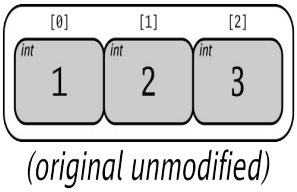

main
[]int
ptr
len 3
cap 3
[0]
[1]
[2]
int
2
int
3
int
4
MODIFY
[]int
ptr
len 3
cap 3

However, once an `append` is added, a new backing array is allocated, and the two slice variables no longer point to the same memory. `addOneToAll` is making its changes to a slice that only exists within that function.

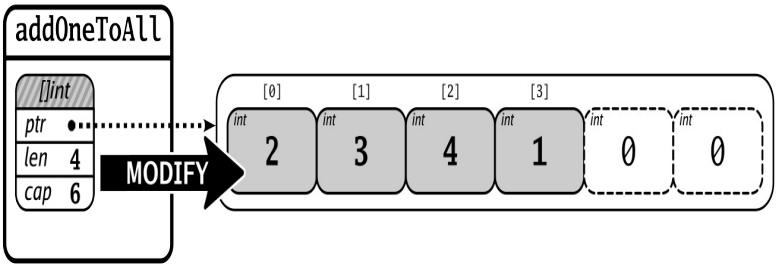
Figure 4.25. `addOneToAll` Modifying A New Slice After Append







(original unmodified)



You can try and get around this by adding extra capacity in `main` before calling the function, but this doesn’t quite work either:

Listing 4.44. Attempting To Avoid The Append Allocation With Extra Capacity

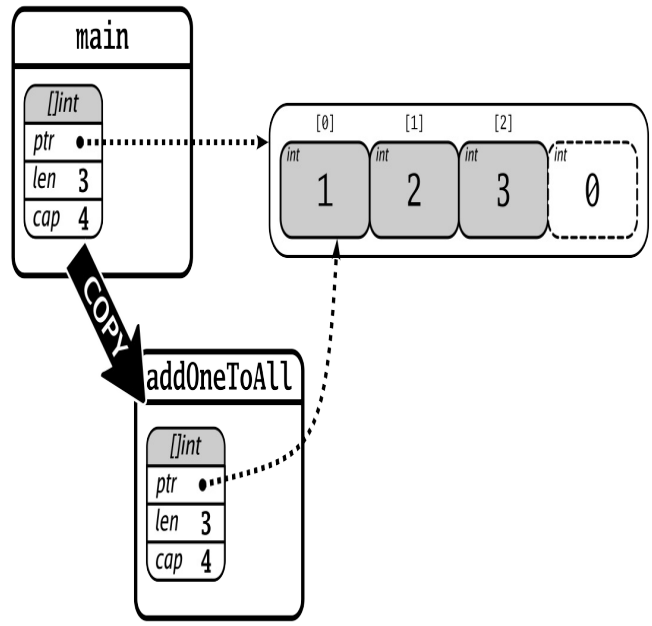
```
func main(){
        // Create A Slice With Extra Capacity
        slice := make([]int, 0, 4)
        slice[0] = 1
        slice[1] = 2
        slice[2] = 3

        addOneToAll(slice)
        fmt.Println(slice)
}
```

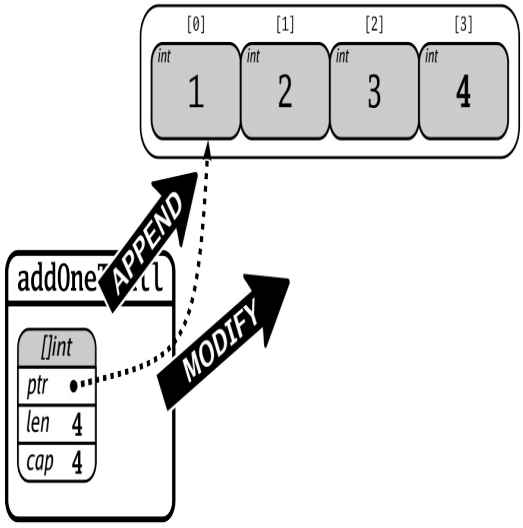
Listing 4.45. Output After `addOneToAll` With Extra Capacity

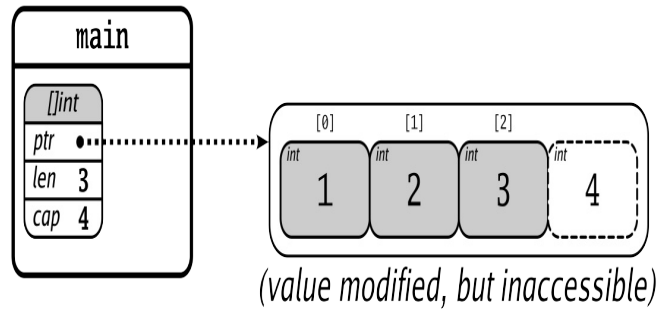
```
[2 3 4]
```

The values were incremented, but the final value is missing. This is because the length of the original slice in `main` remains the same, meaning it cannot "see" the new value.

Figure 4.26. `addOneToAll` Appending And Modifying, But New Value Is Inaccessible







(value modified, but inaccessible)

The right thing to do in this case is to either append to slices before passing them on, or to return the appended slice from the function, similar to how `append` works:

Listing 4.46. Returning An Appended Slice From `addOneToAll`

```
func addOneToAll(s []int) []int {
        s = append(s, 0)
        for i := range s {
                s[i] += 1
        }
        return s
}

func main() {
        slice := []int{1, 2, 3}
        slice = addOneToAll(slice)
        fmt.Println(slice)
        // output: [2 3 4 1]
}
```

