# Chapter 7 – Arrays

In this chapter we start with examining data structures that contain a number of items, so called *collections*, such as arrays 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 V array is very much the same, but has a few peculiarities.

## 7.1 Making and using arrays

### 7.1.1 Declarations

An array is a numbered and fixed-length sequence of data items (elements) of the same *single type:* an array is a *homogeneous data structure*. The data type is determined by the type of the first element that it contains.

This type can be anything from primitive types like integers, strings, and so on, to self-defined types.

The items can be accessed (and changed) through their *index* (the 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 always in the C-family of languages). The items are stored in contiguous positions in memory.

The number of items, also called the *length* len or size of the array, is fixed and must be given when declaring the array. The length can be specified as a constant expression, that must evaluate to a non-negative integer value. It has to be determined at compile time in order to allocate the memory.

A declaration of a literal array is as follows: nums := [1, 2, 3, 4, 5]

which can be visualized in memory as:

5

4

3

2

1

nums

index 0 1 2 3 4

Figure 7.1: Array in memory

Listing 7.1 – arrays1.v:

fn main() {

nums := [1, 2, 3, 4, 5]

println(nums[0]) // => 1

println(nums[1]) // => 2

println(nums[4]) // => 5

println(nums) // => [1, 2, 3, 4, 5]

println(nums**.len**) // => 5

println(nums[5]) // V panic: index out of range (i == 5, a.len == 5)

   // array with different types:

   nums2 := [1, 2, 'abc'] // => arrays1.v:23:24: bad array element type `string` instead of `int`

}

The length of array nums len(nums) is 5, and the index ranges from 0 to len(nums)-1 .

The first element is given by nums[0], the 3rd element is given by nums[2] ; in general the element at index i is given by nums[i]. The last element is given by: nums[len(nums)-1]

println(nums) prints out the array because array has a built-in str() method, which println uses.

Convert an array to a string with  str := arr.str()

The array nums is of type []int.

Possible errors:

1) Only valid indexes can be used. When using an index equal to or greater than len(nums) the compiler gives the error, in the case of println(nums[5]):

V panic: index out of range (i == 5, a.len == 5)

2) All items of an array have to be of the same type. If not, the compiler signals this error with e.g.:

23:24: bad array element type `string` instead of `int`

An empty array can be declared as: a := []Type , like []int or []string:

Unlike with ordinary variables, we need to supply the array item’s type.

arr := []int

println(arr.len) // => 0

println(arr) // => []

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 items default to 0.

When declared as a type in function arguments or structs, it is also written as []type, where type is the type of the first item of the array or the type of the field.

A trailing comma is allowed at the end of the array, so adding new items is easy, see trailing\_comma.v

fn main() {

    n := [

      1,

      2,

      3,

    ]

x := [

      'John'

    , 'Bob'

    , 'Julie'

    ,]

    println(x)  // => ["John", "Bob", "Julie"]

    println(n)  // => [1, 2, 3]

}

V has also fixed size arrays, for example: buf := [100]byte

An array can also be a constant:

const (

numbers = [1, 2, 3]

)

?? An alias for an array of ints is: array\_int

#### Mutable arrays

V arrays are *dynamic*; items can be changed or added to them. In that case the array must be declared as mut.

Then you can assign a value to an array-item at index i with: nums[i] = value*.*

Listing 7.2 – mut\_arrays.v:

fn main() {

**mut** nums := [1, 2, 3, 4, 5]

println(nums)// [1, 2, 3, 4, 5]

**nums[2] = 7**

println(nums)// [1, 2, 7, 4, 5]

nums[2] = 'V' // cannot use type `string` as type `int` in assignment

}

?? To insert an item at a certain position pos, use arr.insert(pos, item)

?? To delete an item at position pos, use arr.delete(pos):

mut even\_numbers := [2, 4, 6, 8, 10]

even\_numbers.delete(3)

println(even\_numbers) // => [2, 4, 6, 10]

#### Pre-allocated arrays

You can also preallocate an array with a certain item (here 0) value and a certain length (here 5):

Listing 7.3 – prealloc\_arrays.v:

a := [0].repeat(5)

println(a)  // => [0, 0, 0, 0, 0]

To pre-allocate an array of a certain length len and with a value val for all items, write: pre\_arr := [val].repeat(len)

nr\_ids := 13

ids := [1].repeat(nr\_ids)

println(ids) // => [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]

As the notation indicates, you can also repeat an array:

arr := [1, 2].repeat(5)

println(arr) // => [1, 2, 1, 2, 1, 2, 1, 2, 1, 2]

If these arrays will be changed, you have to declare them as mut.

