DARKER CORNERS OF GO

RYTIS BIELIUNAS

Table of Contents

Introduction

When I was learning Go first, I read some introduction book and a language specification and I already knew several other programming languages and yet after all that it felt like I really don't know enough about Go to do real world work. I felt I don't have a deep enough knowledge of how things are done in Go-land and I'd probably need to fall into many traps before I can feel confident with it.

I was right.

While simplicity is at the core of Go philosophy you'll find it nevertheless enables numerous creative ways of shooting yourself in the foot.

Since I've now used Go to create production applications for several years and on the account of the many holes in my feet I thought I'd put together a text for the fellow noob students of Go.

My goal is to collect in one place various things in Go that might be surprising to new developers, and perhaps shed some light on the more unusual features of Go. I hope that would save the reader lots of Googling and debugging time, and possibly prevent some expensive bugs.

I think this text will be most useful if you already know at least Go syntax. It would be best if you're an intermediate or experienced programmer who already knows other programming languages and are looking to learn Go.

If you find errors or I didn't include your favorite Go-surprise please let me know at rytbiel@gmail.com

Big thanks and karma points to Vytautas Shaltenis and Jon Forrest for helping make this book better.

Chapter 1: Code formatting

gofmt

Much of Go code formatting is forced upon you by the gofmt tool. Gofmt makes automatic changes to source files, such as sorting import declarations and applying indentation. It's the best thing since sliced bread as it saves man-years of developers arguing about things that hardly matter. For instance, it uses tabs for indentation, spaces for alignment, and that’s the end of that story.

You are free to not use gofmt tool at all, but if you do use it you can't configure it to a particular formatting style. The tool provides zero code formatting options, and that's the point - to provide one “good enough” uniform formatting style. It might be nobody’s favorite style, but Go developers decided that uniformity is better than perfection.

Some benefits of shared style and automatic code formatting are:

- No time spent in code reviews on formatting issues.
- It saves you from colleagues with OCD about where the opening brace should go or what should be used for indentation. All that passion and energy can be used more productively.
- Code is easier to write: minor formatting details are sorted for you.
- Code is easier to read: you don’t need to mentally parse someone else’s unfamiliar formatting style.

Most popular IDEs have plugins for Go that run gofmt automatically when saving a source file.

Third party tools such as goformat allow custom code style formatting in Go. If you really must have that.

Long lines

Gofmt will not try to break up long lines of code. There are third party tools such as golines that do that.

Opening bracket

An opening bracket must be placed at the end of the line in Go. Interestingly this is not enforced by gofmt, but rather is a side effect of how Go lexical analyzer is implemented. With or without gofmt an opening bracket can't be placed on a new line:

```
package main

// missing function body
func main()
// syntax error: unexpected semicolon or newline
before {
{
}

// all good!
func main() {
}
```

Commas in multi-line declarations

Go requires a comma before a new line when initializing a slice, array, map, or struct. Trailing commas are allowed in many languages and encouraged in some style guides. In Go they're mandatory. That way lines can be rearranged or new lines added without modifying unrelated lines. This means less noise in code review diffs:

```
// all of these are OK
a := []int{1, 2}

b := []int{1, 2,}
```

```
c := []int{
    1,
    2}

d := []int{
    1,
    2,
}

// syntax error without trailing comma
e := []int{
    1,
    // syntax error: unexpected newline,
expecting comma or }
    2
}
```

Same thing with structs:

```
type s struct {
    One int
    Two int
}

f := s{
    One: 1,
    // syntax error: unexpected newline,
expecting comma or }
    Two: 2
}
```

Chapter 2: Imports

Unused imports

Go programs with unused imports don't compile. This is a deliberate feature of the language since importing packages slows down the compiler. In large programs unused imports could make a significant impact on compilation time.

To keep the compiler happy during development you can reference the package in some way:

```
package main

import (
    "fmt"
    "math"
)

// Reference unused package
var _ = math.Round
func main() {
    fmt.Println("Hello")
}
```

Goimports

A better solution is to use the goimports tool. Goimports removes unreferenced imports. What's even better is that it tries to automatically find and add missing ones:

```
package main

import "math" // imported and not used: "math"

func main() {
```

```
    fmt.Println("Hello") // undefined: fmt
}
```

After running goimports:

```
./goimports main.go
```

```
package main

import "fmt"

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

Go plugins for most popular IDEs run goimports automatically when saving a source file.

Underscore imports

Underscore imports reference a package solely for the package's side effects. This means the package creates package-level variables and runs package [init functions](#):

```
package package1

func package1Function() int {
    fmt.Println("Package 1 side-effect")
    return 1
}

var globalVariable = package1Function()
func init() {
    fmt.Println("Package 1 init side effect")
}
```

And in package2:

```
package package2

import _ package1
```

This prints the messages and initializes globalVariable:

```
Package 1 side-effect

Package 1 init side effect
```

Importing a package multiple times (e.g. in the main package and also in other packages that main references) only runs init functions once.

Underscore imports are used in Go runtime libraries. For instance, importing net/http/pprof calls its init function to expose HTTP endpoints that can provide debugging information:

```
import _ "net/http/pprof"
```

Dot imports

Dot imports allow identifiers in the imported package to be accessed without a qualifier:

```
package main

import (
    "fmt"
    . "math"
)

func main() {
    fmt.Println(Sin(3)) // references math.Sin
}
```

There's an open debate if dot imports should be removed from the language. The Go team does not recommend using them anywhere except in test packages:

> *It makes the programs much harder to read because it is unclear whether a name like Quux is a top-level identifier in the current package or in an imported package.*
>
> *https://golang.org/doc/faq*

Also, if you use the go-lint tool it shows a warning when a dot import is used outside of test files and you can't turn it off easily.

One use case that the Go team does recommend is in tests that can't be made part of the package being tested because of circular dependencies:

```
// foo_test package tests for foo package
package foo_test
import (
    "bar/testutil" // also imports "foo"
    . "foo"
)
```

This test file can't be part of the foo package, because it references bar/testutil, which in turn references foo. This would create a circular dependency.

The first thing to consider in this case is if perhaps there's a better way to structure the packages to avoid the circular dependency. It may or may not make sense to move what bar/testutil uses from foo to a third package that both foo and bar/testutil could import. That would then allow tests to be written as normal in the foo package.

If restructuring doesn't make sense and tests are moved to a separate package with a dot import, the foo_test package can at least pretend to be part of the foo package. With the caveat that it can't access unexported types and functions from the foo package.

Arguably, there is a good use case for dot imports in the domain specific languages. For instance the Goa framework uses it for configuration. It wouldn't look great without the dot import:

```
package design

import . "goa.design/goa/v3/dsl"

// API describes the global properties of the API
server.
var _ = API("calc", func() {
    Title("Calculator Service")
    Description("HTTP service for adding numbers,
a goa teaser")
    Server("calc", func() {
        Host("localhost", func() {
URI("http://localhost:8088") })
    })
})
```

Chapter 3: Variables

Unused variables

Go programs with unused variables do not compile:

> *The presence of an unused variable may indicate a bug [...] Go refuses to compile programs with unused variables or imports, trading short-term convenience for long-term build speed and program clarity.*
>
> *https://golang.org/doc/faq*

Exceptions to that rule are global variables and function arguments:

```
package main

var unusedGlobal int // this is ok

func f1(unusedArg int) { // unused function
arguments are also ok
    // error: a declared but not used
    a, b := 1,2
    // b is used here, but a is only assigned to,
doesn't count as “used”
    a = b
}
```

Short variable declaration

Short variable declarations only work inside functions:

```
package main

v1 := 1 // error: non-declaration statement
outside function body
```

```
var v2 = 2 // this is ok

func main() {
    v3 := 3 // this is ok
    fmt.Println(v3)
}
```

They also don't work when setting field values:

```
package main

type myStruct struct {
    Field int
}

func main() {
    var s myStruct
    // error: non-name s.Field on the left side
of :=
    s.Field, newVar := 1, 2
    var newVar int
    s.Field, newVar = 1, 2 // this is actually ok
}
```

Variable shadowing

Sadly, variable shadowing is allowed in Go. It is something you need to constantly watch out for as it can cause issues that are hard to spot. This happens because as a convenience Go allows using short variable declaration if at least one of the variables is new:

```
package main

import "fmt"
```

```
func main() {
    v1 := 1
    // v1 isn't actually redeclared here, only
gets a new value set
    v1, v2 := 2, 3
    fmt.Println(v1, v2) // prints 2, 3
}
```

If however the declaration is inside another code block, it would declare a new variable, potentially causing serious bugs:

```
package main

import "fmt"

func main() {
    v1 := 1
    if v1 == 1 {
        v1, v2 := 2, 3
        fmt.Println(v1, v2) // prints 2, 3
    }
    fmt.Println(v1) // prints 1 !
}
```

For a more realistic example, let's say you have a function that returns an error:

```
package main

import (
    "errors"
    "fmt"
)
```

```
func func1() error {
   return nil
}

func errFunc1() (int, error) {
   return 1, errors.New("important error")
}

func returnsErr() error {
    err := func1()
    if err == nil {
        v1, err := errFunc1()
        if err != nil {
            fmt.Println(v1, err) // prints: 1
important error
        }
    }
    return err // this returns nil!
}

func main() {
    fmt.Println(returnsErr()) // prints nil
}
```

One solution to this would be to not use short variable declarations inside of nested code blocks:

```
func returnsErr() error {
    err := func1()
    var v1 int
    if err == nil {
```

```
        v1, err = errFunc1()
        if err != nil {
            fmt.Println(v1, err) // prints: 1
important error
        }
    }

    return err // returns "important error"
}
```

Or, in the example above, even better would be to exit early:

```
func returnsErr() error {
    err := func1()
    if err != nil {
        return err
    }

    v1, err := errFunc1()
    if err != nil {
        fmt.Println(v1, err) // prints: 1
important error
        return err
    }

    return nil
}
```

There are also tools that can help. There was experimental shadowed variable detection in go vet tool but it was removed. At the time of writing this is how to install and run the tool:

```
go get -u
golang.org/x/tools/go/analysis/passes/shadow/cmd/
shadow

go vet -vettool=$(which shadow)
```

Prints:

```
.\main.go:20:7: declaration of "err" shadows
declaration at line 17
```

Chapter 4: Operators

Operator precedence

Go operator precedence is different than in other languages:

```
Precedence   Operator
5            *, /, %, <<, >>, &, &^
4            +, -, | , ^
3            ==, !=, <, <=, >, >=
2            &&
1            ||
```

Compare it to C based languages:

```
Precedence   Operator
10           *, /, %
9            +, -
8            <<, >>
7            <, <=, >, >=
6            ==, !=
5            &
4            ^
3            |
2            &&
1            ||
```

This can lead to different results for the same expression:

```
In Go: 1 << 1 + 1 // (1<<1)+1 = 3
In C: 1 << 1 + 1 // 1<<(1+1) = 4
```

Increment and decrement

Unlike many other languages Go doesn't have prefix increment or decrement operators:

```
var i int
++i // syntax error: unexpected ++, expecting }
--i // syntax error: unexpected --, expecting }
```

While Go does have postfix versions of these operators they're not allowed in expressions:

```
slice := []int{1,2,3}
i := 1
slice[i++] = 0 // syntax error: unexpected ++,
expecting:
```

Ternary operator

One of things commonly missed in Go is a ternary operator:

```
result := a ? b : c
```

There is none, don’t look for it. You must use if-else instead. Go language designers decided this operator often results in ugly code and it's better not to have it at all.

Bitwise NOT

In Go the XOR operator ^ is used as a unary NOT operator instead of the ~ symbol like in many other languages.

```
In Go: ^1 // -2
In C: ~1 // -2
```

The binary XOR operator is still used as the XOR operator.

```
3^1 // 2
```

Chapter 5: Constants

iota

iota starts constant numbering in Go. It doesn't mean “start from zero” as someone might expect. It is an index of a constant in the current const block:

```
const (
    myconst = "c"
    myconst2 = "c2"
    two = iota // 2
)
```

Using iota twice doesn't reset the numbering:

```
const (
    zero = iota // 0
    one // 1
    two = iota // 2
)
```

Chapter 6: Slices and Arrays

Slices and Arrays

In Go slices and arrays serve a similar purpose. They are declared nearly the same way:

```
package main

import "fmt"

func main() {
    slice := []int{1, 2, 3}
    array := [3]int{1, 2, 3}
    // let the compiler work out array length
    // this will be an equivalent of [3]int
    array2 := [...]int{1, 2, 3}
    fmt.Println(slice, array, array2)
}
```

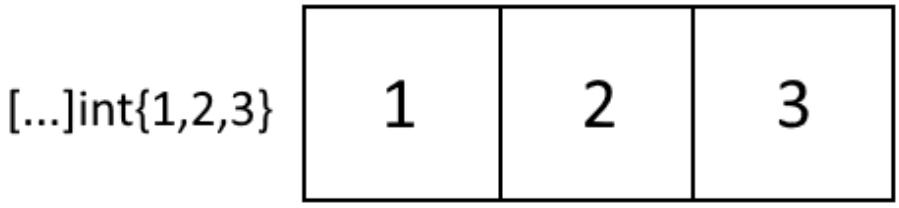
```
[1 2 3] [1 2 3] [1 2 3]
```

Slices feel like arrays with useful functionality on top. They use pointers to arrays internally in their implementation. Slices however are so much more convenient that arrays are rarely used directly in Go.

Arrays

An array is a typed sequence of memory of a fixed length. Arrays of different length are considered to be different incompatible types.
Unlike in C, array elements are initialized to zero values when an array is created so there's no need to do that explicitly. Also unlike in C a Go array is a value type. It's not a pointer to the first element of a

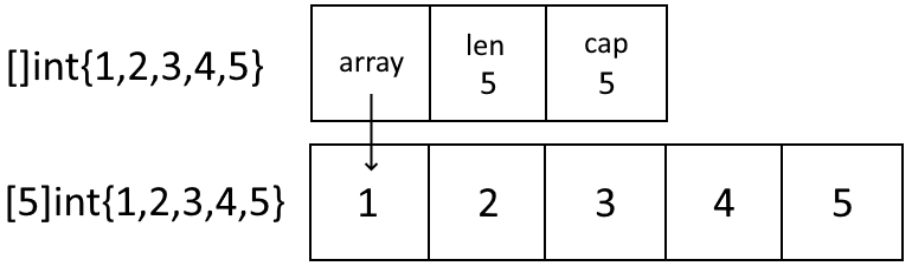
block of memory. If you pass an array into a function, the whole array will be copied. You can still pass a pointer to an array to not have it copied.



Slices

A slice is a descriptor of an array segment. It's a very useful data structure, but perhaps slightly unusual. There are several ways to shoot yourself in a foot with it, all of which can be avoided if you know how slice works internally. Here's the actual definition of a slice in Go source code:

```
type slice struct {
    array unsafe.Pointer
    len   int
    cap   int
}
```

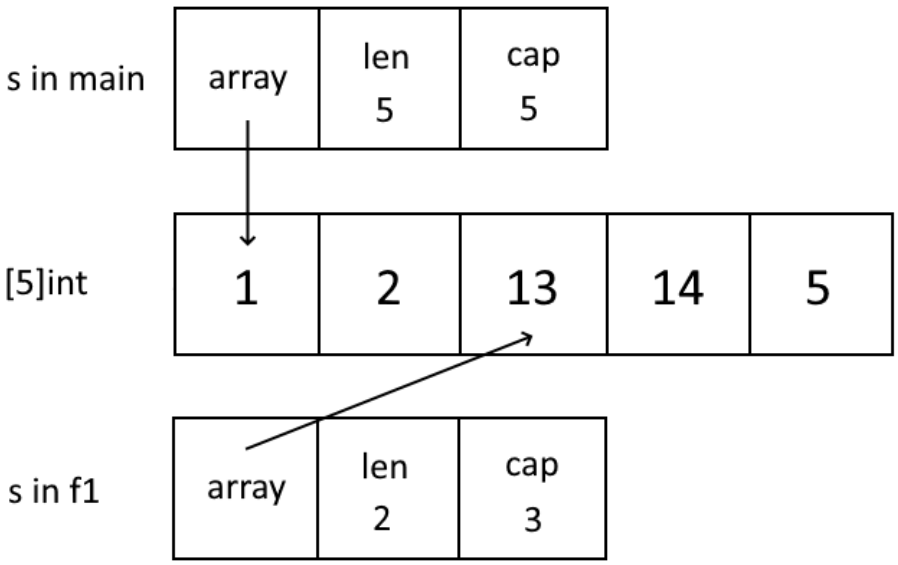


This has interesting implications. A slice itself is a value type, but it references the array it uses with a pointer. Unlike with an array, if you pass a slice to a function you would get a copy of array pointer, len, and cap properties (the first block in the image above), but the data in the array itself wouldn't be copied. Both copies of the slice would

point to the same array. The same thing happens when you “slice” a slice. Slicing creates a new slice, which still points to the same array:

```
package main

import "fmt"

func f1(s []int) {
    // slicing the slice creates a new slice
    // but does not copy the array data
    s = s[2:4]
    // modifying the sub-slice
    // changes the array of slice in main
function as well
    for i := range s {
        s[i] += 10
    }
    fmt.Println("f1", s, len(s), cap(s))
}

func main() {
    s := []int{1, 2, 3, 4, 5}
    // passing a slice as an argument
    // makes a copy of the slice properties
(pointer, len and cap)
    // but the copy shares the same array
    f1(s)
    fmt.Println("main", s, len(s), cap(s))
}
```

```
f1 [13 14] 2 3

main [1 2 13 14 5] 5 5
```



If you're unaware of what a slice is you might assume that it's a value type, and be surprised that f1 "corrupted" the data in the slice in main.

Getting a copy of a slice including its data

To get a copy of a slice with its data you need to do a bit of work. You could copy the elements manually to a new slice or use copy or append:

```
package main

import "fmt"

func f1(s []int) {
    s = s[2:4]
    s2 := make([]int, len(s))
    copy(s2, s)
```

```
    // or if you prefer less efficient, but more
concise version:
    // s2 := append([]int{}, s[2:4]...)
    for i := range s2 {
        s2[i] += 10
    }

    fmt.Println("f1", s2, len(s2), cap(s2))
}

func main() {
    s := []int{1, 2, 3, 4, 5}
    f1(s)
    fmt.Println("main", s, len(s), cap(s))
}
```

```
f1 [13 14] 2 3

main [1 2 3 4 5] 5 5
```

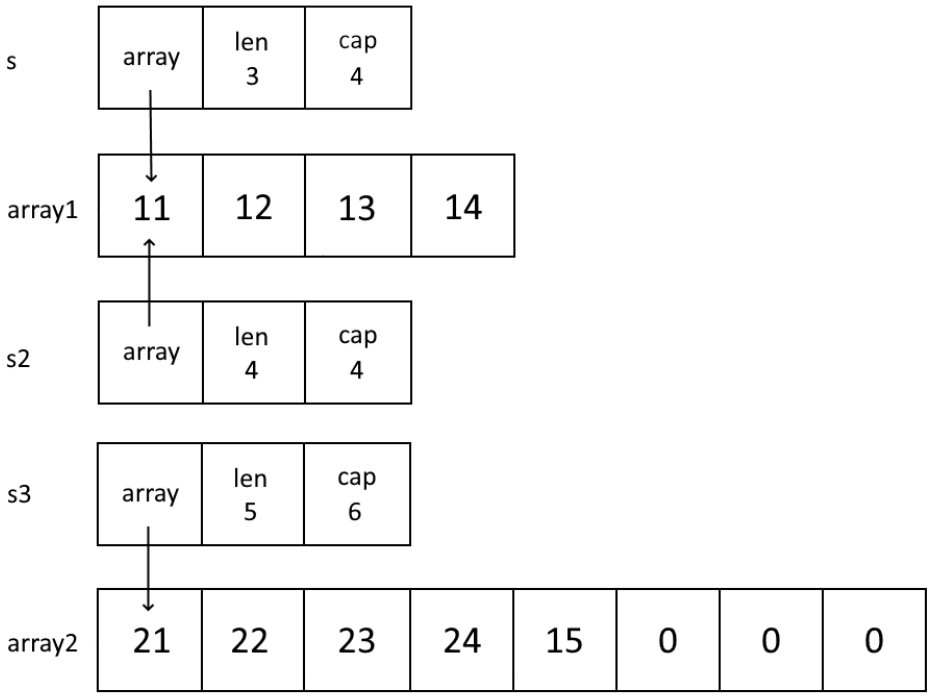
Growing slices with append

So messing with slices does things to pointer, len, and cap values, and all copies of a slice share the same array. Until they don't. The most useful property of a slice is that it manages growing the array. When it needs to grow past the capacity of the existing array, an entirely new array needs to be allocated. If you expected your two copies of a slice to share the array this could also be a trap:

```
package main

import "fmt"
```

```
func main() {
    // make a slice with length 3 and capacity 4
    s := make([]int, 3, 4)

    // initialize to 1,2,3
    s[0] = 1
    s[1] = 2
    s[2] = 3

    // capacity of the array is 4
    // adding one more number fits in the initial
array
    s2 := append(s, 4)

    // modify the elements of the array
    // s and s2 still share the same array
    for i := range s2 {
        s2[i] += 10
    }

    fmt.Println(s, len(s), cap(s))    // [11 12
13] 3 4
    fmt.Println(s2, len(s2), cap(s2)) // [11 12
13 14] 4 4

    // this append grows the array past its
capacity
    // new array must be allocated for s3
    s3 := append(s2, 5)
```

```
    // modify the elements of the array to see
the result
    for i := range s3 {
        s3[i] += 10
    }

    fmt.Println(s, len(s), cap(s)) // still the
old array [11 12 13] 3 4
    fmt.Println(s2, len(s2), cap(s2)) // the old
array [11 12 13 14] 4 4

    // array was copied on last append [21 22 23
24 15] 5 8
    fmt.Println(s3, len(s3), cap(s3))
}
```



nil slices

Slices don't have to be checked for nil values and don't always need to be initialized. Functions such as len, cap, and append work fine on a nil slice:

```
package main

import "fmt"

func main() {
    var s []int // nil slice
    fmt.Println(s, len(s), cap(s)) // [] 0 0
    s = append(s, 1)
    fmt.Println(s, len(s), cap(s)) // [1] 1 1
}
```

An empty slice is not the same thing as a nil slice:

```
package main

import "fmt"

func main() {
    var s []int // this is a nil slice
    s2 := []int{} // this is an empty slice

    // looks like the same thing here:
    fmt.Println(s, len(s), cap(s)) // [] 0 0
    fmt.Println(s2, len(s2), cap(s2)) // [] 0 0

    // but s2 is actually allocated somewhere
    fmt.Printf("%p %p", s, s2) // 0x0 0x65ca90
}
```

If you care very much about performance, memory usage, and such, initializing an empty slice might be less ideal than using a nil slice.

Make trap-ish

To create a new slice you can use make with a slice type, initial length, and capacity of the slice. The capacity parameter is optional:

```
func make([]T, len, cap) []T
```

It's a bit too easy to do this:

```
package main

import (
    "fmt"
)

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

```
[0 0 0 1 2 3]
```

Nah this would never happen to me. I know that the second argument of make for a slice is length, not capacity, I hear you say…

Unused slice array data

Because slicing an array creates a new slice but shares the underlying array, it's possible to keep around a lot more data in memory then you might want or expect. Here's a silly example:

```
package main
```

```
import (
    "bytes"
    "fmt"
    "io/ioutil"
    "os"
)
func getExecutableFormat() []byte {
    // read our own executable file into memory
    bytes, err := ioutil.ReadFile(os.Args[0])
    if err != nil {
        panic(err)
    }
    return bytes[:4]
}

func main() {
    format := getExecutableFormat()
    if bytes.HasPrefix(format, []byte("ELF")) {
        fmt.Println("linux executable")
    } else if bytes.HasPrefix(format,
[]byte("MZ")) {
        fmt.Println("windows executable")
    }
}
```

In the code above, as long as that format variable is in scope and can't be garbage collected, the whole executable (probably few megabytes of data) will be kept in memory. To fix it make a copy of the bytes that are actually needed.

Multidimensional slices

There's no such thing in Go at this point. There might be some day, but at the moment you either need to manually use a single dimension slice as multidimensional slice by working out the element indexes yourself or use “jagged” slices (a jagged slice is a slice of slices):

```
package main

import "fmt"

func main() {
    x := 2
    y := 3
    s := make([][]int, y)
    for i := range s {
        s[i] = make([]int, x)
    }
    fmt.Println(s)
}
```

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

Chapter 7: Strings and Byte Arrays

Strings in Go

Internally a Go string is defined like this:

```
type StringHeader struct {
    Data uintptr
    Len  int
}
```

A string itself is a value type, it has a pointer to a byte array and a fixed length. Zero byte in a string doesn't mark the end of the string as it does in C. There can be any data inside of a string. Usually that data is encoded as a UTF-8 string, but it doesn’t have to be.

Strings can't be nil

Strings are never nil in Go. The default value of a string is an empty string, not nil:

```
package main

import "fmt"

func main() {
    var s string
    fmt.Println(s == "") // true

    s = nil // error: cannot use nil as type
string in assignment
}
```

Strings are immutable (sort of)

Go does not want you to modify strings:

```
package main

func main() {
    str := "darkercorners"
    str[0] = 'D' // error: cannot assign to
str[0]
}
```

Immutable data is easier to reason about and thus creates fewer gotchas. The downside is that every time you want to add or remove something from a string a brand new string has to be allocated. If you really want to, it’s possible to modify a string through the unsafe package, but if you find yourself going that way you're likely being too clever.

The most common case when you might want to worry about allocations is when many strings need to be added together. There's a strings.Builder type for that purpose that allocates memory in batches rather than every time when adding a string:

```
package main

import (
    "strconv"
    "strings"
    "testing"
)

func BenchmarkString(b *testing.B) {
    var str string
    for i := 0; i < b.N; i++ {
        str += strconv.Itoa(i)
    }
}
```

```
func BenchmarkStringBuilder(b *testing.B) {
    var str strings.Builder
    for i := 0; i < b.N; i++ {
        str.WriteString(strconv.Itoa(i))
    }
}
```

```
BenchmarkString-8 401053 147346 ns/op 1108686
B/op 2 allocs/op

BenchmarkStringBuilder-8 29307392 44.9 ns/op 52
B/op 0 allocs/op
```

In this example using strings.Builder is 3000 times faster than simply adding strings (and allocating new memory each time).

