Arrays and Slices

02-201 / 02-601

Arrays

Arrays Store Lists of Variables

3	12	3	3	7	8	10	-2	30	6	11	11	11	32	64	80	99	-1	0	12

- A list of filenames
- A list of prime numbers
- A column of data from a spreadsheet
- A collection of DNA sequences
- Factors of a number
- etc.

Arrays are fundamental *data structures*
Useful whenever you have a collection of things you want to work with together.

Declaring Arrays

```
var a [10]int
var b [100]string
var c [10*10]float64
```

Declares arrays of the given type and *length*.

a[0] a[5] a[i] a[i+3]

Expression inside the [] must be constant when array is declared
(it can't depend on variables or function calls):

```
var d [10-6 + 2]int    // ok
var size int = 10000
var e [size]int        // ERROR! "size" is not a constant
```

array elements:	13	18	-2	10	11	10	-22	8	8	7	-30	-33	-22	12	99	98	97	6	-3	2
index into array:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Accessing and Changing Array Elements

Array *elements* can be accessed by putting their *index* between [ ] following the array name:

```
fmt.Println(a[7],a[8])
a[0] = 10
b[30] = “hi there”
i := 12 + 2
c[i] = 3.1
c[2*i] = c[i]
```

os.Args[i]

x[i] can appear on left-hand side of assignment to set a value.

- The length of an array can be found with **len**(x), where x is an array.
- Array indices **start at 0!** The first element is x[0].
- The last element is at index **len(x) - 1.**
- It’s an error to try to access elements past the end of the array:

```
var d [100]int
d[0] = 2                // ok
d[99] = 70              // ok
var j int = 100
fmt.Println(d[j])       // ERROR!
```

```
d[len(d)-1] = 3   // OK
d[len(d)] = 3     // ERROR!
d[-60] = 7        // ERROR!
```

Computing Prime Numbers

The “Sieve of Eratosthenes” is a very old algorithm for finding prime numbers:

```
func primeSieve() {
    var isComposite [100000000]bool // isComposite[i] will be true if i is not prime
    var biggestPrime = 2 // will hold the biggest prime found so far
    for biggestPrime < len(isComposite) {
        fmt.Println(biggestPrime)
        // knock out all multiples of biggestPrime
        for i := 2*biggestPrime; i < len(isComposite); i += biggestPrime {
            isComposite[i] = true
        }
        // find the next biggest non-composite number
        biggestPrime++
        for biggestPrime < len(isComposite) && isComposite[biggestPrime] {
           biggestPrime++
        }
    }
}
```

This will print all the prime numbers ≤ 100,000,000.

Why does this work?

At start of outer for loop:

all array elements start at false

isComposite:	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F	F
index into array:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

biggestPrime

First inner for loop sets all multiples of biggestPrime to be TRUE:

	F	F	F	F	**T**	F	T	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

biggestPrime

Second inner for loop increments biggestPrime until it finds a non-composite number:

	F	F	F	F	**T**	F	T	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F	**T**	F
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

biggestPrime

Next time through the outer loop, multiples of 3 will be marked as composite, etc.

Shortcut && and ||

Consider this loop from primeSieve():

```
for biggestPrime < len(isComposite) && isComposite[biggestPrime] {
    biggestPrime++
}
```

What happens when biggestPrime == len(isComposite)?

- The green (first) condition is **false**
- The red (second) condition is an ERROR
- So does this program have a bug? No:

The && and || operators work from left to right and ***stop once their truth value can be determined***.

Once the green condition is **false**, there's no way for the whole expression to be **true**, so in that case, the red condition is never evaluated.

For Range

```
var list [10]int

for i := range list {
   list[i] = -(i - 6)*(i-6)
}
fmt.Println(list)
```

iterates over the indices of array. i will equal 0,1,2,…,9 in turn.

will print:

```
[-36 -25 -16 -9 -4 -1 0 -1 -4 -9]
```

```
var max, pos int
for j, v := range list {
   if j == 0 || v > max {
       max = v
       pos = j
   }
}

fmt.Println("Max value is", max, "at",  pos)
```

if you provide 2 variables, **for…range** will iterate over the indices and values of the array:

```
j   v
0 -36
1 -25
2 -16
3 -9
4 -4
5 -1
6 0
7 -1
8 -4
9 -9
```

Use **for … range** to avoid having to compute indices yourself.