Listing 4.47. Output After `addOneToAll` With Returned Slice

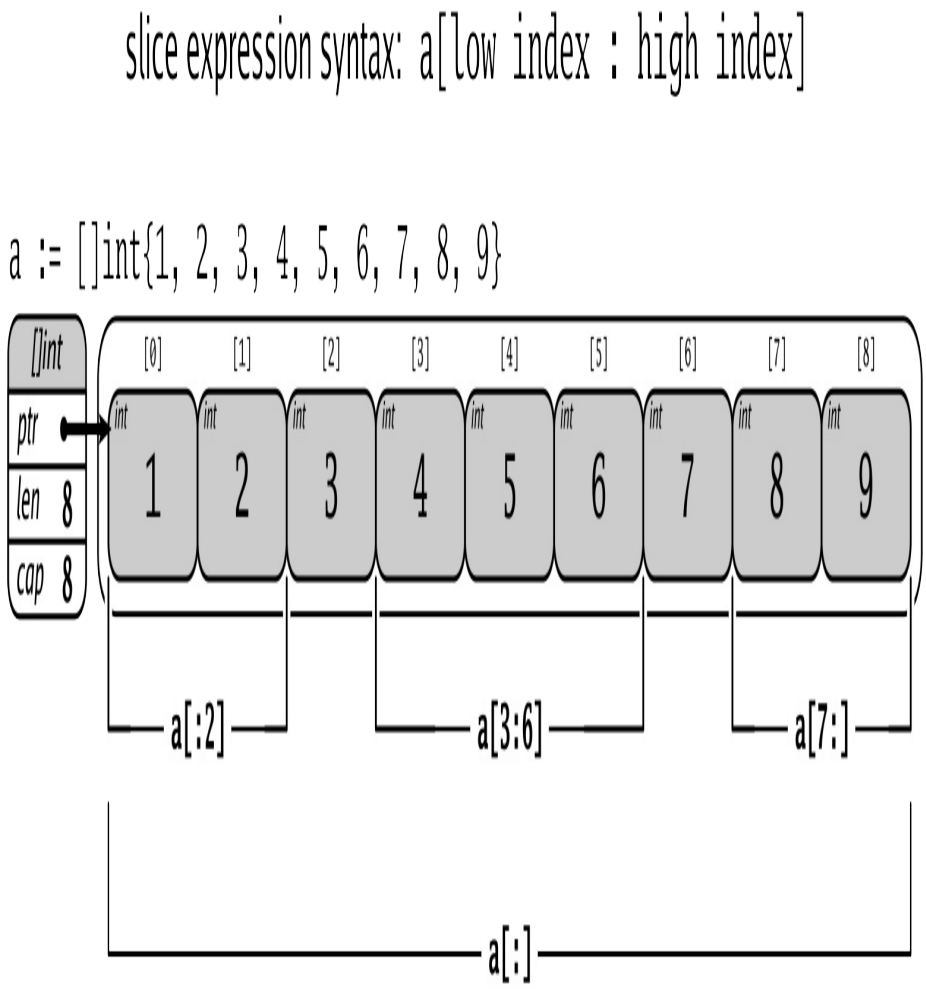
```
[2 3 4 1]
```

4.2.6 Slice Expressions

One of the other things that makes slices so flexible is that they can be created from sections of other slices or arrays while still sharing the same storage. This is done with *slice expressions* which allow you to specify a range of values from a slice, creating a window into the existing slice.

Slice expressions take the form of `a[low : high]`, where `a` is an existing slice or array, and `low` and `high` are indices representing the range of values you want in the new slice. You can leave out one or both of these values, and they will default to `0`, and `len(a)`, respectively, making it easy to specify ranges "starting from" or "up to" a particular index.

Figure 4.27. Slice Expression Syntax Illustrated

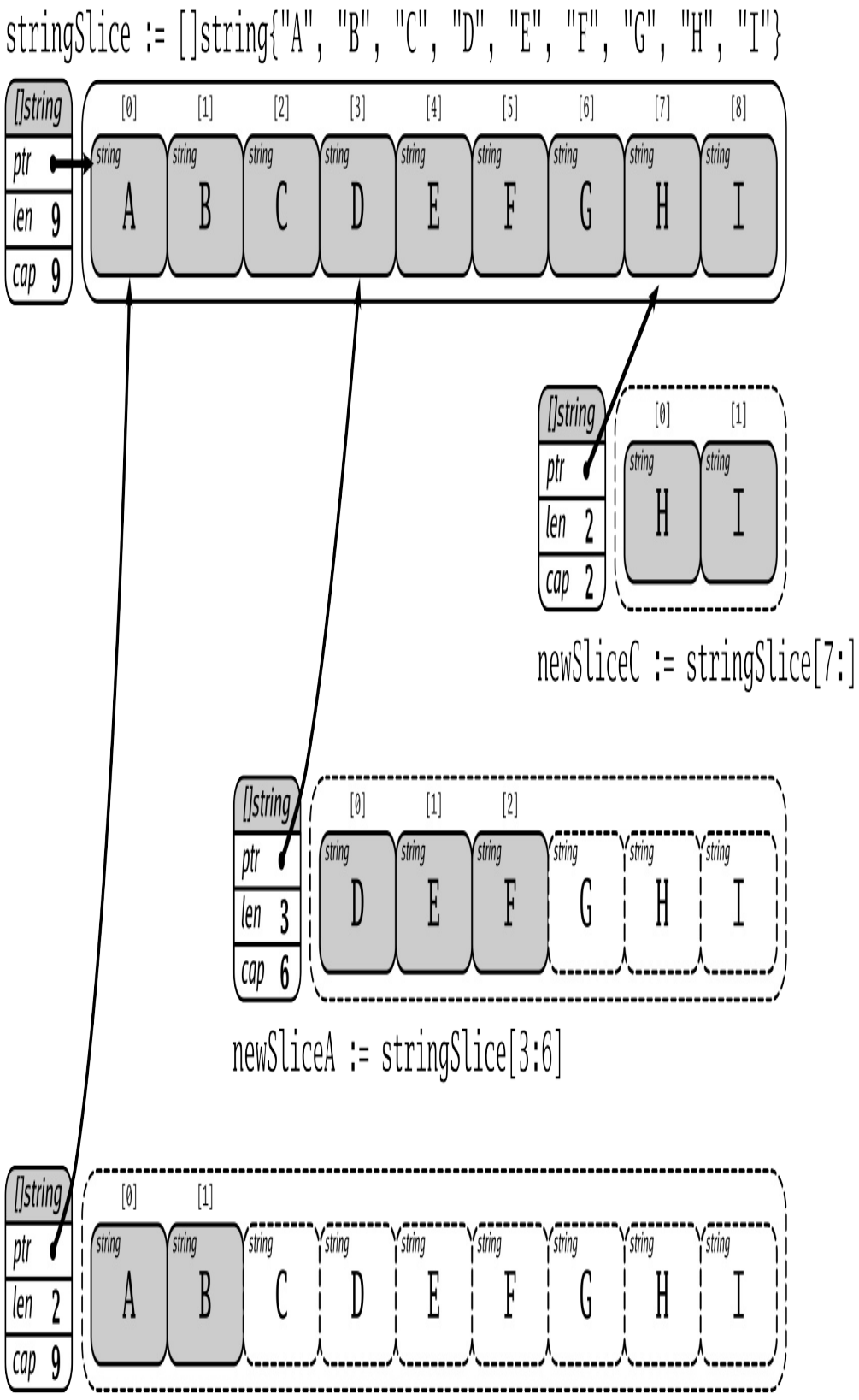


When used with a slice, a slice expression will create a new slice, pointing to the `low` index in the backing array with a length value set based on the `high` value.

