CHAPTER 3

Composite Types

In the previous chapter, you looked at literals and predeclared variable types: numbers, booleans, and strings. In this chapter, you'll learn about the composite types in Go, the built-in functions that support them, and the best practices for working with them.

Arrays—Too Rigid to Use Directly

Like most programming languages, Go has arrays. However, arrays are rarely used directly in Go. You'll learn why in a bit, but first let's quickly cover array declaration syntax and use.

All elements in the array must be of the type that's specified. There are a few declaration styles. In the first, you specify the size of the array and the type of the elements in the array:

```
var x [3]int
```

This creates an array of three `int`s. Since no values were specified, all of the elements (`x[0]`, `x[1]`, and `x[2]`) are initialized to the zero value for an `int`, which is (of course) 0. If you have initial values for the array, you specify them with an *array literal*:

```
var x = [3]int{10, 20, 30}
```

If you have a *sparse array* (an array where most elements are set to their zero value), you can specify only the indices with nonzero values in the array literal:

```
var x = [12]int{1, 5: 4, 6, 10: 100, 15}
```

This creates an array of 12 `int`s with the following values: [1, 0, 0, 0, 0, 4, 6, 0, 0, 0, 100, 15].

When using an array literal to initialize an array, you can replace the number that specifies the number of elements in the array with `...`:

```
var x = [...]int{10, 20, 30}
```

You can use `==` and `!=` to compare two arrays. Arrays are equal if they are the same length and contain equal values:

```
var x = [...]int{1, 2, 3}
var y = [3]int{1, 2, 3}
fmt.Println(x == y) // prints true
```

Go has only one-dimensional arrays, but you can simulate multidimensional arrays:

```
var x [2][3]int
```

This declares `x` to be an array of length 2 whose type is an array of `int`s of length 3. This sounds pedantic, but some languages have true matrix support, like Fortran or Julia; Go isn't one of them.

Like most languages, arrays in Go are read and written using bracket syntax:

```
x[0] = 10
fmt.Println(x[2])
```

You cannot read or write past the end of an array or use a negative index. If you do this with a constant or literal index, it is a compile-time error. An out-of-bounds read or write with a variable index compiles but fails at runtime with a *panic* (you'll learn more about panics in "panic and recover" on page 218).

Finally, the built-in function `len` takes in an array and returns its length:

```
fmt.Println(len(x))
```

Earlier I said that arrays in Go are rarely used explicitly. This is because they come with an unusual limitation: Go considers the *size* of the array to be part of the *type* of the array. This makes an array that's declared to be `[3]int` a different type from an array that's declared to be `[4]int`. This also means that you cannot use a variable to specify the size of an array, because types must be resolved at compile time, not at runtime.

What's more, *you can't use a type conversion to directly convert arrays of different sizes to identical types*. Because you can't convert arrays of different sizes into each other, you can't write a function that works with arrays of any size and you can't assign arrays of different sizes to the same variable.

You'll learn how arrays work behind the scenes when I discuss memory layout in Chapter 6.

Because of these restrictions, don't use arrays unless you know the exact length you need ahead of time. For example, some of the cryptographic functions in the standard library return arrays because the sizes of checksums are defined as part of the algorithm. This is the exception, not the rule.

This raises the question: why is such a limited feature in the language? The main reason arrays exist in Go is to provide the backing store for *slices*, which are one of the most useful features of Go.

Slices

Most of the time, when you want a data structure that holds a sequence of values, a slice is what you should use. What makes slices so useful is that you can grow slices as needed. This is because the length of a slice is *not* part of its type. This removes the biggest limitations of arrays and allows you to write a single function that processes slices of any size (I'll cover function writing in Chapter 5). After going over the basics of using slices in Go, I'll cover the best ways to use them.

Working with slices looks a lot like working with arrays, but subtle differences exist. The first thing to notice is that you don't specify the size of the slice when you declare it:

```
var x = []int{10, 20, 30}
```

Using `[...]` makes an array. Using `[]` makes a slice.

This creates a slice of three `int`s using a *slice literal*. Just as with arrays, you can also specify only the indices with nonzero values in the slice literal:

```
var x = []int{1, 5: 4, 6, 10: 100, 15}
```

This creates a slice of 12 `int`s with the following values: [1, 0, 0, 0, 0, 4, 6, 0, 0, 0, 100, 15].

You can simulate multidimensional slices and make a slice of slices:

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

You read and write slices using bracket syntax, and, just as with arrays, you can't read or write past the end or use a negative index:

```
x[0] = 10
fmt.Println(x[2])
```

So far, slices have seemed identical to arrays. You start to see the differences between arrays and slices when you look at declaring slices without using a literal:

```
var x []int
```

This creates a slice of `int`s. Since no value is assigned, `x` is assigned the zero value for a slice, which is something you haven't seen before: `nil`. I'll talk more about `nil` in Chapter 6, but it is slightly different from the `null` that's found in other languages. In Go, `nil` is an identifier that represents the lack of a value for some types. Like the untyped numeric constants you saw in the previous chapter, `nil` has no type, so it can be assigned or compared against values of different types. A `nil` slice contains nothing.

A slice is the first type you've seen that isn't *comparable*. It is a compile-time error to use `==` to see if two slices are identical or `!=` to see if they are different. The only thing you can compare a slice with using `==` is `nil`:

```
fmt.Println(x == nil) // prints true
```

Since Go 1.21, the `slices` package in the standard library includes two functions to compare slices. The `slices.Equal` function takes in two slices and returns `true` if the slices are the same length, and all of the elements are equal. It requires the elements of the slice to be comparable. The other function, `slices.EqualFunc`, lets you pass in a function to determine equality and does not require the slice elements to be comparable. You'll learn about passing functions into functions in "Passing Functions as Parameters" on page 107. The other functions in the `slices` package are covered in "Adding Generics to the Standard Library" on page 201.

```
x := []int{1, 2, 3, 4, 5}
y := []int{1, 2, 3, 4, 5}
z := []int{1, 2, 3, 4, 5, 6}
s := []string{"a", "b", "c"}
fmt.Println(slices.Equal(x, y)) // prints true
fmt.Println(slices.Equal(x, z)) // prints false
fmt.Println(slices.Equal(x, s)) // does not compile
```

The `reflect` package contains a function called `DeepEqual` that can compare almost anything, including slices. It's a legacy function, primarily intended for testing. Before the inclusion of `slices.Equal` and `slices.EqualFunc`, `reflect.DeepEqual` was often used to compare slices. Don't use it in new code, as it is slower and less safe than using the functions in the `slices` package.

len

Go provides several built-in functions to work with slices. You've already seen the built-in `len` function when looking at arrays. It works for slices too. Passing a `nil` slice to `len` returns 0.

