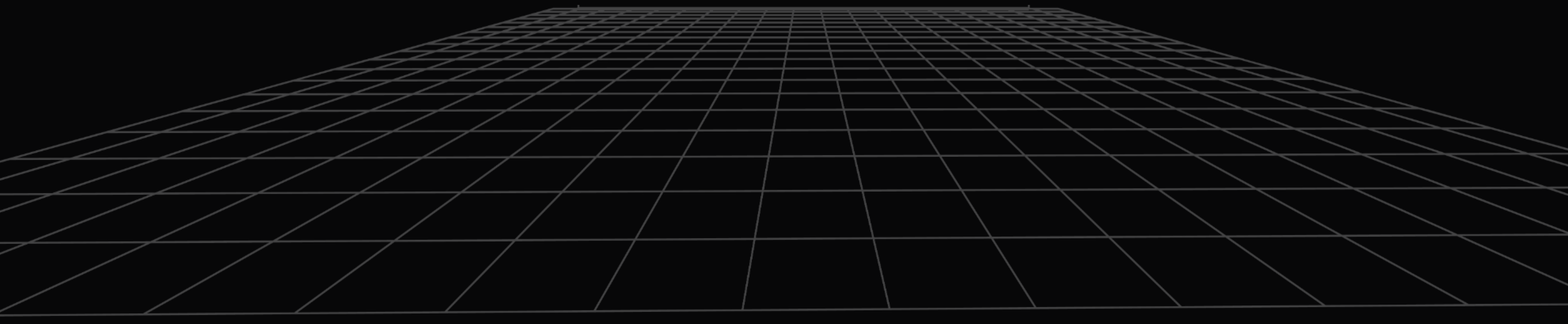


МАССИВЫ И СРЕЗЫ

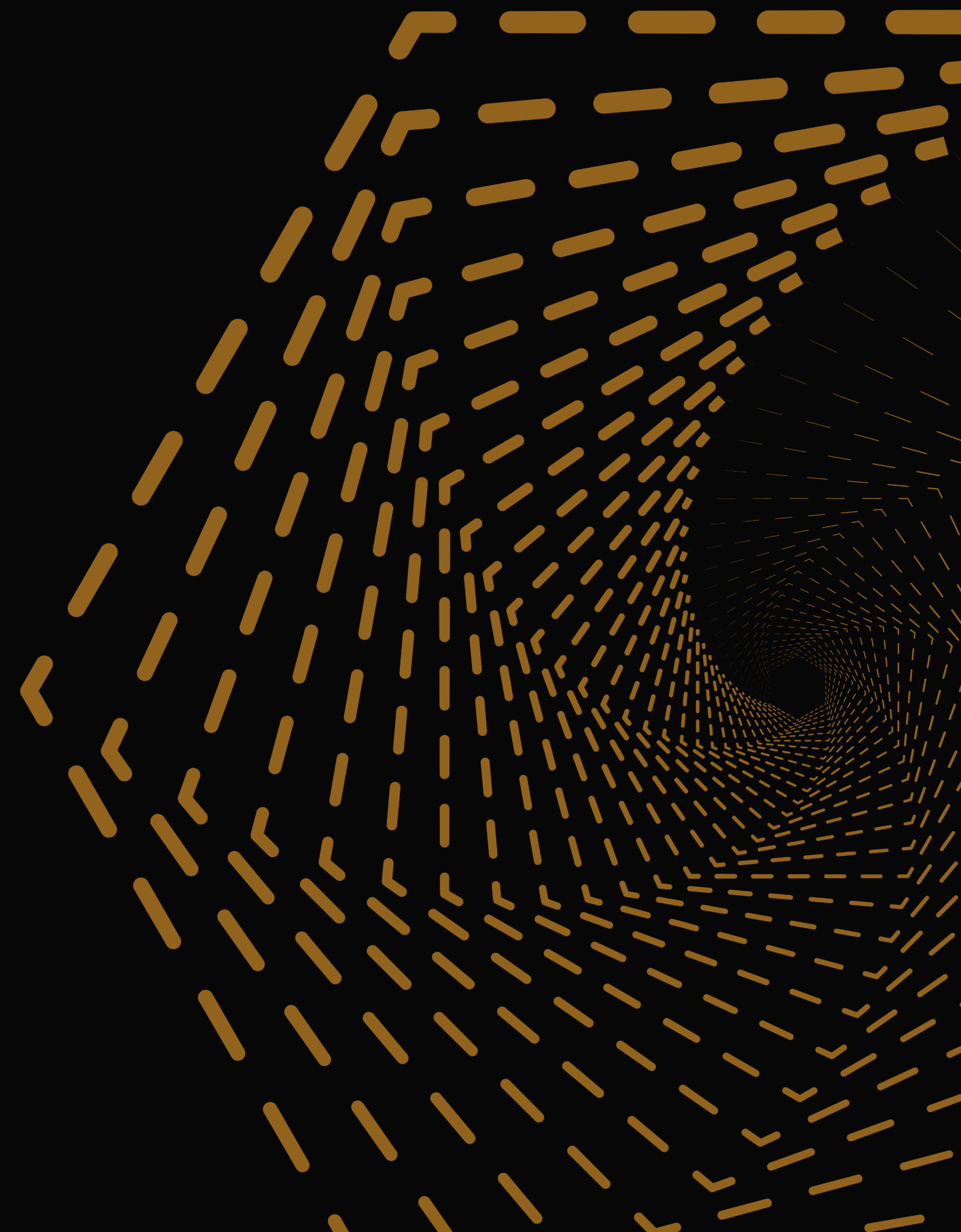


ПРОВЕРЬ ЗАПИСЬ



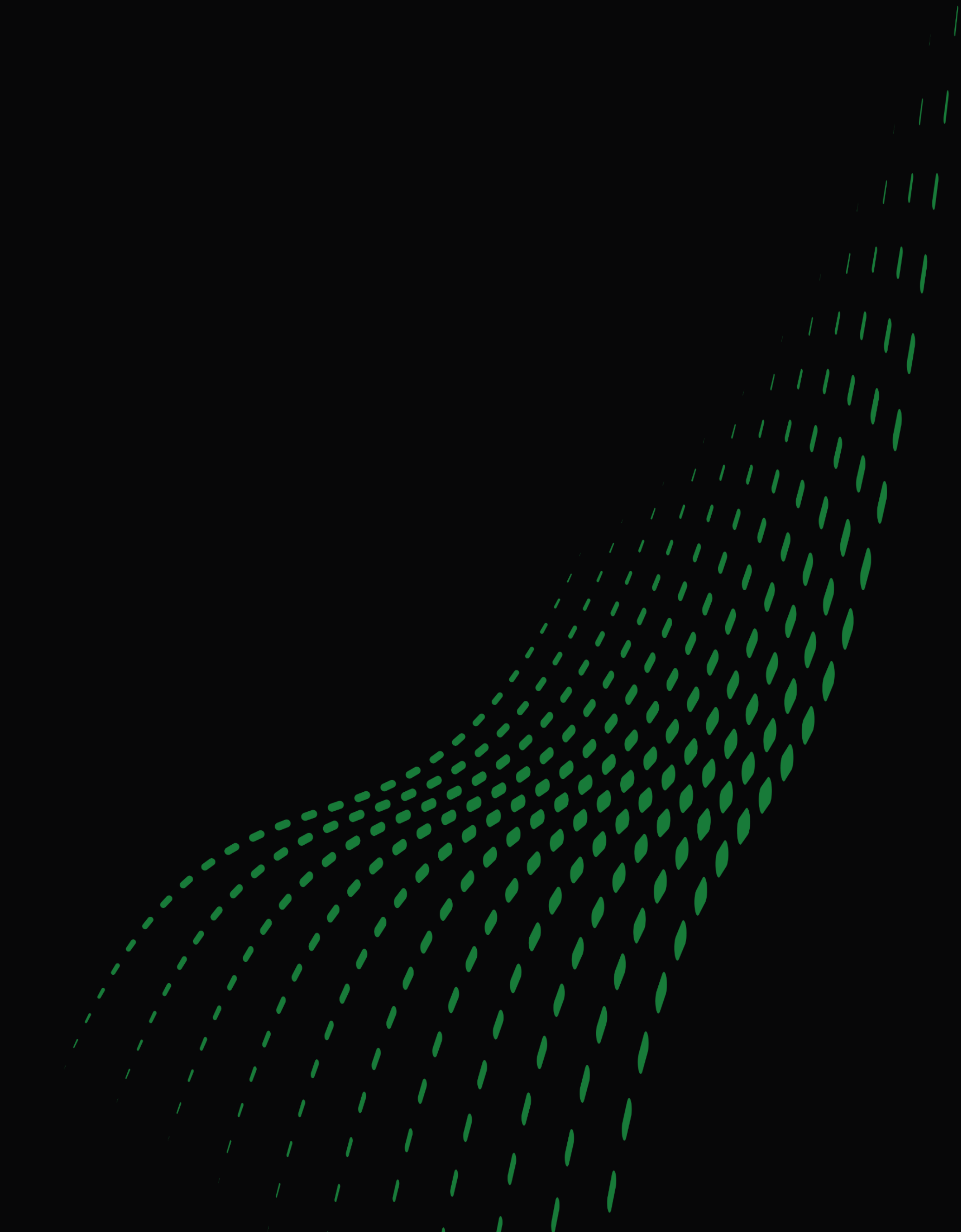
ПРАВИЛА ЗАНЯТИЯ

1. вопросы в чате можно задавать
в любое время
2. вопросы голосом задаем
по поднятой руке в Zoom
3. ответы на вопросы будут
в запланированных местах



