PART IV

Composite Types

Gopher Level

Intermediate

PART IV

Composite Types

Arrays
Collection
of
Elements
Indexable
Fixed Length

Slices
Collection
of
Elements
Indexable
Dynamic length

String Internals
Byte Slices
ASCII & Unicode
Encoding & Decoding

Maps
Collection
of
Indexable
Key-Value Pairs

Structs
Groups
different types of
variables together

PART IV

Composite Types

Arrays

Collection
of
Elements
Indexable
Fixed Length

Slices

Collection
of
Elements
Indexable
Dynamic length

String Internals

Byte Slices
ASCII & Unicode
Encoding & Decoding

Maps

Collection
of
Indexable
Key-Value Pairs

Structs

Groups
different types of
variables together

ARRAYS

What you're going to learn in this section?

- ★ What is an array?
- ★ Getting and Setting Array Elements
- ★ Array Literals — Easy way to create arrays
- ★ Comparing Arrays
- ★ Assigning Arrays
- ★ Multi-Dimensional Arrays
- ★ Keyed Elements
- ★ Named vs Unnamed Types

ARRAYS

By the end of the section, you're going to build awesome projects again!

😀 MOODLY 😞

Random moods

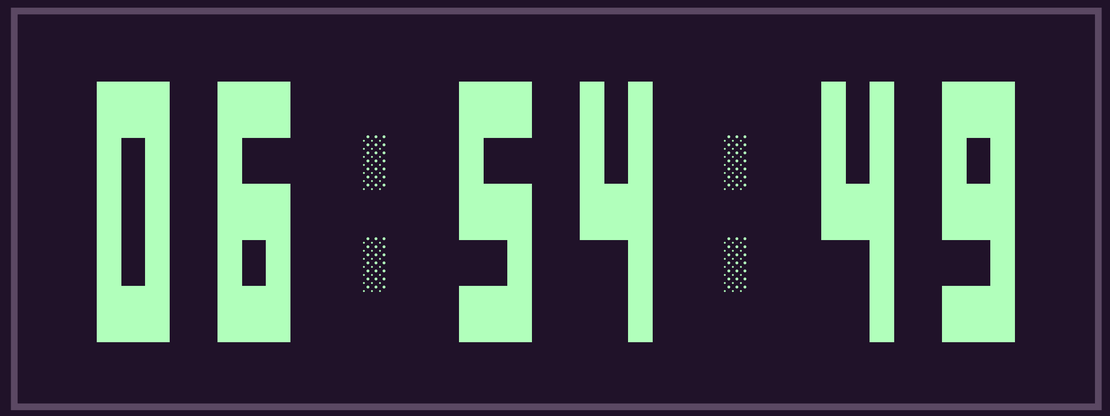
```
$ go run main.go inanc positive
inanc feels awesome 😎

$ go run main.go inanc negative
inanc feels sad 😞
```

ARRAYS

By the end of the section, you're going to build awesome projects again!

DIGITAL CLOCK

Animated & Runs on CLI ⌨



Why Arrays?

Most of the time
we need to work
with
multiple values

It's hard to do this manually!

```
a   := 54
b   := 143
... := ...
oneThousand := 1000

sum := a + b + ... + oneThousand
```

It's easy to do it with an array or a slice!

```
nums := [...]int{54, 143, ... , 1000}

var sum int
for _, num := range nums {
    sum += num
}
```

This is only some of the benefits of arrays

WHAT'S AN ARRAY?