Listing 4.48. Creating Slices From An Existing Slice

```
stringSlice := []string{"A", "B", "C", "D", "E", "F", "G", "H", "

newSliceA := stringSlice[3:6] // [D E F]
newSliceB := stringSlice[:2]  // [A B] - "up to" index 2
newSliceC := stringSlice[7:]  // [H I] - "starting from" index 7
```

Figure 4.28. New Slices Created With Slice Expressions

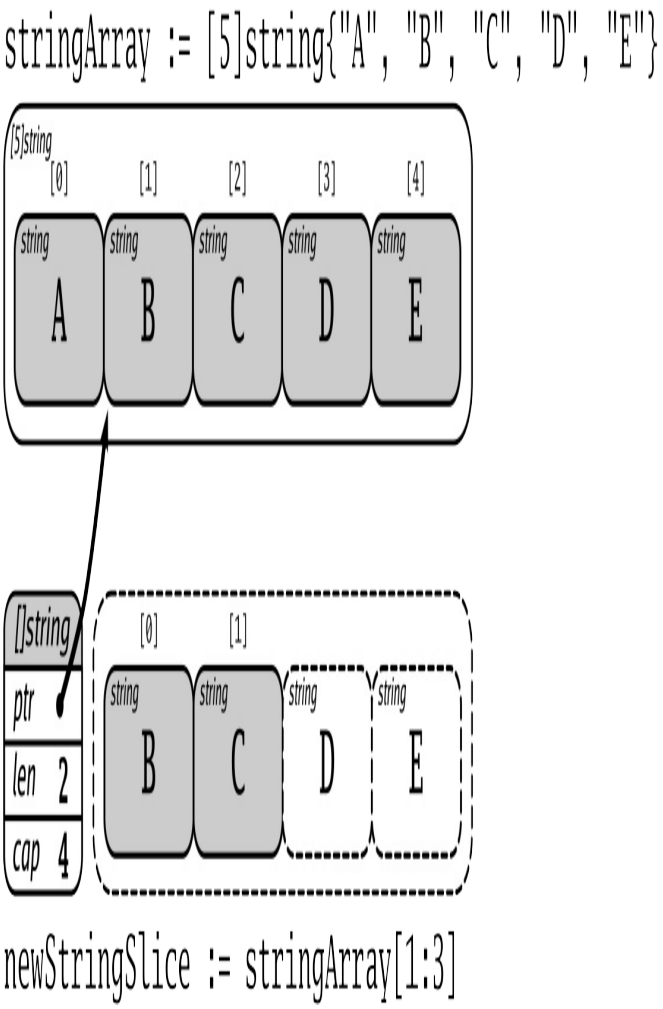

stringSlice := []string{"A", "B", "C", "D", "E", "F", "G", "H", "I"}
[]string
ptr
len 9
cap 9
[0] [1] [2] [3] [4] [5] [6] [7] [8]
string A
string B
string C
string D
string E
string F
string G
string H
string I
[]string
ptr
len 2
cap 2
[0] [1]
string H
string I
newSliceC := stringSlice[7:]
[]string
ptr
len 3
cap 6
[0] [1] [2]
string D
string E
string F
string G
string H
string I
newSliceA := stringSlice[3:6]
[]string
ptr
len 2
cap 9
[0] [1]
string A
string B
string C
string D
string E
string F
string G
string H
string I
newSliceB := stringSlice[:2]

When used with an array, a slice expression will create a new slice using the array for its storage:

Listing 4.49. Creating A Slice From An Array.

```
stringArray := [5]string{"A", "B", "C", "D", "E"}
newStringSlice := stringArray[1:3] // [B C]
```

Figure 4.29. New Slice Created From An Array



Slice Expressions With Capacity

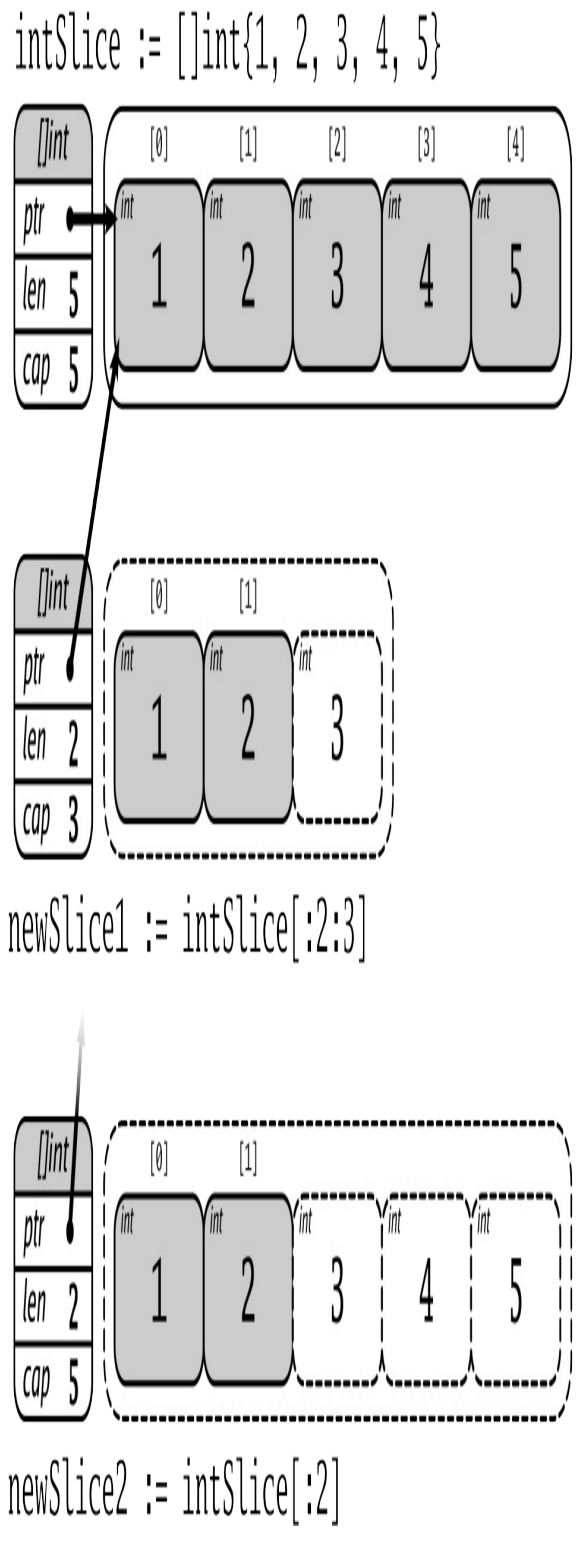
By default, slices created with slice expressions will inherit the remaining

portion of the slice or array they came from as capacity. Normally this is not an issue, but under certain circumsances, you might wish to restrict the capacity of the new slice so that it cannot be appended to past a certain point. For this, you can use a slice expression with three indices provided in the form of `a[low : high : max]`. In this form, the capacity of the new slice will cut off at the `max` index, preventing any modification of the original slice beyond this point.

