An Introduction to Programming in Go
Copyright © 2012 by Caleb Doxsey

ISBN: 978-1478355823

Cover art by Abigail Doxsey Anderson.

Contents

1 Getting Started

Computer programming is the art, craft and science of writing programs which define how computers operate. This book will teach you how to write computer pro grams using a programming language designed by Google named Go.

Go is a general purpose programming language with advanced features and a clean syntax. Because of its wide availability on a variety of platforms, its robust well-documented common library, and its focus on good software engineering principles, Go is an ideal language to learn as your first programming language.

The process we use to write software using Go (and most programming languages) is fairly straightfor ward:

1. Gather requirements

2. Find a solution

3. Write source code to implement the solution

4. Compile the source code into an

executable

5. Run and test the program to make sure it works

This process is iterative (meaning its done many times) and the steps usually overlap. But before we write our first program in Go there are a few prerequi site concepts we need to understand.

1.1 Files and Folders

A file is a collection of data stored as a unit with a name. Modern operating systems (like Windows or Mac OSX) contain millions of files which store a large variety of different types of information – everything from text documents to executable programs to multi media files.

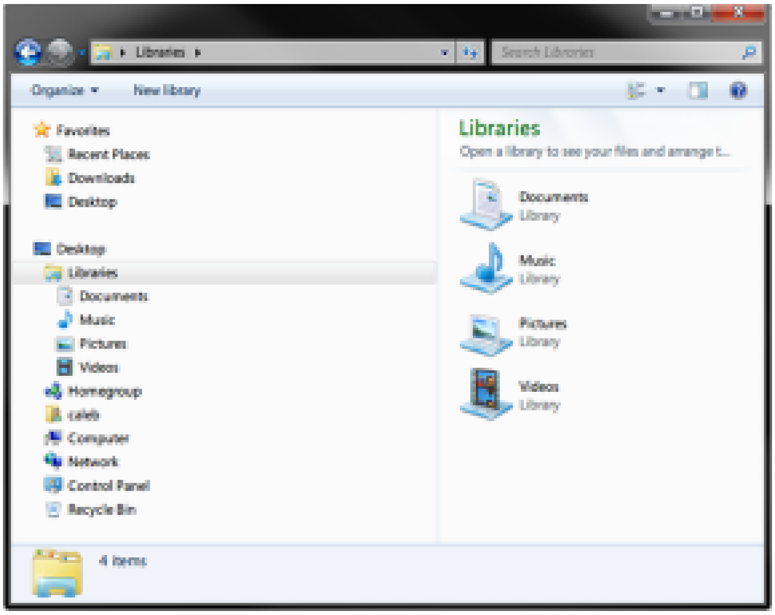
All files are stored in the same way on a computer: they all have a name, a definite size (measured in bytes) and an associated type. Typically the file's type is signified by the file's extension – the part of the file name that comes after the last `.`. For example a file with the name `hello.txt` has the extension `txt` which is used to represent textual data.

Folders (also called directories) are used to group files together. They can also contain other folders. On Win-

dows file and folder paths (locations) are represented with the `\` (backslash) character, for example: `C:\Users\john\example.txt`. `example.txt` is the file name, it is contained in the folder `john`, which is itself contained in the folder `Users` which is stored on drive `C` (which represents the primary physical hard drive in Windows). On OSX (and most other operating sys tems) file and folder paths are represented with the `/` (forward slash) character, for example: `/Users/john/example.txt`. Like on Windows `example.txt` is the file name, it is contained in the folder `john`, which is in the folder `Users`. Unlike Win dows, OSX does not specify a drive letter where the file is stored.

Getting Started 4 **Windows**

On Windows files and folders can be browsed using Windows Explorer (accessible by double-clicking “My Computer” or typing win+e):



5 Getting Started **OSX**

On OSX files and folders can be browsed using Finder (accessible by clicking the Finder icon – the face icon in the lower left bar):

1.2 The Terminal

Most of the interactions we have with computers today are through sophisticated graphical user interfaces (GUIs). We use keyboards, mice and touchscreens to interact with visual buttons or other types of controls that are displayed on a screen.

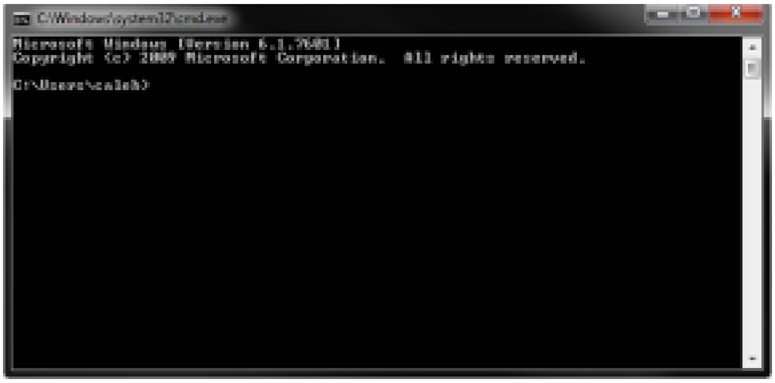
It wasn't always this way. Before the GUI we had the terminal – a simpler textual interface to the computer

where rather than manipulating buttons on a screen we issued commands and received replies. We had a conversation with the computer.

And although it might appear that most of the comput ing world has left behind the terminal as a relic of the past, the truth is that the terminal is still the funda mental user interface used by most programming lan guages on most computers. The Go programming lan guage is no different, and so before we write a program in Go we need to have a rudimentary understanding of how a terminal works.

Windows

In Windows the terminal (also known as the command line) can be brought up by typing the windows key + r (hold down the windows key then press r), typing `cmd.exe` and hitting enter. You should see a black win dow appear that looks like this:



```
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation.  All rights reserved.

C:\Users\caleb>
```

By default the command line starts in your home di rectory. (In my case this is `C:\Users\caleb`) You issue commands by typing them in and hitting enter. Try entering the command `dir`, which lists the contents of a directory. You should see something like this:

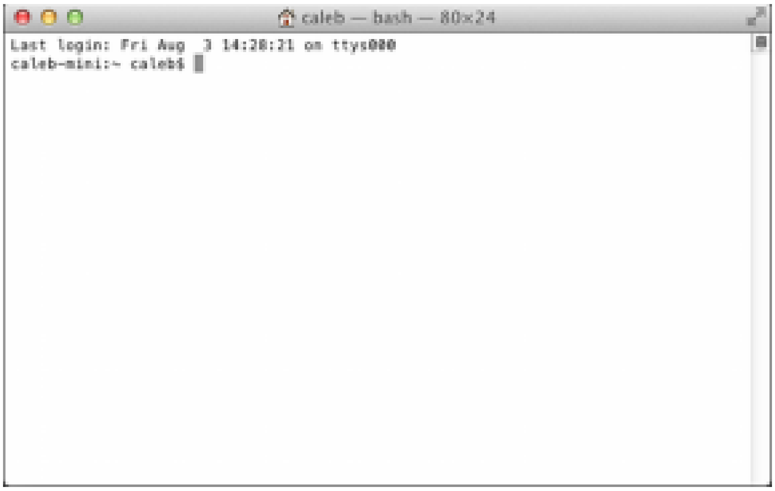
```
C:\Users\caleb>dir
 Volume in drive C has no label.
 Volume Serial Number is B2F5-F125
```

Followed by a list of the files and folders contained in your home directory. You can change directories by us ing the command `cd`. For example you probably have a folder called `Desktop`. You can see its contents by en tering `cd Desktop` and then entering `dir`. To go back to your home directory you can use the special directory name `..` (two periods next to each other): `cd ..`. A sin gle period represents the current folder (known as the working folder), so `cd .` doesn't do anything. There are

a lot more commands you can use, but this should be enough to get you started.