Array is a fixed length container for the same type of values

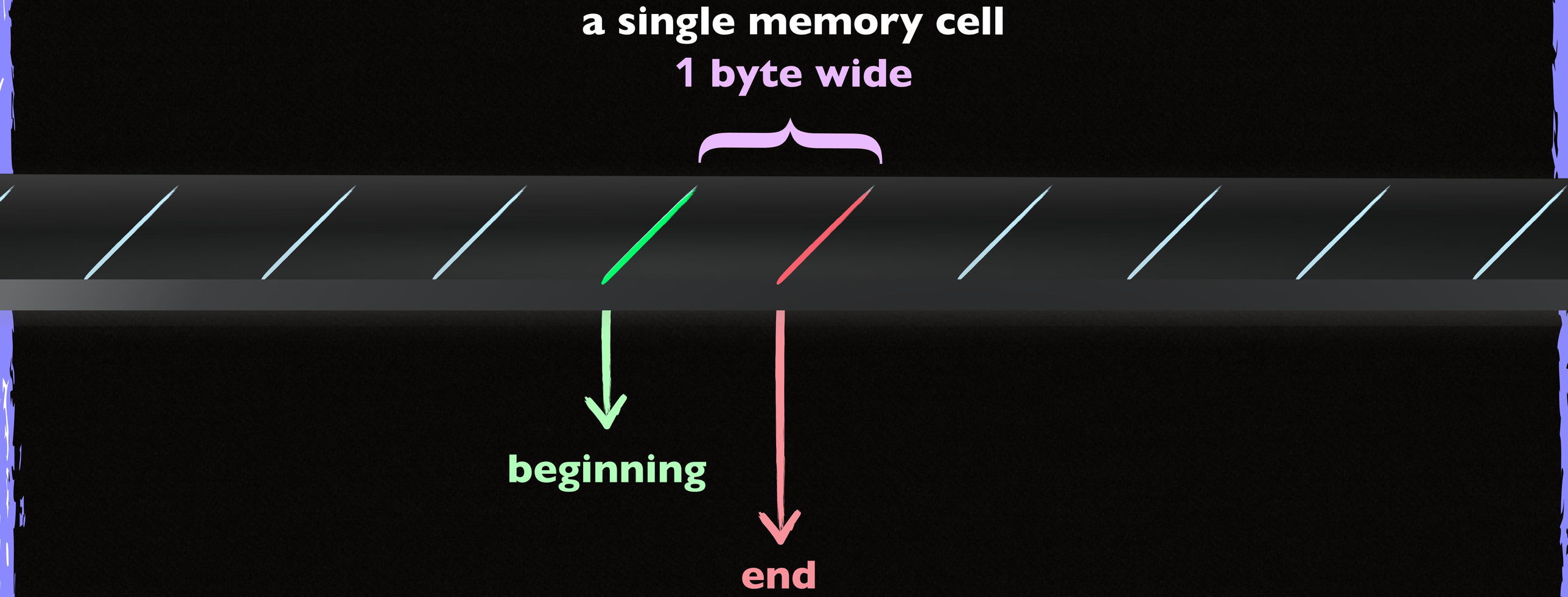
COMPUTER MEMORY

Computer memory stores values in memory cells

Think of this grid like the computer memory

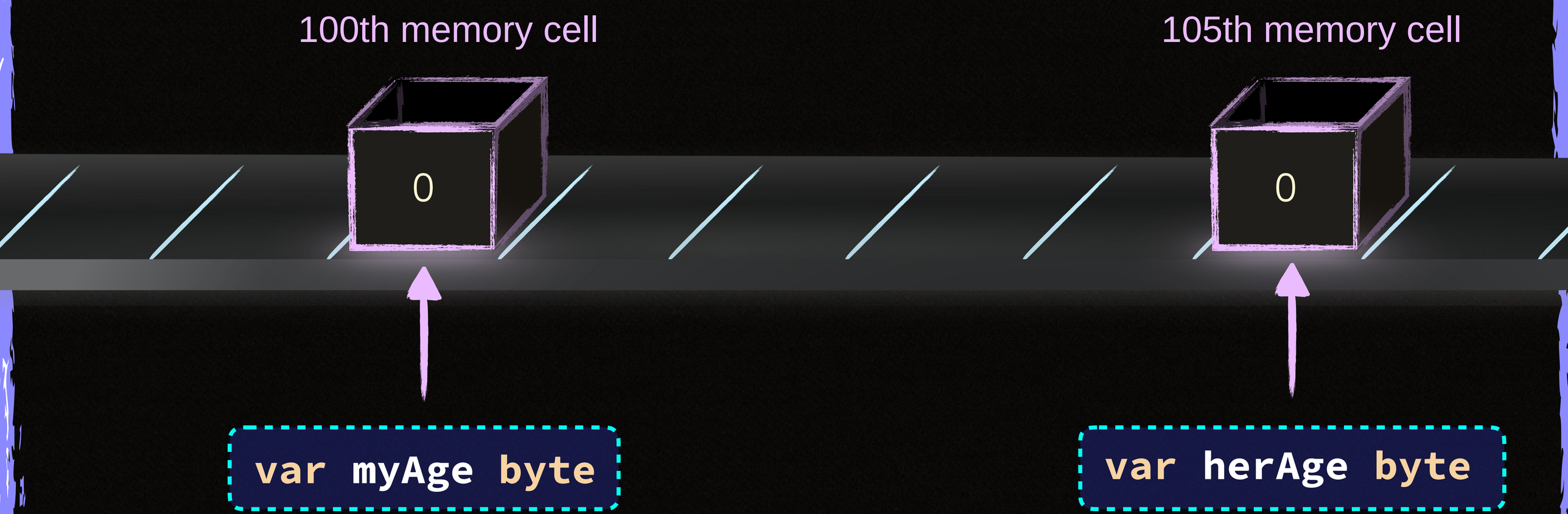
MEMORY CELLS

Each memory cell occupies 1 byte



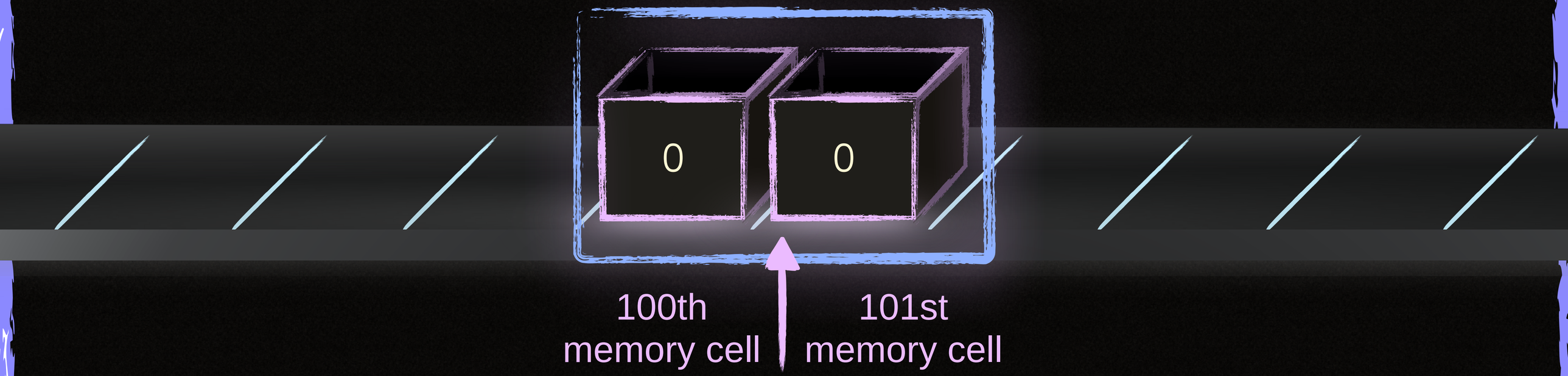
ALLOCATION

Variables can be allocated in different locations in memory



ARRAY

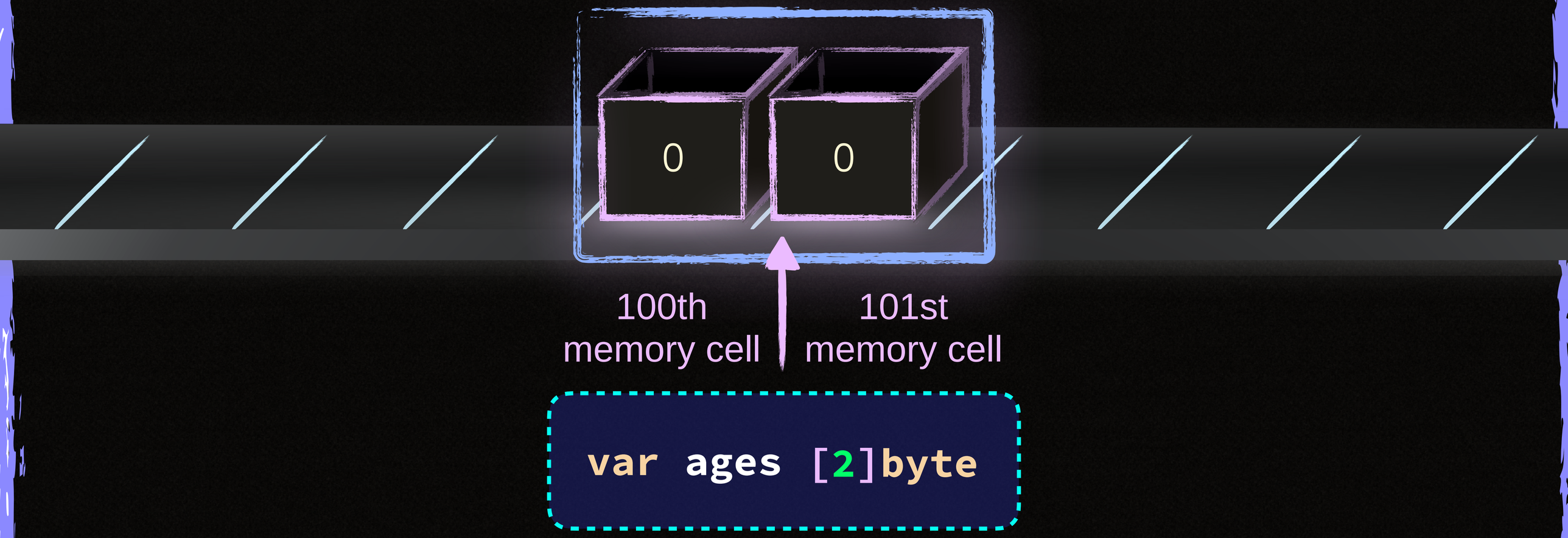
An array stores its elements in contiguous memory cells




```
var ages [2]byte
```

ARRAY

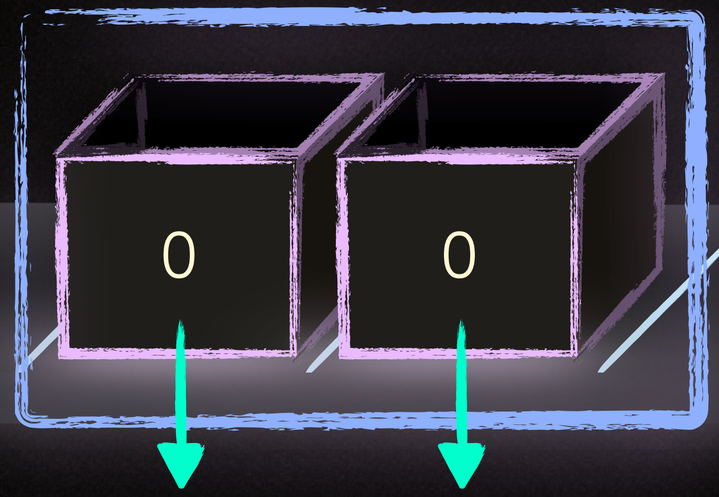
Efficiency: CPU Cache-Lines, Fast Access, 1-o-1 representation of memory...



ARRAY

The size of an array is equal to the total size of its elements

```
var ages [2]byte
```

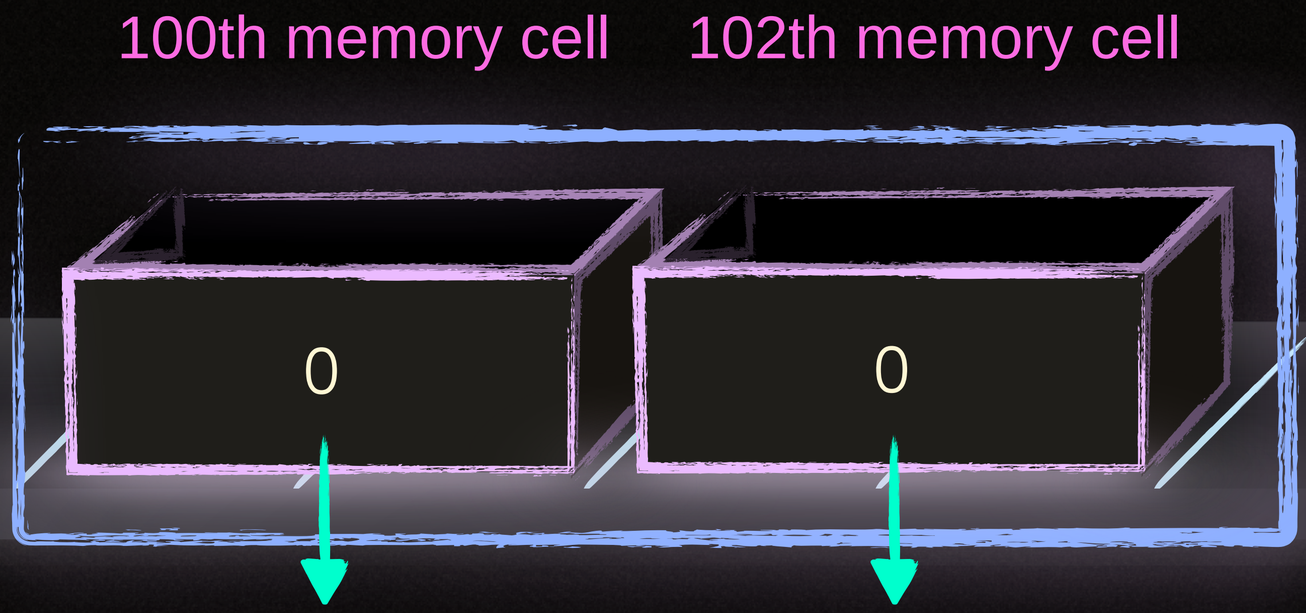


1 byte + 1 byte = 2 bytes

ARRAY

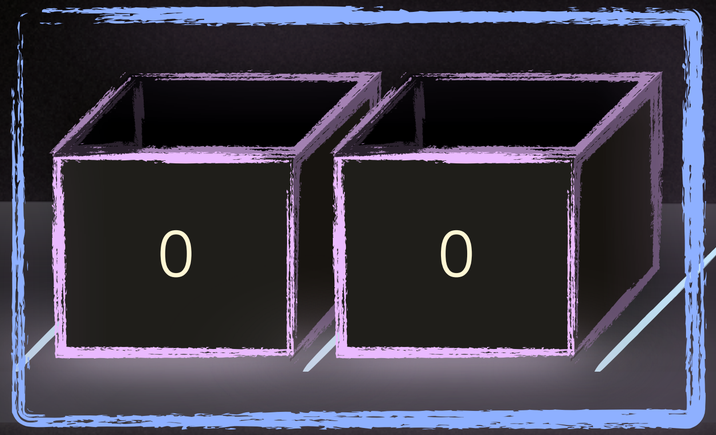
The size of an array is equal to the total size of its elements

```
var ages [2]int16
```



ARRAY'S LENGTH

It determines the number of elements that an array can store



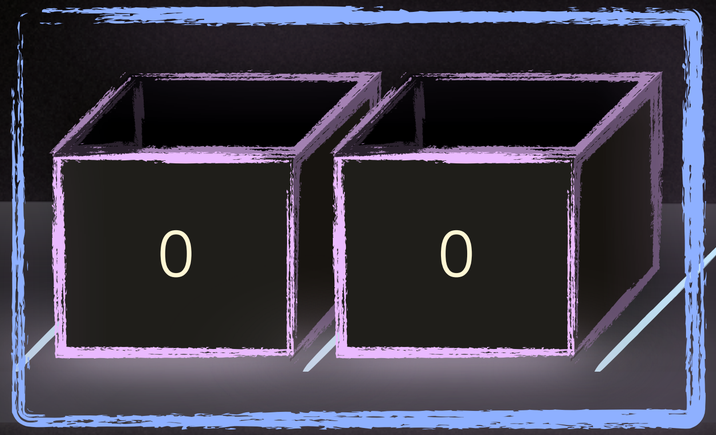

```
var ages [2]byte
```

length of this array

length should always be declared!

ARRAY'S LENGTH

It can only be a constant value — Array's type belongs to compile-time




```
var ages [1 + 1]byte
```

length of this array

You can use constant values and constant expressions!

ARRAY'S LENGTH

Array's length determines its maximum capacity

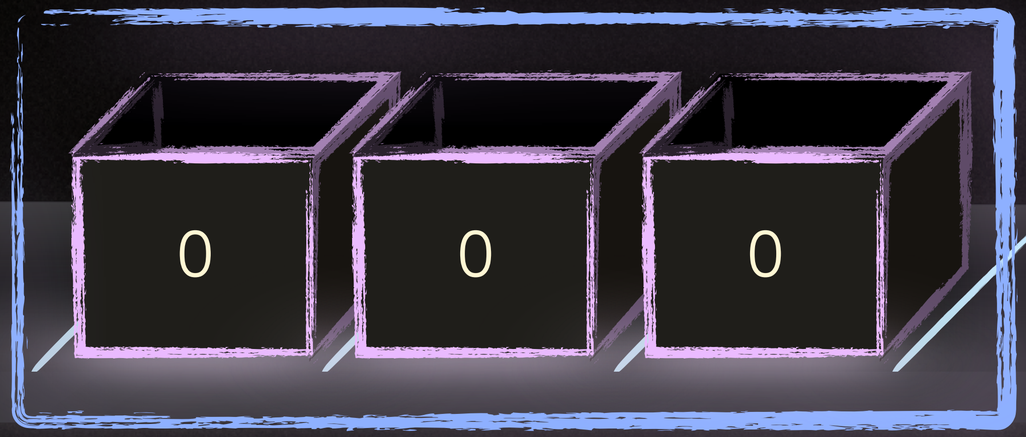

```
var ages [2]byte
```

length of this array

You cannot store more than 2 elements in this array....

ARRAY'S LENGTH

If you need more capacity you can declare a new array



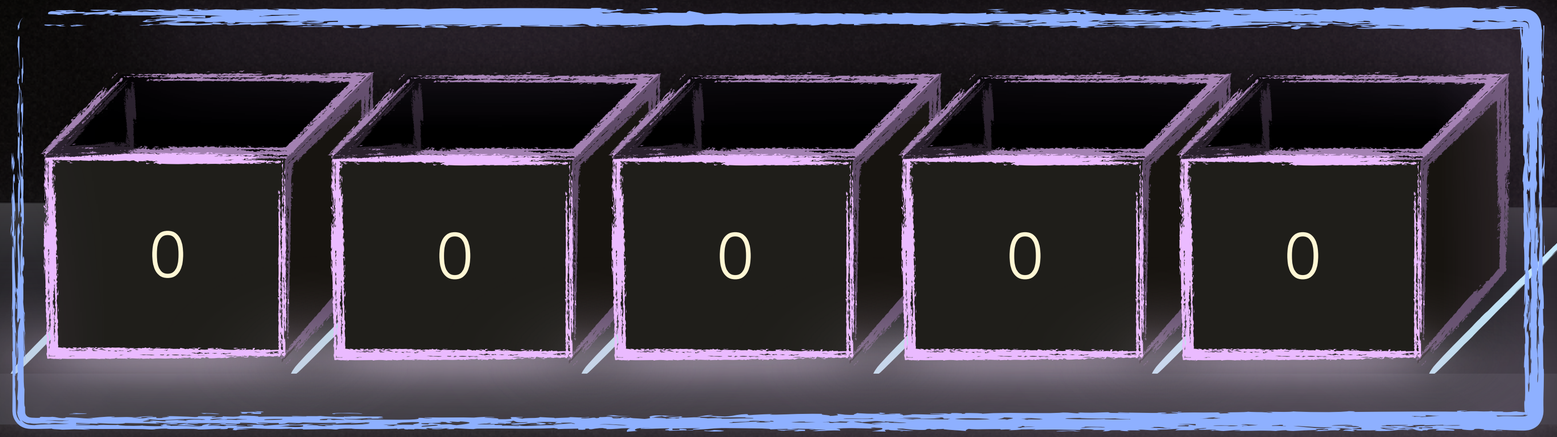

```
var ages [3]byte
```

length of this array

You can only store 3 elements in this array....