ПЛАН ЛЕКЦИИ

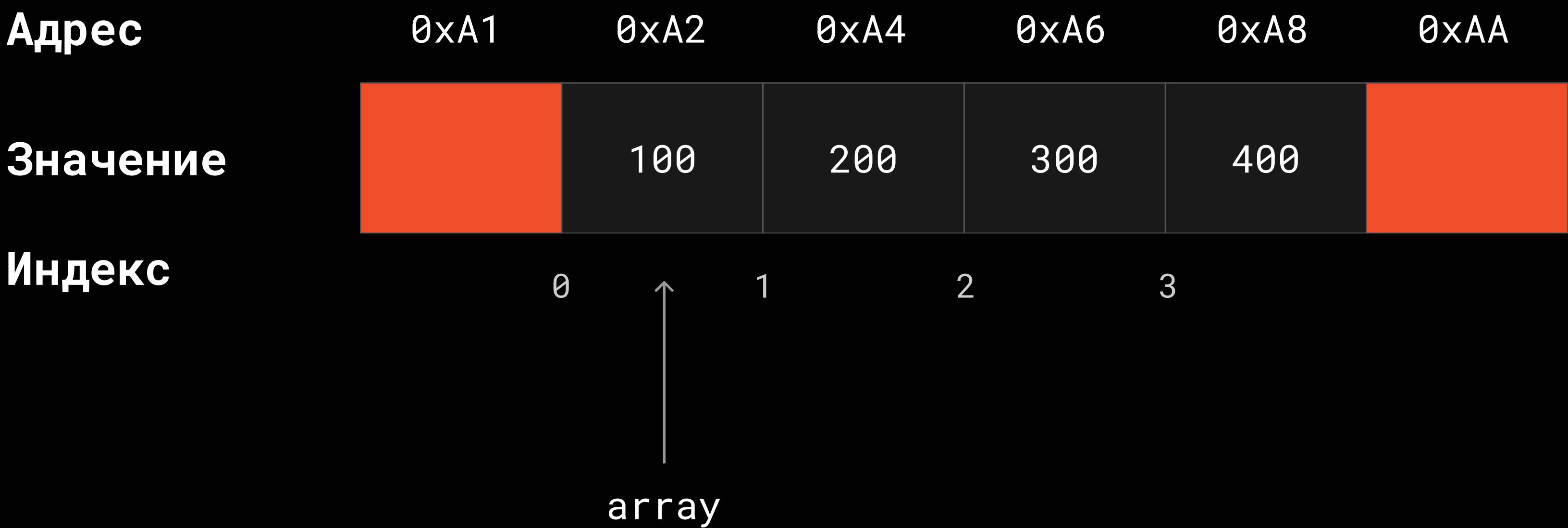
1. Массивы
2. Срезы
3. Получение подсрезов (нарезка)
4. Тонкости срезов



МАССИВЫ

МАССИВ

Структура данных фиксированного размера, хранящая
последовательно элементы одного типа
(каждый элемент имеет индекс от 0 до n-1)



СОЗДАНИЕ МАССИВА



```
1  var data1 [5]int           // [0 0 0 0 0]
2  var data2 [2][5]int        // [[0 0 0 0 0] [0 0 0 0 0]]
3  data3 := [...]int{1, 2, 3} // [1 2 3]
4  data4 := [5]int{1, 2, 3}   // [1 2 3 0 0]
5  data5 := [5]int{3: 4}      // [0 0 0 4 0]
6  data6 := [5]int{2: 5, 6, 1: 7} // [0 7 5 6 0]
```

API



```
1 value := data[5] // чтение из массива
2 data[5] = 10000 // запись в массив
3 len(data) // получение длины массива
4 cap(data) // получение емкости массива
5 p := &data // получение указателя на массив
6 s := data[1:4] // получение среза из массива
```

EXAMPLE

Array operations

МАССИВ $\neq$ УКАЗАТЕЛЬ И ВНУТРИ НЕГО НЕТ НИКАКОГО УКАЗАТЕЛЯ

Это структура данных у которой есть только
адрес начала, длина и размер элементов



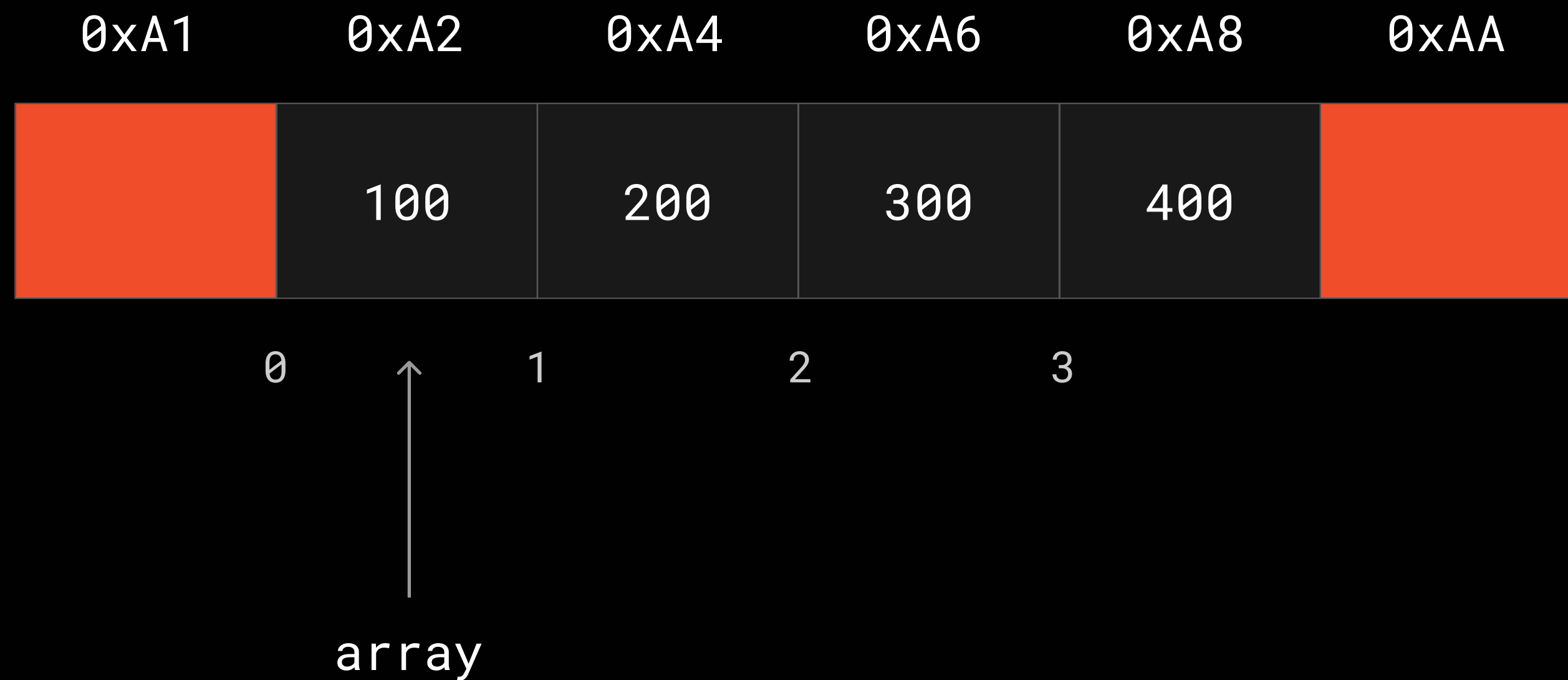
```
1 var values = [...]int{  
2     100, 200, 300, 400, 500,  
3 }
```



```
1 var value1 int = 100  
2 var value2 int = 200  
3 var value3 int = 300  
4 var value4 int = 400  
5 var value5 int = 500
```