Functions like `len` are built into Go because they can do things that can't be done by the functions that you can write. You've already seen that `len`'s parameter can be any type of array or any type of slice. You'll soon see that it also works for strings and maps. In "Channels" on page 291, you'll see it working with channels. Trying to pass a variable of any other type to `len` is a compile-time error. As you'll see in Chapter 5, Go doesn't let developers write a function that accepts any string, array, slice, channel, or map, but rejects other types.

append

The built-in `append` function is used to grow slices:

```
var x []int
x = append(x, 10) // assign result to the variable that's passed in
```

The `append` function takes at least two parameters, a slice of any type and a value of that type. It returns a slice of the same type, which is assigned to the variable that was passed to `append`. In this example, you are appending to a `nil` slice, but you can append to a slice that already has elements:

```
var x = []int{1, 2, 3}
x = append(x, 4)
```

You can append more than one value at a time:

```
x = append(x, 5, 6, 7)
```

One slice is appended onto another by using the `...` operator to expand the source slice into individual values (you'll learn more about the `...` operator in "Variadic Input Parameters and Slices" on page 95):

```
y := []int{20, 30, 40}
x = append(x, y...)
```

It is a compile-time error if you forget to assign the value returned from `append`. You might be wondering why as it seems a bit repetitive. I will talk about this in greater detail in Chapter 5, but Go is a *call-by-value* language. Every time you pass a parameter to a function, Go makes a copy of the value that's passed in. Passing a slice to the `append` function actually passes a copy of the slice to the function. The

function adds the values to the copy of the slice and returns the copy. You then assign the returned slice back to the variable in the calling function.

Capacity

As you've seen, a slice is a sequence of values. Each element in a slice is assigned to consecutive memory locations, which makes it quick to read or write these values. The length of a slice is the number of consecutive memory locations that have been assigned a value. Every slice also has a *capacity*, which is the number of consecutive memory locations reserved. This can be larger than the length. Each time you append to a slice, one or more values is added to the end of the slice. Each value added increases the length by one. When the length reaches the capacity, there's no more room to put values. If you try to add additional values when the length equals the capacity, the `append` function uses the Go runtime to allocate a new backing array for the slice with a larger capacity. The values in the original backing array are copied to the new one, the new values are added to the end of the new backing array, and the slice is updated to refer to the new backing array. Finally, the updated slice is returned.

The Go Runtime

Every high-level language relies on a set of libraries to enable programs written in that language to run, and Go is no exception. The Go runtime provides services like memory allocation and garbage collection, concurrency support, networking, and implementations of built-in types and functions.

The Go runtime is compiled into every Go binary. This is different from languages that use a virtual machine, which must be installed separately to allow programs written in those languages to function. Including the runtime in the binary makes it easier to distribute Go programs and avoids worries about compatibility issues between the runtime and the program. The drawback of including the runtime in the binary is that even the simplest Go program produces a binary that's about 2 MB.

When a slice grows via `append`, it takes time for the Go runtime to allocate new memory and copy the existing data from the old memory to the new. The old memory also needs to be garbage collected. For this reason, the Go runtime usually increases a slice by more than one each time it runs out of capacity. The rule as of Go 1.18 is to double the capacity of a slice when the current capacity is less than 256. A bigger slice increases by `(current_capacity + 768)/4`. This slowly converges at 25% growth (a slice with capacity of 512 will grow by 63%, but a slice with capacity 4,096 will grow by only 30%).

Just as the built-in `len` function returns the current length of a slice, the built-in `cap` function returns the current capacity of a slice. It is used far less frequently than `len`. Most of the time, `cap` is used to check if a slice is large enough to hold new data, or if a call to `make` is needed to create a new slice.

You can also pass an array to the `cap` function, but `cap` always returns the same value as `len` for arrays. Don't put it in your code, but save this trick for Go trivia night.

Let's take a look at how adding elements to a slice changes the length and capacity. Run the code in Example 3-1 on The Go Playground or in the *sample_code/len_cap* directory in the Chapter 3 repository.

Example 3-1. Understanding capacity

```
var x []int
fmt.Println(x, len(x), cap(x))
x = append(x, 10)
fmt.Println(x, len(x), cap(x))
x = append(x, 20)
fmt.Println(x, len(x), cap(x))
x = append(x, 30)
fmt.Println(x, len(x), cap(x))
x = append(x, 40)
fmt.Println(x, len(x), cap(x))
x = append(x, 50)
fmt.Println(x, len(x), cap(x))
```

When you build and run the code, you'll see the following output. Notice how and when the capacity increases:

```
[] 0 0
[10] 1 1
[10 20] 2 2
[10 20 30] 3 4
[10 20 30 40] 4 4
[10 20 30 40 50] 5 8
```

While it's nice that slices grow automatically, it's far more efficient to size them once. If you know how many things you plan to put into a slice, create it with the correct initial capacity. You do that with the `make` function.

make

You've already seen two ways to declare a slice, using a slice literal or the `nil` zero value. While useful, neither way allows you to create an empty slice that already has a length or capacity specified. That's the job of the built-in `make` function. It allows you to specify the type, length, and, optionally, the capacity. Let's take a look:

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

This creates an `int` slice with a length of 5 and a capacity of 5. Since it has a length of 5, `x[0]` through `x[4]` are valid elements, and they are all initialized to 0.