ARRAY'S LENGTH

It determines the number of elements that an array can store




```
var ages [5]byte
```

length of this array

You can only store 5 elements in this array....

ARRAY'S LENGTH

It can't be a negative number

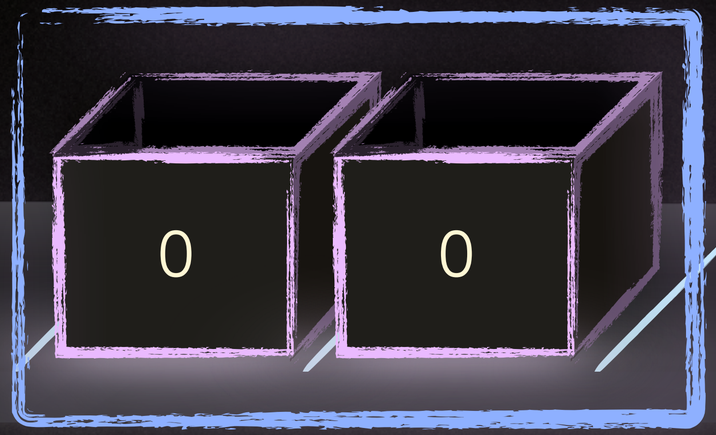
```
var ages [-1]byte
```

length of this array

negative length is meaningless

ARRAY'S ELEMENT TYPE

It determines the type of values that an array can store



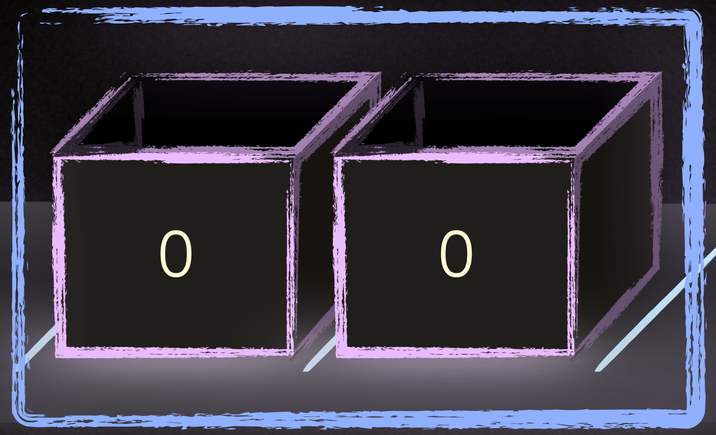

```
var ages [2]byte
```

element type

it always have to be declared!

ARRAY'S ELEMENT TYPE

It determines the type of values that an array can store



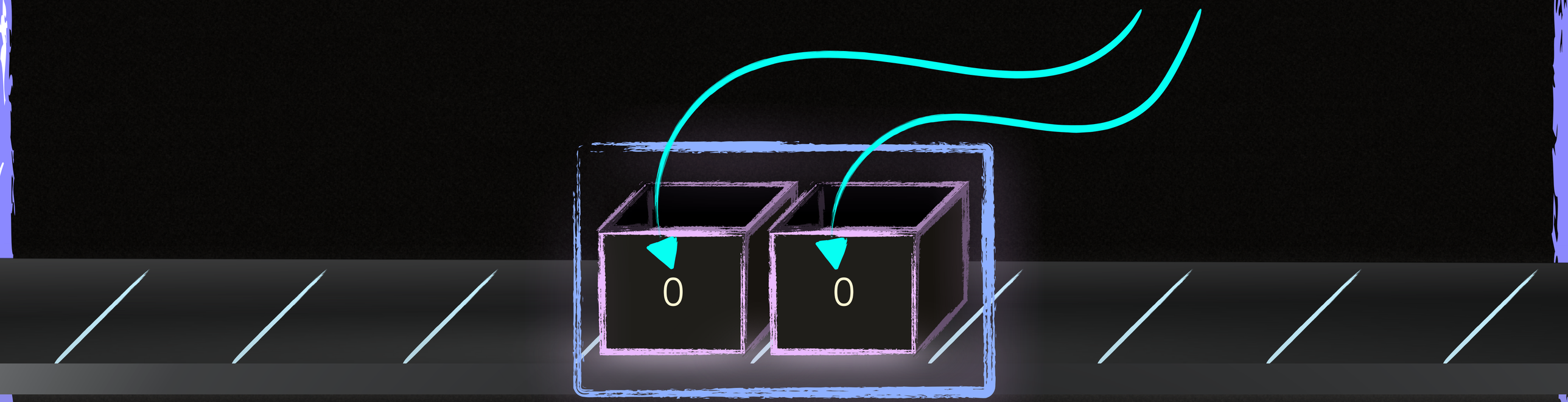

```
var ages [2]byte
```

element type

you can only store "byte" values in this array

ARRAY'S ELEMENT TYPE

Go automatically sets uninitialized array elements to their Zero Values



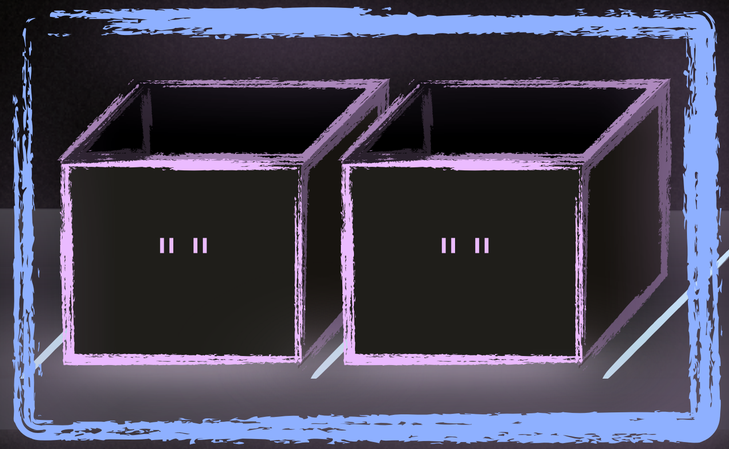
this array's element type is **byte**; its elements are not initialized; so, they're 0

```
var ages [2]byte
```

element type

ARRAY'S ELEMENT TYPE

You can use any type as an element type



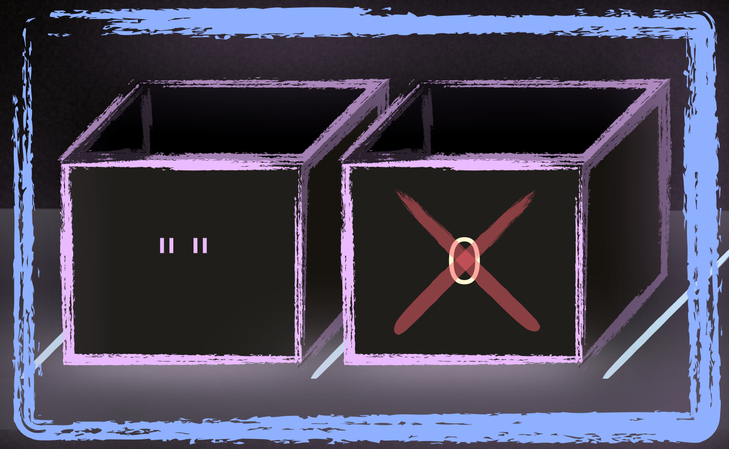

```
var tags [2]string
```

element type

you can only store "string" values in this array

ARRAY'S ELEMENT TYPE

You can't store different type of values in an array



```
var tags [2]string
```

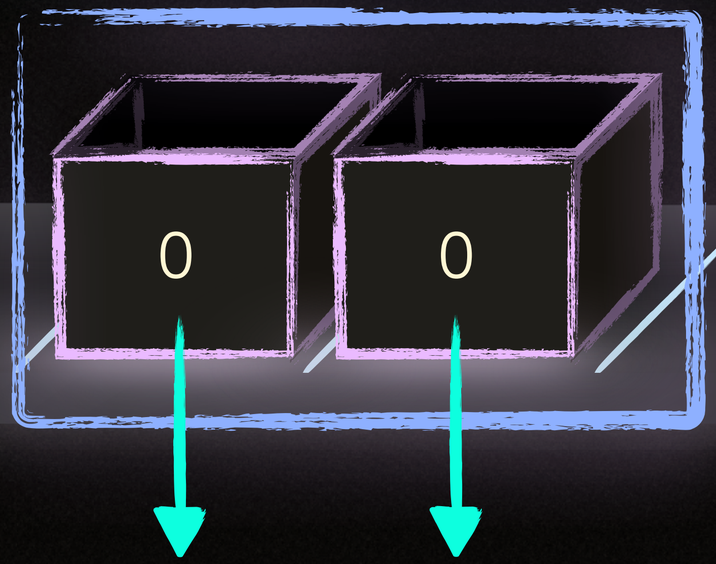
element type

all the values in an array should have the same type

UNNAMED VARIABLES

Each array element is actually an unnamed variable

```
var ages [2]byte
```



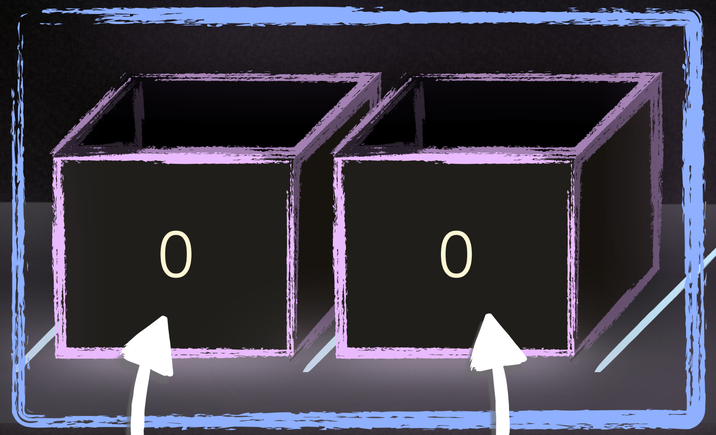
Unnamed Variables

Go creates unnamed variables depending on array's element type

GETTING ELEMENTS

You can get an array element using an index expression

```
var ages [2]byte
```



~~ages[-1]~~

index >= 0

ages[0]

ages[1]

len(ages) → 2

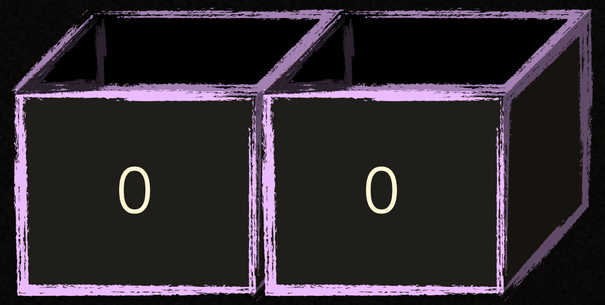
~~ages[2]~~

index < len(array)

SETTING ELEMENTS

You can set an element using the assignment operators

```
var ages [2]int

ages[0]  =  6
ages[1] -=  3

ages[0]  =  "Can I?"
```



type mismatch!

ages is an int array

ARRAY LITERAL

You can use it to create an array + along with its elements

ARRAY LITERAL

Isn't it cumbersome to set elements of an array like this?...

```
var books [4]string
books[0] = "Kafka's Revenge"
books[1] = "Stay Golden"
books[2] = "Everythingship"
books[3] = "Kafka's Revenge 2nd Edition"
```

ARRAY LITERAL

Creates & Initializes a new array with the given values

```
var books = [4]string{
  "Kafka's Revenge",
  "Stay Golden",
  "Everythingship",
  "Kafka's Revenge 2nd Edition",
}
```

ARRAY LITERAL

If you've the elements already: Use this syntax

books's type:
[4]string

```
books := [4]string{
    "Kafka's Revenge",
    "Stay Golden",
    "Everythingship",
    "Kafka's Revenge 2nd Edition",
}
```

ARRAY LITERAL

Array literal is one of the composite literals

```
[4]string{
  "Kafka's Revenge",
  "Stay Golden",
  "Everythingship",
  "Kafka's Revenge 2nd Edition",
}
```

Composite Literals are used to
create new composite values (*like arrays*)
along with their elements on the fly

ARRAY LITERAL

First: You need to provide the type of the composite literal

Array's Type
[4]string

```
[4]string{
    "Kafka's Revenge",
    "Stay Golden",
    "Everythingship",
    "Kafka's Revenge 2nd Edition",
}
```

Composite Literals are used to
create new composite values (*like arrays*)
along with their elements on the fly

ARRAY LITERAL

Third: You need to separate the elements using commas

```
[4]string{
  "Kafka's Revenge",
  "Stay Golden",
  "Everythingship",
  "Kafka's Revenge 2nd Edition",
}
```

The Last Comma is *only* needed
when you want to type the ending curly brace
in a new line

ARRAY LITERAL

You don't need to type the last comma here

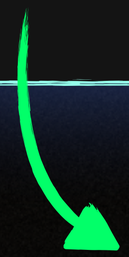
```
[4]string{"Kafka's Revenge", "Stay Golden"}
```

The **last element**, and,
the **ending curly brace**
are **on the same line**

ARRAY LITERAL

What happens when you don't initialize some of the elements?

```
[4]string{"Kafka's Revenge", "Stay Golden"}
```



The length is 4 but there are 2 elements here?!