Listing 4.50. Slice Expressions With And Without A `max` Value

```
intSlice = []int{1, 2, 3, 4, 5}
newSliceA = intSlice[:2]   // [1 2]
newSliceB = intSlice[:2:3] // [1 2] (same values, less capacity)
```

Figure 4.30. New Slices Created With And Without A `max` Value



Slice Expressions In Action

Slice Expressions can be useful for passing only a portion of a slice to a function that operates on slices. For example, let's say that you want to write a function to average the values in a slice of `float64` values. If you write it using `range` and `len`, you can get meaningful values from slices of any size.

Listing 4.51. Function To Return The Average Value Of A Slice Of `float64`

```
// avg returns the average of a []float64.
func avg(fs []float64) float64 {
        var sum float64
        // Add each value to the sum.
        for i := range fs {
                sum += fs[i]
        }
        // Divide the sum by the length of the slice.
        return sum /float64(len(fs))
}
```

If you need to create an average from a large slice, of values, like metrics or sensor readings, you can choose to extract only a subset of values to pass to the function:

Listing 4.52. Averagigng Only The First 10 Entries In A Slice Using A Slice Expression

```
func getMeasurements() []float64 {
        // Simulate a large slice with 10 values and a bunch of z
        return []float64{3.3, 1.5, 3.5, 2.9, 2.7, 3.0, 3.1, 4.5,
}

func main(){
        measurements := getMeasurements()
        fmt.Printf("averaging 10 out of of %d\n measurements", le
        averageMeasurement := avg(measurements[:10])
        fmt.Printf("average measurement is: %f\n", averageMeasure
}
```

Listing 4.53. Output From Averaging A Subset Of A Slice

```
averaging 10 out of of 100 measurements
average measurement is: 3.090000
```

4.2.7 Copying Slices with `copy`

While `append` and slice expressions provide a fast and flexible way to create new slices and save on allocations, sometimes you will want to duplicate the contents of a slice into a new slice, and this is what the built-in `copy` function is for.

Listing 4.54. The Copy Function

```
func copy(dst, src []Type) int
```

The `copy` function takes two slices of the same type and copies the elements of `src` into `dst`, returning the number of elements copied. The one thing to keep in mind when using `copy` is that, when the lengths of slices are different, it will only use the shorter of the two. If the source slice is shorter, `copy` will copy all of the the elements from the source slice, leaving the rest of the elements in the destination slice untouched. If the destination slice is shorter, `copy` will only copy enough values to fill the slice. This often trips people up, since, unlike `append`, `copy` will not do any allocations for you; if either of the slices has a length of `0`, nothing will be copied.

Listing 4.55. Using `copy` On Slices Of Different Lengths

```
sliceLen6 := []int{0, 0, 0, 0, 0, 0}
sliceLen4 := []int{1, 1, 1, 1}
sliceLen2 := []int{2, 2, 2}
sliceLen0 := []int{}

copy(sliceLen6, sliceLen4)
fmt.Println(sliceLen6) // output: [1 1 1 1 0 0]

copy(sliceLen2, sliceLen4)
fmt.Println(sliceLen2) // output: [1 1 1]

copy(sliceLen0, sliceLen4)
fmt.Println(sliceLen0) // output: []
```

Figure 4.31. Copy Illustrated


ILLUSTRATION
COMING

Copy can be useful if you want to create a separate copy of a slice expression, or you want to get a copy of a slice before or after calls to `append` that might modify it. You just need to ensure that you allocate a destination slice of the correct length to hold the copy. This is where `make` can be particularly helpful, since you can use it to create a new slice of exactly the right length to hold the copied elements:

Listing 4.56. Using `copy` To Create A Separate Copy Of A Slice Expression

```
intSlice := []int{1, 2, 3, 4, 5, 6}
subSlice := intSlice[2:4] //[3 4]
subSliceCopy := make([]int, len(subSlice))
copy(subSliceCopy, subSlice)

// Change a value in the original slice.
intSlice[2] = 10
// Change is visible in subSlice, but not in subSliceCopy
fmt.Println(subSlice)     //output: [10 4]
fmt.Println(subSliceCopy) //output: [3 4]
```

4.2.8 Avoiding Runtime Panics When Accessing Slice Values

Because the size of a slice isn't set until the program is running, you can sometimes run into a situation where a value you're expecting to find isn't there.

As an example, let's say you have a function to print the top three finishers in a race:

Listing 4.57. Function To Print The Top Three Items

```
func printMedalists(raceName string, contestants []string) {
        fmt.Printf("Results for %s:\n", raceName)
        fmt.Printf("Gold Medal: %s\n", contestants[0])
        fmt.Printf("Silver Medal: %s\n", contestants[1])
        fmt.Printf("Bronze Medal: %s\n", contestants[3])
}
```

This function expects a sorted list with at least three entries, which it gets directly from the slice by index. If you pass a slice with more than three

values, it works fine, however a slice with less than three values will cause a runtime panic:

Listing 4.58. Passing Different Lengths Of Slices to `printMedalists`

```
func main() {
        race1 := []string{"Sally", "Steve", "Jo", "Adam"}
        printMedalists("Race 1", race1)

        race2 := []string{"Jon", "Jen"}
        printMedalists("Race 2", race2)
}
```

Listing 4.59. Output From `printMedalists` With A Runtime Panic

```
Results for Race 1:
Gold Medal: Sally
Silver Medal: Steve
Bronze Medal: Adam
Results for Race 2:
Gold Medal: Jon
Silver Medal: Jen
panic: runtime error: index out of range [3] with length 2
```

The reason for the panic is that there's really no graceful way for the Go runtime to handle a missing element that you explicitly asked for, so the only thing it can do is stop the program immediately and tell you what went wrong.