One common beginner mistake is to try to populate those initial elements using `append`:

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

The 10 is placed at the end of the slice, *after* the zero values in elements 0–4 because `append` always increases the length of a slice. The value of `x` is now [0 0 0 0 0 10], with a length of 6 and a capacity of 10 (the capacity was doubled as soon as the sixth element was appended).

You can also specify an initial capacity with `make`:

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

This creates an `int` slice with a length of 5 and a capacity of 10.

You can also create a slice with zero length but a capacity that's greater than zero:

```
x := make([]int, 0, 10)
```

In this case, you have a non-`nil` slice with a length of 0 but a capacity of 10. Since the length is 0, you can't directly index into it, but you can append values to it:

```
x := make([]int, 0, 10)
x = append(x, 5,6,7,8)
```

The value of `x` is now [5 6 7 8], with a length of 4 and a capacity of 10.

Never specify a capacity that's less than the length! It is a compile-time error to do so with a constant or numeric literal. If you use a variable to specify a capacity that's smaller than the length, your program will panic at runtime.

Emptying a Slice

Go 1.21 added a `clear` function that takes in a slice and sets all of the slice's elements to their zero value. The length of the slice remains unchanged. The following code:

```
s := []string{"first", "second", "third"}
fmt.Println(s, len(s))
clear(s)
fmt.Println(s, len(s))
```

prints out:

```
[first second third] 3
[  ] 3
```

(Remember, the zero value for a string is an empty string ""!)

Declaring Your Slice

Now that you've seen all these ways to create slices, how do you choose which slice declaration style to use? The primary goal is to minimize the number of times the slice needs to grow. If it's possible that the slice won't need to grow at all, use a `var` declaration with no assigned value to create a `nil` slice, as shown in Example 3-2.

Example 3-2. Declaring a slice that might stay `nil`

```
var data []int
```

You can create a slice using an empty slice literal:

```
var x = []int{}
```

This creates a slice with zero length and zero capacity. It is confusingly different from a `nil` slice. Because of implementation reasons, comparing a zero-length slice to `nil` returns `false`, while comparing a `nil` slice to `nil` returns `true`. For simplicity, favor `nil` slices. A zero-length slice is useful only when converting a slice to JSON. You'll look at this more in "encoding/json" on page 327.

If you have some starting values, or if a slice's values aren't going to change, then a slice literal is a good choice (see Example 3-3).

Example 3-3. Declaring a slice with default values

```
data := []int{2, 4, 6, 8} // numbers we appreciate
```

If you have a good idea of how large your slice needs to be, but don't know what those values will be when you are writing the program, use `make`. The question then becomes whether you should specify a nonzero length in the call to `make` or specify a zero length and a nonzero capacity. There are three possibilities:

- If you are using a slice as a buffer (you'll see this in "io and Friends" on page 319), then specify a nonzero length.
- If you are *sure* you know the exact size you want, you can specify the length and index into the slice to set the values. This is often done when transforming values in one slice and storing them in a second. The downside to this approach is that if you have the size wrong, you'll end up with either zero values at the end of the slice or a panic from trying to access elements that don't exist.

- In other situations, use `make` with a zero length and a specified capacity. This allows you to use `append` to add items to the slice. If the number of items turns out to be smaller, you won't have an extraneous zero value at the end. If the number of items is larger, your code will not panic.

The Go community is split between the second and third approaches. I personally prefer using `append` with a slice initialized to a zero length. It might be slower in some situations, but it is less likely to introduce a bug.

`append` always increases the length of a slice! If you have specified a slice's length using `make`, be sure that you mean to append to it before you do so, or you might end up with a bunch of surprise zero values at the beginning of your slice.

Slicing Slices

A *slice expression* creates a slice from a slice. It's written inside brackets and consists of a starting offset and an ending offset, separated by a colon (:). The starting offset is the first position in the slice that is included in the new slice, and the ending offset is one past the last position to include. If you leave off the starting offset, 0 is assumed. Likewise, if you leave off the ending offset, the end of the slice is substituted. You can see how this works by running the code in Example 3-4 on The Go Playground or in the *sample_code/slicing_slices* directory in the Chapter 3 repository.

Example 3-4. Slicing slices

```
x := []string{"a", "b", "c", "d"}
y := x[:2]
z := x[1:]
d := x[1:3]
e := x[:]
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
fmt.Println("d:", d)
fmt.Println("e:", e)
```

It gives the following output:

```
x: [a b c d]
y: [a b]
z: [b c d]
d: [b c]
e: [a b c d]
```

When you take a slice from a slice, you are *not* making a copy of the data. Instead, you now have two variables that are sharing memory. This means that changes to an element in a slice affect all slices that share that element. Let's see what happens when you change values. You can run the code in Example 3-5 on The Go Playground or in the *sample_code/slice_share_storage* directory in the Chapter 3 repository.

Example 3-5. Slices with overlapping storage

```
x := []string{"a", "b", "c", "d"}
y := x[:2]
z := x[1:]
x[1] = "y"
y[0] = "x"
z[1] = "z"
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

You get the following output:

```
x: [x y z d]
y: [x y]
z: [y z d]
```

Changing `x` modified both `y` and `z`, while changes to `y` and `z` modified `x`.

Slicing slices gets extra confusing when combined with `append`. Try out the code in Example 3-6 on The Go Playground or in the *sample_code/slice_append_storage* directory in the Chapter 3 repository.

Example 3-6. `append` makes overlapping slices more confusing

```
x := []string{"a", "b", "c", "d"}
y := x[:2]
fmt.Println(cap(x), cap(y))
y = append(y, "z")
fmt.Println("x:", x)
fmt.Println("y:", y)
```

Running this code gives the following output:

```
4 4
x: [a b z d]
y: [a b z]
```

What's going on? Whenever you take a slice from another slice, the subslice's capacity is set to the capacity of the original slice, minus the starting offset of the subslice within the original slice. This means elements of the original slice beyond the end of the subslice, including unused capacity, are shared by both slices.

When you make the `y` slice from `x`, the length is set to 2, but the capacity is set to 4, the same as `x`. Since the capacity is 4, appending onto the end of `y` puts the value in the third position of `x`.

This behavior creates some odd scenarios, with multiple slices appending and overwriting each other's data. See if you can guess what the code in Example 3-7 prints out, then run it on The Go Playground or in the *sample_code/confusing_slices* directory in the Chapter 3 repository to see if you guessed correctly.

Example 3-7. Even more confusing slices

```
x := make([]string, 0, 5)
x = append(x, "a", "b", "c", "d")
y := x[:2]
z := x[2:]
fmt.Println(cap(x), cap(y), cap(z))
y = append(y, "i", "j", "k")
x = append(x, "x")
z = append(z, "y")
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

To avoid complicated slice situations, you should either never use `append` with a subslice or make sure that `append` doesn't cause an overwrite by using a *full slice expression*. This is a little weird, but it makes clear how much memory is shared between the parent slice and the subslice. The full slice expression includes a third part, which indicates the last position in the parent slice's capacity that's available for the subslice. Subtract the starting offset from this number to get the subslice's capacity. Example 3-8 shows the first four lines from the previous example, modified to use full slice expressions.

Example 3-8. The full slice expression protects against `append`

```
x := make([]string, 0, 5)
x = append(x, "a", "b", "c", "d")
y := x[:2:2]
z := x[2:4:4]
```

Try out this code on The Go Playground or in the *sample_code/full_slice_expression* directory in the Chapter 3 repository. Both `y` and `z` have a capacity of `2`. Because you limited the capacity of the subslices to their lengths, appending additional elements onto `y` and `z` created new slices that didn't interact with the other slices. After this code runs, `x` is set to `[a b c d x]`, `y` is set to `[a b i j k]`, and `z` is set to `[c d y]`.

Be careful when taking a slice of a slice! Both slices share the same memory, and changes to one are reflected in the other. Avoid modifying slices after they have been sliced or if they were produced by slicing. Use a three-part slice expression to prevent `append` from sharing capacity between slices.

copy

If you need to create a slice that's independent of the original, use the built-in `copy` function. Let's take a look at a simple example, which you can run on The Go Playground or in the *sample_code/copy_slice* directory in the Chapter 3 repository:

```
x := []int{1, 2, 3, 4}
y := make([]int, 4)
num := copy(y, x)
fmt.Println(y, num)
```

You get this output:

```
[1 2 3 4] 4
```

The `copy` function takes two parameters. The first is the destination slice, and the second is the source slice. The function copies as many values as it can from source to destination, limited by whichever slice is smaller, and returns the number of elements copied. The *capacity* of `x` and `y` doesn't matter; it's the length that's important.

You can also copy a subset of a slice. The following code copies the first two elements of a four-element slice into a two-element slice:

```
x := []int{1, 2, 3, 4}
y := make([]int, 2)
num := copy(y, x)
```

The variable `y` is set to [1 2], and `num` is set to 2.

You could also copy from the middle of the source slice:

```
x := []int{1, 2, 3, 4}
y := make([]int, 2)
copy(y, x[2:])
```

You are copying the third and fourth elements in `x` by taking a slice of the slice. Also note that *you don't assign the output of `copy` to a variable*. If you don't need the number of elements copied, you don't need to assign it.

The `copy` function allows you to copy between two slices that cover overlapping sections of an underlying slice:

```
x := []int{1, 2, 3, 4}
num := copy(x[:3], x[1:])
fmt.Println(x, num)
```

In this case, you are copying the last three values in `x` on top of the first three values of `x`. This prints out [2 3 4 4] 3.

You can use `copy` with arrays by taking a slice of the array. You can make the array either the source or the destination of the copy. You can try out the following code on The Go Playground or in the *sample_code/copy_array* directory in the Chapter 3 repository:

```
x := []int{1, 2, 3, 4}
d := [4]int{5, 6, 7, 8}
y := make([]int, 2)
copy(y, d[:])
fmt.Println(y)
copy(d[:], x)
fmt.Println(d)
```

The first call to `copy` copies the first two values in array `d` into slice `y`. The second copies all of the values in slice `x` into array `d`. This produces the following output:

```
[5 6]
[1 2 3 4]
```

Converting Arrays to Slices

Slices aren't the only thing you can slice. If you have an array, you can take a slice from it using a slice expression. This is a useful way to bridge an array to a function that takes only slices. To convert an entire array into a slice, use the `[:]` syntax:

```
xArray := [4]int{5, 6, 7, 8}
xSlice := xArray[:]
```

You can also convert a subset of an array into a slice:

```
x := [4]int{5, 6, 7, 8}
y := x[:2]
z := x[2:]
```

Be aware that taking a slice from an array has the same memory-sharing properties as taking a slice from a slice. If you run the following code on The Go Playground or in the *sample_code/slice_array_memory* directory in the Chapter 3 repository:

```
x := [4]int{5, 6, 7, 8}
y := x[:2]
z := x[2:]
x[0] = 10
fmt.Println("x:", x)
fmt.Println("y:", y)
fmt.Println("z:", z)
```

you get this output:

```
x: [10 6 7 8]
y: [10 6]
z: [7 8]
```

Converting Slices to Arrays

Use a type conversion to make an array variable from a slice. You can convert an entire slice to an array of the same type, or you can create an array from a subset of the slice.

When you convert a slice to an array, the data in the slice is copied to new memory. That means that changes to the slice won't affect the array, and vice versa.

The following code:

```
xSlice := []int{1, 2, 3, 4}
xArray := [4]int(xSlice)
smallArray := [2]int(xSlice)
xSlice[0] = 10
fmt.Println(xSlice)
fmt.Println(xArray)
fmt.Println(smallArray)
```

prints out:

```
[10 2 3 4]
[1 2 3 4]
[1 2]
```

The size of the array must be specified at compile time. It's a compile-time error to use `[...]` in a slice to array type conversion.