Example from tetris.v: res := [Block{}].repeat(4) // array of len 4 of struct value

#### String arrays

Here is an example with an array of strings, of type []string:

Listing 7.3 – string\_array.v:

fn main() {

names := ['Melissa', 'John']

println(names) // ["Melissa", "John"]

}

Notice that in the println output, strings are delimited by double quotes.

#### The in operator

To check whether a certain item is present in the array, use the in operator:

        println('John' in names ) //  => true

        println('Alex' in names)  //  => false

When the item is present it returns true, otherwise false.

Other examples:

if 'version' **in** os.args

int(moore\_sum **in** [2, 3]) // in game\_of\_life.v

sum := 3

println(int(sum in [2, 3])) // 1

So in expressions can be used in conditions.

It can also be useful to write clearer and more compact boolean expressions.

Compare:

if parser.token == .plus || parser.token == .minus ||

parser.token == .div || parser.token == .mult {

...

}

With:

if parser.token in [.plus, .minus, .div, .mult] {

...

}

#### Appending items:

To add items to the end of the array, use the << operator:

Listing 7.3C – append\_array.v:

fn main() {

mut cities := ['Berlin', 'Tokio']

println(cities) // ["Berlin", "Tokio"]

cities << 'Paris'

cities << 'London'

println(cities) // => ["Berlin", "Tokio", "Paris", "London"]

println(cities.len) // => 4

cities << 42 // expected type `string`, but got `int`

// 2nd example:

mut s := []string

s << 'V is '

s << 'awesome'

println(s) // ["V is ", "awesome"]

}

Of course, when using << the array has to be declared with mut, and you can only append items of the array item type.

You can also append an array:

mut nums := [0]

nums << [1, 2, 3]

println(nums) // => [0, 1, 2, 3]

To add an array of n elements with value 0 do:

n:= 4

nums << [0].repeat(n)

println(nums) // => [0, 1, 2, 3, 0, 0, 0, 0]

Append a struct to an array of structs: (see module flag)

fs.flags << Flag{

    name: n,

    abbr: a,

    usage: u,

    val\_desc: vd

  }

Chaining << doesn’t work:

mut a := [1, 2, 3, 4]

a << 5 << 6 << 7 << 8

println(a) // => [1, 2, 3, 4, 10485760]

10485760 is the result of shift left operations (5 << 6 << 7 << 8) :)  
So it's in fact a << (5 << 6 << 7 << 8) with << as the left-shift operator.

#### Looping over an array

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

* for initializing or modifying the items of the array,
* for printing the values,

or in general:

* for processing each item in succession.

A basic example is given in Listing 7.4 – for\_arrays.go :

fn main() {

mut arr1 := [1, 2, 3, 4, 5]

**for i:=0;** **i < arr1.len; i++** {

arr1[i] = i \* 2

println('Item at index $i is ${arr1[i]}')

}

}

Output: Item at index 0 is 0

Item at index 1 is 2

Item at index 2 is 4

Item at index 3 is 6

Item at index 4 is 8

Of crucial importance here is the condition in the for-loop: i < arr1.len

i **<=** arr1.len would give an index out of range-error.

***IDIOM***: for i:=0; i < arr1.len; i++ {

arr1[i]= …

}

The for-in construct is more elegant for using with arrays:

Listing 7.5 – forin\_arrays.go:

fn main() {

  numbers := [1, 2, 3, 4, 5]

  for num in numbers {

    print(num)            // => 12345

  }

  println('')

  names := ['Sam', 'Peter']

**for i, name in names** { // index, value form

    println('$i) $name')  // =>      0) Sam

  }                       //         1) Peter

  for \_, name in names {  // \_ discards the index, but can be left out entirely

    print('$name - ')     // => Sam - Peter -

  }

  println('')

**for name in names** {

    print('$name - ')     // => Sam - Peter -

  }

  println('')

**for ix, \_ in names** { // \_ discards the name

    print('$ix ')      // => 0 1

  }

}

Note that the iteration variable is read-only.

Or as the iconic V Hello-World version:

Listing 7.5B – hello\_developers.v:

fn main() {

  areas := ['game', 'web', 'tools', 'science', 'systems',

            'embedded', 'drivers', 'GUI', 'mobile']

  for area in areas {

    println('Hello, $area developers!')

  }

}

/\* Output:

Hello, game developers!

Hello, web developers!

Hello, tools developers!

Hello, science developers!

Hello, systems developers!

Hello, embedded developers!

Hello, drivers developers!

Hello, GUI developers!

Hello, mobile developers!

