CMSC 245 – Principles of Programming Languages
Lab#4: Arrays, Slices, Functions, Command Line Arguments

In this lab we will learn about arrays, slices, and functions in Go. We recommend that you try all the features presented in this lab using the Go Playground (https://play.golang.org/), a web-based Go learning tool where you can enter short Go programs and run them. As you proceed, whenever you have any questions, go ahead and try them out in code ro get your answers. This is the best way to learn Go.

Structure of a Go Program

The general structure of a Go program is shown below:

```
package main

<import packages>

func main() {
   <code for main() goes here>
} // main()
```

To run a Go program use the command:

```
> go run <filename>.go
```

Arrays

In Go, like in C or Java, an array is an indexed sequence of elements of the same type. For example, here are some simple declarations:

```
var a [10]int
data := [5]int{10, 20, 30, 40, 50}
primes := [5]{int
         2,
         3,
         5,
         7,
         11,
}
```

Here is a short synopsis of the above definitions:

- **a** is an array of 10 integers (**int**). Go initializes all elements to 0.
- **data** is an array of 5 integers and initialized using the values supplied.
- **primes** is an array of 5 integers. The values are each provided on a separate line. The comma (**,**) following the last value (**11**) is required. Why?
- In all cases index begins with 0. To access the elements of a you write **a[0]**, **a[1]**, …, **a[9]**.

- The array is created at the time of definition. And, its size remains fixed once created. That is, the size of the array is part of its type.
- As long as two arrays are the same size and type, they can be assigned/copied using the assignment (**=**) operator:

  ```
  primes := [5]int{2, 3, 5, 7, 11}
  var p [5]int
  p = primes
  ```

  `p[]` will contain a copy of all the elements of **`primes[]`** after the last statement.
- Elements of an array can be printed using the **`fmt.Println()`** function:

  ```
  fmt.Println(primes)
  [2 3 5 7 11]
  ```

- To iterate through all elements of an array:

  ```
  sum := 0
  for i:= 0; I < len(a); i++ {
      sum = sum + a[i]
  ```

  Or, you can use a "range" loop:

  ```
  sum := 0
  for _, x := range a {
      sum = sum + x
  }
  ```

  The expression "**`range a`**" returns two values (*index*, *element at index*). However, since we will not be using the *index*, we need to use the anonymous variable (**_**).

Slices

Slices get around the limitations of fixed size arrays (though they are also of fixed size!). They are like arrays, and can be defined without specifying a size:

```
var a []int
```

Above, no length is specified. In fact, an array of length 0 is created. In order to create a slice of a given length you can either copy a slice of another array into it:

```
a = primes[0:3]
fmt.Println(a)
[2 3 5]
```

Now the length of **`a[]`** will be 3. However, it is NOT a copy. I.e. any change to **`a[]`** will also affect **`primes[]`**. See below.

Alternately, you can create a new slice by using the **`make()`** function:

```
a = make([]int, 3)
```

Creates an empty array of 3 integers. If you want to create a slice that has a certain length, but also provide some capacity to grow, you can use:

```
a = make([]int, 3, 6)
```

It creates an array of three elements (initialized to 0), but with a capacity of 6. That is, after the above length of a will be 3. But, you will be able to assign a fourth, fifth, and a sixth element to **`a[]`**:

```
A[3] = 10
a[4] = 11
fmt.Println(a)
[0 0 0 10 11]
fmt.Println(len(a))
5
```

Like arrays, slices can be assigned:

```
primes := [5]int{2, 3, 5, 7, 11}
var p [5]int

p = primes[:]
```

However, this is a shallow copy. **`p[]`** and **`primes[]`** refer to the same array. In order to create a copy of **`primes[]`** into **`p[]`** (a deep copy), you have to use the **`copy()`** function:

```
copy(p, primes)
```

Otherwise, slices can be accessed, and processed, just like arrays. Slices are useful when writing functions that take array parameters. We will see that in the next section.

Functions

Functions in Go require you to specify the number and type of all parameters, as well as the return type. Here is a simple function:

```
func max(a int, b int) int {
    if a > b {
        return a
    } else {
        return b
    }
} // max()
```

The syntax is:

func *function-name*(*parameters*) *return-spec* {
   function-body
}

Array/Slice Parameters
To compute the maximum value in an array, you will need to specify the array formal parameter as a slice (or the type of formal and actual array parameter will have to match, but that is too restrictive and seldom used):

```
func max(a []int) int {
   m := a[0]
   for i:=1; i < len(a); i++ {
      if a[i] > m {
            m = a[i]
      }
   }
   return m
}
```

Now, you can call max() with any slice as its actual parameter as long as it is a slice of integers.