While the size of the array can be smaller than the size of the slice, it cannot be bigger. Unfortunately, the compiler cannot detect this, and your code will panic at runtime if you specify an array size that's bigger than the length (not the capacity) of the slice. The following code:

```
panicArray := [5]int(xSlice)
fmt.Println(panicArray)
```

panics at runtime with the message:

```
panic: runtime error: cannot convert slice with length 4 to array
     or pointer to array with length 5
```

I haven't talked about pointers yet, but you can also use a type conversion to convert a slice into a pointer to an array:

```
xSlice := []int{1,2,3,4}
xArrayPointer := (*[4]int)(xSlice)
```

After converting a slice to an array pointer, the storage between the two is shared. A change to one will change the other:

```
xSlice[0] = 10
xArrayPointer[1] = 20
fmt.Println(xSlice) // prints [10 20 3 4]
fmt.Println(xArrayPointer) // prints &[10 20 3 4]
```

Pointers are covered in Chapter 6.

You can try all of the array type conversions on The Go Playground or in the *sample_code/array_conversion* directory in the Chapter 3 repository.

In "Arrays—Too Rigid to Use Directly" on page 37, I mentioned that you can't use arrays as function parameters when the size of the array being passed in might vary. Technically, you can work around this limitation by converting an array to a slice, converting the slice to an array of a different size, and then passing the second array in to a function. The second array must be shorter than the first array, or your program will panic. While this might be helpful in a pinch, if you find yourself doing this frequently, strongly consider changing your function's API to take a slice instead of an array.

Strings and Runes and Bytes

Now that I've talked about slices, we can look at strings again. You might think that a string in Go is made out of runes, but that's not the case. Under the covers, Go uses a sequence of bytes to represent a string. These bytes don't have to be in any particular character encoding, but several Go library functions (and the `for-range` loop that I discuss in the next chapter) assume that a string is composed of a sequence of UTF-8-encoded code points.

According to the language specification, Go source code is always written in UTF-8. Unless you use hexadecimal escapes in a string literal, your string literals are written in UTF-8.

Just as you can extract a single value from an array or a slice, you can extract a single value from a string by using an *index expression*:

```
var s string = "Hello there"
var b byte = s[6]
```

Like arrays and slices, string indexes are zero-based; in this example, `b` is assigned the numeric value of the seventh position in `s`, which is `116` (the UTF-8 value of a lowercase t).

The slice expression notation that you used with arrays and slices also works with strings:

```
var s string = "Hello there"
var s2 string = s[4:7]
var s3 string = s[:5]
var s4 string = s[6:]
```

This assigns "o t" to `s2`, "Hello" to `s3`, and "there" to `s4`. You can try out this code on The Go Playground or in the *sample_code/string_slicing* directory in the Chapter 3 repository.

While it's handy that Go allows you to use slicing notation to make substrings and use index notation to extract individual entries from a string, you should be careful when doing so. Since strings are immutable, they don't have the modification problems that slices of slices do. There is a different problem, though. A string is composed of a sequence of bytes, while a code point in UTF-8 can be anywhere from one to four bytes long. The previous example was entirely composed of code points that are one byte long in UTF-8, so everything worked out as expected. But when dealing with languages other than English or with emojis, you run into code points that are multiple bytes long in UTF-8:

```
var s string = "Hello 🌞"
var s2 string = s[4:7]
var s3 string = s[:5]
var s4 string = s[6:]
```

In this example, `s3` will still be equal to "Hello." The variable `s4` is set to the sun emoji. But `s2` is not set to "o 🌞." Instead, you get "o  ." That's because you copied only the first byte of the sun emoji's code point, which is not a valid code point on its own.

Go allows you to pass a string to the built-in `len` function to find the length of the string. Given that string index and slice expressions count positions in bytes, it's not surprising that the length returned is the length in bytes, not in code points:

```
var s string = "Hello 🌞"
fmt.Println(len(s))
```

This code prints out 10, not 7, because it takes four bytes to represent the sun with smiling face emoji in UTF-8. You can run these sun emoji examples on The Go Playground or in the *sample_code/sun_slicing* directory in the Chapter 3 repository.

Even though Go allows you to use slicing and indexing syntax with strings, you should use it only when you know that your string contains only characters that take up one byte.

Because of this complicated relationship among runes, strings, and bytes, Go has some interesting type conversions between these types. A single rune or byte can be converted to a string:

```
var a rune    = 'x'
var s string  = string(a)
var b byte    = 'y'
var s2 string = string(b)
```

A common bug for new Go developers is to try to make an `int` into a `string` by using a type conversion:

```
var x int = 65
var y = string(x)
fmt.Println(y)
```

This results in `y` having the value "A," not "65." As of Go 1.15, `go vet` blocks a type conversion to string from any integer type other than `rune` or `byte`.

A string can be converted back and forth to a slice of bytes or a slice of runes. Try Example 3-9 on The Go Playground or in the *sample_code/string_to_slice* directory in the Chapter 3 repository.

Example 3-9. Converting strings to slices

```
var s string = "Hello, 🌞"
var bs []byte = []byte(s)
var rs []rune = []rune(s)
fmt.Println(bs)
fmt.Println(rs)
```

When you run this code, you see the following:

```
[72 101 108 108 111 44 32 240 159 140 158]
[72 101 108 108 111 44 32 127774]
```

The first output line has the string converted to UTF-8 bytes. The second has the string converted to runes.

Most data in Go is read and written as a sequence of bytes, so the most common string type conversions are back and forth with a slice of bytes. Slices of runes are uncommon.