OSX

In OSX the terminal can be reached by going to Finder → Applications → Utilities → Terminal. You should see a window like this:



```
Last login: Fri Aug  3 14:28:21 on ttys000
caleb-mini:~ caleb$ 
```

By default the terminal starts in your home directory. (In my case this is `/Users/caleb`) You issue commands by typing them in and hitting enter. Try entering the command `ls`, which lists the contents of a directory. You should see something like this:

```
 caleb-min:~ caleb$ ls
Desktop Downloads Movies Pictures Documents
Library Music Public
```

These are the files and folders contained in your home directory (in this case there are no files). You can change directories using the `cd` command. For example you probably have a folder called `Desktop`. You can see its contents by entering `cd Desktop` and then entering `ls`. To go back to your home directory you can use the special directory name `..` (two periods next to each other): `cd ..`. A single period represents the current folder (known as the working folder), so `cd .` doesn't do anything. There are a lot more commands you can use, but this should be enough to get you started.

1.3 Text Editors

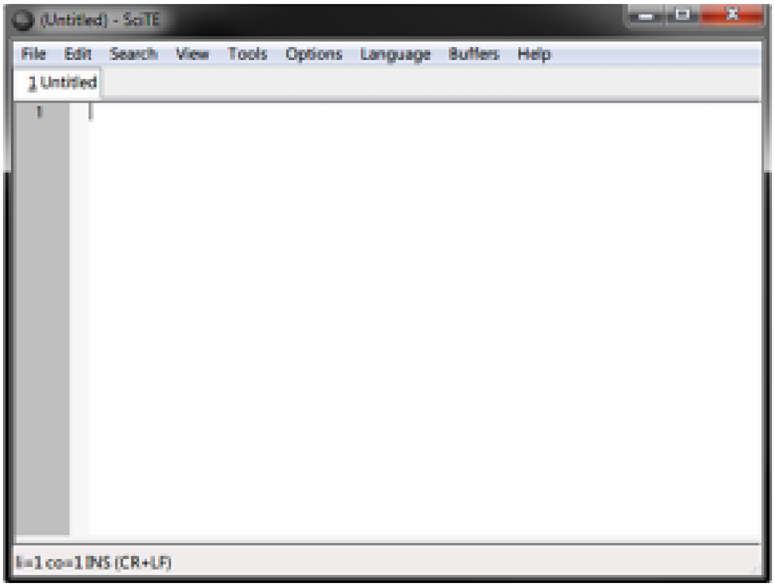
The primary tool programmers use to write software is a text editor. Text editors are similar to word process ing programs (Microsoft Word, Open Office, …) but un like such programs they don't do any formatting, (No bold, italic, …) instead they operate only on plain text. Both OSX and Windows come with text editors but they are highly limited and I recommend installing a better one.

To make the installation of this software easier an in-

staller is available at the book's website: `http://www.golang-book.com/`. This installer will in stall the Go tool suite, setup environmental variables and install a text editor.

Windows

For windows the installer will install the Scite text ed itor. You can open it by going to Start → All Programs → Go → Scite. You should see something like this:



li=1 co=1 INS (CR+LF)

The text editor contains a large white text area where text can be entered. To the left of this text area you can see the line numbers. At the bottom of the window

is a status bar which displays information about the file and your current location in it (right now it says that we are on line 1, column 1, text is being inserted normally, and we are using windows-style newlines).

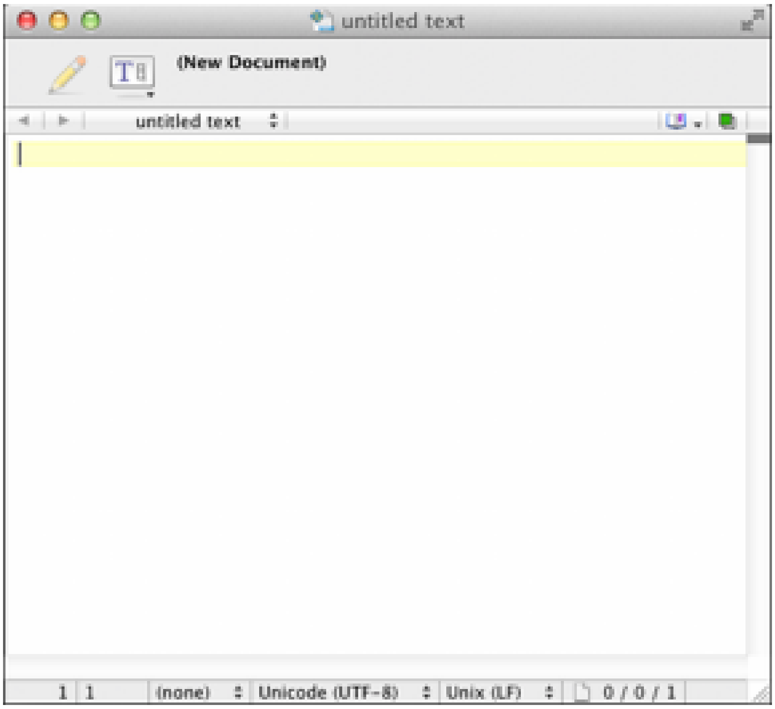
You can open files by going to File → Open and brows ing to your desired file. Files can be saved by going to File → Save or File → Save As.

