Declaration and Initialization

This lesson describes the important concepts of array, i.e., how to use, declare and initialize them.

WE'LL COVER THE FOLLOWING

Introduction

In this chapter, we start by examining data-structures that contain a number of items, called *collections*, such as **arrays** (slices) and **maps**. Here, the Python influence is obvious. The array-type indicated by the **[]** notation is well-known in almost every programming language as the basic workhorse in applications.

The Go array is very much the same but has a few peculiarities. It is not as dynamic as in C, but to compensate, Go has the slice type. This is an abstraction built on top of Go's array type. So to understand slices, we must first understand arrays. Arrays have their place, but they are a bit inflexible. Therefore, you don't see them too often in Go code. Slices, though, are everywhere, and they are built on arrays to provide greater power and convenience.

Concept

An **array** is a *numbered* and *fixed-length* sequence of data items (elements) of the same type (it is a homogeneous data structure). This type can be anything from primitive types like integers or strings to self-defined types. The length must be a constant expression, that must evaluate to a non-negative integer value. It is part of the type of the array, so arrays declared as `[5]int` and `[10]int` differ in type. The items can be *accessed* and *changed* through their *index* (their position). The index starts from **0**, so the 1st element has index 0,

the 2nd index 1, and so on (arrays are zero-based as usual in the C-family of languages).

The number of items, also called the *length* (called len) or *size* of the array, is fixed and must be given when declaring the array (length has to be determined at compile time in order to allocate the memory).

> **Remark:** The maximum array length is 2Gb.

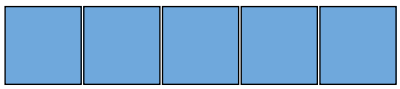
The format of the declaration is:

```
var identifier [len]type
```

For example:

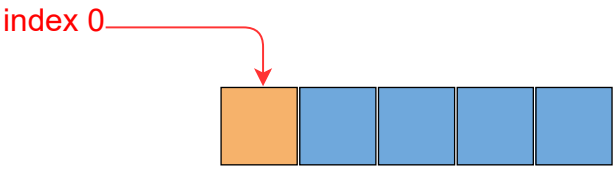
```
var arr1 [5]int
```

Let's visualize `arr1` in memory

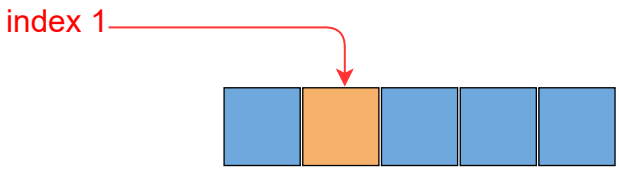


Array in Memory

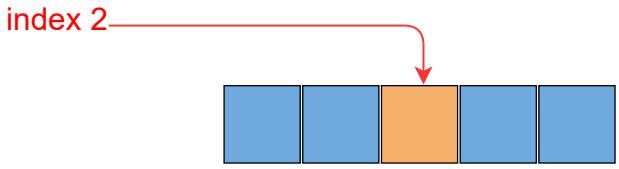
1 of 6



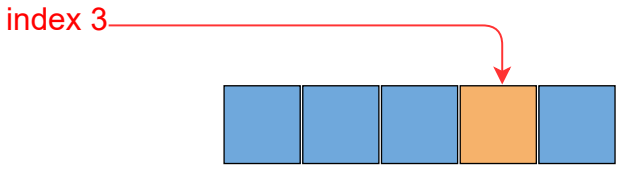
Array in Memory



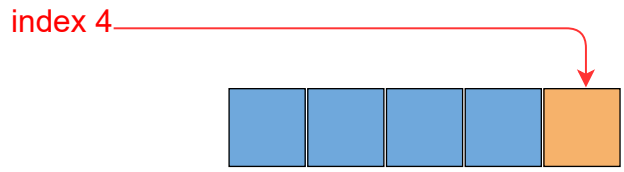
Array in Memory



Array in Memory



Array in Memory



Array in Memory

6 of 6

Each compartment contains an integer. When declaring an array, each item in it is automatically initialized with the default zero-value of the type. Here, all the items default to **0**. The length of `arr1` is **5**, and the index ranges from **0** to **len(arr1)-1** (4 in this case). The first element is given by `arr1[0]`, and the 3rd element is given by `arr1[2]`; in general, the element at index `i` is given by `arr1[i]`. The last element is given by:

```
arr1[len(arr1)-1]
```

Assigning a value to an array-item at index `i`, is done with:

```
arr[i] = value
```

Arrays are mutable, which means that they can be changed.

Only valid indexes can be used. When using an index *equal* to or *greater* than **len(arr1)** and if the compiler can detect this, the message `"index out of bounds"` is given; otherwise, the code compiles just fine. Executing the program will give the runtime error: `"index out of range"`. A runtime error means the program will crash and give a certain message. In Go terminology, this is called a *panic*. We will discuss these in much more detail in Chapter 11.