Named Return Types
In Go, you can specify a variable name for the return type:

```
func max(a []int) (m int) {
   m = a[0]
   for i:=1; i < len(a); i++ {
      if a[i] > m {
            m = a[i]
      }
   }
```

```
    return
}
```

There are three important differences between the function we wrote above:

1. In the function header, we now specify the name of the returned variable: **`(m int)`**
2. Because that is equivalent to a local variable declaration, we changed the initial assignment: **`m = a[0]`**
3. The **`return`** statement no longer needs to mention the name of the returned variable (since it is declared in the function header). Though, Go will also allow: **`return m`**

More importantly, **`max()`** can now be used to compute the largest value in any integer slice of any length.

```
primes := []int{2, 3, 5, 7, 11}
p := []int[42, 32, 67]

fmt.Println(max(primes))
fmt.Println(max(p))

11
67
```

Multiple Return Values
Functions can also return more than one return value. For example, if we wanted to find out the value and index of the largest value in a slice:

```
func max(a []int) (i int, m int) {
	m = a[0]
	i = 0
	for j := 0; j < len(a); j++ {
		if a[j] > m {
			m = a[j]
			i = j
		}
	}
	return
}
```

You can now call the function max as shown below:

```
index, value := max(primes)
```

Many functions in Go take advantage of this feature.

Variable Number of Arguments – Variadic Functions

Go allows you to write functions that may take an unknown number of parameters. For example, to compute the largest of two or more numbers:

```
a = max(x, y)
b = max(x, y, z)
c = max(w, x, y, z)
```

Such functions are written as shown below:

```
func max(a int, args ...int) (m int) {
	m = a
	for _, x := range args {
		if x > m {
			m = x
		}
	}
	return
}
```

Above, the ellipsis (…) before **`int`** in the formal parameter specifies that the parameter **`args`** could be zero or more integers. We can then traverse **`args`** as if it were a slice (**`[]int`**). Additionally, wherever a *variadic parameter* is used, we can pass a slice as an actual parameter by following the name of the slice with ellipsis:

```
m := max(2, primes…)
```

Finally, Go allows you to define local functions within a function. We will not cover that here.

Command Line Arguments

Like in C, C++, Java, Python, Go programs accept command line arguments. The **os** package provides the string slice **Args[]**, that contains one string for each command line argument. **Args[0]** is the command itself.

The program below prints out the **Args[]** slice, followed by the integer value of the first command line argument (a number):

```
package main
import (
    "fmt"
    "os"
    "strconv"
)

func main() {
    fmt.Println(os.Args)
    if n, err := strconv.Atoi(os.Args[1]); err == nil {
        fmt.Println(n)
    }
}
```

strconv.Atoi() converts a string containing a number to an **int**. It returns two values, the number itself, and an error value (in **err**) in case there it was unable to convert the string into an **int**. The error value returned is **nil** if successful. The program above, tests to see if **err** is **nil** before printing the value of the integer command line argument.

The flags package provides more elaborate command line argument parsing when flags might be supplied. We will leave that for self-study for another time.

Exercise (Assignment 4)

1. Collecting Cards in Candy: A candy company wishes to run a promotion in the next quarter. Every candy wrapper will contain a picture of a player from the Womens National Soccer Team. There are 28 players on the roster, plus 7 coaches, making a total of 35 pictures. Pictures will be randomly inserted in each candy wrapper. The company hopes that kids buying their candy will engage in collecting all 35 pictures and placing them in a picture album (sold separately by the company). Each picture will carry a unique number (from 0..34).

Before running with the idea the company wants to examine how this promotion might improve sales of its candy.

Write a program in Go that performs a simulation to determine the number of candies a kid would have to purchase in order to collect all 35 cards. Your program should input the number of picture (say 35) on the command line argument:

```
$ go run collector.go 35
XXX
```

The program will print out the expected number of candies (**XXX**) that kids will purchase to collect all 35 cards. Here is a possible design:

1. Write a function **`collect(c)`** that will perform a simulation of collecting **`c`** cards. It will return the total number of candies bought.
2. Once written, you can use **`collect()`** to perform several trials and keep track of the number of candies bought:

   ```
   n = Number of cards to collect
   trials = 1000000
   sum = 0
   do trials times:
     sum = sum + collect(n)
   output sum/trials as the average number of candies to buy in order to collect n cards
   ```

   As the number of trials increases, the data will start to converge. Run your program to perform 1000,000 trials.
3. Run the program with 1000,000 trials to compute the number of candies required to be bought in order to collect 35, 100 pictures, and 500 pictures.