As you work in a text editor it is useful to learn key board shortcuts. The menus list the shortcuts to their right. Here are a few of the most common:

• Ctrl + S – save the current file
• Ctrl + X – cut the currently selected text (remove it and put it in your clipboard so it can be pasted later) • Ctrl + C – copy the currently selected text • Ctrl + V – paste the text currently in the clipboard • Use the arrow keys to navigate, Home to go to the beginning of the line and End to go to the end of the line
• Hold down shift while using the arrow keys (or Home and End) to select text without using the mouse
• Ctrl + F – brings up a find in file dialog that you can use to search the contents of a file

OSX

For OSX the installer installs the Text Wrangler text editor:



Like Scite on Windows Text Wrangler contains a large white area where text is entered. Files can be opened by going to File → Open. Files can be saved by going to File → Save or File → Save As. Here are some useful keyboard shortcuts: (Command is the ⌘ key)

• Command + S – save the current file

• Command + X – cut the currently selected text (re move it and put it in your clipboard so it can be pasted later)

• Command + C – copy the currently selected text • Command + V – paste the text currently in the clip board

• Use the arrow keys to navigate

- Command + F – brings up a find in file dialog that you can use to search the contents of a file

1.4 Go Tools

Go is a compiled programming language, which means source code (the code you write) is translated into a language that your computer can understand. There fore before we can write a Go program, we need the Go compiler.

The installer will setup Go for you automatically. We will be using version 1 of the language. (More informa tion can be found at `http://www.golang.org`)

Let's make sure everything is working. Open up a ter minal and type the following:

```
go version
```

You should see the following:

```
go version go1.0.2
```

Your version number may be slightly different. If you get an error about the command not being recognized try restarting your computer.

The Go tool suite is made up of several different com mands and sub-commands. A list of those commands is available by typing:

```
go help
```

We will see how they are used in subsequent chapters.

2 Your First Program

Traditionally the first program you write in any pro gramming language is called a “Hello World” program – a program that simply outputs `Hello World` to your terminal. Let's write one using Go.

First create a new folder where we can store our pro gram. The installer you used in chapter 1 created a folder in your home directory named `Go`. Create a folder named `~/Go/src/golang-book/chapter2`. (Where `~` means your home directory) From the terminal you can do this by entering the following commands:

```
mkdir Go/src/golang-book
mkdir Go/src/golang-book/chapter2
```

Using your text editor type in the following:

```
package main

import "fmt"

// this is a comment

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

Make sure your file is identical to what is shown here and save it as `main.go` in the folder we just created. Open up a new terminal and type in the following:

```
cd Go/src/golang-book/chapter2
go run main.go
```

You should see `Hello World` displayed in your termi nal. The `go run` command takes the subsequent files (separated by spaces), compiles them into an exe cutable saved in a temporary directory and then runs the program. If you didn't see `Hello World` displayed you may have made a mistake when typing in the pro gram. The Go compiler will give you hints about where the mistake lies. Like most compilers, the Go compiler is extremely pedantic and has no tolerance for mis takes.

2.1 How to Read a Go Program

Let's look at this program in more detail. Go programs are read top to bottom, left to right. (like a book) The first line says this:

```
package main
```

This is know as a “package declaration”. Every Go pro gram must start with a package declaration. Packages are Go's way of organizing and reusing code. There are two types of Go programs: executables and libraries. Executable applications are the kinds of programs that we can run directly from the terminal. (in Windows they end with `.exe`) Libraries are collections of code that we package together so that we can use them in other programs. We will explore libraries in more de tail later, for now just make sure to include this line in any program you write.

The next line is a blank line. Computers represent newlines with a special character (or several charac ters). Newlines, spaces and tabs are known as white space (because you can't see them). Go mostly doesn't care about whitespace, we use it to make programs easier to read. (You could remove this line and the pro gram would behave in exactly the same way)

Then we see this:

```
import "fmt"
```

The `import` keyword is how we include code from other packages to use with our program. The `fmt` package (shorthand for format) implements formatting for in put and output. Given what we just learned about packages what do you think the `fmt` package's files would contain at the top of them?

Notice that `fmt` above is surrounded by double quotes. The use of double quotes like this is known as a “string literal” which is a type of “expression”. In Go strings represent a sequence of characters (letters, numbers, symbols, ...) of a definite length. Strings are described in more detail in the next chapter, but for now the im portant thing to keep in mind is that an opening `"` character must eventually be followed by another `"` character and anything in between the two is included in the string. (The `"` character itself is not part of the string)

The line that starts with `//` is known as a comment. Comments are ignored by the Go compiler and are there for your own sake (or whoever picks up the source code for your program). Go supports two differ-

ent styles of comments: `//` comments in which all the text between the `//` and the end of the line is part of the comment and `/* */` comments where everything between the `*`s is part of the comment. (And may in clude multiple lines)

After this you see a function declaration:

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

Functions are the building blocks of a Go program. They have inputs, outputs and a series of steps called statements which are executed in order. All functions start with the keyword `func` followed by the name of the function (`main` in this case), a list of zero or more “parameters” surrounded by parentheses, an optional return type and a “body” which is surrounded by curly braces. This function has no parameters, doesn't re turn anything and has only one statement. The name `main` is special because it's the function that gets called when you execute the program.

The final piece of our program is this line:

```
fmt.Println("Hello World")
```

This statement is made of three components. First we access another function inside of the `fmt` package called `Println` (that's the `fmt.Println` piece, `Println` means Print Line). Then we create a new string that contains `Hello World` and invoke (also known as call or execute) that function with the string as the first and only argument.

At this point we've already seen a lot of new terminol ogy and you may be a bit overwhelmed. Sometimes its helpful to deliberately read your program out loud. One reading of the program we just wrote might go like this:

> Create a new executable program, which references the `fmt` library and contains one function called `main`. That function takes no arguments, doesn't re turn anything and does the following: Access the `Println` function contained inside of the `fmt` pack age and invoke it using one argument – the string `Hello World`.

The `Println` function does the real work in this pro gram. You can find out more about it by typing the fol lowing in your terminal:

```
godoc fmt Println
```

Among other things you should see

this:

```
Println formats using the default formats for
its operands and writes to standard output.
Spaces are always added between operands and
a  newline is appended. It returns the number
of  bytes written and any write error
encountered.
```

Go is a very well documented programming language but this documentation can be difficult to understand unless you are already familiar with programming lan guages. Nevertheless the `godoc` command is extremely useful and a good place to start whenever you have a question.

Back to the function at hand, this documentation is telling you that the `Println` function will send what ever you give to it to standard output – a name for the output of the terminal you are working in. This func tion is what causes `Hello World` to be displayed.

In the next chapter we will explore how Go stores and represents things like `Hello World` by learning about types.

Problems

1. What is whitespace?

2. What is a comment? What are the two ways of writing a comment?

3. Our program began with `package main`. What would the files in the `fmt` package begin with?

4. We used the `Println` function defined in the `fmt` package. If we wanted to use the `Exit` function from the `os` package what would we need to do?

5. Modify the program we wrote so that instead of printing `Hello World` it prints `Hello, my name is` followed by your name.

3 Types

In the last chapter we used the data type string to store `Hello World`. Data types categorize a set of re lated values, describe the operations that can be done on them and define the way they are stored. Since types can be a difficult concept to grasp we will look at them from a couple different perspectives before we see how they are implemented in Go.

Philosophers sometimes make a distinction between types and tokens. For example suppose you have a dog named Max. Max is the token (a particular instance or member) and dog is the type (the general concept). “Dog” or “dogness” describes a set of properties that all dogs have in common. Although oversimplistic we might reason like this: All dogs have 4 legs, Max is a dog, therefore Max has 4 legs. Types in programming languages work in a similar way: All strings have a length, x is a string, therefore x has a length.

In mathematics we often talk about sets. For example: ℝ (the set of all real numbers) or ℕ (the set of all natu ral numbers). Each member of these sets shares prop

erties with all the other members of the set. For exam ple all natural numbers are associative: “for all natu ral numbers a, b, and c, a + (b + c) = (a + b) + c and a × (b × c) = (a × b) × c.” In this way sets are similar to types in programming languages since all the values of a particular type share certain properties.

Go is a statically typed programming language. This means that variables always have a specific type and that type cannot change. Static typing may seem cum bersome at first. You'll spend a large amount of your time just trying to fix your program so that it finally compiles. But types help us reason about what our pro gram is doing and catch a wide variety of common mis takes.

Go comes with several built-in data types which we will now look at in more detail.

3.1 Numbers

Go has several different types to represent numbers. Generally we split numbers into two different kinds: integers and floating-point numbers.

Integers

Integers – like their mathematical counterpart – are

numbers without a decimal component. (…, -3, -2, -1, 0, 1, …) Unlike the base-10 decimal system we use to represent numbers, computers use a base-2 binary sys tem.

Our system is made up of 10 different digits. Once we've exhausted our available digits we represent larger numbers by using 2 (then 3, 4, 5, …) digits put next to each other. For example the number after 9 is 10, the number after 99 is 100 and so on. Computers do the same, but they only have 2 digits instead of 10. So counting looks like this: 0, 1, 10, 11, 100, 101, 110, 111 and so on. The other difference between the num ber system we use and the one computers use is that all of the integer types have a definite size. They only have room for a certain number of digits. So a 4 bit in teger might look like this: 0000, 0001, 0010, 0011, 0100. Eventually we run out of space and most com puters just wrap around to the beginning. (Which can result in some very strange behavior)

Go's integer types are: `uint8`, `uint16`, `uint32`, `uint64`, `int8`, `int16`, `int32` and `int64.` 8, 16, 32 and 64 tell us how many bits each of the types use. `uint` means “un signed integer” while `int` means “signed integer”. Un signed integers only contain positive numbers (or zero). In addition there two alias types: `byte` which is the same as `uint8` and `rune` which is the same as

`int32`. Bytes are an extremely common unit of mea surement used on computers (1 byte = 8 bits, 1024 bytes = 1 kilobyte, 1024 kilobytes = 1 megabyte, …) and therefore Go's `byte` data type is often used in the definition of other types. There are also 3 machine de pendent integer types: `uint`, `int` and `uintptr`. They are machine dependent because their size depends on the type of architecture you are using.

Generally if you are working with integers you should just use the `int` type.

Floating Point Numbers

Floating point numbers are numbers that contain a decimal component (real numbers). (1.234, 123.4, 0.00001234, 12340000) Their actual representation on a computer is fairly complicated and not really neces sary in order to know how to use them. So for now we need only keep the following in mind:

1. Floating point numbers are inexact. Occasion ally it is not possible to represent a number. For example computing `1.01 - 0.99` results in `0.020000000000000018` – A number extremely close to what we would expect, but not exactly the same.

2. Like integers floating point numbers have a cer tain size (32 bit or 64 bit). Using a larger sized floating point number increases it's precision. (how many digits it can represent)

3. In addition to numbers there are several other values which can be represented: “not a num ber” (`NaN`, for things like `0/0`) and positive and negative infinity. (`+∞` and `-∞`)

Go has two floating point types: `float32` and `float64` (also often referred to as single precision and double precision respectively) as well as two additional types for representing complex numbers (numbers with imaginary parts): `complex64` and `complex128`. Generally we should stick with `float64` when working with float ing point numbers.

Example

Let's write an example program using numbers. First create a folder called `chapter3` and make a `main.go` file containing the following:

```
package main

import "fmt"

func main() {
 fmt.Println("1 + 1 =", 1 + 1)
}
```

If you run the program and you should see this:

```
$ go run main.go
1 + 1 = 2
```

Notice that this program is very similar to the pro gram we wrote in chapter 2. It contains the same pack age line, the same import line, the same function dec laration and uses the same `Println` function. This time instead of printing the string `Hello World` we print the string `1 + 1 =` followed by the result of the expression `1 + 1`. This expression is made up of three parts: the numeric literal `1` (which is of type `int`), the `+` operator (which represents addition) and another numeric lit eral `1`. Let's try the same thing using floating point numbers:

```
fmt.Println("1 + 1 =", 1.0 + 1.0)
```

Notice that we use the `.0` to tell Go that this is a floating point number instead of an integer. Running this program will give you the same result as before.

In addition to addition Go has several other operators:

`+`	addition
`-`	subtraction
`*`	multiplicatio n
`/`	division
`%`	remainder

3.2 Strings

As we saw in chapter 2 a string is a sequence of char acters with a definite length used to represent text. Go strings are made up of individual bytes, usually one for each character. (Characters from other languages like Chinese are represented by more than one byte)

String literals can be created using double quotes `"Hello World"` or back ticks `` `Hello World` ``. The differ ence between these is that double quoted strings can not contain newlines and they allow special escape se quences. For example `\n` gets replaced with a newline and `\t` gets replaced with a tab character.

Several common operations on strings include finding the length of a string: `len("Hello World")`, accessing an individual character in the string: `"Hello World"[1]`, and concatenating two strings together: `"Hello " + "World"`. Let's modify the program we cre ated earlier to test these out:

```
package main

import "fmt"

func main() {
 fmt.Println(len("Hello World"))
 fmt.Println("Hello World"[1])
 fmt.Println("Hello " + "World")
}
```

A few things to notice:

1. A space is also considered a character, so the string's length is 11 not 10 and the 3rd line has `"Hello "` instead of `"Hello"`.

2. Strings are "indexed" starting at 0 not 1. `[1]` gives you the 2nd element not the 1st. Also notice that you see `101` instead of `e` when you run this program. This is because the character is repre sented by a byte (remember a byte is an integer).

One way to think about indexing would be to show it like this instead: `"Hello World"`$_1$. You'd read that as "The string Hello World sub 1," "The string Hello World at 1" or "The second character of the string Hello World".

3. Concatenation uses the same symbol as addi tion. The Go compiler figures out what to do based on the types of the arguments. Since both sides of the `+` are strings the compiler assumes you mean concatenation and not addition. (Ad dition is meaningless for strings)

3.3 Booleans

A boolean value (named after George Boole) is a spe cial 1 bit integer type used to represent true and false (or on and off). Three logical operators are used with boolean values:

&&	and
||	or
!	not

Here is an example program showing how they can be used:

```
func main() {
 fmt.Println(true && true)
 fmt.Println(true && false)
 fmt.Println(true || true)
 fmt.Println(true || false)
 fmt.Println(!true)
}
```

Running this program should give you:

```
$ go run main.go
true
false
true
true
false
```

We usually use truth tables to define how these opera tors work:

Expression	**Value**
`true && true`	`true`
`true && false`	`false`
`false && true`	`false`
`false && false`	`false`

Expression	**Value**
`true || true`	`true`

`true || false`	`true`
`false || true`	`true`
`false || false`	`false`

Expression	**Value**
`!true`	`false`
`!false`	`true`

These are the simplest types included with Go and form the foundation from which all later types are built.

Problems

1. How are integers stored on a computer?

2. We know that (in base 10) the largest 1 digit number is 9 and the largest 2 digit number is 99. Given that in binary the largest 2 digit num ber is 11 (3), the largest 3 digit number is 111 (7) and the largest 4 digit number is 1111 (15) what's the largest 8 digit number? (*hint*: 10^1-1 = 9 and 10^2-1 = 99)

3. Although overpowered for the task you can use Go as a calculator. Write a program that com putes `321325 × 424521` and prints it to the ter

minal. (Use the `*` operator for multiplication)

4. What is a string? How do you find its length?

5. What's the value of the expression `(true && false) || (false && true) || !(false && false)`?

4 Variables

Up until now we have only seen programs that use lit eral values (numbers, strings, etc.) but such programs aren't particularly useful. To make truly useful pro grams we need to learn two new concepts: variables and control flow statements. This chapter will explore variables in more detail.

A variable is a storage location, with a specific type and an associated name. Let's change the program we wrote in chapter 2 so that it uses a variable:

```
package main

import "fmt"

func main() {
 var x string = "Hello World"
 fmt.Println(x)
}
```

Notice that the string literal from the original program still appears in this program, but rather than send it directly to the `Println` function we assign it to a vari

able instead. Variables in Go are created by first using the `var` keyword, then specifying the variable name (`x`), the type (`string`) and finally assigning a value to the variable (`Hello World`). The last step is optional so an alternative way of writing the program would be like this:

```
package main

import "fmt"

func main() {
 var x string
 x = "Hello World"
 fmt.Println(x)
}
```

Variables in Go are similar to variables in algebra but there are some subtle differences:

First when we see the `=` symbol we have a tendency to read that as “x equals the string Hello World”. There's nothing wrong with reading our program that way, but it's better to read it as “x takes the string Hello World” or “x is assigned the string Hello World”. This distinc tion is important because (as their name would sug gest) variables can change their value throughout the lifetime of a program. Try running the following:

37 Variables

```
package main

import "fmt"

func main() {
 var x string
 x = "first"
 fmt.Println(x)
 x = "second"
 fmt.Println(x)
}
```

In fact you can even do this:

```
var x string
x = "first "
fmt.Println(x)
x = x + "second"
fmt.Println(x)
```

This program would be nonsense if you read it like an algebraic theorem. But it makes sense if you are care ful to read the program as a list of commands. When we see `x = x + "second"` we should read it as “assign the concatenation of the value of the variable x and the string literal second to the variable x.” The right side of the `=` is done first and the result is then assigned to the left side of the `=`.

The `x = x + y` form is so common in programming that

Go has a special assignment statement: `+=`. We could have written `x = x + "second"` as `x += "second"` and it would have done the same thing. (Other operators can be used the same way)

Another difference between Go and algebra is that we use a different symbol for equality: `==`. (Two equal signs next to each other) `==` is an operator like `+` and it returns a boolean. For example:

```
var x string = "hello"
 var y string = "world"
 fmt.Println(x == y)
```

This program should print `false` because `hello` is not the same as `world`. On the other hand:

```
var x string = "hello"
 var y string = "hello"
 fmt.Println(x == y)
```

This will print `true` because the two strings are the same.

Since creating a new variable with a starting value is so common Go also supports a shorter statement:

```
x := "Hello World"
```

Notice the `:` before the `=` and that no type was speci fied. The type is not necessary because the Go compiler is able to infer the type based on the literal value you assign the variable. (Since you are assigning a string literal, `x` is given the type `string`) The compiler can also do inference with the `var` statement:

```
var x = "Hello World"
```

The same thing works for other types:

```
x := 5
 fmt.Println(x)
```

Generally you should use this shorter form whenever possible.

4.1 How to Name a Variable

Naming a variable properly is an important part of software development. Names must start with a letter and may contain letters, numbers or the `_` (underscore) symbol. The Go compiler doesn't care what you name a variable so the name is meant for your (and others)

benefit. Pick names which clearly describe the vari able's purpose. Suppose we had the following:

```
x := "Max"
 fmt.Println("My dog's name is", x)
```

In this case x is not a very good name for a variable. A better name would be:

```
name := "Max"
 fmt.Println("My dog's name is", name)
```

or even:

```
dogsName := "Max"
 fmt.Println("My dog's name is", dogsName)
```

In this last case we use a special way to represent mul tiple words in a variable name known as lower camel case (also know as mixed case, bumpy caps, camel back or hump back). The first letter of the first word is lowercase, the first letter of the subsequent words is uppercase and all the other letters are lowercase.

4.2 Scope

Going back to the program we saw at the beginning of the chapter:

```
package main

import "fmt"

func main() {
 var x string = "Hello World"
 fmt.Println(x)
}
```

Another way of writing this program would be like this:

```
package main

import "fmt"

var x string = "Hello World"

func main() {
 fmt.Println(x)
}
```

Notice that we moved the variable outside of the main function. This means that other functions can access this variable:

```
var x string = "Hello World"

func main() {
 fmt.Println(x)
}

func f() {
 fmt.Println(x)
}
```

The `f` function now has access to the `x` variable. Now suppose that we wrote this instead:

```
func main() {
 var x string = "Hello World"
 fmt.Println(x)
}

func f() {
 fmt.Println(x)
}
```

If you run this program you should see an error:

```
.\main.go:11: undefined: x
```

The compiler is telling you that the `x` variable inside of the `f` function doesn't exist. It only exists inside of the `main` function. The range of places where you are al lowed to use `x` is called the *scope* of the variable. Ac-

cording to the language specification “Go is lexically scoped using blocks". Basically this means that the variable exists within the nearest curly braces `{ }` (a block) including any nested curly braces (blocks), but not outside of them. Scope can be a little confusing at first; as we see more Go examples it should become more clear.

4.3 Constants

Go also has support for constants. Constants are basi cally variables whose values cannot be changed later. They are created in the same way you create variables but instead of using the `var` keyword we use the `const` keyword:

```
package main

import "fmt"

func main() {
 const x string = "Hello World"
 fmt.Println(x)
}
```

This:

```
const x string = "Hello World"
 x = "Some other string"
```

Results in a compile-time error:

```
.\main.go:7: cannot assign to x
```

Constants are a good way to reuse common values in a program without writing them out each time. For ex ample `Pi` in the `math` package is defined as a constant.

4.4 Defining Multiple Variables

Go also has another shorthand when you need to de fine multiple variables:

```
var (
 a = 5
 b = 10
 c = 15
)
```

Use the keyword `var` (or `const`) followed by parenthe ses with each variable on its own line.

4.5 An Example Program

Here's an example program which takes in a number entered by the user and doubles it:

```
package main

import "fmt"

func main() {
    fmt.Print("Enter a number: ")
    var input float64
    fmt.Scanf("%f", &input)

    output := input * 2

    fmt.Println(output)
}
```

We use another function from the `fmt` package to read the user input (`Scanf`). `&input` will be explained in a later chapter, for now all we need to know is that `Scanf` fills input with the number we enter.

Problems

1. What are two ways to create a new variable?

2. What is the value of `x` after running: `x := 5; x += 1`?

3. What is scope and how do you determine the scope of a variable in Go?

4. What is the difference between `var` and `const`?

5. Using the example program as a starting point, write a program that converts from Fahrenheit

into Celsius. (`C = (F - 32) * 5/9`)

6. Write another program that converts from feet into meters. (`1 ft = 0.3048 m`)

5 Control Structures

Now that we know how to use variables it's time to start writing some useful programs. First let's write a program that counts to 10, starting from 1, with each number on its own line. Using what we've learned so far we could write this:

```
package main

import "fmt"

func main() {
 fmt.Println(1)
 fmt.Println(2)
 fmt.Println(3)
 fmt.Println(4)
 fmt.Println(5)
 fmt.Println(6)
 fmt.Println(7)
 fmt.Println(8)
 fmt.Println(9)
 fmt.Println(10)
}
```

Or this:

```
package main
import "fmt"

func main() {
 fmt.Println(`1
2
3
4
5
6
7
8
9
10`)
}
```

But both of these programs are pretty tedious to write. What we need is a way of doing something multiple times.

5.1 For

The `for` statement allows us to repeat a list of state ments (a block) multiple times. Rewriting our previous program using a `for` statement looks like this:

```
package main

import "fmt"

func main() {
 i := 1
 for i <= 10 {
 fmt.Println(i)
 i = i + 1
 }
}
```

First we create a variable called `i` that we use to store the number we want to print. Then we create a `for` loop by using the keyword `for`, providing a conditional expression which is either `true` or `false` and finally supplying a block to execute. The for loop works like this:

1. We evaluate (run) the expression `i <= 10` ("i less than or equal to 10"). If this evaluates to true then we run the statements inside of the block. Otherwise we jump to the next line of our pro gram after the block. (in this case there is noth ing after the for loop so we exit the program)

2. After we run the statements inside of the block we loop back to the beginning of the for state ment and repeat step 1.

The `i = i + 1` line is extremely important, because without it `i <= 10` would always evaluate to `true` and our program would never stop. (When this happens this is referred to as an infinite loop)

As an exercise lets walk through the program like a computer would:

- Create a variable named `i` with the value 1
- Is `i <= 10`? Yes.
- Print `i`
- Set `i` to `i + 1` (`i` now equals 2)
- Is `i <= 10`? Yes.
- Print `i`
- Set `i` to `i + 1` (`i` now equals 3)
- …
- Set `i` to `i + 1` (`i` now equals 11)
- Is `i <= 10`? No.
- Nothing left to do, so exit

Other programming languages have a lot of different types of loops (while, do, until, foreach, …) but Go only has one that can be used in a variety of different ways. The previous program could also have been written like this:

```
func main() {
 for i := 1; i <= 10; i++ {
 fmt.Println(i)
 }
}
```

Now the conditional expression also contains two other statements with semicolons between them. First we have the variable initialization, then we have the con dition to check each time and finally we “increment” the variable. (adding 1 to a variable is so common that we have a special operator: `++`. Similarly subtracting 1 can be done with `--`)

We will see additional ways of using the for loop in later chapters.

5.2 If

Let's modify the program we just wrote so that instead of just printing the numbers 1-10 on each line it also specifies whether or not the number is even or odd. Like this:

```
1 odd
2 even
3 odd
4 even
5 odd
6 even
7 odd
8 even
9 odd
10 even
```

First we need a way of determining whether or not a number is even or odd. An easy way to tell is to divide the number by 2. If you have nothing left over then the number is even, otherwise it's odd. So how do we find the remainder after division in Go? We use the `%` oper ator. `1 % 2` equals `1`, `2 % 2` equals `0`, `3 % 2` equals `1` and so on.

Next we need a way of choosing to do different things based on a condition. For that we use the `if` state ment:

```
if i % 2 == 0 {
 // even
} else {
 // odd
}
```

An `if` statement is similar to a `for` statement in that it

has a condition followed by a block. If statements also have an optional `else` part. If the condition evaluates to `true` then the block after the condition is run, other wise either the block is skipped or if the `else` block is present that block is run.

If statements can also have `else if` parts:

```
if i % 2 == 0 {
 // divisible by 2
} else if i % 3 == 0 {
 // divisible by 3
} else if i % 4 == 0 {
 // divisible by 4
}
```

The conditions are checked top down and the first one to result in true will have its associated block exe cuted. None of the other blocks will execute, even if their conditions also pass. (So for example the number 8 is divisible by both 4 and 2, but the `// divisible by 4` block will never execute because the `// divisible by 2` block is done first)

Putting it all together we have:

```
func main() {
 for i := 1; i <= 10; i++ {
      if i % 2 == 0 {
 fmt.Println(i, "even")
 } else {
 fmt.Println(i, "odd")
 }
 }
}
```

Let's walk through this program:

• Create a variable `i` of type `int` and give it the value `1` • Is `i` less than or equal to `10`? Yes: jump to the block

• Is the remainder of `i` ÷ `2` equal to `0`? No: jump to the `else` block

• Print `i` followed by `odd`

• Increment `i` (the statement after the condition) • Is `i` less than or equal to `10`? Yes: jump to the block • Is the remainder of `i` ÷ `2` equal to `0`? Yes: jump to the `if` block

• Print `i` followed by `even`

• …

The remainder operator, while rarely seen outside of elementary school, turns out to be really useful when programming. You'll see it turn up everywhere from zebra striping tables to partitioning data sets.

5.3 Switch

Suppose we wanted to write a program that printed the English names for numbers. Using what we've learned so far we might start by doing this:

```
if i == 0 {
 fmt.Println("Zero")
} else if i == 1 {
 fmt.Println("One")
} else if i == 2 {
 fmt.Println("Two")
} else if i == 3 {
 fmt.Println("Three")
} else if i == 4 {
 fmt.Println("Four")
} else if i == 5 {
 fmt.Println("Five")
}
```

Since writing a program in this way would be pretty tedious Go provides another statement to make this easier: the `switch` statement. We can rewrite our pro gram to look like this:

```
switch i {
case 0: fmt.Println("Zero")
case 1: fmt.Println("One")
case 2: fmt.Println("Two")
case 3: fmt.Println("Three")
case 4: fmt.Println("Four")
case 5: fmt.Println("Five")
default: fmt.Println("Unknown Number")
}
```

A switch statement starts with the keyword `switch` fol lowed by an expression (in this case `i`) and then a se ries of `case`s. The value of the expression is compared to the expression following each `case` keyword. If they are equivalent then the statement(s) following the `:` is executed.

Like an if statement each case is checked top down and the first one to succeed is chosen. A switch also supports a default case which will happen if none of the cases matches the value. (Kind of like the else in an if statement)

These are the main control flow statements.
Additional statements will be explored in later

chapters.

Problems

1. What does the following program print:

```
i := 10
if i > 10 {
 fmt.Println("Big")
} else {
 fmt.Println("Small")
}
```

2. Write a program that prints out all the numbers evenly divisible by 3 between 1 and 100. (3, 6, 9, etc.)

3. Write a program that prints the numbers from 1 to 100. But for multiples of three print "Fizz" in stead of the number and for the multiples of five print "Buzz". For numbers which are multiples of both three and five print "FizzBuzz".

6 Arrays, Slices and Maps

In chapter 3 we learned about Go's basic types. In this chapter we will look at three more built-in types: ar rays, slices and maps.

6.1 Arrays

An array is a numbered sequence of elements of a sin gle type with a fixed length. In Go they look like this:

```
var x [5]int
```

`x` is an example of an array which is composed of 5 `int`s. Try running the following program:

```
package main

import "fmt"

func main() {
    var x [5]int
    x[4] = 100
    fmt.Println(x)
}
```

You should see this:

```
[0 0 0 0 100]
```

`x[4] = 100` should be read "set the 5th element of the array x to 100". It might seem strange that `x[4]` repre sents the 5th element instead of the 4th but like strings, arrays are indexed starting from 0. Arrays are ac cessed in a similar way. We could change `fmt.Println(x)` to `fmt.Println(x[4])` and we would get 100.

Here's an example program that uses arrays:

```
func main() {
     var x [5]float64
     x[0] = 98
     x[1] = 93
     x[2] = 77
     x[3] = 82
     x[4] = 83

     var total float64 = 0
     for i := 0; i < 5; i++ {
          total += x[i]
     }
     fmt.Println(total / 5)
}
```

This program computes the average of a series of test scores. If you run it you should see `86.6`. Let's walk through the program:

• First we create an array of length 5 to hold our test scores, then we fill up each element with a grade • Next we setup a for loop to compute the total score • Finally we divide the total score by the number of el ements to find the average

This program works, but Go provides some features we can use to improve it. First these 2 parts: `i < 5` and `total / 5` should throw up a red flag for us. Say we

changed the number of grades from 5 to 6. We would also need to change both of these parts. It would be better to use the length of the array instead:

```
    var total float64 = 0
    for i := 0; i < len(x); i++ {
          total += x[i]
    }
    fmt.Println(total / len(x))
```

Go ahead and make these changes and run the pro gram. You should get an error:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:19: invalid operation: total /
5  (mismatched types float64 and int)
```

The issue here is that `len(x)` and `total` have different types. `total` is a `float64` while `len(x)` is an `int`. So we need to convert `len(x)` into a `float64`:

```
fmt.Println(total / float64(len(x)))
```

This is an example of a type conversion. In general to convert between types you use the type name like a function.

Another change to the program we can make is to use a special form of the for loop:

```
var total float64 = 0
for i, value := range x {
     total += value
}
fmt.Println(total / float64(len(x)))
```

In this for loop `i` represents the current position in the array and `value` is the same as `x[i]`. We use the key word `range` followed by the name of the variable we want to loop over.

Running this program will result in another error:

```
$ go run tmp.go
# command-line-arguments
.\tmp.go:16: i declared and not used
```

The Go compiler won't allow you to create variables that you never use. Since we don't use `i` inside of our loop we need to change it to this:

```
var total float64 = 0
for _, value := range x {
     total += value
}
fmt.Println(total / float64(len(x)))
```

A single _ (underscore) is used to tell the compiler that we don't need this. (In this case we don't need the iter ator variable)

Go also provides a shorter syntax for creating arrays:

```
x := [5]float64{ 98, 93, 77, 82, 83 }
```

We no longer need to specify the type because Go can figure it out. Sometimes arrays like this can get too long to fit on one line, so Go allows you to break it up like this:

```
x := [5]float64{
    98,
    93,
    77,
    82,
    83,
}
```

Notice the extra trailing , after 83. This is required by

Go and it allows us to easily remove an element from the array by commenting out the line:

```
x := [4]float64{
    98,
    93,
    77,
    82,
    // 83,
}
```

This example illustrates a major issue with arrays: their length is fixed and part of the array's type name. In order to remove the last item, we actually had to change the type as well. Go's solution to this problem is to use a different type: slices.

6.2 Slices

A slice is a segment of an array. Like arrays slices are indexable and have a length. Unlike arrays this length is allowed to change. Here's an example of a slice:

```
var x []float64
```

The only difference between this and an array is the missing length between the brackets. In this case `x` has been created with a length of `0`.

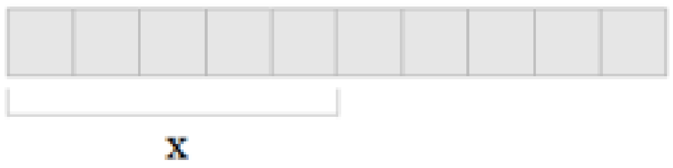
If you want to create a slice you should use the built-in `make` function:

```
x := make([]float64, 5)
```

This creates a slice that is associated with an underly ing `float64` array of length 5. Slices are always associ ated with some array, and although they can never be longer than the array, they can be smaller. The `make` function also allows a 3rd parameter:

```
x := make([]float64, 5, 10)
```

10 represents the capacity of the underlying array which the slice points to:



Another way to create slices is to use the `[low : high]` expression:

```
arr := []float64{1,2,3,4,5}
x := arr[0:5]
```

`low` is the index of where to start the slice and `high` is the index where to end it (but not including the index itself). For example while `arr[0:5]` returns `[1,2,3,4,5]`, `arr[1:4]` returns `[2,3,4]`.

For convenience we are also allowed to omit `low`, `high` or even both `low` and `high`. `arr[0:]` is the same as `arr[0:len(arr)]`, `arr[:5]` is the same as `arr[0:5]` and `arr[:]` is the same as `arr[0:len(arr)]`.