UTF-8

UTF-8 is the most commonly used encoding for Unicode. Unicode uses four bytes (32 bits) to represent each *code point*, the technical name for each character and modifier. Given this, the simplest way to represent Unicode code points is to store four bytes for each code point. This is called UTF-32. It is mostly unused because it wastes so much space. Due to Unicode implementation details, 11 of the 32 bits are always zero. Another common encoding is UTF-16, which uses one or two 16-bit (2-byte) sequences to represent each code point. This is also wasteful; much of the content in the world is written using code points that fit into a single byte. And that's where UTF-8 comes in.

UTF-8 is clever. It lets you use a single byte to represent the Unicode characters whose values are below 128 (which includes all of the letters, numbers, and punctuation commonly used in English), but expands to a maximum of four bytes to represent Unicode code points with larger values. The result is that the *worst* case for UTF-8 is the same as using UTF-32. UTF-8 has some other nice properties. Unlike UTF-32 and UTF-16, you don't have to worry about little-endian versus big-endian. It also allows you to look at any byte in a sequence and tell if you are at the start of a UTF-8 sequence or somewhere in the middle. That means you can't accidentally read a character incorrectly.

The only downside is that you cannot randomly access a string encoded with UTF-8. While you can detect if you are in the middle of a character, you can't tell how many characters in you are. You need to start at the beginning of the string and count. Go doesn't require a string to be written in UTF-8 but strongly encourages it. You'll see how to work with UTF-8 strings in upcoming chapters.

Fun fact: UTF-8 was invented in 1992 by Ken Thompson and Rob Pike, two of the creators of Go.

Rather than use the slice and index expressions with strings, you should extract substrings and code points from strings using the functions in the `strings` and `unicode/utf8` packages in the standard library. In the next chapter, you'll see how to use a `for-range` loop to iterate over the code points in a string.

Maps

Slices are useful when you have sequential data. Like most languages, Go provides a built-in data type for situations where you want to associate one value to another. The map type is written as `map[keyType]valueType`. Let's take a look at a few ways to declare maps. First, you can use a `var` declaration to create a map variable that's set to its zero value:

```
var nilMap map[string]int
```

In this case, `nilMap` is declared to be a map with `string` keys and `int` values. The zero value for a map is `nil`. A `nil` map has a length of 0. Attempting to read a `nil` map always returns the zero value for the map's value type. However, *attempting to write to a `nil` map variable causes a panic*.

You can use a `:=` declaration to create a map variable by assigning it a *map literal*:

```
totalWins := map[string]int{}
```

In this case, you are using an empty map literal. This is not the same as a `nil` map. It has a length of 0, but you can read and write to a map assigned an empty map literal. Here's what a nonempty map literal looks like:

```
teams := map[string][]string {
    "Orcas": []string{"Fred", "Ralph", "Bijou"},
    "Lions": []string{"Sarah", "Peter", "Billie"},
    "Kittens": []string{"Waldo", "Raul", "Ze"},
}
```

A map literal's body is written as the key, followed by a colon (:), then the value. A comma separates each key-value pair in the map, even on the last line. In this example, the value is a slice of strings. The type of the value in a map can be anything. There are some restrictions on the types of the keys that I'll discuss in a bit.

If you know how many key-value pairs you intend to put in the map but don't know the exact values, you can use `make` to create a map with a default size:

```
ages := make(map[int][]string, 10)
```

Maps created with `make` still have a length of 0, and they can grow past the initially specified size.

Maps are like slices in several ways:

- Maps automatically grow as you add key-value pairs to them.
- If you know how many key-value pairs you plan to insert into a map, you can use `make` to create a map with a specific initial size.
- Passing a map to the `len` function tells you the number of key-value pairs in a map.

- The zero value for a map is `nil`.
- Maps are not comparable. You can check if they are equal to `nil`, but you cannot check if two maps have identical keys and values using == or differ using `!=`.

The key for a map can be any comparable type. This means *you cannot use a slice or a map as the key for a map*.

When should you use a map, and when should you use a slice? You should use slices for lists of data when the data should be processed sequentially or the order of the elements is important.

Maps are useful when you need to organize values using something other than an increasing integer value, such as a name.

What Is a Hash Map?

In computer science, a *map* is a data structure that associates (or maps) one value to another. Maps can be implemented in several ways, each with its own trade-offs. The map that's built into Go is a *hash map*, or *hash table*. If you aren't familiar with the concept, Chapter 5 in *Grokking Algorithms* by Aditya Bhargava (Manning) describes what a hash table is and why they are so useful.

It's great that Go includes a hash map implementation as part of the runtime, because building your own is hard to get right. If you'd like to learn more about how Go does it, watch "Inside the Map Implementation", a talk from GopherCon 2016 by Keith Randall.

Go doesn't require (or even allow) you to define your own hash algorithm or equality definition. Instead, the Go runtime that's compiled into every Go program has code that implements hash algorithms for all types that are allowed to be keys.

Reading and Writing a Map

Let's look at a short program that declares, writes to, and reads from a map. You can run the program in Example 3-10 on The Go Playground or in the *sample_code/map_read_write* directory in the Chapter 3 repository.

Example 3-10. Using a map

```
totalWins := map[string]int{}
totalWins["Orcas"] = 1
totalWins["Lions"] = 2
fmt.Println(totalWins["Orcas"])
fmt.Println(totalWins["Kittens"])
totalWins["Kittens"]++
```

```
fmt.Println(totalWins["Kittens"])
totalWins["Lions"] = 3
fmt.Println(totalWins["Lions"])
```

When you run this program, you'll see the following output:

```
1
0
1
3
```

You assign a value to a map key by putting the key within brackets and using `=` to specify the value, and you read the value assigned to a map key by putting the key within brackets. Note that you cannot use `:=` to assign a value to a map key.

When you try to read the value assigned to a map key that was never set, the map returns the zero value for the map's value type. In this case, the value type is an `int`, so you get back a 0. You can use the `++` operator to increment the numeric value for a map key. Because a map returns its zero value by default, this works even when there's no existing value associated with the key.