array[idx] = *(array + idx * elem_size):

- array[0] = *(array + 0 * 2) // *(0xA02 + 0 * 2) = 0xA2
- array[1] = *(array + 1 * 2) // *(0xA02 + 1 * 2) = 0xA4
- array[2] = *(array + 2 * 2) // *(0xA02 + 2 * 2) = 0xA6
- array[3] = *(array + 3 * 2) // *(0xA02 + 3 * 2) = 0xA8

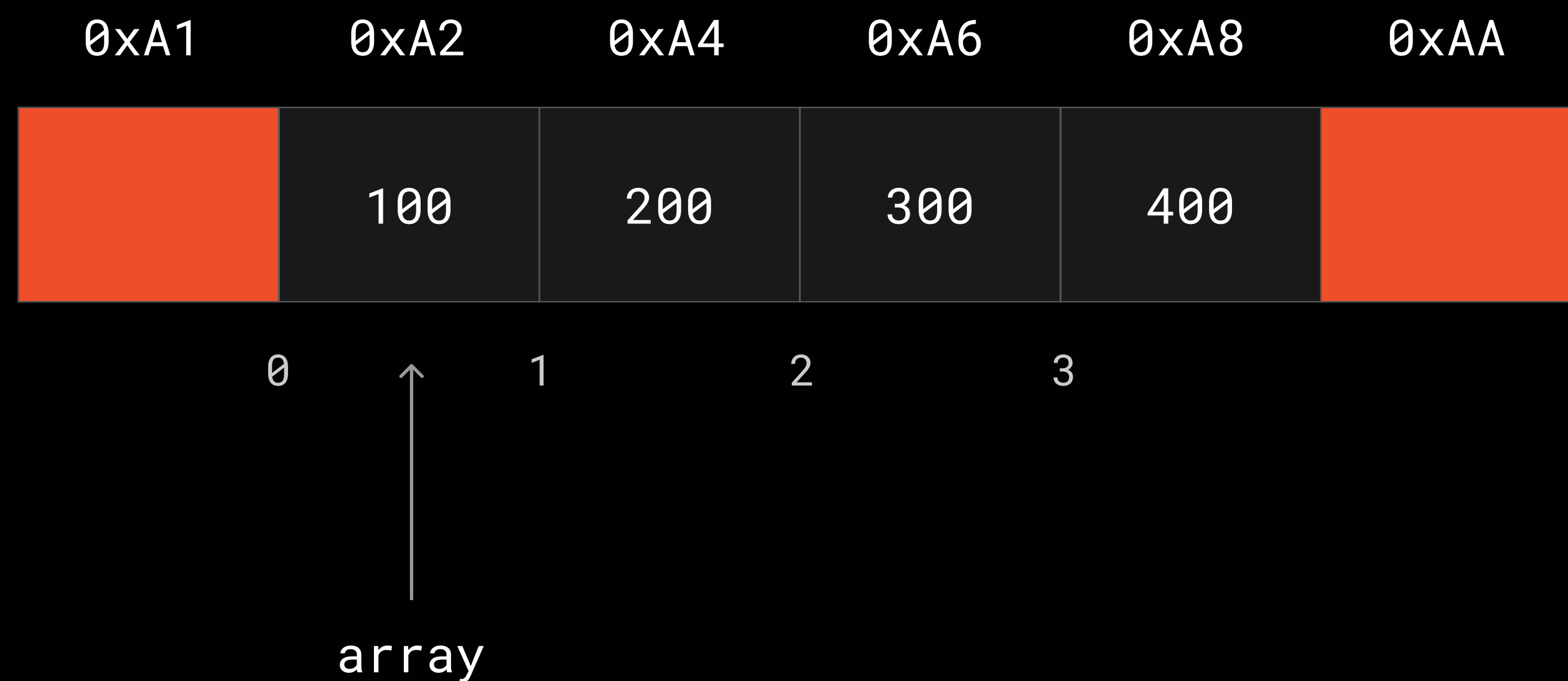


EXAMPLE

Access by index

МАССИВ В GO

Его длина, как и размер элементов является частью типа (не хранятся рядом с данными)



EXAMPLE

Arrays with different types

ВЫЗОВ ФУНКЦИИ LEN() ДЛЯ МАССИВА ВЫЧИСЛЯЕТСЯ НА ЭТАПЕ КОМПИЛЯЦИИ

А не в рантайме



```
1 data := [5]int{ 1, 2, 3, 4, 5 }  
2 length := len(data) // => 5
```

МАССИВ В GO

Не имеет постоянного расположения в памяти
на все время жизни программы (может
переезжать, когда растёт стек горутины)

EXAMPLE

Array moving

КАК ИТЕРИРОВАТЬСЯ ПО МАССИВУ?