There are some cases when the Go compiler will optimize out these allocations:

1. When comparing a string to a byte slice: str == string(byteSlice)
2. When []byte keys are used to lookup entries in map[string]: m[string(byteSlice)]
3. In range clauses where a string is converted to bytes: for i, v := range []byte(str) {...}

New versions of the Go compiler will likely add more optimizations so if performance is important it's always best to use benchmarks and the profiler.

string vs. []byte

One way to modify a string is to first convert it to a byte slice and then back to a string. As shown in the example below converting a

string to a byte slice and back copies the whole string and byte slice. The original string isn't changed:

```
package main

import (
    "fmt"
)

func main() {
    str := "darkercorners"
    bytes := []byte(str)

    bytes[0] = 'D'

    str2 := string(bytes)

    bytes[6] = 'C'

    // prints: darkercorners Darkercorners DarkerCorners
    fmt.Println(str, str2, string(bytes))
}
```

Using unsafe package it's possible (but apparently unsafe) to modify the string directly without allocating memory.

> *Packages that import unsafe may be non-portable and are not protected by the Go 1 compatibility guidelines.*
>
> *https://golang.org/pkg/unsafe/*

```
package main

import (
    "fmt"
```

```
    "unsafe"
)

func main() {
    buf := []byte("darkercorners")
    buf[0] = 'D'

    // make a string that points to the same data
as buf byte slice
    str := *(*string)(unsafe.Pointer(&buf))

    // modifying byte slice
    // it now points to the same memory as the
string does
    // str is modified here as well
    buf[6] = 'C'

    // DarkerCorners DarkerCorners
    fmt.Println(str, string(buf))
}
```

UTF-8 things

Unicode and UTF-8 are hairy subjects. To learn about how Unicode and UTF-8 work in general you might want to read Joel Spolsky's blog post The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!).

As a short recap:

1. Unicode is “an international encoding standard for use with different languages and scripts, by which each letter, digit, or symbol is assigned a unique numeric value that applies across different platforms and programs”. Essentially it’s a

big table of "code points". It contains most (but not all) the characters of all languages. Each code point is an index in that table which you can sometimes see specified with the U+ notation such as U+0041 for letter A.

2. Usually code point means a character, for instance the Chinese character 竜 (U+2EEF), but it can be a geometric shape or a character modifier (such as an umlaut for letters like German ä, ö, and ü). For some reason, it can even be a poo icon (U+1F4A9).
3. UTF-8 is one of the ways (and the most common one) to encode elements of that big Unicode table into actual bytes that computers can work with.
4. A single Unicode code point can take between 1 and 4 bytes when encoded in UTF-8.
5. Numbers and Latin letters (a-z, A-Z, 0-9) are encoded in 1 byte. Letters of many other languages will take more than 1 byte in UTF-8 encoding.
6. If you're unaware of #5, your Go programs might break once someone uses them with other languages. Unless of course you carefully read the rest of this chapter.

String encoding in Go

A string in Go is an array of bytes. Any bytes. A string itself knows nothing about encodings, and doesn't have to be UTF-8 encoded. Although some library functions and even one language feature (for-range loops, described below) assume it is.

It's not uncommon to believe that Go strings are all UTF-8. One thing that adds a lot to this confusion are string literals. While strings themselves don't have any specific encoding, the Go compiler always interprets source code as UTF-8.

When a string literal is defined, your editor will save it, same as the rest of the code, as a UTF-8 encoded Unicode string. That's what will be compiled into your program once Go parses it. Neither the

compiler or Go string handling code had anything to do with the string ending up encoded as UTF-8 - it's simply how the text editor wrote the string to disk:

```
package main

import (
    "fmt"
)

func main() {
    // a string literal with Unicode characters
    s := "English 한국어"

    // prints the expected Unicode string:
English 한국어
    fmt.Println(s)
}
```

Just to prove a point, here's how you could define a string that's not UTF-8:

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    s := "\xe2\x28\xa1"

    fmt.Println(utf8.ValidString(s)) // false
```

```
    fmt.Println(s) // ◆(◆
}
```

Rune type

A Unicode code point is represented in Go with the type "rune" which is a 32 bit integer.

String length

Calling len on a string returns the number of bytes in a string rather than the number of characters.

Getting the number of characters can be quite involved. Counting the runes in the string may or may not be good enough:

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    s := "한국어" // 3 Korean characters, encoded in 9 bytes

    byteLen := len(s)
    runeLen := utf8.RuneCountInString(s)
    runeLen2 := len([]rune(s)) // same thing as doing RuneCountInString

    fmt.Println(byteLen, runeLen, runeLen2) // prints 9 3 3
}
```

Unfortunately, some Unicode characters span multiple code points and therefore multiple runes. Scary things need to be done to work out the human perceived number of characters in a Unicode string, as explained in Unicode standard. Go libraries don't really provide a simple way to do it. Here is one of the ways:

```
package main

import (
    "fmt"
    "unicode/utf8"
    "golang.org/x/text/unicode/norm"
)

func normlen(s string) int {
    var ia norm.Iter
    ia.InitString(norm.NFKD, s)
    nc := 0

    for !ia.Done() {
        nc = nc + 1
        ia.Next()
    }

    return nc
}

func main() {
    str := "é‟ // a particularly strange
character

    fmt.Printf(
```

```
        "%d bytes, %d runes, %d actual
character",
        len(str),
        utf8.RuneCountInString(str),
        normlen(str))
}
```

```
7 bytes, 4 runes, 1 actual character
```

String index operator vs. for … range

In short, the string index operator returns the byte at the index in the byte array containing the string. For … range iterates over runes in a string interpreting the string as UTF-8 encoded text:

```
package main

import (
    "fmt"
)

func main() {
    s := "touché"

    // prints every byte
    // touchÃ©
    for i := 0; i < len(s); i++ {
        fmt.Print(string(s[i]))
    }
    fmt.Println()

    // prints every rune
```

```
    // touché
    for _, r := range s {
        fmt.Print(string(r))
    }
    fmt.Println()

    // convert a string to rune slice to access
by index
    // touché
    r := []rune(s)
    for i := 0; i < len(r); i++ {
        fmt.Print(string(r[i]))
    }
}
```

Chapter 8: Maps

Map iteration order is random (no really)

Technically map iteration order is “undefined”. Go maps use a hash table internally so map iteration would normally be done in whatever order the map elements are laid out in that table. This order can't be relied on and changes as a hash table grows when new elements are added to the map. In the early days of Go this was a serious trap for programmers who didn’t read the instructions and relied on maps being iterated in a certain order. To help spot these issues early on rather than in production, Go developers made map iterations random-ish:

```
package main

import "fmt"

func main() {
    // add sequential elements
    // to make it seem like maybe maps are
iterated in order
    m := map[int]int{0: 0, 1: 1, 2: 2, 3: 3, 4:
4, 5: 5}
    for i := 0; i < 5; i++ {
        for i := range m {
            fmt.Print(i, " ")
        }
        fmt.Println()
    }

    // add more elements
```

```
    // to make the hash table of the map grow and
reorder elements
    m[6] = 6
    m[7] = 7
    m[8] = 8

    for i := 0; i < 5; i++ {
        for i := range m {
            fmt.Print(i, " ")
        }
        fmt.Println()
    }
}
```

```
3 4 5 0 1 2

5 0 1 2 3 4

0 1 2 3 4 5

1 2 3 4 5 0

0 1 2 3 4 5

0 1 3 6 7 2 4 5 8

1 3 6 7 0 4 5 8 2

2 4 5 8 0 1 3 6 7
```

```
0 1 3 6 7 2 4 5 8

0 1 3 6 7 2 4 5 8
```

In the above example when the map is initialized with items 1 through 5 they're added to the hash table in that order. The first five printed lines are all numbers 0 to 5 written sequentially. Go only randomizes which element the iteration starts from. Adding more elements to the map makes the hash table for the map grow, which reorders the whole hash table. The last 5 printed lines are no longer in any kind of obvious order. If you must, you can find out all about it in the source code of Go maps.

