Array Literals and Parameters

This lesson explains array literals and handling arrays as parameters to functions in detail.

WE'LL COVER THE FOLLOWING

- Array literals
  - 1st variant
  - 2nd variant
  - 3rd variant
  - 4th variant
- Passing an array to a function

Array literals

When the values (or some of them) of the items are known beforehand, a simpler initialization exists using the **{ , ,}** notation called **array literals** (or constructors) instead of initializing every item in the **[ ]**= way. Let's see some variants.

1st variant

Specify explicitly the size `n` of the array with `[n]`. For example:

```
var arrAge = [5]int{18, 20, 15, 22, 16}
```

Here is another example:

```
var arr = [10]int{ 1, 2, 3 }
```

This is an array of **10** elements with the 1st three *different* from 0.

2nd variant

The size of the array is determined by the number of elements between curly

The size of the array is determined by the number of elements between curly brackets. For example:

```
var arrAge2 = []int{18, 20, 15, 22, 16}
```

This array has a length of 5. So, you see the size can be omitted. In that case, the number of items between curly brackets becomes the size of the array.

3rd variant

The [...] notation, for example:

```
var arrLazy = [...]int{5, 6, 7, 8, 22}
```

`...` indicates the compiler has to count the number of items to obtain the length of the array. However, `[...]int` is not a type, so this is illegal:

```
var arrLazy [...]int = [...]int{5, 6, 7, 8, 22}
```

If the `...` is omitted then a slice is created.

4th variant

The `index: value` syntax can be followed. For example:

```
var arrKeyValue = [5]string{3: "Chris", 4: "Ron"}
```

Passing an array to a function

Passing big arrays to a function quickly eats up a lot of memory because arrays are copied when passing. There are 2 ways to prevent this:

- Pass a pointer to the array.
- Use a slice of the array (we'll discuss slices in the next section).

The following example illustrates the first solution:

```
package main
import "fmt"

func main() {
  array := [3]float64{7.0, 8.5, 9.1}
  x := Sum(&array) // Note the explicit address-of operator
  // to pass a pointer to the array
  fmt.Printf("The sum of the array is: %f", x)
}


func Sum(a *[3]float64) (sum float64) {
  for _, v := range a { // dereferencing *a to get back to the array is not necessary!
    sum += v
  }
  return
}
```

Pointer to Array in Function Call

In the above code, there is a function `Sum` that takes an array and calculates the sum of its elements and returns their sum. Look at its header at **line 11**: `func Sum(a *[3]float64) (sum float64)`. The function is taking the pointer to the array `a` of length **3**, whose elements are of type **float64**, and returning a **float64** number `sum`. Now, look at the `main`, where we declared an array called `array` as: `array := [3]float64{7.0, 8.5, 9.1}` at **line 5**. In the next line, we call the `Sum` function and pass `&array` because `Sum` accepts a pointer to the array and stores the result in `x`. At **line 8**, we are printing `x` to verify the result.

Now, you are familiar with the variations and use of arrays in functions. In the next lesson, you have to write a program to solve a problem.