OFF BY ONE ERROR

Ошибка на единицу или ошибка неучтённой единицы – **логическая ошибка в алгоритме**, включающая в частности дискретный вариант нарушения граничных условий



```
1 for i := 0; i < len(data); i++ {  
2     // ...  
3 }
```



```
1 for _, value := range data {  
2     ...  
3 }  
4  
5 for idx, _ := range data {  
6     ...  
7 }  
8  
9 for idx := range data {  
10     ...  
11 }  
12  
13 for _, _ = range data {  
14     ...  
15 }  
16  
17 for range data {  
18     ...  
19 }
```

EXAMPLE

Incorrect range by array

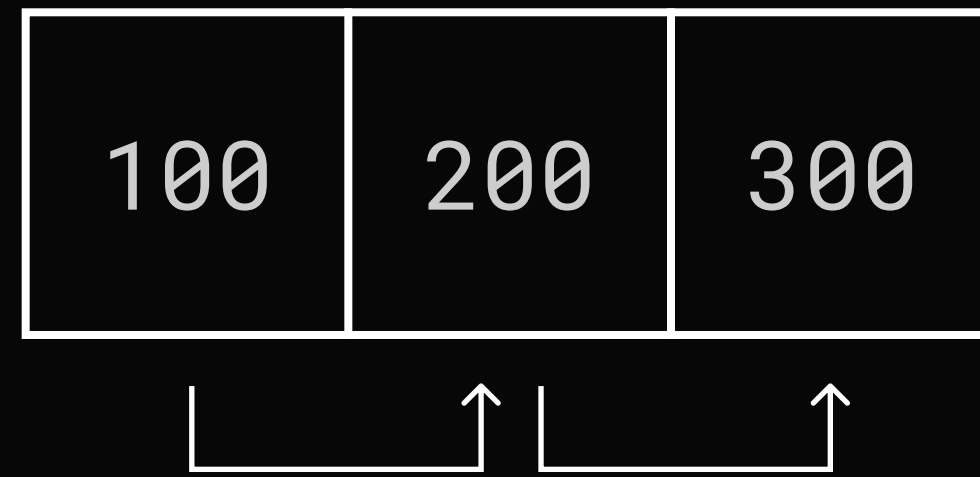
Начиная с Go 1.22 – уникальный экземпляр будет создан **для каждой итерационной переменной в каждой итерации цикла**