The comma ok Idiom

As you've seen, a map returns the zero value if you ask for the value associated with a key that's not in the map. This is handy when implementing things like the `totalWins` counter you saw earlier. However, you sometimes do need to find out if a key is in a map. Go provides the *comma ok idiom* to tell the difference between a key that's associated with a zero value and a key that's not in the map:

```
m := map[string]int{
    "hello": 5,
    "world": 0,
}
v, ok := m["hello"]
fmt.Println(v, ok)

v, ok = m["world"]
fmt.Println(v, ok)

v, ok = m["goodbye"]
fmt.Println(v, ok)
```

Rather than assign the result of a map read to a single variable, with the comma ok idiom you assign the results of a map read to two variables. The first gets the value associated with the key. The second value returned is a bool. It is usually named `ok`. If `ok` is `true`, the key is present in the map. If `ok` is `false`, the key is not present. In this example, the code prints out `5 true`, `0 true`, and `0 false`.

The comma ok idiom is used in Go when you want to differentiate between reading a value and getting back the zero value. You'll see it again when you read from channels in Chapter 12 and when you use type assertions in Chapter 7.

Deleting from Maps

Key-value pairs are removed from a map via the built-in `delete` function:

```
m := map[string]int{
    "hello": 5,
    "world": 10,
}
delete(m, "hello")
```

The `delete` function takes a map and a key and then removes the key-value pair with the specified key. If the key isn't present in the map or if the map is `nil`, nothing happens. The `delete` function doesn't return a value.

Emptying a Map

The `clear` function that you saw in "Emptying a Slice" on page 44 works on maps also. A cleared map has its length set to zero, unlike a cleared slice. The following code:

```
m := map[string]int{
    "hello": 5,
    "world": 10,
}
fmt.Println(m, len(m))
clear(m)
fmt.Println(m, len(m))
```

prints out:

```
map[hello:5 world:10] 2
map[] 0
```

Comparing Maps

Go 1.21 added a package to the standard library called `maps` that contains helper functions for working with maps. You'll learn more about this package in "Adding Generics to the Standard Library" on page 201. Two functions in the package are useful for comparing if two maps are equal, `maps.Equal` and `maps.EqualFunc`. They are analogous to the `slices.Equal` and `slices.EqualFunc` functions:

```
m := map[string]int{
    "hello": 5,
    "world": 10,
```

```
}
n := map[string]int{
    "world": 10,
    "hello": 5,
}
fmt.Println(maps.Equal(m, n)) // prints true
```

Using Maps as Sets

Many languages include a set in their standard library. A *set* is a data type that ensures there is at most one of a value, but doesn't guarantee that the values are in any particular order. Checking to see if an element is in a set is fast, no matter how many elements are in the set. (Checking to see if an element is in a slice takes longer, as you add more elements to the slice.)

Go doesn't include a set, but you can use a map to simulate some of its features. Use the key of the map for the type that you want to put into the set and use a `bool` for the value. The code in Example 3-11 demonstrates the concept. You can run it on The Go Playground or in the *sample_code/map_set* directory in the Chapter 3 repository.

Example 3-11. Using a map as a set

```
intSet := map[int]bool{}
vals := []int{5, 10, 2, 5, 8, 7, 3, 9, 1, 2, 10}
for _, v := range vals {
    intSet[v] = true
}
fmt.Println(len(vals), len(intSet))
fmt.Println(intSet[5])
fmt.Println(intSet[500])
if intSet[100] {
    fmt.Println("100 is in the set")
}
```

You want a set of `int`s, so you create a map where the keys are of `int` type and the values are of `bool` type. You iterate over the values in `vals` using a `for-range` loop (which I discuss in "The for-range Statement" on page 76) to place them into `intSet`, associating each `int` with the boolean value `true`.

We wrote 11 values into `intSet`, but the length of `intSet` is 8, because you cannot have duplicate keys in a map. If you look for 5 in `intSet`, it returns `true`, because there is a key with the value 5. However, if you look for 500 or 100 in `intSet`, it returns `false`. This is because you haven't put either value into `intSet`, which causes the map to return the zero value for the map value, and the zero value for a `bool` is `false`.

If you need sets that provide operations like union, intersection, and subtraction, you can either write one yourself or use one of the many third-party libraries that provide the functionality. (You'll learn more about using third-party libraries in Chapter 10.)

Some people prefer to use `struct{}` for the value when a map is being used to implement a set. (I'll discuss structs in the next section.) The advantage is that an empty struct uses zero bytes, while a boolean uses one byte.

The disadvantage is that using a `struct{}` makes your code clumsier. You have a less obvious assignment, and you need to use the comma ok idiom to check if a value is in the set:

```
intSet := map[int]struct{}{}
vals := []int{5, 10, 2, 5, 8, 7, 3, 9, 1, 2, 10}
for _, v := range vals {
    intSet[v] = struct{}{}
}
if _, ok := intSet[5]; ok {
    fmt.Println("5 is in the set")
}
```

Unless you have very large sets, the difference in memory usage will not likely be significant enough to outweigh the disadvantages.

Structs

Maps are a convenient way to store some kinds of data, but they have limitations. They don't define an API since there's no way to constrain a map to allow only certain keys. Also, all values in a map must be of the same type. For these reasons, maps are not an ideal way to pass data from function to function. When you have related data that you want to group together, you should define a *struct*.

If you already know an object-oriented language, you might be wondering about the difference between classes and structs. The difference is simple: Go doesn't have classes, because it doesn't have inheritance. This doesn't mean that Go doesn't have some of the features of object-oriented languages it just does things a little differently. You'll learn more about the object-oriented features of Go in Chapter 7.

Most languages have a concept that's similar to a struct, and the syntax that Go uses to read and write structs should look familiar:

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

```
    pet  string
}
```

A struct type is defined with the keyword `type`, the name of the struct type, the keyword `struct`, and a pair of braces (`{}`). Within the braces, you list the fields in the struct. Just as you put the variable name first and the variable type second in a `var` declaration, you put the struct field name first and the struct field type second. Also note that unlike in map literals, no commas separate the fields in a struct declaration. You can define a struct type inside or outside of a function. A struct type that's defined within a function can be used only within that function. (You'll learn more about functions in Chapter 5.)