Checking if a map key exists

Accessing a map element that doesn't exist returns the zero value for the map value type. If it's a map of integers it would return 0, for reference types it would return nil. To check if an element exists in the map, sometimes a zero value is enough. For instance if you have a map of pointers to a struct, then getting a nil value when accessing a map means the element you looked for isn't in the map. In case of a map of bool values for example, the default value isn't enough to tell if an elements value is "false" or if it's missing in the map. Accessing a map element returns an optional second argument which shows if the element was actually in the map:

```
package main

import "fmt"

func main() {
    m := map[int]bool{1: false, 2: true, 3: true}

    // prints false, but not clear if the value
of
```

```
    // the element is false or map item doesn't
exist
    // and the default was returned
    fmt.Println(m[1])

    val, exists := m[1]
    fmt.Println(val, exists) // prints false true
}
```

Map is a pointer

While a slice type is a struct (a value type) that has a pointer to an array, a map itself is a pointer. The zero value of a slice is already fully usable. You can use append to add elements and get the slice's length. A map is different. Go developers would like to make map zero values fully usable, but they couldn't figure out how to implement that efficiently. The map keyword in Go is an alias for the *runtime.hmap type. Its zero value is nil. A nil map can be read from, but it can't be written to:

```
package main

import "fmt"

func main() {
    var m map[int]int // a nil map

     // taking len of a nil map is OK. prints 0
    fmt.Println(len(m))
    // reading nil map is OK. prints 0 (the
default of the map value type)
    fmt.Println(m[10])
    m[10] = 1 // panic: assignment to entry in
nil map
```

```
}
```

Nil maps can be read from because map items are accessed with functions like this one (from runtime/map.go):

```
func mapaccess1(t *maptype, h *hmap, key
unsafe.Pointer) unsafe.Pointer
```

This function checks if map is nil and returns a zero value for the map type if it is. Notice it can't create a map. To create a fully usable map make must be called:

```
package main

import "fmt"

func main() {
    m := make(map[int]int)
    m[10] = 11          // now all is well
    fmt.Println(m[10]) // prints 11
}
```

Since a map is a pointer, passing it to a function passes a pointer to the same map data structure:

```
package main

import "fmt"

func f1(m map[int]int) {
    m[5] = 123
}

func main() {
    m := make(map[int]int)
```

```
    f1(m)
    fmt.Println(m[5]) // prints 123
}
```

When a pointer to a map is passed to a function the value of the pointer is copied (Go passes everything by value, including pointers). Creating a new map inside of a function would change the value of the copy of the pointer. So this won't work:

```
package main

import "fmt"

func f1(m map[int]int) {
    m = make(map[int]int)
    m[5] = 123
}

func main() {
    var m map[int]int
    f1(m)
    fmt.Println(m[5])      // prints 0
    fmt.Println(m == nil) // true
}
```

struct{} type

Go doesn't have a set data structure (something like a map with keys but without values, as implemented as std::set in C++ or HashSet in C#). It's easy enough to use a map instead. One small trick is to use a struct{} type as a map value:

```
package main

import (
```

```
    "fmt"
)

func main() {
    m := make(map[int]struct{})
    m[123] = struct{}{}
    _, keyexists := m[123]
    fmt.Println(keyexists)
}
```

Often a bool value would be used here, but a map with a struct{} value type would use a bit less memory. struct{} type is actually zero bytes in size:

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    fmt.Println(unsafe.Sizeof(false)) // 1
    fmt.Println(unsafe.Sizeof(struct{}{})) // 0
}
```

Map capacity

A map is a fairly complex data structure. While it's possible to specify its initial capacity when creating it, it's not possible to get its capacity later (at least not with the cap function):

```
package main

import (
```

```
    "fmt"
)

func main() {
    m := make(map[int]bool, 5) // initial
capacity of 5
    fmt.Println(len(m)) // len is fine
    fmt.Println(cap(m)) // invalid argument m
(type map[int]bool) for cap
}
```

Map values are not addressable

A Go map is implemented as a hash table, and hash tables need to move their elements around when the map grows or shrinks. For this reason Go doesn't allow taking the address of a map element:

```
package main

import "fmt"

type item struct {
    value string
}

func main() {
    m := map[int]item{1: {"one"}}

    fmt.Println(m[1].value) // reading a struct
value is fine
    addr := &m[1]           // error: cannot take
the address of m[1]
```

```
    // error: cannot assign to struct field
m[1].value in map
    m[1].value = "two"
}
```

There's [a proposal to allow assignment to a struct field](m[1].value = "two") since in this case the pointer to the value field isn't kept, only assigned through. There are no specific plans for when or if it will be implemented because of “subtle corner cases”.

As a workaround, the whole struct needs to be reassigned back to the map:

```
package main

type item struct {
    value string
}

func main() {
    m := map[int]item{1: {"one"}}
    tmp := m[1]
    tmp.value = "two"
    m[1] = tmp
}
```

Alternatively a map of pointers to a struct will also work. In this case a “value” of m[1] is of a type *item. Go doesn't need to take a pointer to the map value, since the value itself is already a pointer. The hash table will move the pointers around in memory, but if you take a copy of a value of m[1] it will keep pointing to the same item so all is well:

```
package main

import "fmt"
```

```
type item struct {
    value string
}

func main() {
    m := map[int]*item{1: {"one"}}
    // Go doesn't need to take address of m[1]
here
    // as it's a pointer already
    m[1].value = "two"
    fmt.Println(m[1].value) // two

    addr := &m[1] // still same error: cannot
take the address of m[1]
}
```

It's worth noting that slices and arrays don't have this problem:

```
package main

import "fmt"

func main() {
    slice := []string{"one"}

    saddr := &slice[0]
    *saddr = "two"

    fmt.Println(slice) // [two]
}
```

Data races

A regular Go map isn't safe for concurrent access. Maps are often used to share data between goroutines, but access to a map must be synchronized through sync.Mutex, sync.RWMutex, some other memory barrier, or coordinated with Go channels to prevent concurrent access, with the following exception:

> *Map access is unsafe only when updates are occurring. As long as all goroutines are only reading—looking up elements in the map, including iterating through it using a `for range` loop—and not changing the map by assigning to elements or doing deletions, it is safe for them to access the map concurrently without synchronization.*
>
> *https://golang.org/doc/faq*

```
package main

import (
    "math/rand"
    "time"
)

func readWrite(m map[int]int) {
    // do some random reads and writes to the map
    for i := 0; i < 100; i++ {
        k := rand.Int()
        m[k] = m[k] + 1
    }
}

func main() {
    m := make(map[int]int)

    // start goroutines to read and write map
concurrently
```

```
    for i := 0; i < 10; i++ {
        go readWrite(m)
    }

    time.Sleep(time.Second)
}
```

```
fatal error: concurrent map read and map write

fatal error: concurrent map writes

…
```

In this case map access could be synchronized with a Mutex. The code below will work as expected:

```
package main

import (
    "math/rand"
    "sync"
    "time"
)

var mu sync.Mutex

func readWrite(m map[int]int) {
    mu.Lock()
    // defer unlock mutex will unlock mutex
    // even if this goroutine would panic
    defer mu.Unlock()
```

```
    for i := 0; i < 100; i++ {
        k := rand.Int()
        m[k] = m[k] + 1
    }
}

func main() {
    m := make(map[int]int)
    for i := 0; i < 10; i++ {
        go readWrite(m)
    }

    time.Sleep(time.Second)
}
```

sync.Map

There's a specialized version of a map in the sync package that's safe for concurrent use by multiple goroutines. Go documentation however recommends using a regular map with locking or coordination in most cases. sync.Map isn't type safe, and it's similar to a map[interface{}]interface{}. From sync.Map documentation:

> *The Map type is optimized for two common use cases: (1) when the entry for a given key is only ever written once but read many times, as in caches that only grow, or (2) when multiple goroutines read, write, and overwrite entries for disjoint sets of keys. In these two cases, use of a Map may significantly reduce lock contention compared to a Go map paired with a separate Mutex or RWMutex.*
>
> *https://github.com/golang/go/blob/master/src/sync/map.go*

Chapter 9: Loops

Range iterator returns two values

A trap for early beginners is that for-range in Go works a bit differently than its equivalents in other languages. It returns one or two variables, the first of those an iteration index (or a map key if a map is iterated on) and second is the value. If only one variable is used - it's the index:

```
package main

import "fmt"

func main() {
    slice := []string{"one", "two", "three"}

    for v := range slice {
        fmt.Println(v) // 0, 1, 2
    }

    for _, v := range slice {
        fmt.Println(v) // one two three
    }
}
```

For loop iterator variables are reused

In loops the same iterator variable is reused for every iteration. If you take its address it will be the same address each time. This means the value of the iterator variable gets copied over to the same memory location on every iteration. It makes loops more efficient, but also it's one of the most common traps in Go. Here is an example from the Go wiki:

```
package main

import "fmt"

func main() {
    var out []*int
    for i := 0; i < 3; i++ {
        out = append(out, &i)
    }
    fmt.Println("Values:", *out[0], *out[1],
*out[2])
    fmt.Println("Addresses:", out[0], out[1],
out[2])
}
```

```
Values: 3 3 3

Addresses: 0xc0000120e0 0xc0000120e0 0xc0000120e0
```

Ouch. One solution is to declare a new variable inside of the loop. Variables declared inside of a code block are not reused, even in loops:

```
package main

import "fmt"

func main() {
    var out []*int
    for i := 0; i < 3; i++ {
        i := i // copy i into a new variable
```

```
        out = append(out, &i)
    }
    fmt.Println("Values:", *out[0], *out[1],
*out[2])
    fmt.Println("Addresses:", out[0], out[1],
out[2])
}
```

Now it works as expected:

```
Values: 0 1 2

Addresses: 0xc0000120e0 0xc0000120e8 0xc0000120f0
```

In case of for-range clauses both index and value variables are reused.

It's a similar thing with starting goroutines in a loop:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i < 3; i++ {
        go func() {
            fmt.Print(i)
        }()
    }
    time.Sleep(time.Second)
```

```
}
```

```
333
```

Those goroutines are created in this loop, but it takes a bit of time for them to start running. Since they capture the single i variable, Println prints whatever value it has at the time goroutine is executed.