**КАК ПРАВИЛЬНО
МОДИФИЦИРОВАТЬ
ЭЛЕМЕНТЫ МАССИВА
ВО ВРЕМЯ ИТЕРАЦИИ?**

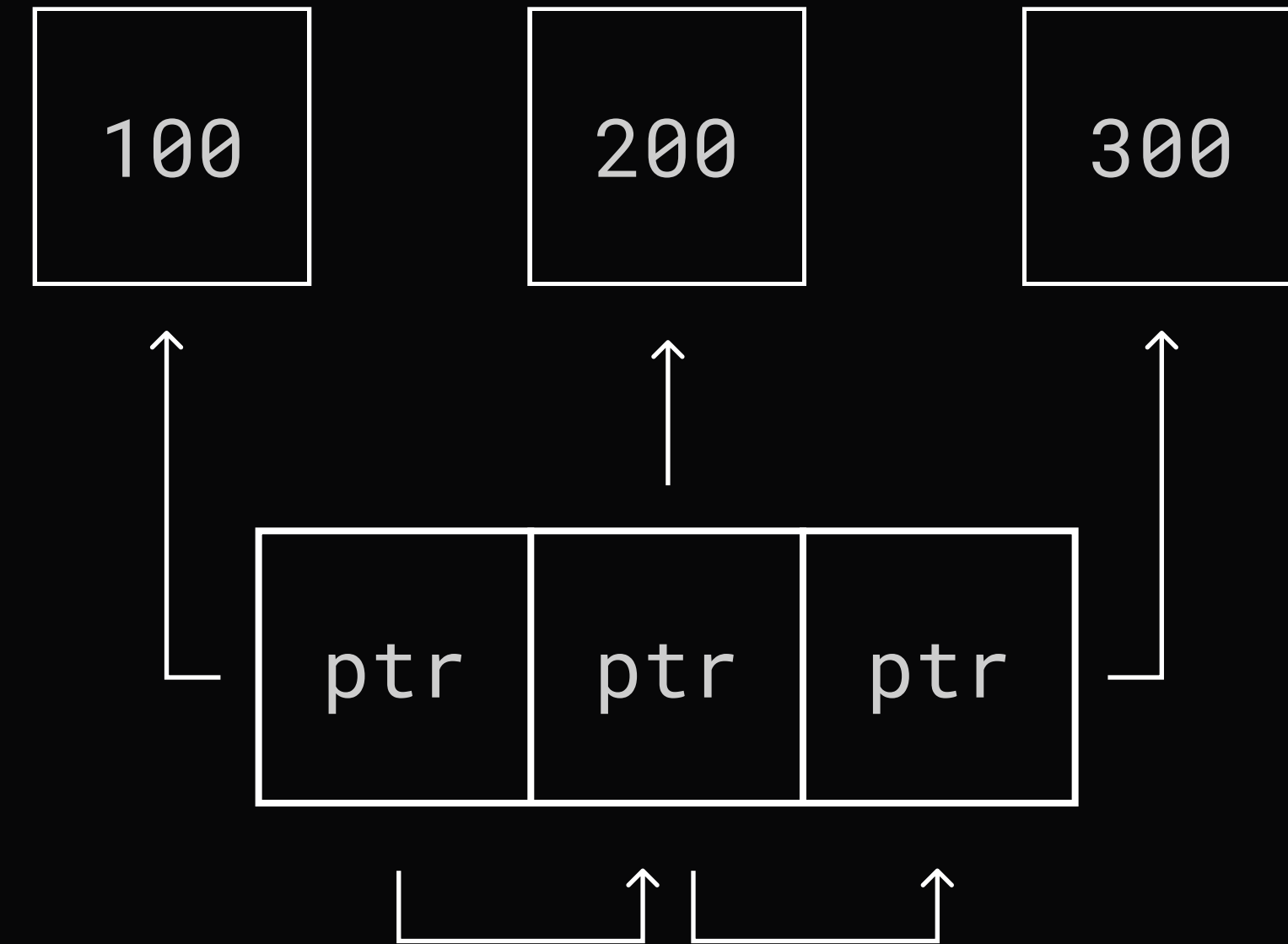
EXAMPLE

Range by array comparison

BenchmarkWithIndices



BenchmarkWithPointers



**ГДЕ АЛЛОЦИРУЮТСЯ
МАССИВЫ?**

GO АЛЛОЦИРУЕТ МАССИВ В СТЕКЕ, ЕСЛИ ЕГО РАЗМЕР МЕНЬШЕ $\leq$ 10 МБ

но не забываем про escape analysis



```
1 var array [N]byte
```

EXAMPLE

Arrays allocation 1

EXAMPLE

Arrays allocation 2

**ЧТО ПРОИСХОДИТ, КОГДА МАССИВ
ПЕРЕДАЕТСЯ В ФУНКЦИЮ?**



```
1 func print(temp [6]int) {  
2     ...  
3 }  
4  
5 func main() {  
6     var data = [6]int{ 1, 2, 3, 4, 5, 6 }  
7     print(data)  
8 }
```

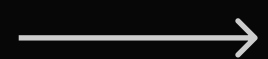
data

array



temp

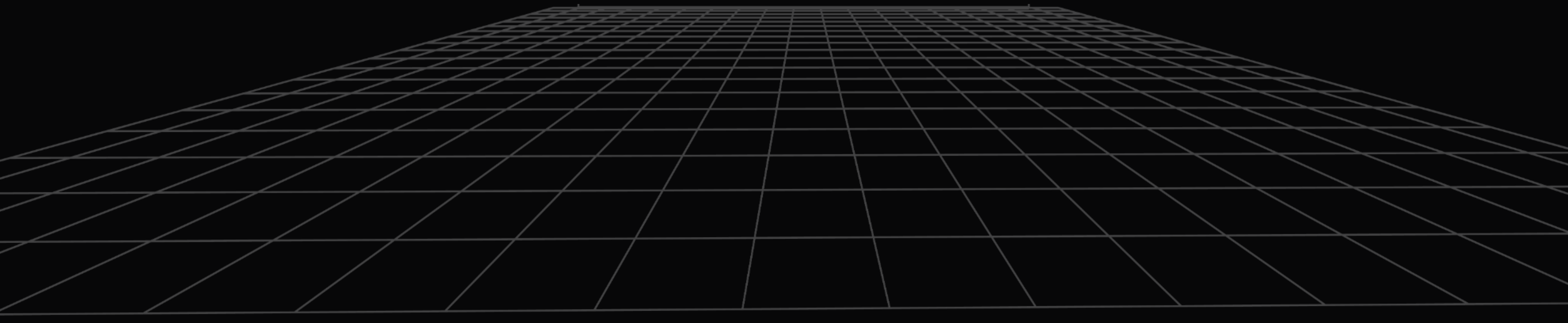
array



В случае необходимости можно использовать
указатели на массив или срезы,
чтобы избавиться от копирования

