A slice variable is declared by not specifying a size in `[]`

```go
var s []int
// at this point s has the special value nil
// and can’t be used as an array
s = make([]int, 10, 20)
```

This creates an array of size 20 with a slice of size 10 inside it.

There is an array behind every slice.

You can think of a slice as a triple: `(array, start, end)`

<table>
<thead>
<tr>
<th>array</th>
<th>start</th>
<th>end</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>10</td>
</tr>
</tbody>
</table>

`make([]type, length, capacity)` creates the array of size `capacity`, and sets `starts = 0, end = length`. 

Length of this slice is 10

Underlying array of size 20
What if we want to make a slice bigger by adding something to the end of it?

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

Note: the syntax is the somewhat redundant:

```go
s = append(s, 5)
```
An Updated `primeSieve()`

```go
define primeSieve(isComposite []bool) {
    var biggestPrime = 2 // will hold the biggest prime found so far
    for biggestPrime < len(isComposite) {
        // 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++
        }
    }
}

define main() {
    var composites []bool = make([]bool, 100000000)
    primeSieve(composites)
    var primeCount int = 0
    var primesList []int = make([]int, 0)
    for i, isComp := range composites {
        if !isComp && i >= 2 {
            primeCount++
            fmt.Println("Number of primes ≤", i, "is", primeCount)
            primesList = append(primesList, i)
        }
    }
}
```
Another Append Example

// take a box and list of 2D points and return the 2D points that lie in the box
func pointsInBox(x1,y1,x2,y2 float64, xs, ys []float64) ([]float64, []float64) {
    var xout = make([]float64, 0)
    var yout = make([]float64, 0)

    for i := range xs {
        if x1 <= xs[i] && xs[i] <= x2 && y1 <= ys[i] && ys[i] <= y2 {
            xout = append(xout, xs[i])
            yout = append(yout, ys[i])
        }
    }
    return xout, yout
}

func main() {
    var x = []float64{-1, 3.2, 7.8, -2.45}
    var y = []float64{-2, -4.0, 3.14, 2.7}

    xlist, ylist := pointsInBox(-5,-5,5,5, x, y)

    for i := range xlist {
        fmt.Println(xlist[i], ylist[i])
    }
}
Array and Slice Literals

Recall: a literal is an explicit value in your program:
- 3 is a integer literal
- “Pittsburgh” is a string literal

Can also write slice literals:
- `[]interface{float64}{3.2, -30, 84, 62}
- `[]interface{int}{1,2,3,6,7,8}

And array literals:
- `[4]interface{float64}{3.2, -30, 84, 62}
- `[6]interface{int}{1,2,3,6,7,8}

Useful if you have a fixed, short list of data.
Multi-dimensional Slices:
Self-Avoiding Walk Example
Example: Self-Avoiding Random Walks

Simulate a random walk on an n-by-n chessboard but don’t allow the walk to visit the same square twice

(n/2, n/2)

Need to keep track of where the walk has been → 2D slice
Creating a 2-D Slice

2-D slices are “slices of slices”. This creates a slice of \( n \) slices, each of which is not yet initialized:

```go
var field [][]bool = make([][]bool, n)
```

To initialize all the slices in field, you must write an explicit loop:

```go
for row := range field {
    field[row] = make([]bool, n)
}
```

Can use field like a 2D array now:

```go
var x, y = len(field)/2, len(field)/2
field[x][y] = true
```
func selfAvoidingRandomWalk(n, steps int) {
    var field [][]bool = make([]][]bool, n)
    for row := range field {
        field[row] = make([]bool, n)
    }
    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, len(field))
        }
        x, y = xnext, ynext
        field[x][y] = true
        fmt.Println(x,y)
    }
}
Bug: What if the walk gets stuck?

```go
func selfAvoidingRandomWalk(n, steps int) {
    var field [][]bool = make([][]bool, n)
    for row := range field {
        field[row] = make([]bool, n)
    }
    var x, y = len(field)/2, len(field)/2

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

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

Add test to stop if stuck

Can initialize a slice using `[]type{value1, value2, ...}`

```go
func stuck(x, y int, field [][]bool) bool {
    var deltas = []int{-1, 0, 1}
    for _, dx := range deltas {
        for _, dy := range deltas {
            nx, ny := x+dx, y+dy
            if inField(nx, n) && inField(ny, n) && !field[nx][ny] {
                return false
            }
        }
    }
    return true
}
```
Subslices: A picture

```go
var s []int
s = make([]int, 10, 20)

len(s[8:15]) == 7
var q []int = s[8:15]
q[0] == -9
q[6] == -15
q[15] == ERROR
s[8] == q[0]
```

Both slices still exist. Both refer to the same underlying array.
Subslices Example

```go
// create a new slice of 0 length
var primes = []int
primes = make([]int, 0)

// add the first prime to our list
primes = append(primes, 2)

// add the next 999 primes to our list
for i := 1; i < 1000; i++ {
    next := getNextPrimeAfter(primes[len(primes)-1])
    primes = append(primes, next)
}

// print out the 27 through 50th prime
fmt.Println(primes[26:51])
```

Assume we have a function `getNextPrimeAfter(n int) int` that gives us the next prime after `n`

`len(primes)-1` is the index of the last element in our primes slice.

Subslice: `A[x:y]` means the part of the slice from index `x` up to (but not including) `y`
Strings
Indexing Strings

Strings work like arrays of `uint8`s in some ways:
You can access elements of string `s` with `s[i]`.
You can iterate through their "letters" using `for...range`
You cannot modify a string once it has been created.

```go
s := "Hi There!"
fmt.Println(s[0])                // prints H
fmt.Println(s[len(s)-1])         // prints !
fmt.Println(s[3:5])              // prints Th
fmt.Println(s[1:])               // prints i There!
fmt.Println(s[:4])               // prints Hi T
s[3] = "t"                       // ERROR! Can't assign to strings

var str string = s[3:6]          // prints The
fmt.Println(str)                 // prints T
```

### Table

<table>
<thead>
<tr>
<th></th>
<th>H</th>
<th>i</th>
<th>T</th>
<th>h</th>
<th>e</th>
<th>r</th>
<th>e</th>
<th>!</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>

`s[0]`  

`s[3:6]`  

`s[x:y]` creates a new string using characters `[x,y)` from `s`.
That is the string ends at character `y-1`.

```go
len(s[x:y]) == y - x
```
Example: Reverse Complementing DNA

The reverse complement of a string of DNA is the string reversed with C ↔ G and A ↔ T.

DNA string $r$: ACGGGATGA
complement of $r$: TGCCCTACT
reverse complement of $r$: TCATCCCGT

A letter is a single character inside single quotes.

// Complement computes the reverse complement of a single given nucleotide. Ns become Ts as if they were As. Any other character induces a panic.
func Complement(c byte) byte {
    if c == 'A' { return 'T' }
    if c == 'C' { return 'G' }
    if c == 'G' { return 'C' }
    if c == 'T' { return 'A' }
    panic(fmt.Errorf("Bad character: %s!", string(c)))
}

Create a byte array $s$
Reverse and complement string $r$, storing the letters into $s$
Convert byte array $s$ into string.
Slices Summary

- Slices work nearly the same as arrays except:
  - You have to explicitly initialize them with `make(type, length)`
  - Now `length` doesn’t need to be known when you write the program.
  - When you use a slice as a function parameter, it is not copied, and the function sees (and can modify) the original slice.

- You have to explicitly write code to create 2-D (or 3-D, etc.) slices.

- You should almost always use slices when you need to create a list of variables.