In this case you might create a new variable inside of the code block as in the previous example or pass the iterator variable as a parameter to the goroutine:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    for i := 0; i < 3; i++ {
        go func(i int) {
            fmt.Print(i)
        }(i)
    }
    time.Sleep(time.Second)
}
```

```
012
```

Here the parameter of the goroutine i is a new variable which is copied from the iterator variable as part of creating the goroutine.

If instead of starting a goroutine the loop would invoke a simple function, the code would work as expected:

```
for i := 0; i < 3; i++ {
    func() {
        fmt.Print(i)
    }()
}
```

```
012
```

Variable i is reused as before. Each of those function calls however won't let the loop continue until the function is finished executing. During that time i will have the expected value.

It gets a little more tricky. Take a look at this example with calling a method on a struct:

```
package main

import (
    "fmt"
    "time"
)

type myStruct struct {
    v int
}

func (s *myStruct) myMethod() {
    // print the value of myStruct and its
address
    fmt.Printf("%v, %p\n", s.v, s)
```

```
}

func main() {
    byValue := []myStruct{{1}, {2}, {3}}
    byReference := []*myStruct{{1}, {2}, {3}}

    fmt.Println("By value")

    for _, i := range byValue {
        go i.myMethod()
    }
    time.Sleep(time.Millisecond * 100)

    fmt.Println("By reference")

    for _, i := range byReference {
        go i.myMethod()
    }
    time.Sleep(time.Millisecond * 100)
}
```

```
By value

3, 0xc000012120

3, 0xc000012120

3, 0xc000012120

By reference

1, 0xc0000120e0
```

```
3, 0xc0000120f0

2, 0xc0000120e8
```

Ouch again. When myStruct is used by reference it works as if there's no trap to begin with! This has to do with how goroutines are created. Goroutine arguments are evaluated at the time the goroutine is created. The method receiver (myStruct of myMethod) is effectively an argument.

When called by value: since the argument s of myMethod is a pointer, the address of i is passed as an argument to the goroutine. The iterator variable as we know is reused so it's the same address each time. When the iterator runs it will copy over a new myStruct value to the same address of the i variable. The values printed are whatever the value the i variable had at the time when the goroutine executed.

When called by reference: the argument is already a pointer so its value is pushed to the stack of the new goroutine at the time of creating the goroutine. This happens to be the address we want and the expected values are printed.

Labeled break and continue

Perhaps less well known feature of Go is the ability to label for, switch, and select statements, and use break and continue on those labels. Here's how you can break from the outer loop:

```
loopi:
    for x := 0; x < 3; x++ {
        for y := 0; y < 3; y++ {
            fmt.Printf(x, y)
            break loopi
        }
```

```
    }
```

```
0 0
```

Continue can be used in a similar way as well:

```
loopi:
    for x := 0; x < 3; x++ {
        for y := 0; y < 3; y++ {
            fmt.Printf(x, y)
            continue loopi
        }
    }
```

```
0 0

1 0

2 0
```

Labels can also be used with switch and select statements. Here a break without a label would only break out of the select statement and into the for loop:

```
package main

import (
    "fmt"
    "time"
)

func main() {
```

```
loop:
    for {
        select {
        case <-time.After(time.Second):
            fmt.Println("timeout reached")
            break loop
        }
    }
    fmt.Println("the end")
}
```

```
timeout reached

the end
```

As mentioned before, switch and select statements can be labeled as well so we can turn the above example around:

```
package main

import (
    "fmt"
    "time"
)

func main() {
myswitch:
    switch {
    case true:
        for {
```

```
            fmt.Println("switch")
            break myswitch // wouldn't be able to
“continue” in this case
        }
    }
    fmt.Println("the end")
}
```

```
switch

the end
```

It’s easy to confuse “labeled statements” in previous examples with a label that goto would be used on. In fact you can use the same label for both break/continue and goto, but the behavior will be different. In the code below, while break would break out of a labeled loop, goto would transfer execution to the location of the label (and cause an infinite loop in the code below):

```
package main

import (
    "fmt"
    "time"
)

func main() {
loop:
    switch {
    case true:
        for {
```

```
            fmt.Println("switch")
            break loop // breaks the “labeled
statement”
        }
    }
    fmt.Println("not the end")
    goto loop // jumps to “loop” label
}
```

```
switch

not the end

switch

not the end

…
```

Chapter 10: Switch and Select

Case statements break by default

Unlike in C based languages case statements in Go break by default. To make case statements fall through use the fallthrough keyword:

```
package main

import (
    "fmt"
    "time"
)

func main() {
    // this won't work, in case of Saturday
nothing will be printed
    switch time.Now().Weekday() {
    case 6: // this case will break out of switch
without doing anything
    case 7:
        fmt.Println("weekend")
    }

    switch time.Now().Weekday() {
    case 1:
        break // this break does nothing because
case would break anyway
    case 2:
        fmt.Println("weekend")
    }
```

```
    // fallthrough keyword will make Saturday
print weekend as well
    switch time.Now().Weekday() {
    case 6:
        fallthrough
    case 7:
        fmt.Println("weekend")
    }

    // case can also have multiple values
    switch time.Now().Weekday() {
    case 6, 7:
        fmt.Println("weekend")
    }

    // conditional breaks are still useful
    switch time.Now().Weekday() {
    case 6, 7:
        day := time.Now().Format("01-02")
        if day == "12-25" ||  day == "12-26" {
            fmt.Println("Christmas weekend")
            break // don't also print "weekend"
        }

        // a regular weekend
        fmt.Println("weekend")
    }
}
```

Labeled breaks

As mentioned before in the chapter about loops, switch and select can also do labeled breaks to break out of the outer loop rather than a switch or select statement itself:

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    s := "The quick brown Waldo fox jumps over
the lazy dog"

findWaldoLoop:
    for _, w := range strings.Split(s, " ") {
        switch w {
        case "Waldo":
            fmt.Println("found Waldo!")
            break findWaldoLoop
        default:
            fmt.Println(w, "is not Waldo")
        }
    }
}
```

```
The is not Waldo

quick is not Waldo
```

```
brown is not Waldo

found Waldo!
```

Chapter 11: Functions

Defer statement

Defer doesn't seem to have big traps, but it’s worth mentioning a few nuances.

From an excellent post by Andrew Gerrand on the subject:

> *A defer statement pushes a function call onto a list. The list of saved calls is executed after the surrounding function returns. Defer is commonly used to simplify functions that perform various clean-up actions.*

The most important points to be aware of:

1. While a deferred function is invoked when the original function returns, its arguments are evaluated at the point of calling defer

```
package main

import (
    "fmt"
)

func main() {
    s := "defer"
    defer fmt.Println(s)
    s = "original"
    fmt.Println(s)
}
```

```
original

defer
```

2. Deferred functions execute in Last In First Out order once the original function returns

```
package main

import (
    "fmt"
)

func main() {
    defer fmt.Println("one")
    defer fmt.Println("two")
    defer fmt.Println("three")
}
```

```
three

two

one
```

3. Deferred functions can access and modify named function arguments

```
package main

import (
    "fmt"
    "time"
)

func timeNow() (t string) {
```

```
    defer func() {
      t = "Current time is: " + t
    }()
  return time.Now().Format(time.Stamp)
}

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

```
Current time is: Feb 13 13:36:44
```

4. Defer doesn't work for code blocks, only for the whole function

Unlike variable declarations defer statements are not scoped to code blocks:

```
package main

import (
    "fmt"
)

func main() {
    for i := 0; i < 9; i++ {
        if i%3 == 0 {
            defer func(i int) {
                fmt.Println("defer", i)
            }(i)
        }
    }
    fmt.Println("exiting main")
```

```
}
```

```
exiting main

defer 6

defer 3

defer 0
```

In this example a deferred function call will be added to the list when i is 0, 3, and 6. But it will only get invoked when the main function exits and not at the end of the if statement.

5. recover() only works inside of deferred functions, it will do nothing in the original function

It doesn't really make sense any other way, but in case you're looking for an equivalent of try…catch statements, there are none in Go. Panics are caught using recover() inside of deferred functions.