FAQ

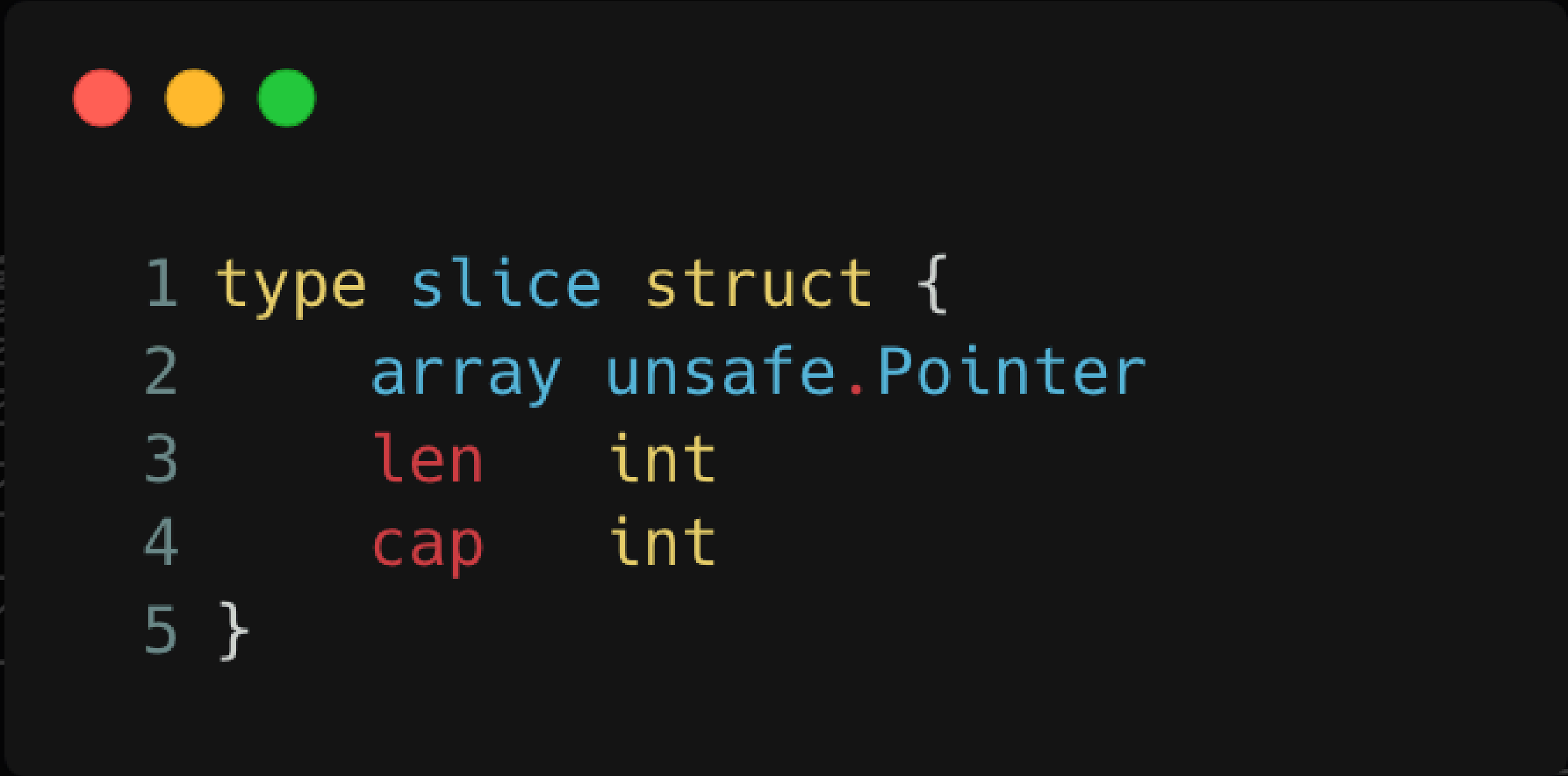
Массивы



СРЕЗЫ

В GO ЗА СРЕЗОМ СТОИТ МАССИВ

это означает, что данные среза
хранятся в структуре данных массив



```
1 type slice struct {  
2     array unsafe.Pointer  
3     len    int  
4     cap    int  
5 }
```

Как ни странно, хоть срез и структура,
но `zero value` для среза `nil`

СОЗДАНИЕ СРЕЗА



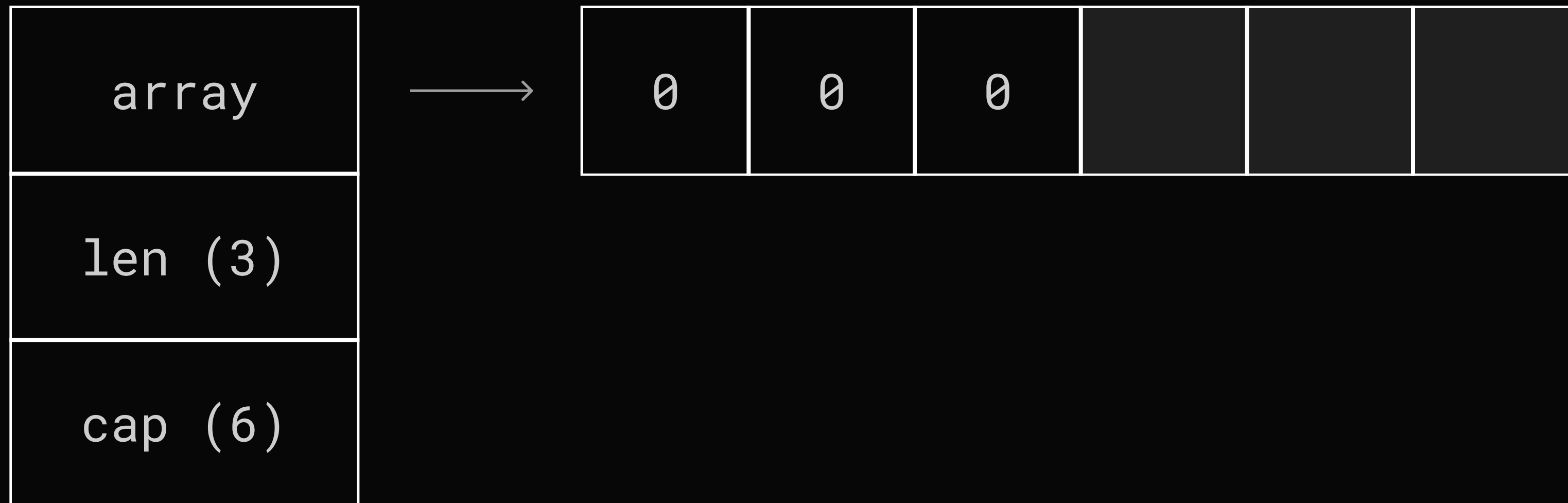
```
1 data = []int{}           // []           len=0 cap=0
2 data = []int{1, 2}       // [1 2]       len=2 cap=2
3 data = []int{5: 10}      // [0 0 0 0 0 10] len=6 cap=6
4 data = make([]int, 6)    // [0 0 0 0 0 0]  len=6 cap=6
5 data = make([]int, 3, 6) // [0 0 0 0 0 0]  len=3 cap=6
```