Out-of-bounds errors on slices like this are one of the most common panic conditions you'll encounter when starting out with Go, but fortunately they are pretty easy to handle with a little foresight.

The first technique is to verify that you have at least the number of items you need by calling `len` on the incoming slice. In this case, you know that the function is expecting three entries, so you can just add a check for a length below that and return an error otherwise:

Listing 4.60. Verifying Slice Size With `len()`

```
func printMedalists2(raceName string, contestants []string) error
```

```
        if len(contestants) < 3 {
                return fmt.Errorf("not enough contestants for %s.
                        raceName, len(contestants))
        }
        fmt.Printf("Results for %s:\n", raceName)
        fmt.Printf("Gold Medal: %s\n", contestants[0])
        fmt.Printf("Silver Medal: %s\n", contestants[1])
        fmt.Printf("Bronze Medal: %s\n", contestants[3])
        return nil
}

func main() {
        race2 := []string{"Jon", "Jen"}

        // Test an invalid race with new error handling.
        err := printMedalists2("Race 2", race2)
        if err != nil {
                log.Println(err)
        }
}
```

Listing 4.61. Output From `printMedalists2` With Slice Length Checking

```
not enough contestants for Race 2. Want: 3 Got: 2
```

By adding a length check, you ensure that accessing an invalid entry is impossible, since the function will just return an error otherwise. It also gives you the opportunity to log a helpful error message with the name of the invalid condition and the number of arguments you actually got. Of course, this means every call to your function needs to do error checking to ensure that nothing went wrong, and this function is pretty inflexible, since races with less than three people are invalid.

A much more flexible technique is to avoid direct index access altogether, and write your code to handle the entire slice with a `range` loop.

Listing 4.62. Using A `range` Statement Instead Of Explicit Indices

```
func printMedalists3(raceName string, contestants []string) {
        fmt.Printf("Results for %s:\n", raceName)
        for place, contestant := range contestants {
                switch place {
                case 0:
```

```
                        fmt.Printf("Gold Medal: %s\n", contestant
                case 1:
                        fmt.Printf("Silver Medal: %s\n", contesta
                case 2:
                        fmt.Printf("Bronze Medal: %s\n", contesta
                default:
                        fmt.Printf("Finisher: %s\n", contestant)
                        // or just return
                }
        }
}

func main() {
        race1 := []string{"Sally", "Steve", "Jo", "Adam"}
        race2 := []string{"Jon", "Jen"}
        race3 := []string{}

        printMedalists3("Race 1", race1)
        // Race 2 only has two contestants, but we still need to
        printMedalists3("Race 2", race2)
        // Throw in a race with no contestants at all.
        printMedalists3("Race 3", race3)
}
```

Listing 4.63. Output From `printMedalists3` With More Flexible Algorithm

```
Results for Race 1:
Gold Medal: Sally
Silver Medal: Steve
Bronze Medal: Jo
Finisher: Adam
Results for Race 2:
Gold Medal: Jon
Silver Medal: Jen
Results for Race 3:
```

By making the code more flexible, you can handle any number of constestants, and even add new features, like printing fourth-place and beyond, after all, sometimes just finishing is a victory! Because this code uses a `range` loop, it will repeat as few or as many times as it needs to, or, in the case of no items at all, it will simply just skip it altogether.

 **Tip**

When working with slices, it is often more flexible to use a range loop than to attempt random index access.

Of course, another option is to just do nothing at all, and let the program panic if the truly unexpected occurs. If you are always expecting your slices to have a certain length, an exception to this might represent a significant enough bug that a panic will give a pretty clear message that something is very wrong (panics have their purpose, after all). However, if there's any valid possibilty that you could be getting slices of an unexpected size, you might want to opt for one of the other techniques.

4.2.9 Nil Slices Versus Empty Slices

It might be tempting to use `if slice == nil` to check if a slice is empty, but this can lead to surprising results. This is because slices are only `nil` if they have not been initialized or if they have been explicitly set to `nil` by the programmer. A slice can have a `len` of `0`, yet still not be a nil slice:

Listing 4.64. Creating Various Slices Of Zero Length

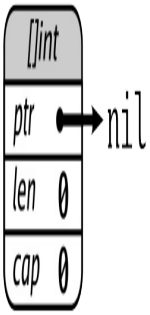
```
// An uninitialized slice is == nil.
var slice []int
fmt.Println(slice == nil) // true
fmt.Println(len(slice))   // 0

// A slice with an explicit len of zero is != nil.
sliceLenZero := make([]int, 0)
fmt.Println(sliceLenZero == nil) // false
fmt.Println(len(sliceLenZero))   // 0

// A slice made from an empty literal is != nil.
sliceEmptyLit := []int{}
fmt.Println(sliceEmptyLit == nil) // false
fmt.Println(len(sliceEmptyLit))   // 0

// A slice of a slice that has a length of zero is != nil.
sliceWithValues := []int{1, 2, 3}
slicedSlice := sliceWithValues[0:0]
fmt.Println(slicedSlice == nil) // false
fmt.Println(len(slicedSlice))   // 0

// The only way to get back to a nil slice is to set it to nil ex
```

```
sliceLenZero = nil
fmt.Println(sliceLenZero == nil) // true
```

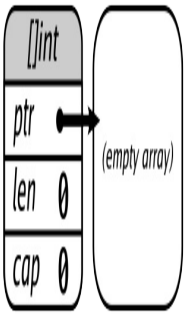
This code illustrates several different slices which are effectively empty, yet are not equal to `nil` for different reasons.

Figure 4.32. Different Zero-length Slices Illustrated

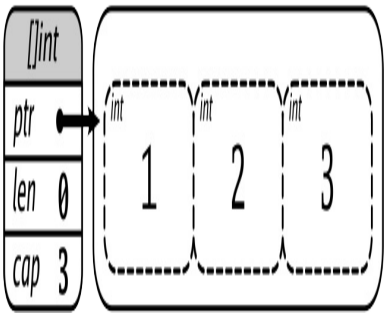
```
var slice []int
```




```
sliceLenZero := make([]int, 0)
sliceEmptyLit := []int{}
```




```
sliceWithValues := []int{1, 2, 3}
slicedSlice := sliceWithValues[:0]
```



In the case of `sliceLenZero` and `sliceEmptyLit`, they become initialized even if the backing array they point to is empty, since there are valid conditions in which this can occur, such as explicit reallocation. In the case of

`slicedSlice`, it still shares a backing array that has values, but the slice header itself has a length of `0`. This can happen when programmatically resizing slices that might not meet some threshold.