Technically, you can scope a struct definition to any block level. You'll learn more about blocks in Chapter 4.

Once a struct type is declared, you can define variables of that type:

```
var fred person
```

Here we are using a `var` declaration. Since no value is assigned to `fred`, it gets the zero value for the `person` struct type. A zero value struct has every field set to the field's zero value.

A *struct literal* can be assigned to a variable as well:

```
bob := person{}
```

Unlike maps, there is no difference between assigning an empty struct literal and not assigning a value at all. Both initialize all fields in the struct to their zero values. There are two styles for a nonempty struct literal. First, a struct literal can be specified as a comma-separated list of values for the fields inside of braces:

```
julia := person{
    "Julia",
    40,
    "cat",
}
```

When using this struct literal format, a value for every field in the struct must be specified, and the values are assigned to the fields in the order they were declared in the struct definition.

The second struct literal style looks like the map literal style:

```
beth := person{
    age:  30,
```

```
    name: "Beth",
}
```

You use the names of the fields in the struct to specify the values. This style has some advantages. It allows you to specify the fields in any order, and you don't need to provide a value for all fields. Any field not specified is set to its zero value.

You cannot mix the two struct literal styles: either all fields are specified with names, or none of them are. For small structs where all fields are always specified, the simpler struct literal style is fine. In other cases, use the names. It's more verbose, but it makes clear what value is being assigned to what field without having to reference the struct definition. It's also more maintainable. If you initialize a struct without using the field names and a future version of the struct adds additional fields, your code will no longer compile.

A field in a struct is accessed with dot notation:

```
bob.name = "Bob"
fmt.Println(bob.name)
```

Just as you use brackets for both reading and writing to a map, you use dotted notation for reading and writing to struct fields.

Anonymous Structs

You can also declare that a variable implements a struct type without first giving the struct type a name. This is called an *anonymous struct*:

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

person.name = "bob"
person.age = 50
person.pet = "dog"

pet := struct {
    name string
    kind string
}{
    name: "Fido",
    kind: "dog",
}
```

In this example, the types of the variables `person` and `pet` are anonymous structs. You assign (and read) fields in an anonymous struct just as you do for a named struct type. Just as you can initialize an instance of a named struct with a struct literal, you can do the same for an anonymous struct as well.

You might wonder when it's useful to have a data type that's associated only with a single instance. Anonymous structs are handy in two common situations. The first is when you translate external data into a struct or a struct into external data (like JSON or Protocol Buffers). This is called *unmarshaling* and *marshaling* data, respectively. You'll learn how to do this in "encoding/json" on page 327.

Writing tests is another place where anonymous structs pop up. You'll use a slice of anonymous structs when writing table-driven tests in Chapter 15.

Comparing and Converting Structs

Whether a struct is comparable depends on the struct's fields. Structs that are entirely composed of comparable types are comparable; those with slice or map fields are not (as you will see in later chapters, function and channel fields also prevent a struct from being comparable).

Unlike in Python or Ruby, in Go there's no magic method that can be overridden to redefine equality and make `==` and `!=` work for incomparable structs. You can, of course, write your own function that you use to compare structs.

Just as Go doesn't allow comparisons between variables of different primitive types, Go doesn't allow comparisons between variables that represent structs of different types. Go does allow you to perform a type conversion from one struct type to another *if the fields of both structs have the same names, order, and types*. Let's see what this means. Given this struct:

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

you can use a type conversion to convert an instance of `firstPerson` to `secondPerson`, but you can't use `==` to compare an instance of `firstPerson` and an instance of `secondPerson`, because they are different types:

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

You can't convert an instance of `firstPerson` to `thirdPerson`, because the fields are in a different order:

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

You can't convert an instance of `firstPerson` to `fourthPerson` because the field names don't match:

```
type fourthPerson struct {
    firstName string
    age       int
}
```

Finally, you can't convert an instance of `firstPerson` to `fifthPerson` because there's an additional field:

```
type fifthPerson struct {
    name          string
    age           int
    favoriteColor string
}
```

Anonymous structs add a small twist: if two struct variables are being compared and at least one has a type that's an anonymous struct, you can compare them without a type conversion if the fields of both structs have the same names, order, and types. You can also assign between named and anonymous struct types if the fields of both structs have the same names, order, and types:

```
type firstPerson struct {
    name string
    age  int
}
f := firstPerson{
    name: "Bob",
    age:  50,
}
var g struct {
    name string
    age  int
}

// compiles -- can use = and == between identical named and anonymous structs
g = f
fmt.Println(f == g)
```

Exercises

The following exercises will test what you've learned about Go's composite types. You can find solutions in the *exercise_solutions* directory in the Chapter 3 Repository.

1. Write a program that defines a variable named `greetings` of type slice of strings with the following values: `"Hello"`, `"Hola"`, `"नमस्कार"`, `"こんにちは"`, and `"Привіт"`. Create a subslice containing the first two values; a second subslice with the second, third, and fourth values; and a third subslice with the fourth and fifth values. Print out all four slices.

2. Write a program that defines a string variable called `message` with the value "`Hi` 👩 `and` 👨" and prints the fourth rune in it as a character, not a number.
3. Write a program that defines a struct called `Employee` with three fields: `firstName`, `lastName`, and `id`. The first two fields are of type `string`, and the last field (`id`) is of type `int`. Create three instances of this struct using whatever values you'd like. Initialize the first one using the struct literal style without names, the second using the struct literal style with names, and the third with a `var` declaration. Use dot notation to populate the fields in the third struct. Print out all three structs.

Wrapping Up

You've learned a lot about composite types in Go. In addition to learning more about strings, you now know how to use the built-in generic container types: slices and maps. You can also construct your own composite types via structs. In the next chapter, you're going to take a look at Go's control structures: `for`, `if/else`, and `switch`. You will also learn how Go organizes code into blocks and how the different block levels can lead to surprising behavior.