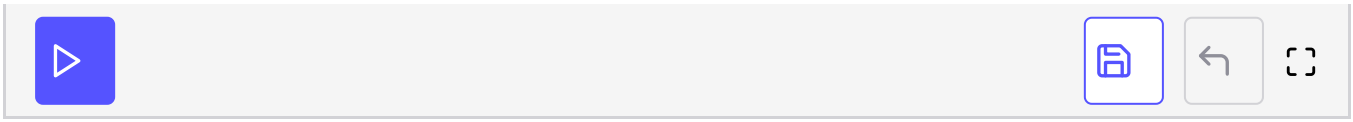
Because of the index, a natural way to loop over an array is to use the *for-construct*:

- for initializing the items of the array
- for printing the values or in general
- for processing each item in succession

The following program demonstrates the basic use of *for* loop with an array.

```
package main
import "fmt"

func main() {
  var arr1 [5]int   // declaring an array
  for i:=0; i < len(arr1); i++ {
    arr1[i] = i * 2 // initializing items of array
  }
  for i:=0; i < len(arr1); i++ {
    fmt.Printf("Item at index %d is %d\n", i, arr1[i])  // printing items of array
  }
}
```



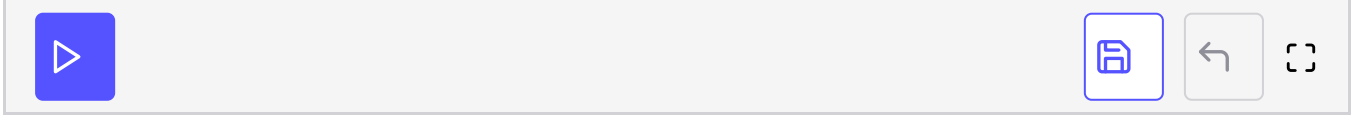
Arrays in Go

In the above code, at **line 5** in `main`, we declare an array `arr1` of *length* **5**, which will contain integral values. Then, we have a *for* loop at **line 6** that will iterate over the `arr1` `len(arr1)` times and will initialize each item in an array as `arr1[i]=i*2` at **line** 7. Again, we have a *for* loop at **line 9** that will iterate over `arr1` `len(arr1)` times and will print the items in array at **line 10**.

We can also use the `range` function and a single *for* loop to write the same program. Run the following program.

```
package main
import "fmt"

func main() {
  var arr1 [5]int   // declaring an array
  for i:= range arr1 {
    arr1[i] = i * 2 // initializing items of array
    fmt.Printf("Item at index %d is %d\n", i, arr1[i])  // printing items of array
  }
}
```



In the above code, at **line 5** in `main`, we declared an array `arr1` of *length* **5**, which will contain integer values. Then, we have a *for* loop at **line 6** that will iterate over the array `len(arr1)` times using the `range` function. It will initialize the items in an array as `arr1[i]=i*2` at **line** 7 and will print the items in array at **line 8**.

An array in Go is a *value* type (not a pointer to the first element like in C/C++). Therefore, it can be created with new():

```
var arr2 = new([5]int)
```

But what is the difference with:

```
var arr1 [5]int
```

Recall that `new(T)` allocates zeroed storage for a new item of type T and returns to its address. Therefore, it returns a pointer to the type T. Thus, `arr2` is of type ***[5]int**, whereas `arr1` is of **type [5]int**. The consequence is that, when assigning an array to another array, a distinct copy in the memory of the array is made. For example, when we say:

```
arr3 := arr1
arr3[2] = 100
```

Then, the arrays have different values. Changing `arr3` after the assignment does not change `arr1`. If you want the change to be possible, use:

```
arr3 := &arr1
```

Then, `*arr3` will always have the same value as `arr1`.

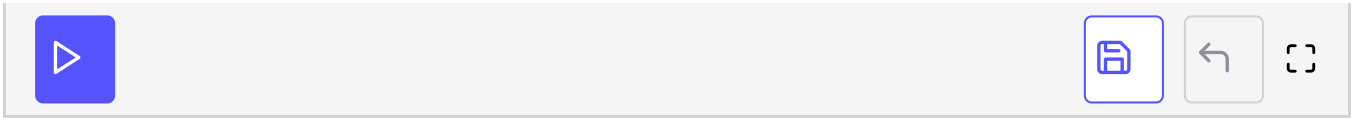
When an array is passed as an argument to a function like in `func1(arr1)`, a copy of the array is made, and `func1` cannot change the original array `arr1`. If this is possible, then `arr1` has to be passed by reference with the **&**-operator, as in `func1(&arr1)`. Look at the following implementation to understand this concept clearly.

```
package main
import "fmt"


func f(a [3]int) { fmt.Println(a) }   // accepts copy

func fp(a *[3]int) { fmt.Println(a) } // accepts pointer

func main() {
  var ar [3]int
  f(ar) // passes a copy of ar
  fp(&ar) // passes a pointer to ar
}
```



Pointer Array

The above code has two basic functions: `f` and `fp`. The function `f` is accepting a *copy* of the array passed to it as a parameter. See its header at **line 4**: `func f(a [3]int)`. The second function, `fp` accepts a pointer to the array that is passed to it as a parameter. See its header at **line 6**: `func fp(a *[3]int)`. Now, look at the `main`. We declare an array `ar` at **line 9** as `var ar [3]int`. In the next line (**line 10**), we pass `ar` to `f`, which prints `[0,0,0]` as it was the copy of `ar`. At **line 11**, we pass `&ar` as a pointer variable to `fp`, which prints `&[0,0,0]` since a pointer variable contains address.

That's it for the brief introduction to arrays in Golang. In the next lesson, you will learn about further variations and how to handle arrays within functions.