For this reason, you should always use a `len` check if you need to find out if a slice is empty, since checking for `nil` equality will not always give you the answer you want.

Explicit `nil` checks should be reserved for those cases when you want to tell the difference between a slice that somehow been has a length of `0`, versus one that was never initialized in the first place.

[10] This is sometimes called *array doubling* or *amortized doubling*

4.3 Maps

Like arrays and slices, maps are collection types for storing multiple values of any type you need, however the way in which the values are indexed is much more flexible.

In arrays and slices, values are set and retrieved based off of an integer index, which represents the value's position in the list. In maps, values are accessed using another value called the *key*, which can be almost any type you want. This lets you retrieve values quickly, using values that are more meaningful to your use case.

For example, let's say you need to write an inventory system for a grocery store, and you need to calculate the weights for shipping items in bulk. This would be kind of a pain with list types like arrays and slices, since you'd have to keep two separate lists for both items names (strings) and weights (ints), but with maps you can store them together:

Listing 4.65. An Example Map

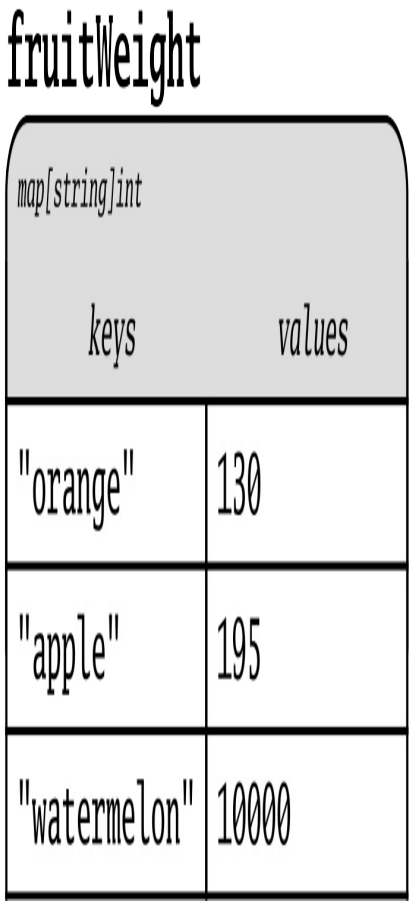
```
// fruitWeight stores the average weight of different kinds of fr
// in grams
fruitWeight := map[string]int {
        "orange"    : 130,
```

```
        "apple"     : 195,
        "watermelon": 10000,
}

// About how heavy are 10 apples and 30 oranges?
fmt.Println(10*fruitWeight["apple"] + 30*fruitWeight["orange"])
// output:
//
```

In this example, the `fruitWeight` map stores the average weight of different kinds of fruit. The key type is `string`, while the value type is `int`, since weights are generally specified in numeric form.

You can think of a map as a bit like a dictionary where definitions (values) are found by looking up the word(key) they are associated with.

Figure 4.33. Example Map Illustrated



In the above example `namedNumbers` is a map that stores `int` values usint a

`string` key, which means that each integer stored in the map is associated with a particular unique string value, which can be used to access it.

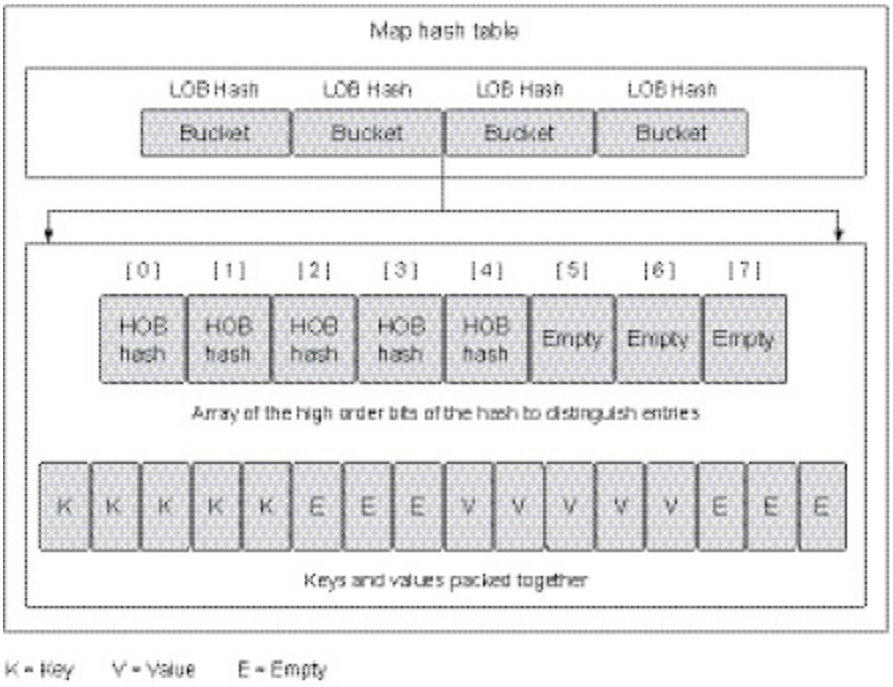
You can think of maps as a bit like a dictionary, where definitions (values) are retrieved by looking up the word (key).

Maps are valuable data structures because they provide a fast way to take a key, such as a string or or a struct, and use it as an index to store and retrieve values from the map.

Figure 4.34. Relationship Between Keys And Values

Internally, maps are implemented using hash tables, which contain a collection of buckets. When you’re storing, removing, or looking up a key/value pair, everything starts with selecting a bucket.

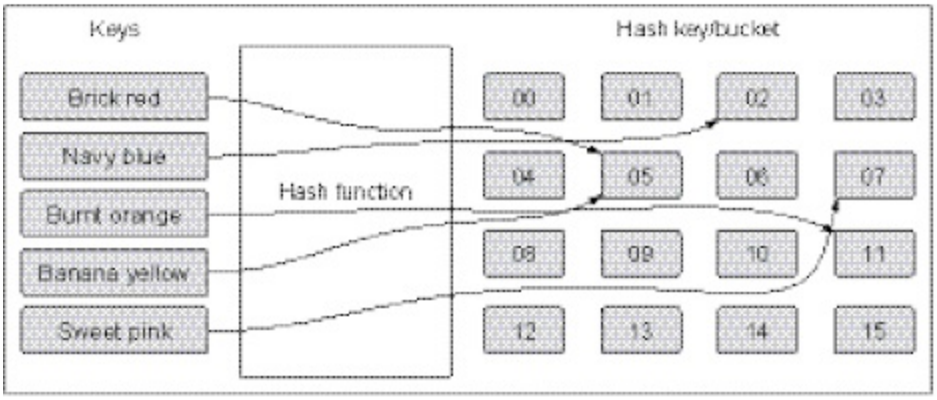
Figure 4.35. Simple Representation Of The Internal Structure Of A Map



This is performed by passing the key specified in your map operation to the map's hash function. The purpose of the hash function is to generate an index that evenly distributes key/value pairs across all available buckets.

