More types: structs, slices, and maps.

Learn how to define types based on existing ones: this lesson covers structs, arrays, slices, and maps.

The Go Authors

https://golang.org

*** Pointers**

Go has pointers.

A pointer holds the memory address of a value.

The type `*T` is a pointer to a `T` value. Its zero value is `nil`.

```
 var p *int
```

The `&` operator generates a pointer to its operand.

```
 i := 42

 p = &i
```

The `*` operator denotes the pointer's underlying value.

```
 fmt.Println(*p) // read i through the pointer p
 *p = 21         // set i through the pointer p
```

This is known as "dereferencing" or "indirecting".

Unlike C, Go has no pointer arithmetic.

*** Structs**

A `struct` is a collection of fields.

*** Struct Fields**

Struct fields are accessed using a dot.

*** Pointers to structs**

Struct fields can be accessed through a struct pointer.

To access the field `X` of a struct when we have the struct pointer `p` we could

write `(*p).X`.

However, that notation is cumbersome, so the language permits us instead to

write just `p.X`, without the explicit dereference.

*** Struct Literals**

A struct literal denotes a newly allocated struct value by listing the values of its fields.

You can list just a subset of fields by using the `Name:` syntax. (And the order of named fields is irrelevant.)

The special prefix `&` returns a pointer to the struct value.

*** Arrays**

The type `[n]T` is an array of `n` values of type `T`.

The expression

```
 var a [10]int
```

declares a variable `a` as an array of ten integers.

An array's length is part of its type, so arrays cannot be resized.

This seems limiting, but don't worry;

Go provides a convenient way of working with arrays.

*** Slices**

An array has a fixed size.

A slice, on the other hand, is a dynamically-sized,

flexible view into the elements of an array.

In practice, slices are much more common than arrays.

The type `[]T` is a slice with elements of type `T`.

A slice is formed by specifying two indices, a low and

high bound, separated by a colon:

```
 a[low : high]
```

This selects a half-open range which includes the first

element, but excludes the last one.

The following expression creates a slice which includes

elements 1 through 3 of `a`:

```
 a[1:4]
```

*** Slices are like references to arrays**

A slice does not store any data,

it just describes a section of an underlying array.

Changing the elements of a slice modifies the

corresponding elements of its underlying array.

Other slices that share the same underlying array will see those changes.

*** Slice literals**

A slice literal is like an array literal without the length.

This is an array literal:

```
[3]bool{true, true, false}
```

And this creates the same array as above,

then builds a slice that references it:

```
[]bool{true, true, false}
```

*** Slice defaults**

When slicing, you may omit the high or low bounds to use their defaults instead.

The default is zero for the low bound and the length of the slice for the high bound.

For the array

```
 var a [10]int
```

these slice expressions are equivalent:

```
 a[0:10]
 a[:10]
 a[0:]
 a[:]
```

*** Slice length and capacity**

A slice has both a _length_ and a _capacity_.

The length of a slice is the number of elements it contains.

The capacity of a slice is the number of elements in the underlying array,

counting from the first element in the slice.

The length and capacity of a slice `s` can be obtained using the expressions

`len(s)` and `cap(s)`.

You can extend a slice's length by re-slicing it,

provided it has sufficient capacity.

Try changing one of the slice operations in the example program to extend it

beyond its capacity and see what happens.

*** Nil slices**

The zero value of a slice is `nil`.

A nil slice has a length and capacity of 0

and has no underlying array.

*** Creating a slice with make**

Slices can be created with the built-in `make` function;

this is how you create dynamically-sized arrays.

The `make` function allocates a zeroed array

and returns a slice that refers to that array:

```
 a := make([]int, 5)  // len(a)=5
```

To specify a capacity, pass a third argument to `make`:

```
 b := make([]int, 0, 5) // len(b)=0, cap(b)=5

 b = b[:cap(b)] // len(b)=5, cap(b)=5
 b = b[1:]      // len(b)=4, cap(b)=4
```

*** Slices of slices**

Slices can contain any type, including other slices.

*** Appending to a slice**

It is common to append new elements to a slice, and so Go provides a built-in
`append` function. The [[/pkg/builtin/#append][documentation]]
of the built-in package describes `append`.

```
 func append(s []T, vs ...T) []T
```

The first parameter `s` of `append` is a slice of type `T`, and the rest are
`T` values to append to the slice.

The resulting value of `append` is a slice containing all the elements of the
original slice plus the provided values.

If the backing array of `s` is too small to fit all the given values a bigger
array will be allocated. The returned slice will point to the newly allocated
array.

(To learn more about slices, read the [[/blog/go-slices-usage-and-internals][Slices: usage and internals]] article.)

*** Range**

The `range` form of the `for` loop iterates over a slice or map.

When ranging over a slice, two values are returned for each iteration.

The first is the index, and the second is a copy of the element at that index.

*** Range continued**

You can skip the index or value by assigning to `_`.

```
    for i, _ := range pow
    for _, value := range pow
```

If you only want the index, you can omit the second variable.

```
    for i := range pow
```

*** Exercise: Slices**

Implement `Pic`. It should return a slice of length `dy`, each element of which is a slice of `dx` 8-bit unsigned integers. When you run the program, it will display your picture, interpreting the integers as grayscale (well, bluescale) values.

The choice of image is up to you. Interesting functions include `(x+y)/2`, `x*y`, and `x^y`.

(You need to use a loop to allocate each `[]uint8` inside the `[][]uint8`.)

(Use `uint8(intValue)` to convert between types.)

*** Maps**

A map maps keys to values.

The zero value of a map is `nil`.

A `nil` map has no keys, nor can keys be added.

The `make` function returns a map of the given type,

initialized and ready for use.

*** Map literals**

Map literals are like struct literals, but the keys are required.

*** Map literals continued**

If the top-level type is just a type name, you can omit it from the elements of the literal.

*** Mutating Maps**

Insert or update an element in map `m`:

```
 m[key] = elem
```

Retrieve an element:

```
 elem = m[key]
```

Delete an element:

```
 delete(m, key)
```

Test that a key is present with a two-value assignment:

```
 elem, ok = m[key]
```

If `key` is in `m`, `ok` is `true`. If not, `ok` is `false`.

If `key` is not in the map, then `elem` is the zero value for the map's element type.

***Note:* If `elem` or `ok` have not yet been declared you could use a short declaration form:**

```
 elem, ok := m[key]
```

*** Exercise: Maps**

Implement `WordCount`. It should return a map of the counts of each “word” in the string `s`. The `wc.Test` function runs a test suite against the provided function and prints success or failure.

You might find [[/pkg/strings/#Fields][strings.Fields]] helpful.

*** Function values**

Functions are values too. They can be passed around just like other values.

Function values may be used as function arguments and return values.

*** Function closures**

Go functions may be closures. A closure is a function value that references variables from outside its body. The function may access and assign to the referenced variables; in this sense the function is "bound" to the variables.

For example, the `adder` function returns a closure. Each closure is bound to its own `sum` variable.

*** Exercise: Fibonacci closure**

Let's have some fun with functions.

Implement a `fibonacci` function that returns a function (a closure) that

returns successive [[https://en.wikipedia.org/wiki/Fibonacci_number][fibonacci numbers]]
(0, 1, 1, 2, 3, 5, ...).

*** Congratulations!**

You finished this lesson!

You can go back to the list of [[/tour/list][modules]] to find what to learn next, or continue with the [[javascript:click('.next-page')][next lesson]].