```
package main

import (
    "fmt"
)

func panickyFunc() {
    panic("panic!")
}

func main() {
    defer func() {
      r := recover()
```

```
        if r != nil {
            fmt.Println("recovered", r)
        }
    }()

    panickyFunc()
    fmt.Println("this will never be printed")
}
```

```
recovered panic!
```

Chapter 12: Goroutines

What are goroutines

For most intents and purposes goroutines can be thought of as lightweight threads. They're quick to start, initially use only 2kb of stack memory (which can grow or shrink). They're managed by the Go runtime (rather than the operating system) and context switching between them is cheap. Goroutines are built for concurrency and when run on multiple hardware threads they will also run in parallel.

> *Concurrency is about dealing with lots of things at once. Parallelism is about doing lots of things at once.*
>
> *Rob Pike*

It's scary how efficient goroutines are and when combined with channels they could well be the best feature of Go. They're everywhere in Go, but an extreme example of a good problem for goroutines could be a server managing lots of concurrent websocket connections. They need to each be managed individually, but they're likely also mostly sitting idle (not using much CPU or memory). Creating a thread for each of them would cause problems once it gets to thousands of connections, while with goroutines hundreds of thousands are possible.

A more detailed post on how goroutines work can be found here.

Running goroutines don't stop a program from exiting

Go program exits when the main function exits. Any goroutines running in the background quietly stop. The following program will exit without printing anything:

```
package main

import (
    "fmt"
```

```
    "time"
)

func goroutine1() {
    time.Sleep(time.Second)
    fmt.Println("goroutine1")
}

func goroutine2() {
    time.Sleep(time.Second)
    fmt.Println("goroutine2")
}

func main() {
    go goroutine1()
    go goroutine2()
}
```

To make sure those goroutines finish some synchronization needs to be added, such as using channels or sync.WaitGroup:

```
package main

import (
    "fmt"
    "sync"
    "time"
)

func goroutine1(wg *sync.WaitGroup) {
    time.Sleep(time.Second)
    fmt.Println("goroutine1")
```

```
    wg.Done()
}

func goroutine2(wg *sync.WaitGroup) {
    time.Sleep(time.Second)
    fmt.Println("goroutine2")
    wg.Done()
}

func main() {
    wg := &sync.WaitGroup{}
    wg.Add(2)
    go goroutine1(wg)
    go goroutine2(wg)
    wg.Wait()
}
```

```
goroutine2

goroutine1
```

A panicking goroutine will crash the whole application

Panics inside goroutines must be handled with defer and recover(). Otherwise the whole application will crash:

```
package main

import (
    "fmt"
    "time"
)
```

```go
func goroutine1() {
    panic("something went wrong")
}

func main() {
    go goroutine1()
    time.Sleep(time.Second)
    fmt.Println("will never get here")
}
```

```
panic: something went wrong


goroutine 6 [running]:

main.goroutine1()

        c:/projects/test/main.go:9 +0x45

created by main.main

        c:/projects/test/main.go:13 +0x45
```

Chapter 13: Interfaces

Checking if an interface variable is nil

This is certainly one of the most common traps in Go. An interface in Go is not simply a pointer to a memory location as in some other languages.

An interface has:

- A static type (the type of the interface itself)
- A dynamic type
- A value

> *"A variable of an interface type is equal to nil when both its dynamic type and value are nil.*

```
package main

import (
    "fmt"
)

type ISayHi interface {
    Say()
}

type SayHi struct{}

func (s *SayHi) Say() {
    fmt.Println("Hi!")
}

func main() {
```

```
    // at this point variable “sayer” only has
the static type of ISayHi
    // dynamic type and value are nil
    var sayer ISayHi

    // as expected sayer equals to nil
    fmt.Println(sayer == nil) // true
    // a nil variable of a concrete type
    var sayerImplementation *SayHi

    // dynamic type of the interface variable is
now SayHi
    // the actual value interface points to is
still nil
    sayer = sayerImplementation

    // sayer no longer equals to nil, because its
dynamic type is set
    // even though the value it points to is nil
    // which isn't what most people would expect
here
    fmt.Println(sayer == nil) // false
}
```

The interface value is set to a nil struct. The interface can't be used for anything, so why isn't it equal to nil? This is another difference of Go when compared to other languages. While calling a method on a nil class in C# will throw an exception, for better or worse, in Go it's allowed. Therefore when an interface has the dynamic type set, it can sometimes be usable even when the value is nil. So you can argue then the interface isn't really "nil":

```go
package main

import (
    "fmt"
)

type ISayHi interface {
    Say()
}

type SayHi struct{}

func (s *SayHi) Say() {
    // this function isn't accessing s
    // even if s is nil this will work
    fmt.Println("Hi!")
}

func main() {
    var sayer ISayHi
    var sayerImplementation *SayHi
    sayer = sayerImplementation

    // the value of SayHi on sayer interface is nil
    // in Go it's OK to call methods on a nil struct
    // this line will work fine, because Say function isn't accessing s
    sayer.Say()
}
```

As odd as this may be, there's no simple way of checking if the value an interface points to is nil. There's a long ongoing discussion on the topic and it doesn't seem to be going anywhere. So for a foreseeable future these are the things you can do:

Least of all evils option #1: never assign nil concrete types to interfaces

If you never assign nil concrete types to an interface variable (with the exception of types that are designed to work with nil receivers), a simple "== nil" check will always work. E.g. never do this:

```
func MyFunc() ISayHi {
    var result *SayHi
    if time.Now().Weekday() == time.Sunday {
      result = &SayHi{}
    }
    // if it's not Sunday, this returns an
interface that's not
    // equal to nil, but has a nil value for its
concrete type
    // (MyFunc() == nil would be false)
    return result
}
```

Return an actual nil instead:

```
func MyBetterFunc() ISayHi {
    if time.Now().Weekday() != time.Sunday {
      // if it's not Sunday
      // MyBetterFunc() == nil would be true
      return nil
  }
```

```
    return &SayHi{}
}
```

This is probably the best available solution even if it's not ideal because everyone then must be aware of it and monitor this in code reviews etc. and in a way do work that a computer could do.

OK in special cases option #2: reflection

If you must, you can check if the underlying value of an interface is nil through reflection. It will be slow and it's probably not a good idea to litter your code with these function calls:

```
func IsInterfaceNil(i interface{}) bool {
    if i == nil {
      return false
    }
    rvalue := reflect.ValueOf(i)
    return rvalue.Kind() == reflect.Ptr &&
rvalue.IsNil()
}
```

Checking if the Kind() of value is a pointer is necessary as IsNil will panic for types that can't be nil (such as a simple int).

Please don't do this option #3: add IsNil to your struct interfaces

This way you can check if an interface is nil without using reflection:

```
type ISayHi interface {
    Say()
    IsNil() bool
}

type SayHi struct{}

func (s *SayHi) Say() {
```

```
    fmt.Println("Hi!")
}

func (s *SayHi) IsNil() bool {
    return s == nil
}
```

Perhaps consider option #1 option #4: asserting concrete type

If you know what type the interface value is supposed to be you can check if its nil by first getting the value of a concrete type with a type switch or type assertion:

```
func main() {
    v := MyFunc()
    fmt.Println(v.(*SayHi) == nil)
}
```

While this might be fine if you really know what you're doing, in many cases this kind of beats the purpose of using interfaces to begin with. Consider what happens when a new implementation of ISayHi is added. Will you need to remember to find this code and add another check for the new struct? Will you be doing this for every new implementation? What if this code is handling a rarely occurring event and not checking for a newly added implementation, and this is only noticed long after the code is in production?

Interfaces are satisfied implicitly

Unlike in many other languages you don’t need to explicitly specify that a struct implements an interface. Compiler can work it out by itself. Which makes a lot of sense and is very convenient in practice:

```
package main

import (
    "fmt"
```

```
)

// an interface
type ISayHi interface {
    Say()
}

// this struct implements ISayHi even if it
doesn't know it
type SayHi struct{}

func (s *SayHi) Say() {
    fmt.Println("Hi!")
}

func main() {
    var sayer ISayHi // sayer is an interface
    sayer = &SayHi{} // SayHi implicitly
implements ISayHi
    sayer.Say()
}
```

It can sometimes be useful to have a compiler check if a struct implements an interface:

```
// verify at compile time that *SayHi implements
ISayHi
var _ ISayHi = (*SayHi)(nil)
```

Type assertions on a wrong type

There is a one variable and two variable versions of type assertion. One variable version panics when the type is not the one being asserted:

```
func main() {
	var sayer ISayHi
	sayer = &SayHi{}

	// t will be a zero value (nil in this
case) of type *SayHi2
	// ok will be false
	t, ok := sayer.(*SayHi2)
	if ok {
	    t.Say()
	}

	// panic: interface conversion:
	// main.ISayHi is *main.SayHi, not
*main.SayHi2
	t2 := sayer.(*SayHi2)
	t2.Say()
}
```

Chapter 14: Inheritance

Redefining vs. embedding types

Go type system is... pragmatic. It is not object-oriented in a sense that C++ or Java is. You cannot really inherit structs or interfaces (there is no subclassing), but you can put them together (embed) to make more complicated structs or interfaces.

> *There's an important way in which embedding differs from subclassing. When we embed a type, the methods of that type become methods of the outer type, but when they are invoked the receiver of the method is the inner type, not the outer one.*
>
> *https://golang.org/doc/effective_go*

Next to embedding types Go allows redefining a type. Redefining inherits the fields of a type, but not its methods:

```
package main

type t1 struct {
    f1 string
}

func (t *t1) t1method() {
}

// embedding type
type t2 struct {
    t1
}

// redefining type
type t3 t1
```

```
func main() {
    var mt1 t1
    var mt2 t2
    var mt3 t3

    // fields are inherited in all the cases
    _ = mt1.f1
    _ = mt2.f1
    _ = mt3.f1

    // these work ok
    mt1.t1method()
    mt2.t1method()

    // mt3.t1method undefined (type t3 has no
field or method t1method)
    mt3.t1method()
}
```

Chapter 15: Equality

Equality in Go

There are different ways to compare things in Go, none of them perfect.

Operators == and !=

The equality operator is the simplest and often most efficient way to compare things in Go, but it only works on certain things. Most notably it doesn't work on slices or maps. Slices and maps can only be compared to nil this way.

Using == you can compare basic types like int and string, and also arrays and structs that have elements in them that can themselves be compared using ==:

```
package main

import "fmt"

type compareStruct1 struct {
    A int
    B string
    C [3]int
}

func main() {
    s1 := compareStruct1{}
    s2 := compareStruct1{}
    fmt.Println(s1 == s2) // works fine, prints
true
}
```

As soon as you add a property to the struct that can't be compared with ==, you need a whole other way to compare:/

```
package main

import "fmt"

type compareStruct2 struct {
    A int
    B string
    C []int // changed type of C from array to
slice
}

func main() {
    s1 := compareStruct2{}
    s2 := compareStruct2{}

    // invalid operation: s1 == s2
    // (struct containing []int can't be
compared)
    fmt.Println(s1 == s2)
}
```

Writing specialized code

If performance is important and you need to compare slightly more complicated types your best bet might be comparing manually:

```
type compareStruct struct {
    A int
    B string
    C []int
}
```

```
func (s *compareStruct) Equals(s2 *compareStruct)
bool {
    if s.A != s2.A || s.B != s2.B || len(s.C) !=
len(s2.C) {
        return false
    }

    for i := 0; i < len(s.C); i++ {
        if s.C[i] != s2.C[i] {
            return false
        }
    }

    return true
}
```

The comparison function like in the code above could be auto-generated, but at the time of writing I'm unaware of any tools that can do that.

reflect.DeepEqual

DeepEqual is the most generic way to compare things in Go, and it can handle most of things. Here is the catch:

```
var (
    c1 = compareStruct{
        A: 1,
        B: "hello",
        C: []int{1, 2, 3},
    }
    c2 = compareStruct{
        A: 1,
```

```
        B: "hello",
        C: []int{1, 2, 3},
    }
)

func BenchmarkManual(b *testing.B) {
    for i := 0; i < b.N; i++ {
        c1.Equals(&c2)
    }
}

func BenchmarkDeepEqual(b *testing.B) {
    for i := 0; i < b.N; i++ {
        reflect.DeepEqual(c1, c2)
    }
}
```

```
BenchmarkManual-8 217182776 5.51 ns/op 0 B/op 0
allocs/op

BenchmarkDeepEqual-8 2175002 559 ns/op 144 B/op 8
allocs/op
```

DeepEqual is 100 times slower in this example than writing manual code for comparing that struct.

Note that DeepEqual will compare unexported (lower cased) fields from a struct as well. Also, two different types would never be considered deeply equal even if they were two different structs with identical fields and values.

Uncomparable things

Some things can't be compared and are considered unequal even to themselves, such as floating-point variables with a NaN value or a func type. If you have such fields in a struct for instance, the struct won't be DeepEqual to itself:

```
func TestF(t *testing.T) {
    x := math.NaN
    fmt.Println(reflect.DeepEqual(x, x)) // false
    fmt.Println(reflect.DeepEqual(TestF, TestF))
// false
}
```

bytes.Equal

bytes.Equal is a specialized way to compare byte slices. It’s much faster than simply comparing two slices with a for loop.

Something worth being aware of, the bytes.Equal function considers empty and nil slices to be equal, while reflect.DeepEqual does not.

Chapter 16: Memory Management

Should structs be passed by value or by reference

Arguments to Go functions are always passed by value. When a struct (or array) type variable is passed into a function the whole struct gets copied. If a pointer to a struct gets passed, then the pointer is copied, but the struct it points to isn't. 8 bytes of memory (for 64bit architectures) get copied instead of whatever the size of the struct is. So does that mean it's better to pass structs as pointers? As always - it depends.

Taking a pointer to a struct (or array) 1) places it in heap memory rather than stack where it would normally be 2) involves a garbage collector to manage that heap allocation.

If you'd like a refresher on stack vs. heap check this stackoverflow thread. For the purposes of this chapter it's enough to know this much: stack - fast, heap - slow.

That means if you allocate structs more than pass them around, it's way faster to let them be copied on stack:

```
package test

import (
    "testing"
)

type myStruct struct {
    a, b, c int64
    d, e, f string
    g, h, i float64
}

func byValue() myStruct {
```

```
    return myStruct{
        a: 1, b: 1, c: 1,
        d: "foo", e: "bar", f: "baz",
        g: 1.0, h: 1.0, i: 1.0,
    }
}

func byReference() *myStruct {
    return &myStruct{
        a: 1, b: 1, c: 1,
        d: "foo", e: "bar", f: "baz",
        g: 1.0, h: 1.0, i: 1.0,
    }
}

func BenchmarkByValue(b *testing.B) {
    var s myStruct

    for i := 0; i < b.N; i++ {
        // make a copy of the whole struct
        // but do it through stack memory
        s = byValue()
    }

    _ = s
}

func BenchmarkByReference(b *testing.B) {
    var s *myStruct

    for i := 0; i < b.N; i++ {
```

```
        // allocate struct on the heap
        // and only return a pointer to it
        s = byReference()
    }

    _ = s
}
```

```
BenchmarkByValue-8 476965734 2.499 ns/op 0 B/op 0 allocs/op

BenchmarkByReference-8 24860521 45.86 ns/op 96 B/op 1 allocs/op
```

Passing by value (and not involving heap or garbage collector) is 18 times faster in this toy example.

To illustrate a point, let's do an opposite toy example allocating the struct once and only passing it to functions:

```
var s = myStruct{
    a: 1, b: 1, c: 1,
    d: "foo", e: "bar", f: "baz",
    g: 1.0, h: 1.0, i: 1.0,
}

func byValue() myStruct {
    return s
}

func byReference() *myStruct {
```

```
    return &s
}
```

```
BenchmarkByValue-8 471494428 2.509 ns/op 0 B/op 0
allocs/op

BenchmarkByReference-8 1000000000 0.2484 ns/op 0
B/op 0 allocs/op
```

When things are only passed around, but not allocated - it's much faster by reference.

For more details check [this great post](#) by Vincent Blanchon.

While this chapter was about which is faster, in many applications clarity and consistency of the code will be more important than performance, but that's a separate discussion. In summary, don't assume that copying things will be slow, and use the excellent Go profiler if performance is important.

A note for C developers

Go is much more strict on memory management. Pointer arithmetic isn't allowed and it's not possible to have dangling pointers. Things like this are perfectly fine:

```
func byReference() *myStruct {
    return &myStruct{
        a: 1, b: 1, c: 1,
        d: "foo", e: "bar", f: "baz",
        g: 1.0, h: 1.0, i: 1.0,
    }
}
```

The Go compiler is smart enough to move that struct to the heap.

Chapter 17: Logging

log.Fatal and log.Panic

When logging with the Go log package there's a trap waiting for you in the log.Fatal and log.Panic functions. Unlike what you might expect of a logging function these don't simply log a message with a different log level, they also terminate the whole application. log.Fatal cleanly exits the application, log.Panic invokes a runtime panic. Here are the actual functions from the Go log package:

```
// Fatal is equivalent to Print() followed by a
call to os.Exit(1).
func Fatal(v ...interface{}) {
    std.Output(2, fmt.Sprint(v...))
    os.Exit(1)
}

// Panic is equivalent to Print() followed by a
call to panic().
func Panic(v ...interface{}) {
    s := fmt.Sprint(v...)
    std.Output(2, s)
    panic(s)
}
```

Chapter 18: Time

time.LoadLocation reads from a file

This is one of my personal favorite traps in Go. To convert between time zones you first need to load location information. It turns out that time.LoadLocation reads from a file every time it's called. Not the best thing to do when formatting each row of a massive CSV report:

```
package main

import (
    "testing"
    "time"
)

func BenchmarkLocation(b *testing.B) {
    for n := 0; n < b.N; n++ {
        loc, _ :=
time.LoadLocation("Asia/Kolkata")
        time.Now().In(loc)
    }
}

func BenchmarkLocation2(b *testing.B) {
    loc, _ := time.LoadLocation("Asia/Kolkata")
    for n := 0; n < b.N; n++ {
        time.Now().In(loc)
    }
}
```

```
BenchmarkLocation-8 16810 76179 ns/op 58192 B/op
14 allocs/op

BenchmarkLocation2-8 188887110 6.97 ns/op 0 B/op
0 allocs/op
```