ZERO VALUES

Go initializes the uninitialized elements to zero values automatically

```
[4]string{"Kafka's Revenge", "Stay Golden"}
```

ZERO VALUES

Go initializes the uninitialized elements to zero values automatically

```
[4]string{"Kafka's Revenge", "Stay Golden"}
```

Array's length is still 4 not 2

ELLIPSIS...

Go finds out the length of the array automatically

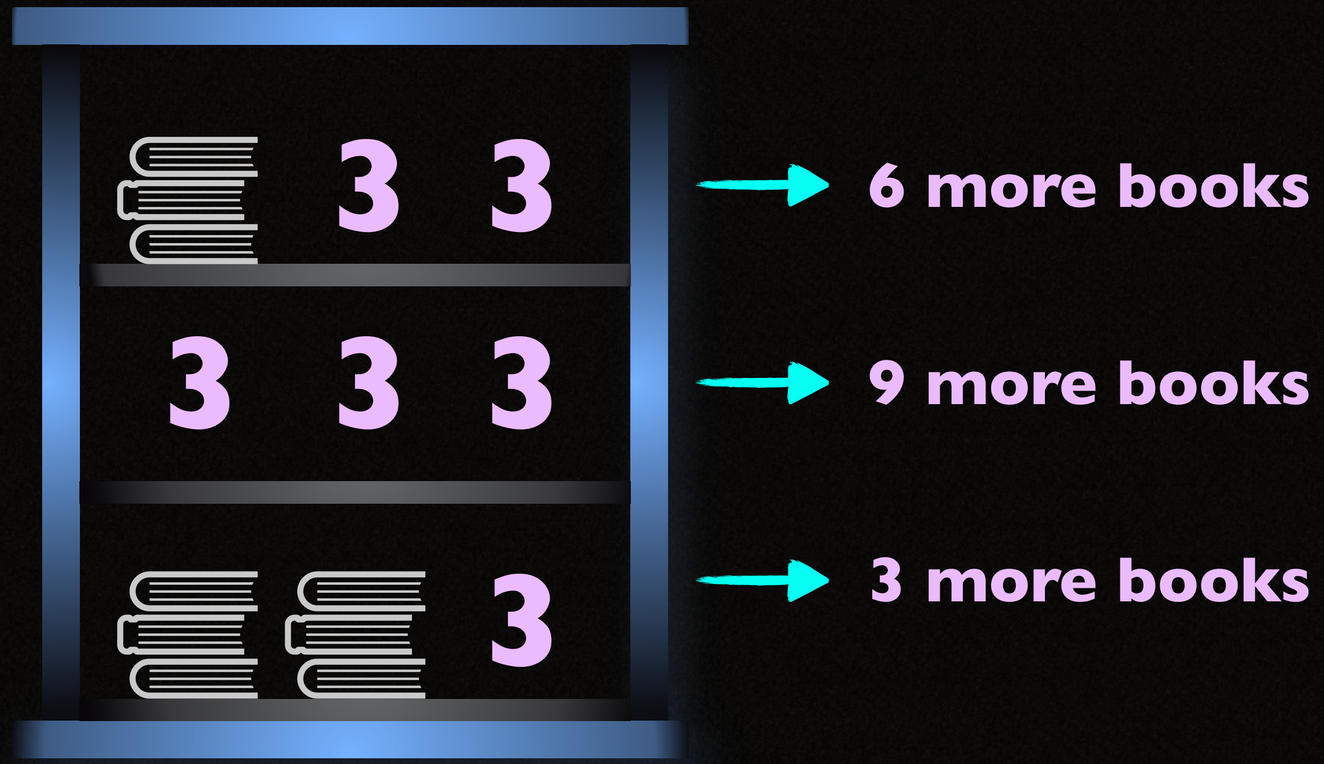
```
[...]string{"Kafka's Revenge", "Stay Golden"}
```

The Length is 2 elements

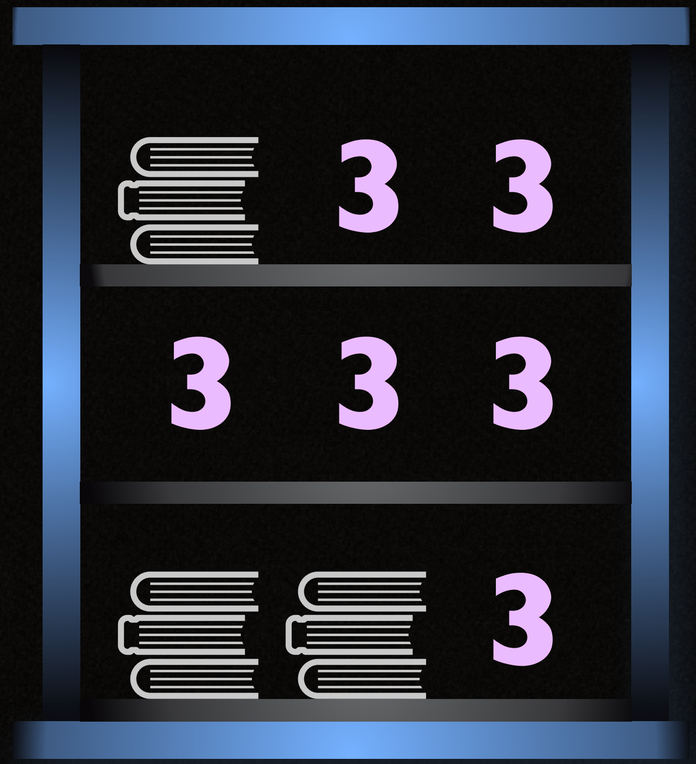
COMPARING ARRAYS

You can only compare arrays that have identical types

Hipster's Love Book Collection



Hipster's Love Book Collection



6 more books

9 more books

3 more books

bookcase
data buffer
database records
...

```
[3]int{6, 9, 3}
```

```
[3]int{6, 9, 3}
```

==

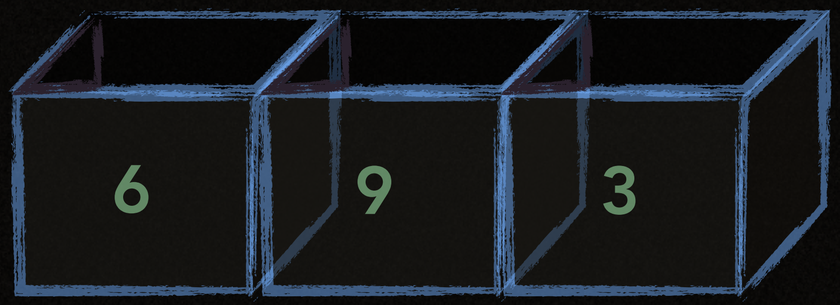
?

```
[3]int{6, 9, 3}
```

COMPARING ARRAYS

Arrays are comparable when their types are identical

```
[3]int{6, 9, 3}
```



==

```
[3]int{6, 9, 3}
```

Comparable

They've identical types

`[3]int`

COMPARING ARRAYS

Arrays are equal when their elements are equal

```
[3]int{6, 9, 3}
```

==

```
[3]int{6, 9, 3}
```

Equal

They have the same elements

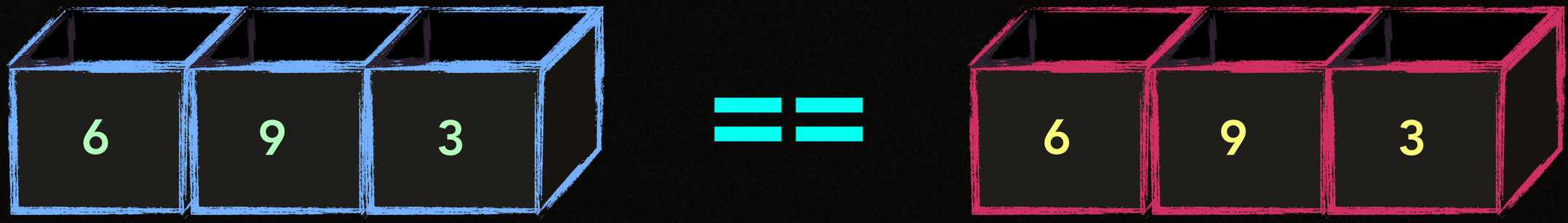
`6`, `9`, and `3`

COMPARING ARRAYS

Go compares every element of the arrays one by one

```
blue := [3]int{6, 9, 3}
```

```
red := [3]int{6, 9, 3}
```




```
blue[0] == red[0]
blue[1] == red[1]
blue[2] == red[2]
```

COMPARING ARRAYS

Arrays are not equal even when their elements are not equal

```
[3]int{6, 9, 3}
```

≠

```
[3]int{3, 9, 6}
```

Not Equal

They have the **same elements**; *but* in a **different order**

COMPARING ARRAYS

Can you compare these arrays?

```
[3]int{6, 9, 3}
```

?

```
[2]int{6, 9}
```

COMPARING ARRAYS

Arrays are not comparable when their types are different

```
[3]int{6, 9, 3}
```

```
[2]int{6, 9}
```