Blank Identifier

```
func sum(A [10]int) {
    var result int
    for i, val := range A {
        result = result + val
    }
    return result
}
```

ERROR! Variable i is declared but never used.

This is an error in Go.

How do we use the for…range loop if we don't want the index?

```
func sum(A [10]int) {
    var result int
    for _, val := range A {
        result = result + val
    }
    return result
}
```

The *blank identifier* _ (a single underscore) can be used when you need to provide a variable name, but don't care about the value.

_ is always "defined" and has whatever type(s) it needs to.

Multidimensional Arrays

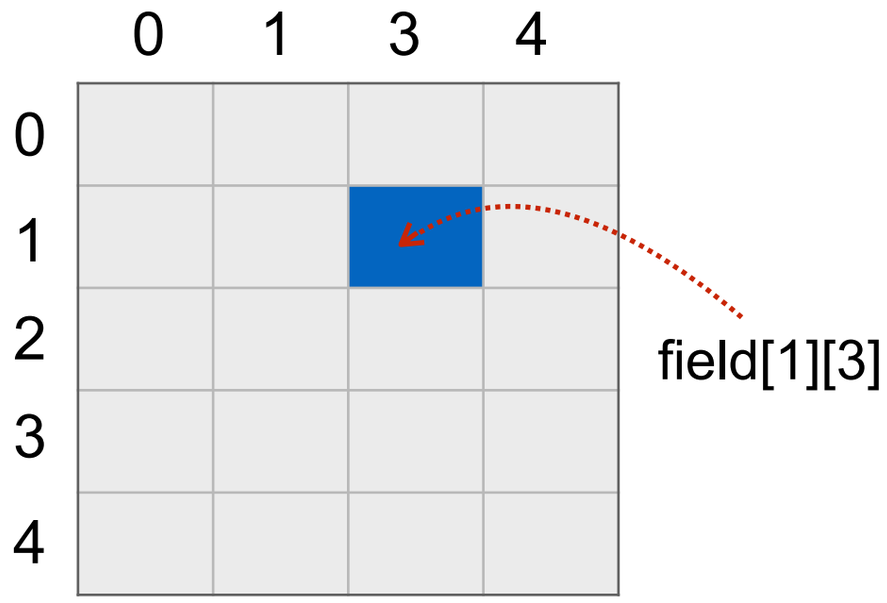
```
func selfAvoidingRandomWalk(steps int) {
    var field [10][10]bool
    var x, y = len(field)/2, len(field)/2

    field[x][y] = true
    fmt.Println(x,y)

    for i := 0; i < steps; i++ {
        // repeat until field is empty
         xnext, ynext := x, y
         for field[xnext][ynext] {
             xnext, ynext = randStep(x, y, 10)
        }
        x, y = xnext, ynext
        field[x][y] = true
        fmt.Println(x,y)
    }
}
```

Declare a 2d array as shown.

Can declare arrays of higher dimension as well.

Repeat until we walk to a square that hasn't been visited

Can reuse our randStep() function from a previous lecture.



Arrays are Copied When Passed to Functions

```
func maxValue(A [10000000]int) int {
    m := 0
    for i := range A {
        if A[i] > m {
            m = A[i]
        }
    }
    return m
}
```

A new array A is created and the contents from numbers is copied over.

This is wasteful of memory if the array is large.

```
func main() {
    var numbers [10000000]int

    // fill numbers with random integers
    for i := range numbers {
        numbers[i] = rand.Int()
    }

    fmt.Println(maxValue(numbers))
}
```

maxValue() will only work for arrays of 10 million elements.

But nothing in the maxValue code wouldn't work for arrays of different sizes.

Slices (up next) fix both of these problems.

Summary

- Arrays store collections of variables of the same type.
- Arrays have a fixed size that is determined when you write your program.

Arrays Summary

- Declare an array variable with: **var** *name* [*size*]*type*
  - *size* must be a constant expression
    (you must know its value when you write your program).
  - *type* can be any type, even another array type (e.g. [10][10]**int**)
- The length of an array can be found with: **len**(*name*)
- name[i] is a variable that is the ith element of the array
  - name[0] is the first element of the array.
- Arrays are *copied* when passed to functions: the function only sees a copy of the array.