The better the distribution, the quicker you can find your key/value pairs as the map grows. If you store 10,000 items in your map, you don't want to ever look at 10,000 key/value pairs to find the one you want. You want to look at the least number of key/value pairs possible. Looking at only 8 key/value pairs in a map of 10,000 items is a good and balanced map. A balanced list of key/value pairs across the right number of buckets makes this possible.

Figure 4.36. Simple Representation Of The Internal Structure Of A Map



The hash key that's generated for a Go map is a bit longer than what it appears in this illustration, but it works the same way. In this example, the keys are strings that represents a color. Those strings are converted into a numeric value within the scope of the number of buckets we have available for storage. The numeric value is then used to select a bucket for storing or finding the specific key/value pair. In the case of a Go map, a portion of the generated hash key, specifically the low order bits (LOB), is used to select the bucket.

There are two data structures that contain the data for the map. First, there's an array with the top eight high order bits (HOB) from the same hash key that was used to select the bucket. This array distinguishes each individual key/value pair stored in the respective bucket. Second, there's an array of bytes that stores the key/value pairs. The byte array packs all the keys and then all the values together for the respective bucket. The packing of the key/value pairs is implemented to minimize the memory required for each bucket.

There are a lot of other low-level implementation details about maps that are outside the scope of this chapter. You don't need to understand all the internals to learn how to create and use maps. Just remember one thing: a map is an unordered collection of key/value pairs.

4.3.1 Declaring Maps

Map declarations take the form of `map[K]E` where `K` is the key type used for looking up values, and `E` is the element type, which is the type of value stored in the map. Just like slices, maps can be created using `make`, or they can be created and initialized at the same time, using map literals, which specify a starting list of key/value pairs.

Listing 4.66. Declaring Maps With `make` And Literals

```
// Declare and initialize a map with make
intToString := make(map[int]string)

// Declare an empty map with a literal.
// (this is identical to using make)
intToString := map[int]string{}

// Declare and populate a map with a literal.
// Keys are separated from values with a colon, and commas separa
// key-value pairs.
intToString := map[int]string{
        1: "one",
        2: "two",
}
```

Figure 4.37. Map Declarations Visualized


ILLUSTRATION
COMING

You can also declare map variables using the `var` keyword, however this comes with a caveat: values cannot be added to or retrieved from a map until it has been intialized, and attempts to do so will result in a runtime panic:

Listing 4.67. Declaring A Map With `var`

```
// Declare a map variable using var
var intToString map[int]string

// Attempt to set a value in the map
intToString[1] = "one" // panic: assignment to entry in nil map
```

The reason for this is that maps declared without a literal are uninitialized, which means they do not have any memory allocated for their internal data structures, and so they aren't set up to store values yet. This is known as a *nil map*. Like nil slices, nil maps are equal to `nil`, which you can check for. However, unlike nil slices, there is nothing like `append` that can add values to a nil map, so it is easier to forget to initialize them and end up with a panic. For this reason, you should prefer using `make` or map literal initialization when you can. If a map value is optional, you need to remember to check for `nil` first.

 **Important**

Nil maps *must* be initialized before they can be used. When in doubt, check to see if the map is `nil` first.

Nil map panics can be avoided by checking to see if a map value is `== nil`, and initializing it if so:

Listing 4.68. Initializing A Map If It Is A Nil Map

```
// Declare a map variable using var
var intToString map[int]string

if intToString == nil {
        intToString = map[int]string{}
}
```

```
// Assignment succeeds after the map has been initialized.
intToString[1] = "one"
```

How Should You Initialize Maps?

Because map literals can create new maps and also optionally populate them with values, they are generally the preferred method to allocate new map values. They are also somewhat shorter, since you don’t need two separate steps to create the map and populate it.

 **Tip**

Map literals are the most flexible and idiomatic way to initialize maps in Go. Use them unless you have a good reason not to.

4.3.2 Accessing Map Values

To set a map value, you need to provide a key and an element of the correct types. If a value already exists for the given key, it will be overwritten.

Listing 4.69. Assigning Values For Different Map Types

```
colors := map[string]string{}
// The key type of colors is string, so the indices are strings
// representing the name of the color.
colors["red"] = "#FF0000"
colors["green"] = "#00FF00"

// Assigning a value using an existing key will overwrite the old
colors["red"] = "#DD2222" // (replaces "#FF0000")

userIDs := map[string]int{}
// The element type of userIDs is int, so the values are integers
userIDs["Andy"] = 1
userIDs["Sarah"] = 2

// Variables can be used for both the key value and the element v
userName := "Brian"
userID := 3
userIDs[userName] = userID
```

When retrieving values from a map, you can access values by their key. If the key does not exist, Go will return the zero value for your element type instead. If you need to test whether a particular key exists, you can use two-variable assignment form, which will return a second boolean value representing whether or not the key was found in the map.

Listing 4.70. Retrieving Values From A Map

```
userIDs := map[string]int{
        "Andy": 1,
        "Sarah": 2,
}

AndyID := userIDs["Andy"] // 1
JenID := userIDs["Jen"] // 0 (key is not in the map)

JeffID, found := userIDs["Jeff"] // 0, false

// You can also omit the value variable if you are just checking
// membership. Using if with a simple-statement if is a common id
if _, found := userIDs["Carmen"]; !found {
        fmt.Println("Didn't find user Carmen")
}
```

Nonexistent Keys And Zero Values

It might seem odd at first that maps will return values for keys that don’t exist, but this actually enables some handy behavior in cases where the existence of the key isn’t important.

One example might be a membership list, using `string` as the key type (representing a member’s name) and `bool` for the element type, representing whether they are a member of the club or not.

Listing 4.71. Using Zero Values For Membership Queries

```
clubMembers := map[string]bool{
        "Susan": true,
        "Aditya": true,
}

for _, name := range []string{"Aditya", "Andy", "Susan"} {
```

```
        if clubMembers[name] {
                fmt.Printf("%s is a member.\n", name)
        } else {
                fmt.Printf("%s is not a member.\n", name)
        }
}

// output:
// Aditya is a member.
// Andy is not a member.
// Susan is a member.
```

Because the zero value of `bool` is false, you can get away with only storing members who are in the club, and all other queries will return `false`. This is a common way to create sets in Go, since you can check for membership with a simple `if` statement.[11]

Another case where this might be useful is when looking up numeric data for some entity that might or might not be tracked yet.