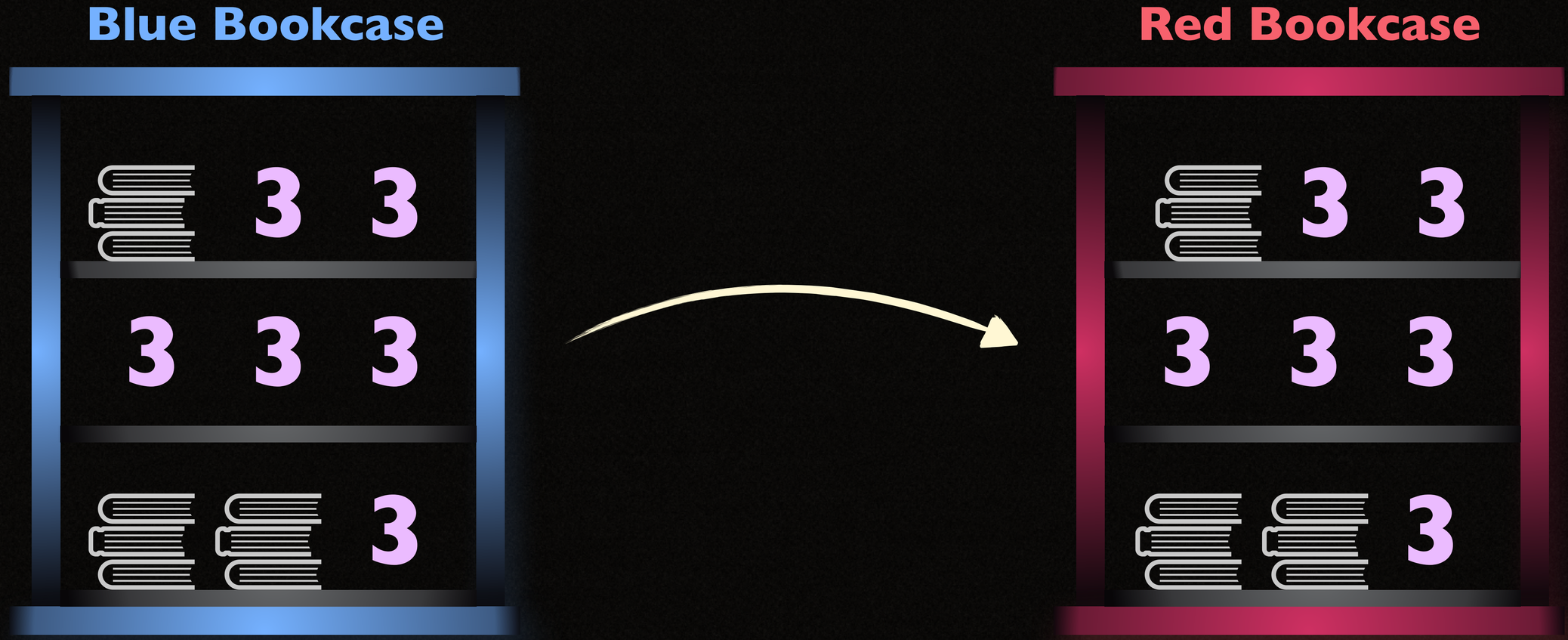
Not Comparable

They have **different types**

ASSIGNING ARRAYS

Remember, if you can compare, then you can assign

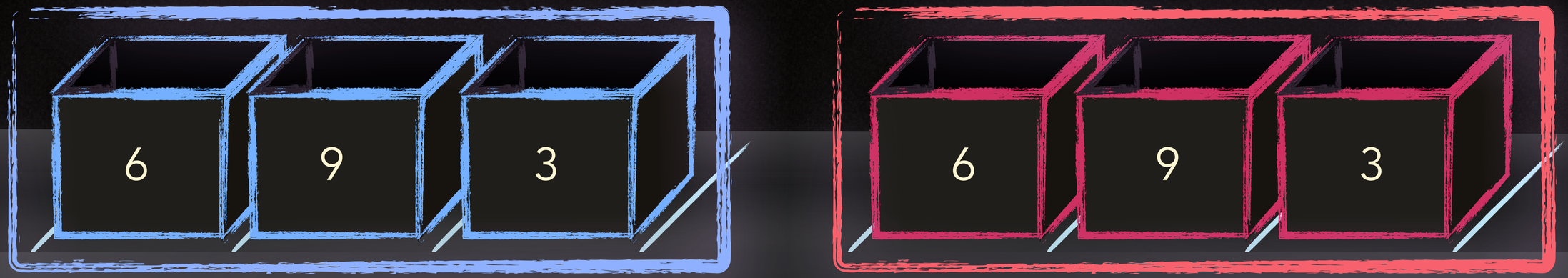
Hipster's Love Book Collection



ARRAYS ARE COPIED

Assigned array's elements will be copied into a new array

allocates
24 bytes of new memory




```
blue := [3]int{6, 9, 3}
```

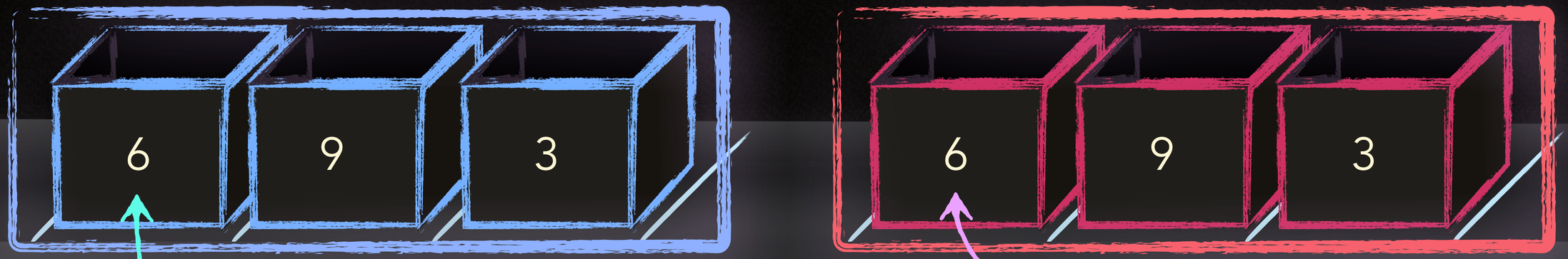
```
red := blue
```

ARRAYS ARE COPIED

Copied array and the Original array are **not** connected

```
blue := [3]int{6, 9, 3}
```

"red" array stays the same
it's a new array

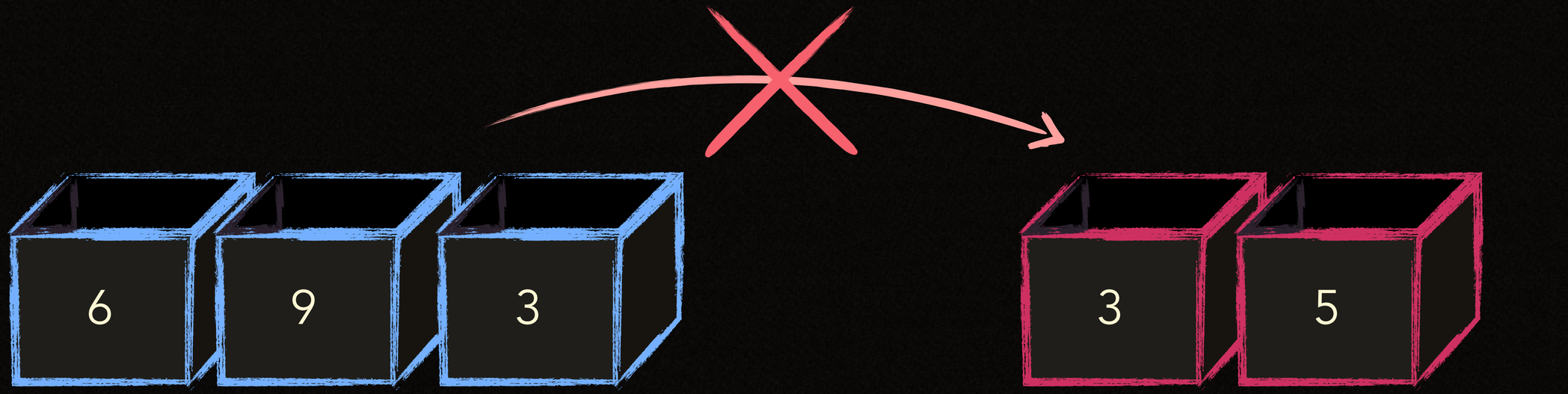



```
red := blue
blue[0] = 10
```

ASSIGNING ARRAYS

Different types of arrays are not assignable

```
red = blue
```




```
blue := [3]int{6, 9, 3}
```

```
red := [2]int{3, 5}
```

MULTI-DIMENSIONAL ARRAYS

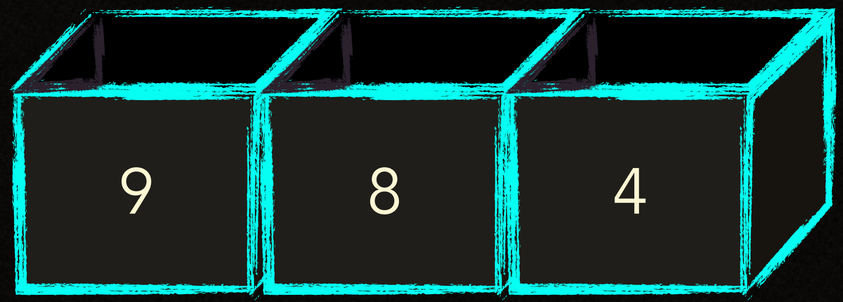
Arrays of Arrays

1st Student's Grades

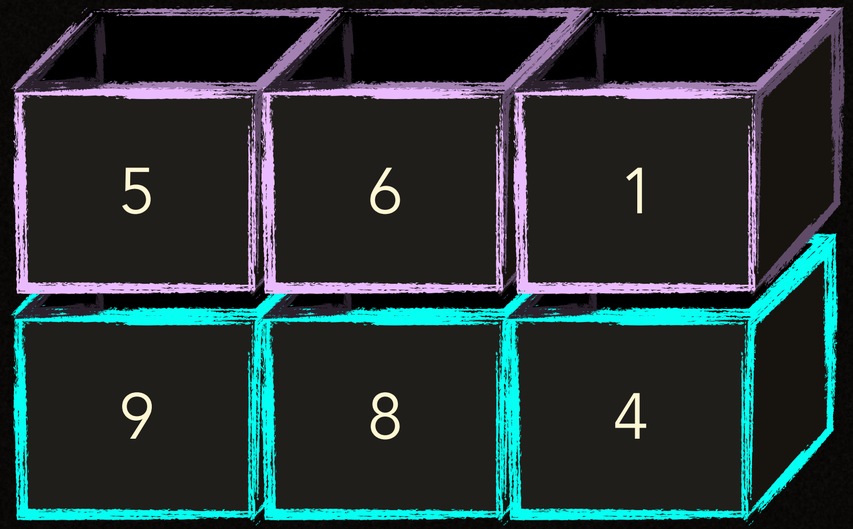
```
[3]int{5, 6, 1}
```

2nd Student's Grades

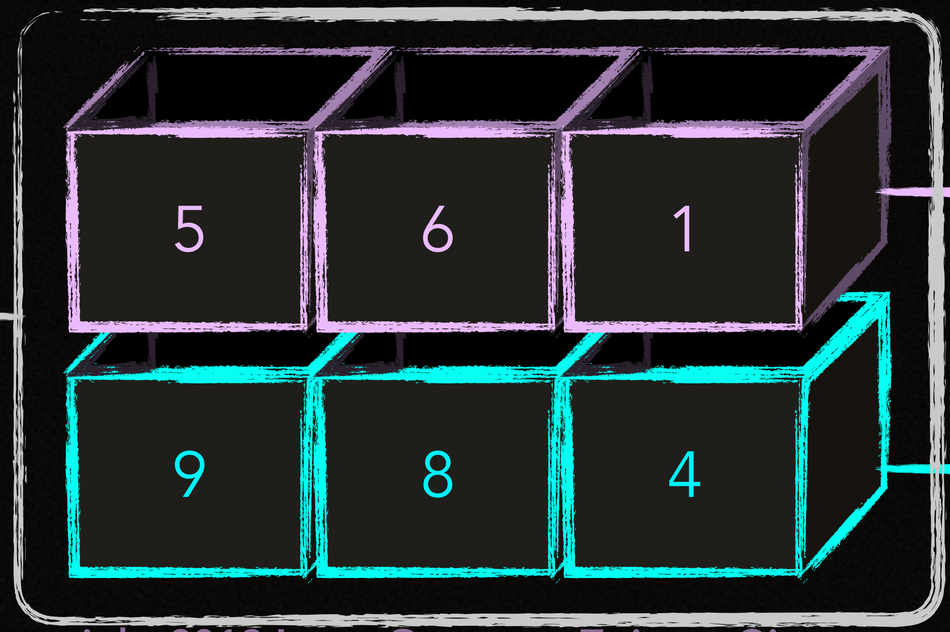
```
[3]int{9, 8, 4}
```



```
[3]int{5, 6, 1}
[3]int{9, 8, 4}
```



```
[2][3]int{
    [3]int{5, 6, 1},
    [3]int{9, 8, 4},
}
```