\*/

***IDIOM:*** for item in arr1 {

…

}

for ix, item in arr1 {

…

}

Here ix is the index in the array. Both ix or item can be discarded by replacing them with \_

Useful applications:

Enumerating the command-line arguments: for arg in os.args (see § 13.2)

#### Slices

(see *slices.v*)

As in Go, V arrays can be sliced, that is: a part of an array can be taken out through [start..end], the item at index end is not included.

To slice the whole array, do: arr[0 .. arr.len]

fn main() {

  arr := [1,2,3,4]

  println(arr[1..2]) // => [2]

  println(arr[0..4]) // => [1, 2, 3, 4]

println(arr[..2]) // => [1, 2]

  str := 'hello'

  println(str[0..2]) // => he

}

This also works for strings, which are arrays of bytes.

#### Copying arrays

Arrays are value types: when assigned, a copy is made. Arrays can grow, so they are allocated on the heap.

Copying arrays works like in C and Go: see array\_memory.v:

fn main() {

  a := [1, 2, 3]

  mut b := a

  loc1 := &a

  loc2 := &b

  println(loc1) // => 000000000061FD60

  println(loc2) // => 000000000061FD40

  b[0] = 0

  println(a)  // => [0, 2, 3]

  println(b)  // => [0, 2, 3]

}

We see that a and b are stored at different locations loc1 and loc2 in memory, but they happen to point to the same array.

So changing b[0] affects both a and b. This is just like arrays in C and Go.

Here is another example: we see that when an array arr1 is assigned to arr3, they can be changed by either variable through indexing.

But as arr4 and arr5 shows, they have different addresses, so they are different copies in memory:

When the whole array arr1 is changed, arr3 is not changed.

Listing 7.6 – array\_value.go:

fn main() {

        mut arr1 := [1, 2, 3]

        println(arr1)      // => [1, 2, 3]

        mut arr3 := arr1

        println(arr3)      // => [1, 2, 3]

        arr1[2] = 100

        println(arr1)      // => [1, 2, 100]

        println(arr3)      // => [1, 2, 100]

        arr3[1] = 77

        println(arr1)      // => [1, 77, 100]

        println(arr3)      // => [1, 77, 100]

        arr4 := &arr1

        println(arr4)      // => 000000000061FCE0

        arr5 := &arr3

        println(arr5)      // => 000000000061FCC0

        println(\*arr4)     // => [1, 77, 100]

arr1 = [1, 2, 108]

        println(\*arr4)     // => [1, 2, 108]

println(arr3) // => [1, 77, 108]

}

There's an array.clone() method if you want a real copy:

  c := a.**clone()**

  println(&c) // => // 000000000061FCD0

  a[2] = 9

  println(a)  // => [0, 2, 9]

  println(b)  // => [0, 2, 9]

  println(c)  // => [0, 2, 3]

When a changes, c retains the original value.

**EXERCISES:**

Exercise 7.2: for\_array.v: Write the loop that fills an array with the loop-counter (from 0 to 15) and then prints that array to the screen.

Exercise: for\_loops.v

Test out 5 ways to loop through the following array: langs := ['python', 'java', 'javascript']

Exercise: Generate a random permutation of the items of an array of integers (see shuffle\_a\_list.v)

To pick a random element from an array arr, do: pick := rand\_pick(arr)

fn rand\_pick(arr []int) int {

return arr[rand.next(arr.len)]

}

Exercise: Pick a random item from an array (see pick\_a\_random\_element\_from\_a\_list.v)

Exercise: Implement a stack structure, specifically the pop operation that removes the last element and shows it (see *stack.v*)

### 7.1.2 Multidimensional arrays

?? Better general explanation and example needed.

The general format is of a 2 dimensional int array is: [][]int

A 3 x 5 multidimensional array can be inititalized as: marr := [[0].repeat(3)].repeat(5)

Here is another example: multidim.v

fn main() {

  m := [[0]]

  n := [ [3] ]

  k := n[0][0]

  println(k) // => 3

}

Suppose we want to initialize the array marr to:

marr = [ [0,1,2], [1,2,3], [2,3,4], [3,4,5], [4,5,6] ]

To do that we need the following code (multidim2.v):

fn main () {

    mut marr := [[0].repeat(3)].repeat(5)

    for i := 0; i < 5; i++ {

        marr[i] = [0].repeat(3)

        for j :=0 ; j < 3; j++ {

            marr[i][j] = i + j

//            println( '$i, $j, ${i + j}' )

        }

    }

    for i := 0; i < 5; i++ { println(marr[i]) }

}

/\* Output:

[0, 1, 2]

[1, 2, 3]

[2, 3, 4]

[3, 4, 5]

[4, 5, 6]

\*/

Explain why the following code: (see multidim2\_wrong.v)

fn main () {

    mut marr := [[0].repeat(3)].repeat(5)

    for i := 0; i < 5; i++ {

        for j := 0 ; j < 3; j++ {

            marr[i][j] = i + j

//            println( '$i, $j, ${i + j}' )

        }

    }

    for i := 0; i < 5; i++ { println(marr[i]) }

}

has as output:

[4, 5, 6]

[4, 5, 6]

[4, 5, 6]

[4, 5, 6]

[4, 5, 6]

Explanation: the array marr in the wrong version contains multiple copies of *the same row object*. When you modify a value, you modify it for all rows. This is the same reasoning as was mode in the section on Copying arrays.

The solution is to make a new object for each row, as we did in multidim2.v

Exercise: Implement a function search which looks for item x in a 2D matrix m. Return indices i, j of the matching cell.

(see return\_wto\_values.v)

Other example: game\_of\_life/life.v

(Created by fuyutarow: <https://github.com/fuyutarow/Conways-Game-of-Life-with-Vlang>)

?? An array of array of ints has type: []array\_int

### 7.1.3 Passing an array to a function

An array can be passed by value, as in the following example, Listing 7.6 – array\_sum.v

fn sum(a []f32) f32 {

  mut sum := 0.0

  for v in a {

    sum += v

  }

  return sum

}

arr := [7.0, 8.5, 9.1]

x := sum(arr)

println('The sum of the array is: $x') // The sum of the array is: 24.600000

Passing big arrays to a function quickly uses up much memory, because they are copied when passing.

To change the original array in a function, pass it as mutable, effectively passing a pointer to the array.

Here is an example: mutate\_array.v:

fn multiply\_by\_2(arr **mut** []int) {

  for i := 0; i < arr.len; i++ {

    arr[i] \*= 2

  }

}

**mut** nums := [1, 2, 3]

multiply\_by\_2(**mut** nums)

println(nums) // => [2, 4, 6]

So mutable function arguments are allowed: but you have to indicate this 3 times:

* The original variable has to be mut
* When calling the function: mut arg
* When defining the function: fn func1(a mut type)

This makes it very clear that the function being called will modify the value.

## 7.2 Array functionalities

These are demonstrated in array\_functions.v:

fn main() {

// sort:

  mut nums := [67, -3, 108, 42, 7]

  println(nums)

  nums.**sort**() //  sort nums in place

  println(nums) // => [-3, 7, 42, 67, 108]

// reverse:

  println(nums.**reverse**()) // => [108, 67, 42, 7, -3]

// filter:

  nums2 := [1, 2, 3, 4, 5, 6]

  even := nums2.**filter**(it % 2 == 0)

  println(even) // => // => [2, 4, 6]

  c := ['v', 'is', 'awesome']

  d := c.filter(it.len > 1)

  println(d) // => // => ['is', 'awesome']

// map:

  words := ['hello', 'world']

  upper := words.**map**(**it**.to\_upper())

  println(upper) // => ['HELLO', 'WORLD']

  a := [1, 2, 3, 4]

  b := a.map(it \* 10)

  println(b) // => [10,20,30,40]

  a2 := ['v', 'is', 'awesome']

  b2 := a2.map(it == 'v')

  println(b2) // => [true, false, false]

// join:

  mut arr := [''].repeat(100)

  for i in 0.. arr.len - 1 {

    arr[i] = "$i"

  }

  println(arr.**join**(' line ')) // => 0 line 1 line 2 line 3 line 4 line 5 line 6 line 7 line ...

// clone:

  arr2 := [1, 2, 4, 5, 4, 6]

  arr\_copy := arr2.**clone**()

  println(arr\_copy) // => [1, 2, 4, 5, 4, 6]

}

sort(): array.sort() changes the array on which it works (for an example with strings: see word\_counter.v)

reverse():

filter(condition): using filter() is like writing a for loop, there's no function call, it is optimized

it is a special variable that takes on each array value in succession.

map(function): see filter

A nice example of its use is: hn\_top.v in § 13.7

fn fetch\_top\_stories(num int) []Story {

  text := http.get\_text('${api}/topstories.json')

  stories\_ids := json.decode([]int, text) or { exit(1) }

  stories\_top\_ids := stories\_ids[..num]

  return stories\_top\_ids.**map**(fetch\_story(**it**))

}

join(str): concatenates all items of an array with the specified string str

array.clone() makes a duplicate of an array