Slices with Functions

This lesson explains handling slices as parameters to the functions in detail.

Passing a slice to a function

If you have a function that must operate on an array, you probably always want to declare the formal parameter to be a slice. When you call the function, slice the array to create (efficiently) a slice reference and pass that. For example, here is a program that sums all elements in an array:

```go
package main
import "fmt"

func sum(a []int) int {   // function that sums integers
  s := 0
  for i := 0; i < len(a); i++ {
    s += a[i]
  }
  return s
}

func main() {
  var arr = [5]int{0,1,2,3,4} // declare an array
  fmt.Println(sum(arr[:]))    // passing slice to the function
}
```

Passing Slice to a Function

In the above program, we have a function `sum`, that takes an array `a` as a parameter and returns the sum of the elements present in `a`. Look at its header at **line 4**. In the `main` function, at **line 13**, we declare an array `arr` of length **5** and pass it to the `sum` function on the next line. We write `arr[:]`, which is enough to pass the array `arr` as the copy.

Creating a slice with `make()`

Often the underlying array is not yet defined. In that case, we can make the slice together with the array using the `make()` function:

```
var slice1 []type = make([]type, len)
```

which can be shortened to:

```
slice1 := make([]type, len)
```

where `len` is the length of the array and also the initial length of the slice. Therefore, for the slice `s2` made with:

```
s2 := make([]int, 10)
```

the following is true:

```
cap(s2) == len(s2) == 10
```

The function `make` takes 2 parameters:

- the type to be created
- the number of items in the slice. If you want `slice1` not to occupy the whole array (with length cap) from the start, but only a number `len` of items, use the form:

```
slice1 := make([]type, len, cap)
```

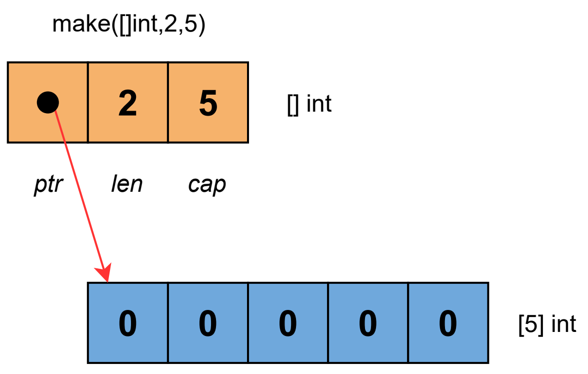
So `make` has the signature:

```
func make([]T, len, cap) // []T with type T and optional parameter cap
```

And the following statements result in the same slice:

```
make([]int, 50, 100)
new([100]int)[0:50]
```

The following figure depicts the making of a slice in memory:



Slice in Memory

The following program is an implementation of making slices through the `make` function.

```
package main
import "fmt"

func main() {
  var slice1 []int = make([]int, 10)
  // load the array/slice:
  for i := 0; i < len(slice1); i++ {
    slice1[i] = 5 * i
  }
  // print the slice:
  for i := 0; i < len(slice1); i++ {
    fmt.Printf("Slice at %d is %d\n", i, slice1[i])
  }
  fmt.Printf("\nThe length of slice1 is %d\n", len(slice1))
  fmt.Printf("The capacity of slice1 is %d\n", cap(slice1))
}
```

Slice via make()

In the program above, we make a slice via the `make` function. Look at **line 5** in `main`. We make an integer type slice `slice1` with the length **10**. The loop at **line 7** is populating the values in `slice1`. Each element at index `i` is given a value of `5*i` at **line 8**. The next *for* loop at **line 11** is printing the slice elements. Then, at **line 14** and **line 15**, we are printing the length and capacity of `slice1` using the `len()` and `cap()` functions, which are both **10**.

Difference between `new()` and `make()`

This is often confusing at first sight: both allocate memory on the heap, but they do different things and apply to different types.

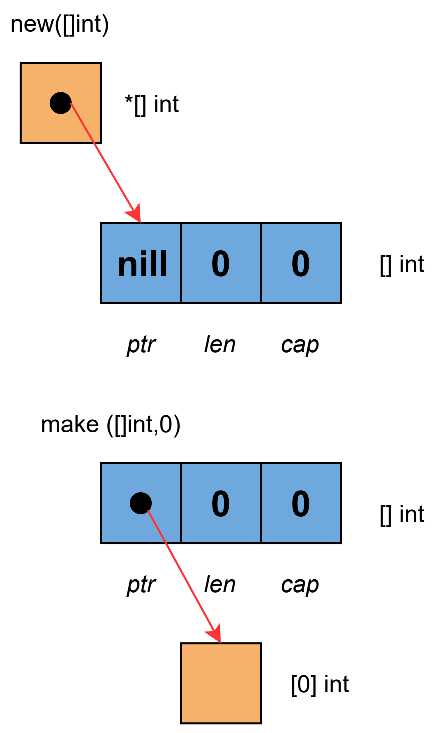
`new(T)`

The function `new(T)` allocates zeroed storage for a new item of type **T** and returns its address as a value of type ***T**. It applies to value types like arrays and structs (see Chapter 8), and it is equivalent to **&T{ }**

`make(T)`

The function `make(T)` returns an initialized value of type **T**. It applies only to the 3 built-in reference types: slices, maps, and channels (see Chapters 6 and Chapter 12).

In other words, `new` allocates and `make` initializes; the following figure illustrates this difference:



Difference b/w new() and make()

The function `new` works as:

```
var p *[]int = new([]int) // *p == nil; with len and cap 0
```

or

```
p := new([]int)
```

Whereas, `make` works as:

```
p := make([]int, 0)
```

Here, our slice is initialized, but it points to an empty array. Both these statements aren't very useful, but the following is:

```
var v []int = make([]int, 10, 50)
or v := make([]int, 10, 50)
```

This allocates an array of **50** integers and then creates a slice `v` with length 10 and the capacity 50, which points to the first 10 elements of the array.

Now, that you are familiar with the use of slices, try your hand at writing a function to solve a problem in the next lesson.