outer array

`[2][3]int`

contains

the inner arrays

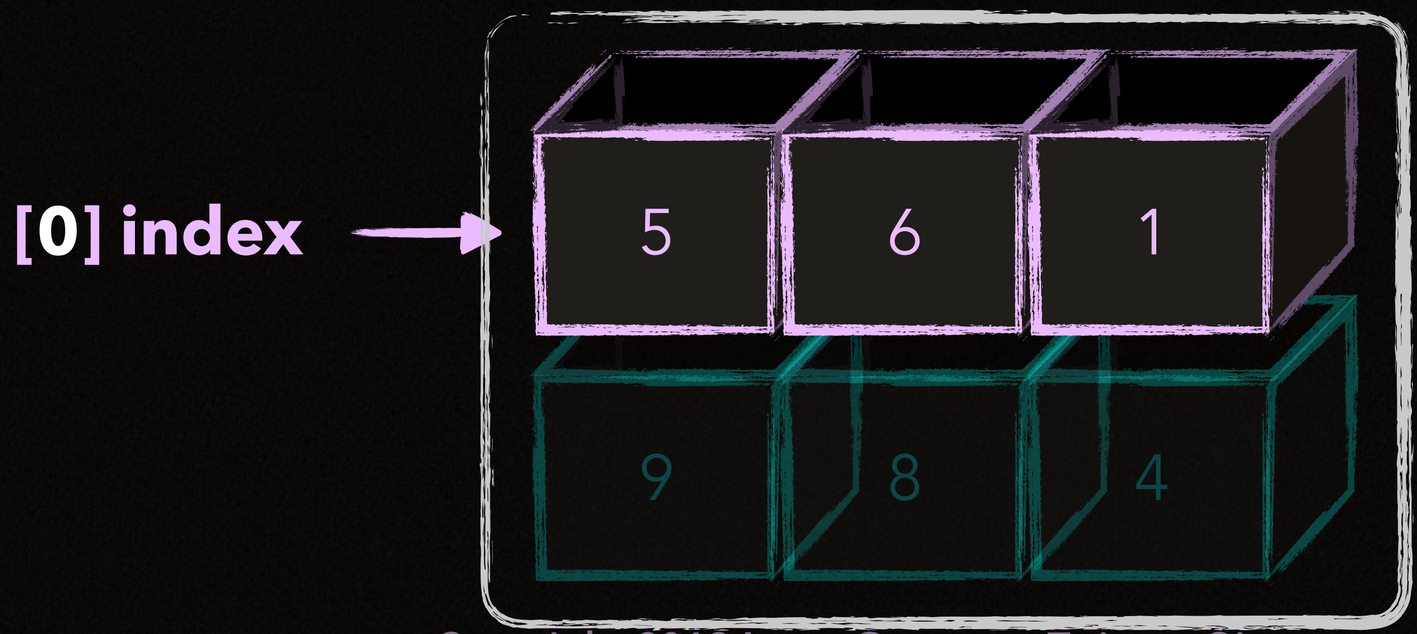
1st inner array

`[3]int`

2nd inner array

`[3]int`

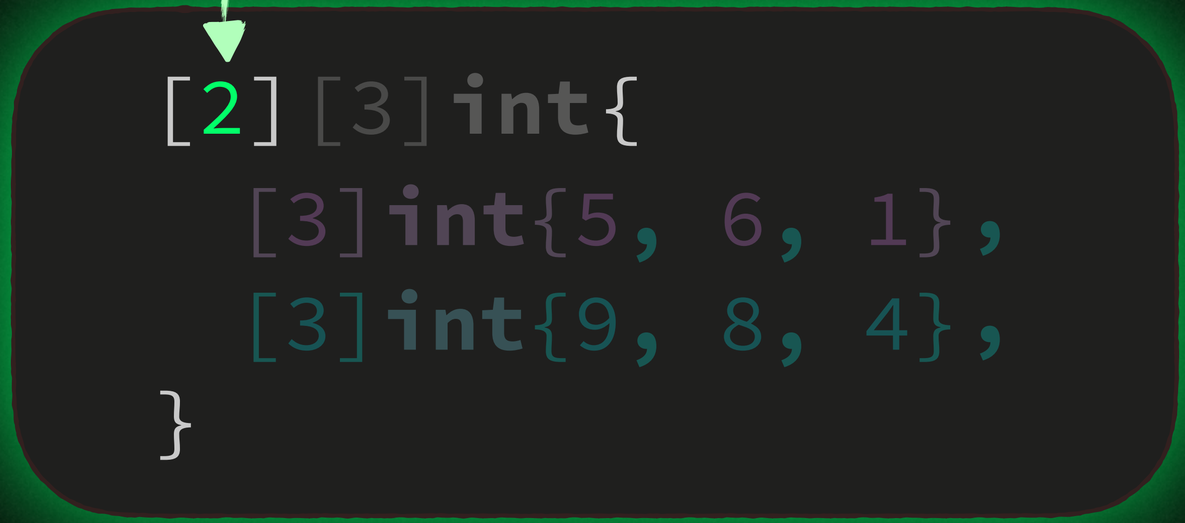
```
[2][3]int{
    [3]int{5, 6, 1},
    [3]int{9, 8, 4},
}
```



```
[2][3]int{
    [3]int{5, 6, 1},
    [3]int{9, 8, 4},
}
```

The length of the array

It can be any length



```
[2][3]int{
    [3]int{5, 6, 1},
    [3]int{9, 8, 4},
}
```

So: "for this array", there can be only 2 inner arrays

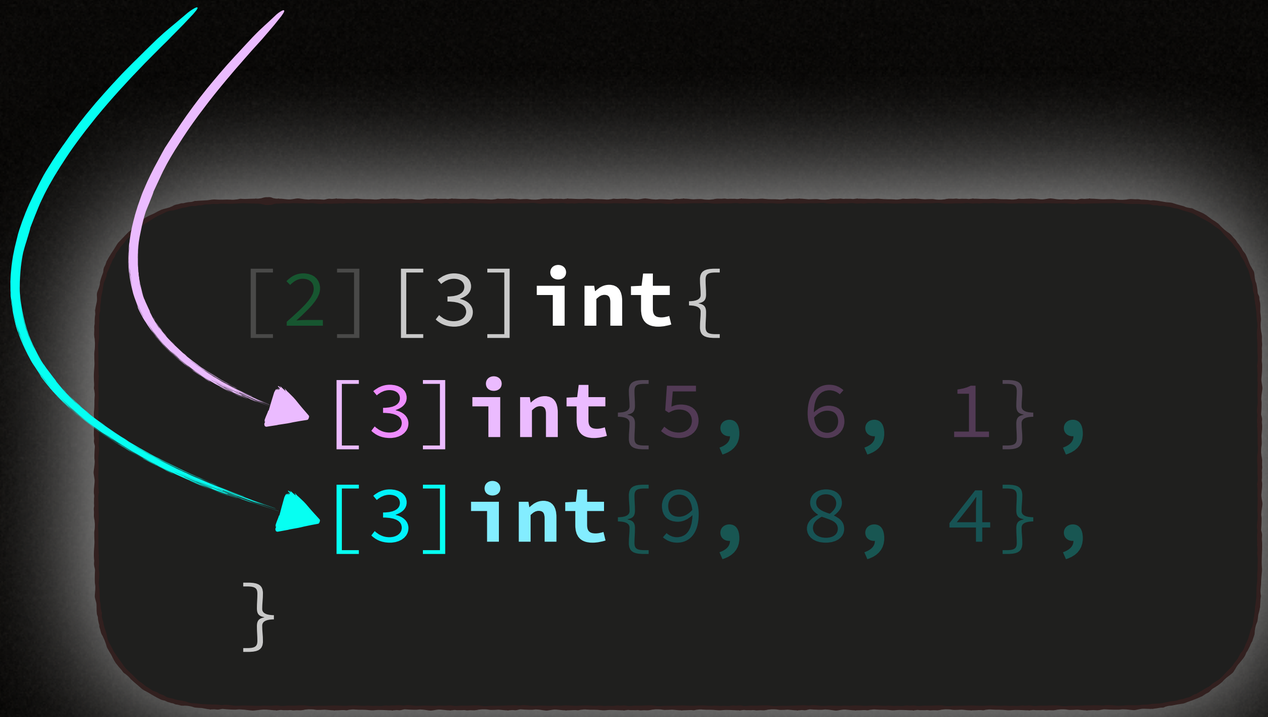
The element type of the array

It can be any type

```
[2][3]int{
    [3]int{5, 6, 1},
    [3]int{9, 8, 4},
}
```

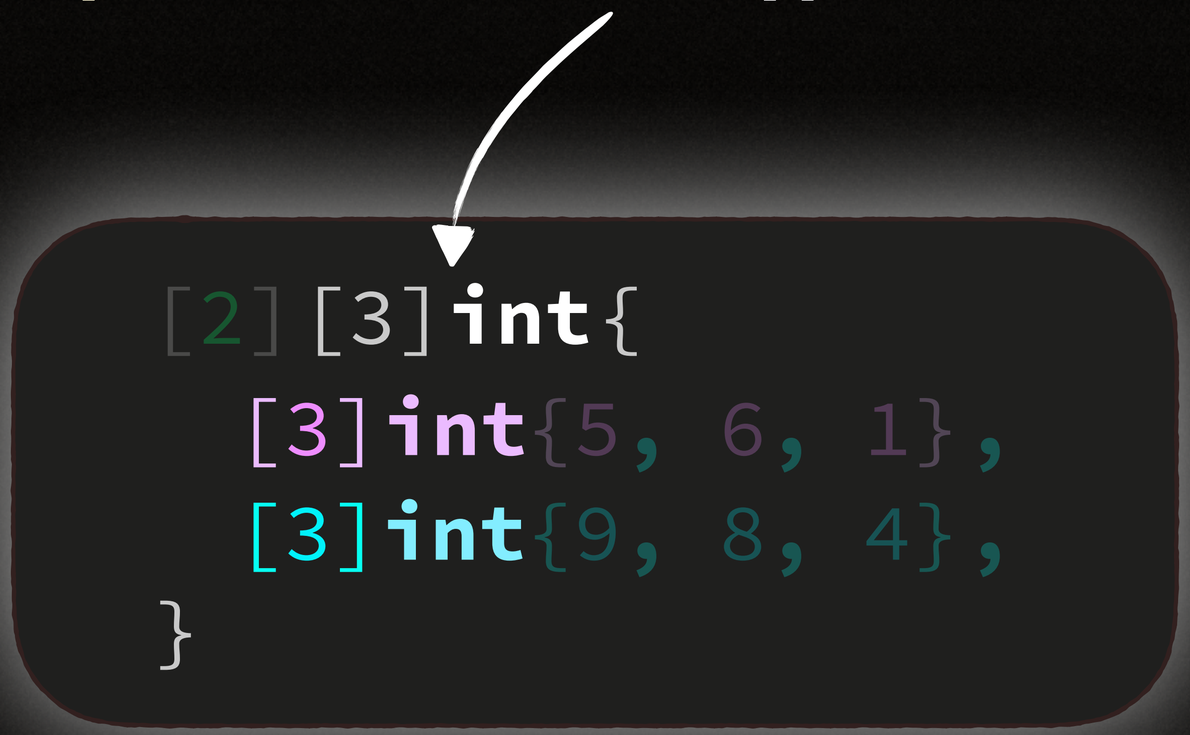
Element type determines
the type of the inner arrays