Listing 4.72. Using The Zero Value With Numeric Data

```
wordsWritten := map[string]int {
        "Andy": 1000,
        "Jen": 500,
}

for _, name := range []string{"Jen", "Andy", "Eve"} {
        fmt.Printf("%s has written %d words\n", name, wordsWritte
}

// output:
// Jen has written 500 words
// Andy has written 1000 words
// Eve has written 0 words
```

In this example, `"Eve"` is not in the map, so you can assume they have written zero words, and can just let the zero value speak for itself.

This attribute makes maps more consistent to work with, and means that value access will never cause a panic. It also allows you to decide whether or not key existence is important. When it is necessary to distinguish between "not in the map" and "in the map, but with a zero value", you can use the two-

value assignment as you need.

4.3.3 Removing Values From Maps.

The built-in `delete` function is used to remove items from a map by key. Removing keys that do not exist is safe, and will not result in an error.

Listing 4.73. Removing A Value From A Map With `delete`

```
countryCodes := map[string]string{
        "FR": "France",
        "CN": "China",
        "ES": "Spain",
        "TX": "Texas",
}
fmt.Println(countryCodes)
// output: map[CN:China ES:Spain FR:France TX:Texas]

// Delete the value associated with key "TX" from the map
delete(countryCodes, "TX")
fmt.Println(countryCodes)
// output: map[CN:China ES:Spain FR:France]

// The "XX" key does not exist, so no changes will be made to the
// no error will result.
delete(countryCodes, "XX") // Key "XX" does not exist, so no acti
fmt.Println(countryCodes)
// output: map[CN:China ES:Spain FR:France]
```

4.3.4 Getting The Number Of Values In A Map

To get the number of values in a map, you can use the `len` built-in.

Listing 4.74. Using `len` to get the number of keys/values In A Map

```
petWeightsKG := map[string]float64 {
        "Baxter": 8.6,
        "Barry": 3.17,
        "Bobby": 9.07,
        "Benny": 1.08,
}

numberOfPets := len(petWeightsKG) // 4
```

4.3.5 Iterating Maps

To iterate a map, you can use a `range` loop which will return a single map key for each iteration of the loop, along with an optional copy of the value.

Listing 4.75. Iterating A Map Using `for` and `range`

```
groupNouns := map[string]string{
        "eagle":    "convocation",
        "cat":      "clowder",
        "kangaroo": "mob",
        "dog":      "pack",
}

// Iterate with keys only.
for animal := range groupNouns {
        fmt.Printf("A group of %ss is called a %s.\n", k, groupNo
}

// Iterate with keys and values.
for animal, group := range groupNouns {
        fmt.Printf("A group of %ss is called a %s.\n", animal, gr
}
```

Because of the way maps are implented, they have no specific order, and the order you will receive keys from the range statement can be different each time.[12]. If you run the above code multiple times, you will likely see the order change:

Listing 4.76. Map Iteration Output From Multiple Runs

```
A group of cats is called a clowder.
A group of kangaroos is called a mob.
A group of dogs is called a pack.
A group of eagles is called a convocation.

A group of eagles is called a convocation.
A group of cats is called a clowder.
A group of kangaroos is called a mob.
A group of dogs is called a pack.
```

Map iteration can also be combined with the `delete` function, allowing you to examine each item of the map, removing it if necessary. This is useful for

"filtering" a map by some criteria:

Listing 4.77. Filtering A Map Using A `range` loop

```
familyAges := map[string]int{
        "Grandpa": 83,
        "Homer":   36,
        "Marge":   34,
        "Bart":    10,
        "Lisa":    8,
        "Maggie":  1,
}

for person, age := range familyAges {
        if age < 18 {
                delete(familyAges, person)
        }
}
fmt.Println(familyAges)
// output: map[Grandpa:83 Homer:36 Marge:34]
```

In the example above, the list is filtered based on whether or not each person is over the age of 18. This is safe to do in Go, provided the map is not being accessed elsewhere at the same time.[13].

4.3.6 Passing Maps To Functions

Like slices, maps are fundamentally reference types, since they are

4.3.7 Key And Value Types

You can use any type you like for map elements, while key types are slightly more restricted. The only requirement for key types is that they must be able to be compared using the == operator. Types that are not comparable in this way cannot be key types, so you cannot use functions, slices or maps for your key type, or any structs that contain them. Everything else is fair game. This leads to some interesting possibilities, such as maps using a struct type as a key and slices as a value:

Listing 4.78. Example Map With A Struct Type Key And A Slice Value

```
favoriteFoods := map[person][]string{}

andy := person{
        firstName: "Andy",
        lastName:  "Walker",
        age:       42,
}

john := person{
        firstName: "John",
        lastName:  "Smith",
        age:       53,
}

favoriteFoods[andy] = []string{"jambalaya", "bulgogi"}
favoriteFoods[john] = []string{"crab cakes", "licorice"}

        johnFoods, found := favoriteFoods[person{
                firstName: "John",
                lastName:  "Smith",
                age:       29,
        }]
        // The "firstName" and "lastName" fields are the same, bu
        // field is different, so there is no match.
        fmt.Println(johnFoods, found) // output: [], false

        // All fields match a key value in the map, so the elemen
        // returned.
        andyFoods, found := favoriteFoods[person{
                firstName: "Andy",
                lastName:  "Walker",
                age:       42,
        }]
        fmt.Println(andyFoods, found) // output: [jambalaya bulgo
```

[11] For larger sets, there is another technique using empty structs that is slightly less readable, but consumes less memory. We'll go over this in Chapter: Generics

[12] This is actually by design, since map order is unspecified, and can differ from architecture to architecture.

[13] You will learn more about concurrent map access safety in Chapter Concurrency

4.4 Summary

- Arrays are constant-sized collections of data that form the basis of the slice and map types.
- Slices are powerful and flexible ordered collection types, and the primary ordered collection type in Go.
- `append` works by returning a new slice header, though it does not always allocate new storage.
- Maps are Go’s associative array or "dictionary" type, and allow you to store values of any type using a key type best suited for your use-case.
- Slices and maps are fundamentally reference types that point to the data they contain, and it helps to be mindful of how they work and when you should modify them.