Slice Functions

Go includes two built-in functions to assist with slices: `append` and `copy`. Here is an example of `append`:

```
func main() {
     slice1 := []int{1,2,3}
     slice2 := append(slice1, 4, 5)
     fmt.Println(slice1, slice2)
}
```

After running this program `slice1` has `[1,2,3]` and `slice2` has `[1,2,3,4,5]`. `append` creates a new slice by taking an existing slice (the first argument) and ap pending all the following arguments to it.

Here is an example of copy:

```
func main() {
     slice1 := []int{1,2,3}
     slice2 := make([]int, 2)
     copy(slice2, slice1)
     fmt.Println(slice1, slice2)
}
```

After running this program `slice1` has `[1,2,3]` and slice2 has [1,2]. The contents of slice1 are copied into slice2, but since slice2 has room for only two ele ments only the first two elements of slice1 are copied.

6.3 Maps

A map is an unordered collection of key-value pairs. Also known as an associative array, a hash table or a dictionary, maps are used to look up a value by its as sociated key. Here's an example of a map in Go:

```
var x map[string]int
```

The map type is represented by the keyword map, fol lowed by the key type in brackets and finally the value type. If you were to read this out loud you would say "x is a map of strings to ints."

Like arrays and slices maps can be accessed using brackets. Try running the following program:

```
var x map[string]int
x["key"] = 10
fmt.Println(x)
```

You should see an error similar to this:

```
panic: runtime error: assignment to entry in
nil map

goroutine 1 [running]:
main.main()
 main.go:7 +0x4d

goroutine 2 [syscall]:
created by runtime.main

C:/Users/ADMINI~1/AppData/Local/Temp/2/bin
di
t269497170/go/src/pkg/runtime/proc.c:221
exit status 2
```

Up till now we have only seen compile-time errors. This is an example of a runtime error. As the name would imply, runtime errors happen when you run the program, while compile-time errors happen when you try to compile the program.

The problem with our program is that maps have to be initialized before they can be used. We should have written this:

```
x := make(map[string]int)
x["key"] = 10
fmt.Println(x["key"])
```

If you run this program you should see `10` displayed. The statement `x["key"] = 10` is similar to what we saw with arrays but the key, instead of being an inte ger, is a string because the map's key type is `string`. We can also create maps with a key type of `int`:

```
x := make(map[int]int)
x[1] = 10
fmt.Println(x[1])
```

This looks very much like an array but there are a few differences. First the length of a map (found by doing `len(x)`) can change as we add new items to it. When first created it has a length of 0, after `x[1] = 10` it has a length of 1. Second maps are not sequential. We have `x[1]`, and with an array that would imply there must be an `x[0]`, but maps don't have this requirement.

We can also delete items from a map using the built-in `delete` function:

```
delete(x, 1)
```

Let's look at an example

program that uses a map:

```
package main

import "fmt"

func main() {
     elements := make(map[string]string)
     elements["H"] = "Hydrogen"
     elements["He"] = "Helium"
     elements["Li"] = "Lithium"
     elements["Be"] = "Beryllium"
     elements["B"] = "Boron"
     elements["C"] = "Carbon"
     elements["N"] = "Nitrogen"
     elements["O"] = "Oxygen"
     elements["F"] = "Fluorine"
     elements["Ne"] = "Neon"

     fmt.Println(elements["Li"])
}
```

`elements` is a map that represents the first 10 chemical elements indexed by their symbol. This is a very com mon way of using maps: as a lookup table or a dictio nary. Suppose we tried to look up an element that doesn't exist:

```
fmt.Println(elements["Un"])
```

If you run this you should see nothing returned. Tech-

nically a map returns the zero value for the value type (which for strings is the empty string). Although we could check for the zero value in a condition (`elements["Un"] == ""`) Go provides a better way:

```
name, ok := elements["Un"]
fmt.Println(name, ok)
```

Accessing an element of a map can return two values instead of just one. The first value is the result of the lookup, the second tells us whether or not the lookup was successful. In Go we often see code like this:

```
if name, ok := elements["Un"]; ok {
    fmt.Println(name, ok)
}
```

First we try to get the value from the map, then if it's successful we run the code inside of the block.

Like we saw with arrays there is also a shorter way to create maps:

```
elements := map[string]string{
     "H": "Hydrogen",
     "He": "Helium",
     "Li": "Lithium",
     "Be": "Beryllium",
     "B": "Boron",
     "C": "Carbon",
     "N": "Nitrogen",
     "O": "Oxygen",
     "F": "Fluorine",
     "Ne": "Neon",
}
```

Maps are also often used to store general information. Let's modify our program so that instead of just storing the name of the element we store its standard state (state at room temperature) as well:

```
func main() {
                    elements :=
        map[string]map[string]string{ "H":
                map[string]string{
                 "name":"Hydrogen",
                  "state":"gas",
            },
            "He": map[string]string{
                 "name":"Helium",
                  "state":"gas",
            },
            "Li": map[string]string{
                 "name":"Lithium",
                "state":"solid",
            },
            "Be": map[string]string{
                 "name":"Beryllium",
                "state":"solid",
            },
            "B": map[string]string{
                 "name":"Boron",
                "state":"solid",
            },
            "C": map[string]string{
                "name":"Carbon",
                "state":"solid",
            },
            "N": map[string]string{
                "name":"Nitrogen",
                  "state":"gas",
            },
            "O": map[string]string{
                "name":"Oxygen",
                  "state":"gas",
            },
            "F": map[string]string{
                "name":"Fluorine",
                  "state":"gas",
            },
            "Ne": map[string]string{
                  "name":"Neon",
                  "state":"gas",
            },
      }

      if el, ok := elements["Li"]; ok {
          fmt.Println(el["name"], el["state"])
      }
  }
```

Notice that the type of our map has changed from `map[string]string` to `map[string]map[string]string`. We now have a map of strings to maps of strings to strings. The outer map is used as a lookup table based on the element's symbol, while the inner maps are used to store general information about the elements. Although maps are often used like this, in chapter 9 we will see a better way to store structured informa tion.

Problems

1. How do you access the 4th element of an array or slice?

2. What is the length of a slice created using: `make([]int, 3, 9)`?

3. Given the array:

```
x := [6]string{"a","b","c","d","e","f"}
```

   what would `x[2:5]` give you?

4. Write a program that finds the smallest number in this list:

```
x := []int{
 48,96,86,68,
 57,82,63,70,
 37,34,83,27,
 19,97, 9,17,
}
```

7 Functions

A function is an independent section of code that maps zero or more input parameters to zero or more output parameters. Functions (also known as procedures or subroutines) are often represented as a black box: (the black box represents the function)

Inputs
Outputs

Until now the programs we have written in Go have used only one function:

```
func main() {}
```

We will now begin writing programs that use more than one function.