Element types of the inner arrays

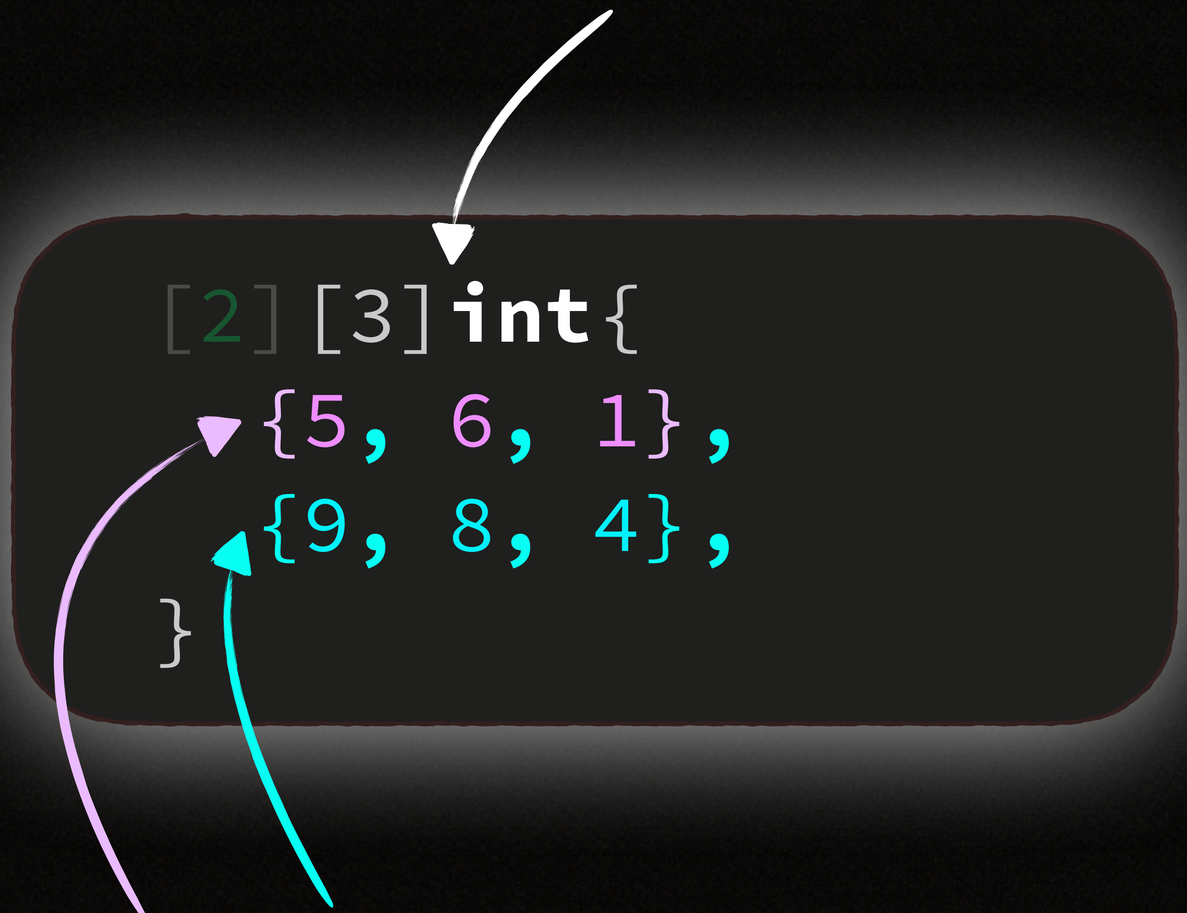


Inner arrays should have the same types

Go already knows about the type of the elements



Go already knows about the type of the elements



So, you can omit the types
Go implicitly adds them for you

KEYED ELEMENTS

Describe the indexes manually

Cryptocurrency exchange ratios

What is 1 Bitcoin in Ethereum?

1 Ethereum is 0.5 Bitcoin

```
rates := [3]float64{
  0.5,
  2.5,
  1.5,
}
```

indexes

0

1

2

KEYED ELEMENTS

Keyed elements describe the index positions

```
rates := [3]float64{
  0: 0.5,
  1: 2.5,
  2: 1.5,
}
```

KEYED ELEMENTS

Each key corresponds to an index of the array

```
rates := [3]float64{
  0: 0.5,
  1: 2.5,
  2: 1.5,
}
```

indexes

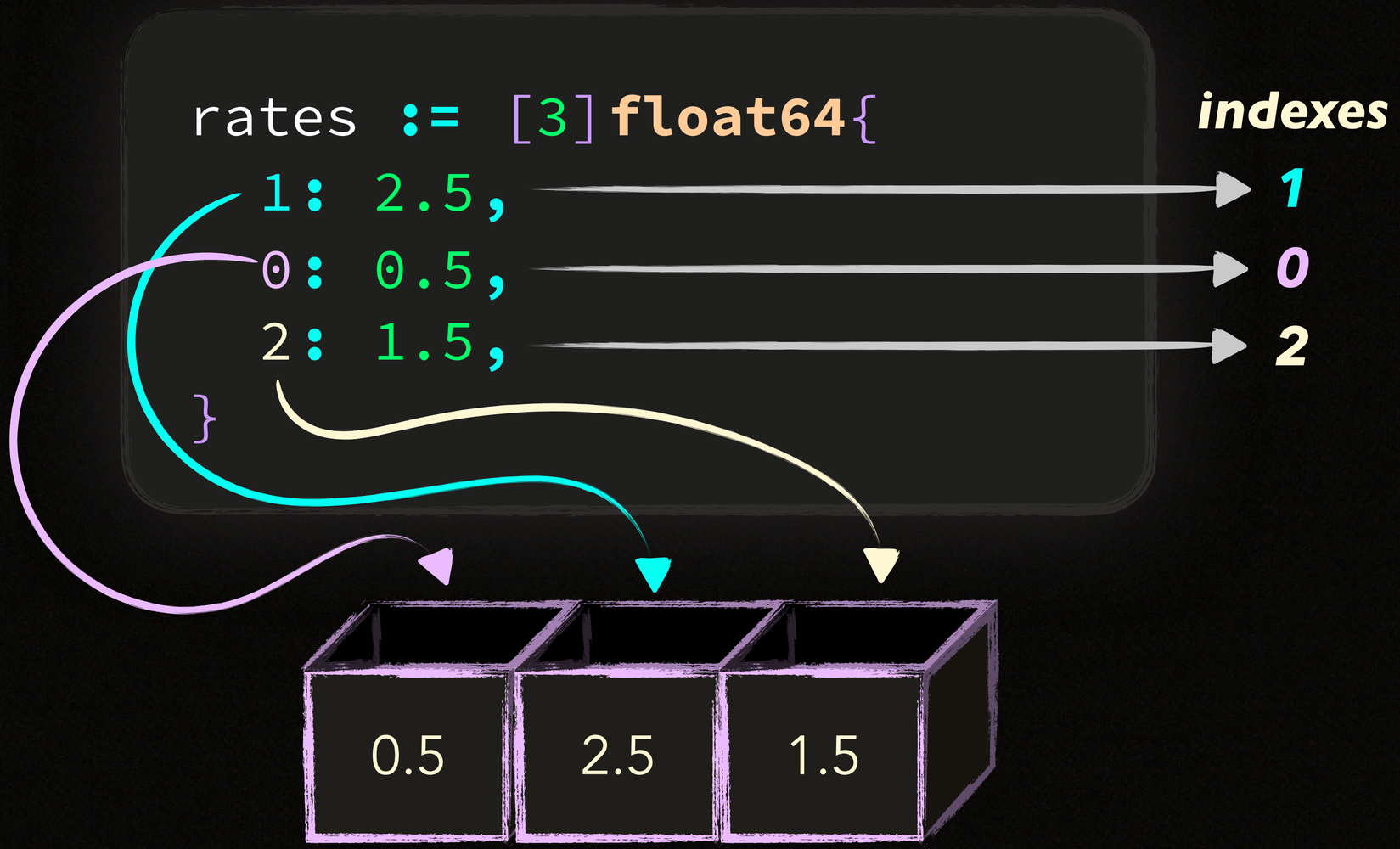
0

1

2

KEYED ELEMENTS

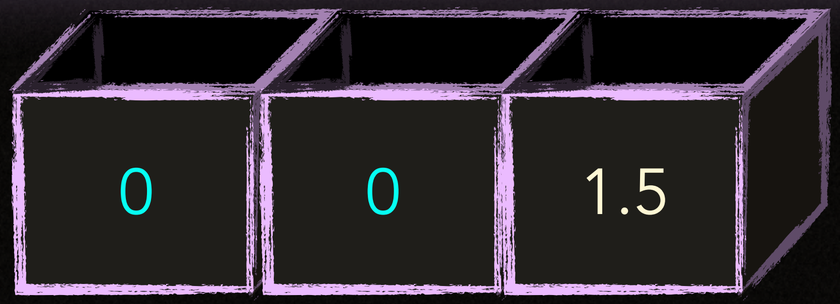
The keyed elements can be in any order



KEYED ELEMENTS

Uninitialized elements will be initialized to their zero values

```
rates := [3]float64{
   2: 1.5,
}
```

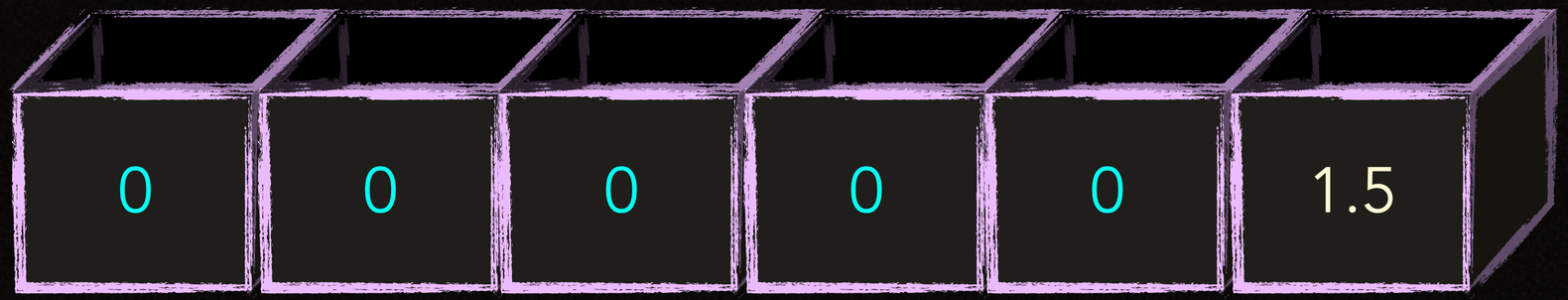


Go initializes
the uninitialized elements
to their zero values

KEYED ELEMENTS

Uninitialized elements will be initialized to their zero values

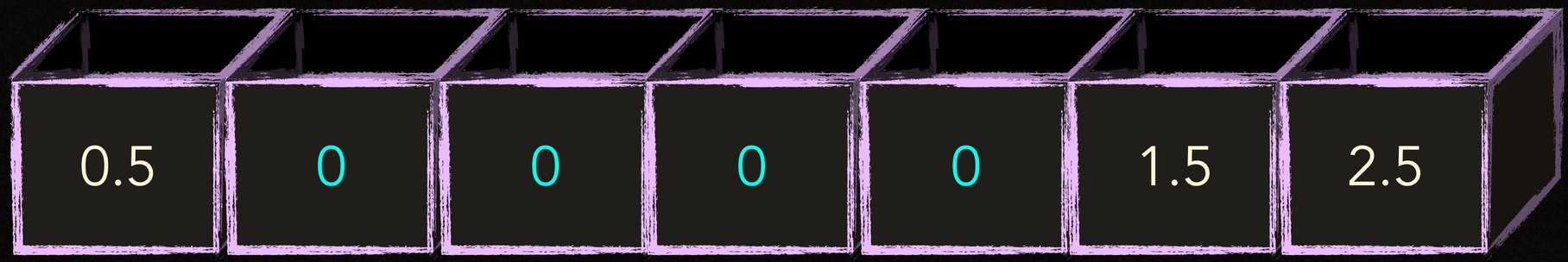
```
rates := [...]float64{
   5: 1.5,
}
```



KEYED ELEMENTS

Keyed elements can determine the index position of the unkeyed elements

```
rates := [...]float64{
  5: 1.5,
  2.5,
  0: 0.5,
}
```



XRATIO

Cryptocurrency exchange ratios

NAMED vs UNNAMED TYPES

Composite Types are Unnamed Types

UNNAMED TYPES

An unnamed composite type's underlying type is itself!

```
[3]int{6, 9, 3}
```

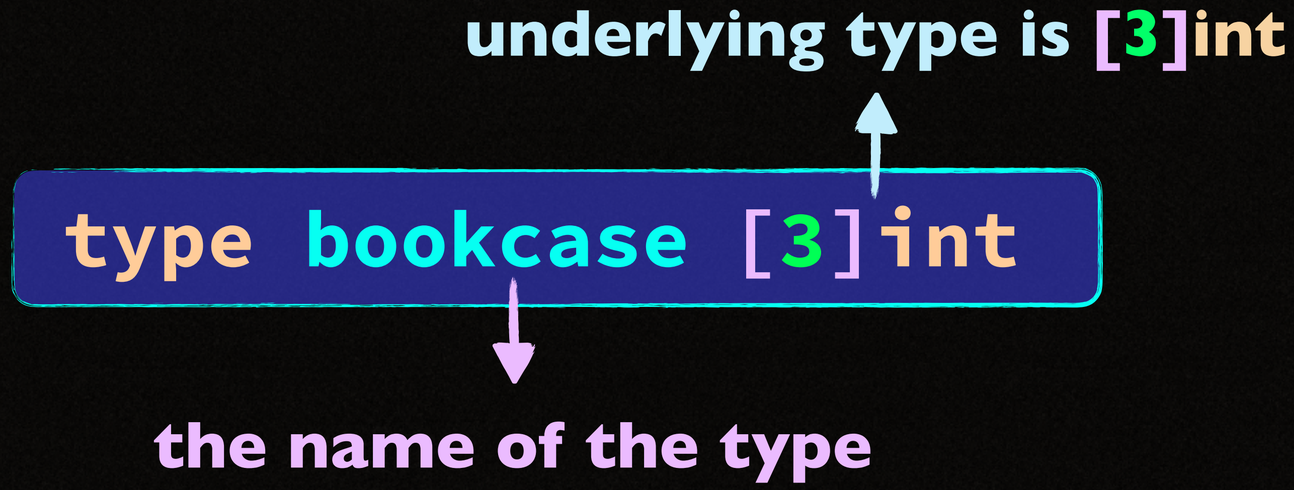
unnamed type

underlying type is itself: [3]int

[It has its own structure]

NAMED TYPE

Type definition creates a new type with a new name



UNNAMED <-> NAMED

Unnamed and Named Typed Arrays comparable

If their underlying types are identical

✓

You don't need to convert these array values

NAMED <-> NAMED

A named type is always a different type than any other type

```
type bookcase [3]int
```

```
type cabinet [3]int
```

```
bookcase{6, 9, 3}
```

```
cabinet{6, 9, 3}
```

NAMED <-> NAMED

They're comparable when converted ✓

```
type bookcase [3]int
```

```
type cabinet [3]int
```

```
bookcase{6, 9, 3}
```

==

✓

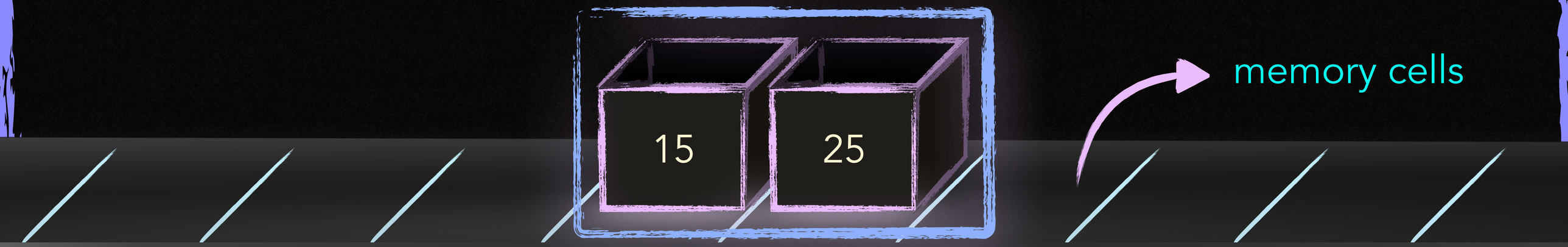
```
bookcase(
    cabinet{6, 9, 3},
)
```

You can only convert
if their underlying types are identical

summary

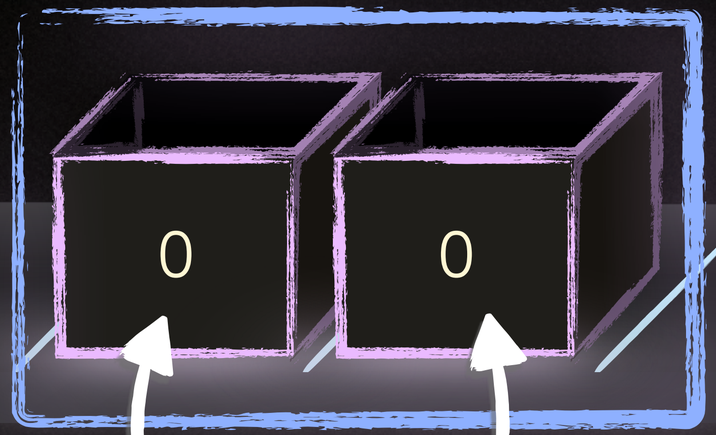
Array is a collection of elements

It stores the same type and the same number of elements in contiguous memory locations



You can access array elements using index expressions

```
var ages [2]byte
```



~~ages[-1]~~

index >= 0

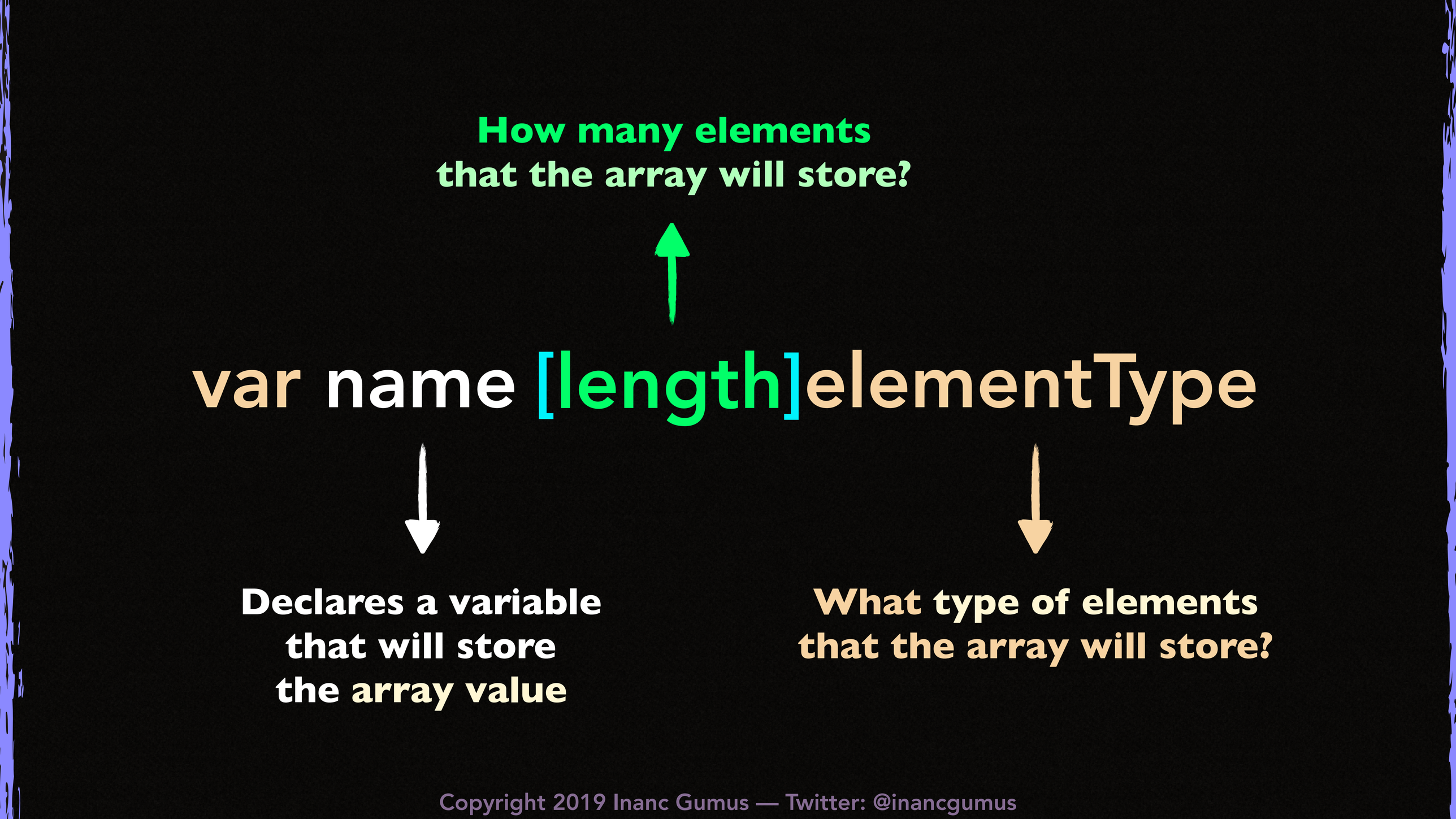
ages[0]

ages[1]

len(ages) → 2

~~ages[2]~~

index < len(array)



var name [length]elementType

This is the type of the array

Its length and its element type
determine the type of the array as a whole

Array Literals

```
[4]string{"Kafka's Revenge", "Stay Golden"}
```

Array's length is still 4 not 2

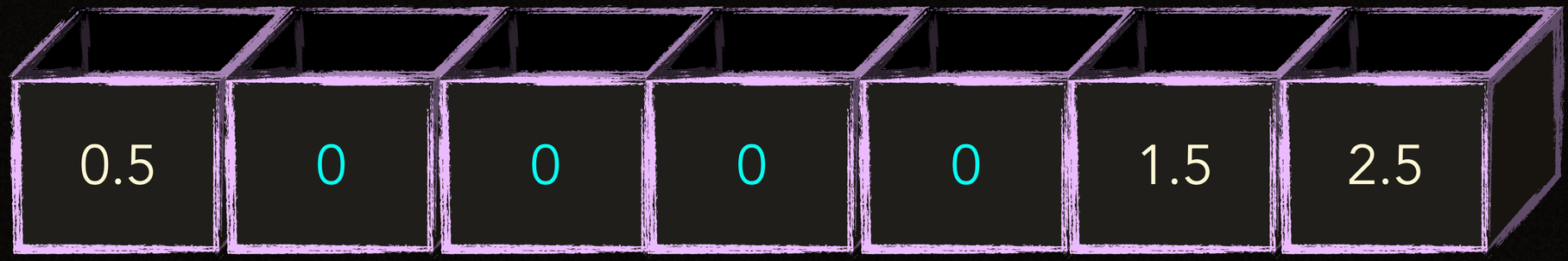
Ellipsis...

```
[...]string{"Kafka's Revenge", "Stay Golden"}
```

The Length is 2 elements

Keyed Elements

```
rates := [...]float64{
  5: 1.5,
  2.5,
  0: 0.5,
}
```



Multi-Dimensional Arrays

```
[2][3]int{
    {5, 6, 1},
    {9, 8, 4},
}
```

An array with two inner [3]int arrays

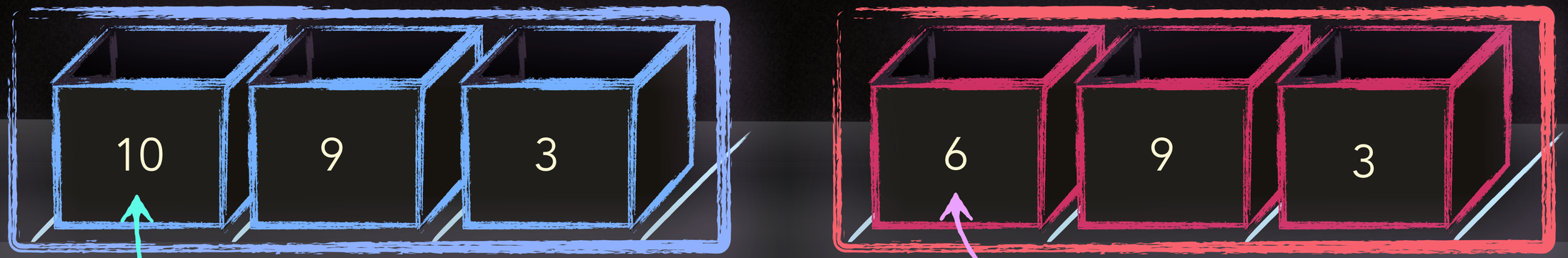
Element type: [3]int

Length: 2

Copied array and the Original array are **not** connected

```
blue := [3]int{6, 9, 3}
```

"red" array stays the same
it's a copy of the blue array




```
red := blue
blue[0] = 10
```

Different types of arrays are neither comparable nor assignable

```
red == blue
```

```
blue = red
```


```
blue := [3]int{6, 9, 3}
```


```
red := [2]int{6, 9}
```

An unnamed composite type's underlying type is itself!

```
[3]int{6, 9, 3}
```

unnamed type

underlying type is itself: [3]int

[It has its own structure]

Unnamed and Named typed values are comparable

If their underlying types are identical

✓

You don't need to convert these array values

PART IV

Composite Types

Arrays ✓

Collection
of
Elements
Indexable
Fixed Length

Slices

Collection
of
Elements
Indexable
Dynamic length

String Internals

Byte Slices
ASCII & Unicode
Encoding & Decoding

Maps

Collection
of
Indexable
Key-Value Pairs

Structs

Groups
different types of
variables together