API практически точно такой же,
как и у массивов

ОСТАЛЬНЫЕ ЭЛЕМЕНТЫ ЗАПОЛНЯЮТСЯ НУЛЯМИ



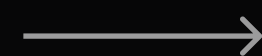
```
1 data := make([]int, 3, 6)
```





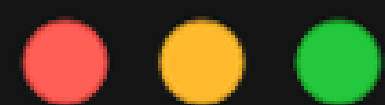
```
1 data = append(data, 2)
```

array
len (4)
cap (6)



0	0	0	2		
---	---	---	---	--	--

ДОБАВЛЕНИЕ В КОНЕЦ



```
1 // data => [1,2,3,4]
2 data = append(data)           // [1,2,3,4]
3 data = append(data, 5)        // [1,2,3,4,5]
4 data = append(data, 5, 6, 7)   // [1,2,3,4,5,6,7]
5 data = append(data, []int{ 5, 6 }...) // [1,2,3,4,5,6]
```

EXAMPLE

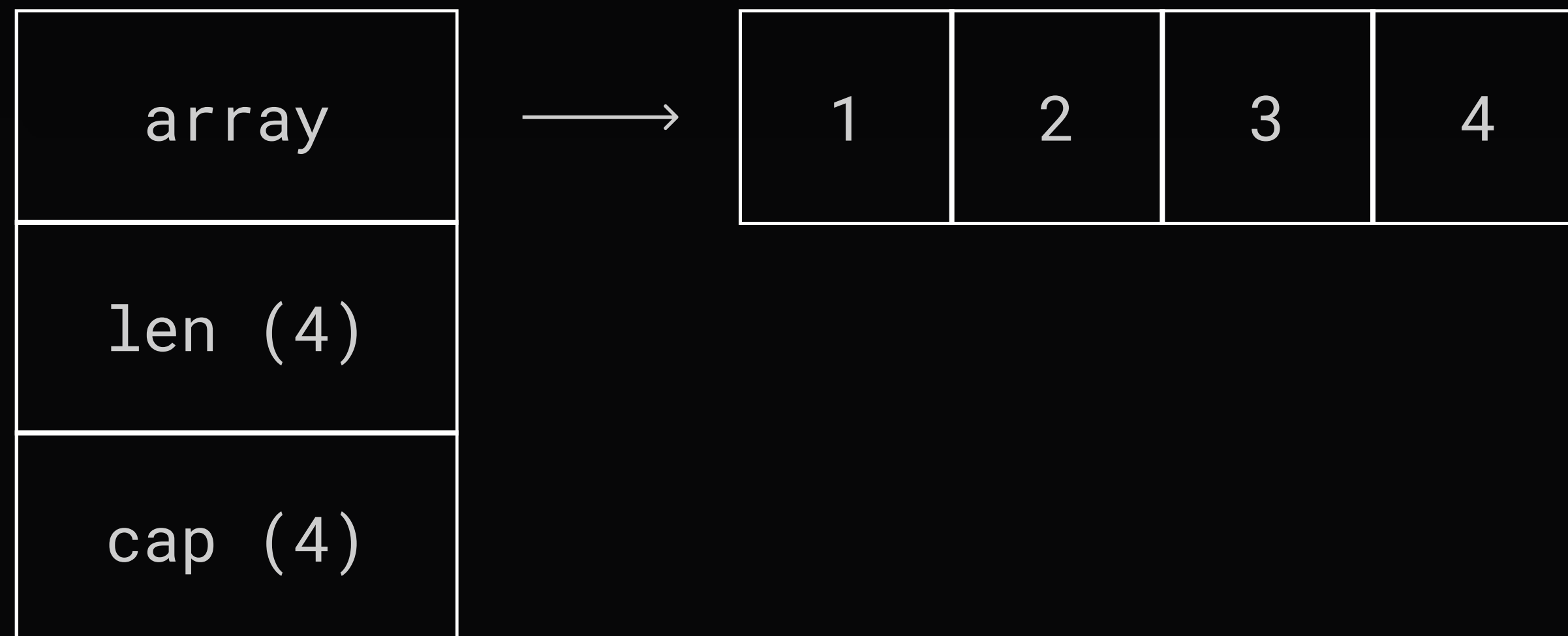
Slice operations

Читать из нулевого среза?	Писать в нулевой срез?	Добавлять в нулевой срез?	Итерироваться по нулевому срезу?
Паника	Паника	Окей	Окей

ЧТО ДЕЛАТЬ, КОГДА ЗАКАНЧИВАЕТСЯ ЕМКОСТЬ?



```
1 data := []int{ 1, 2, 3, 4 }  
2 // data = append(data, 5)
```



АЛГОРИТМ

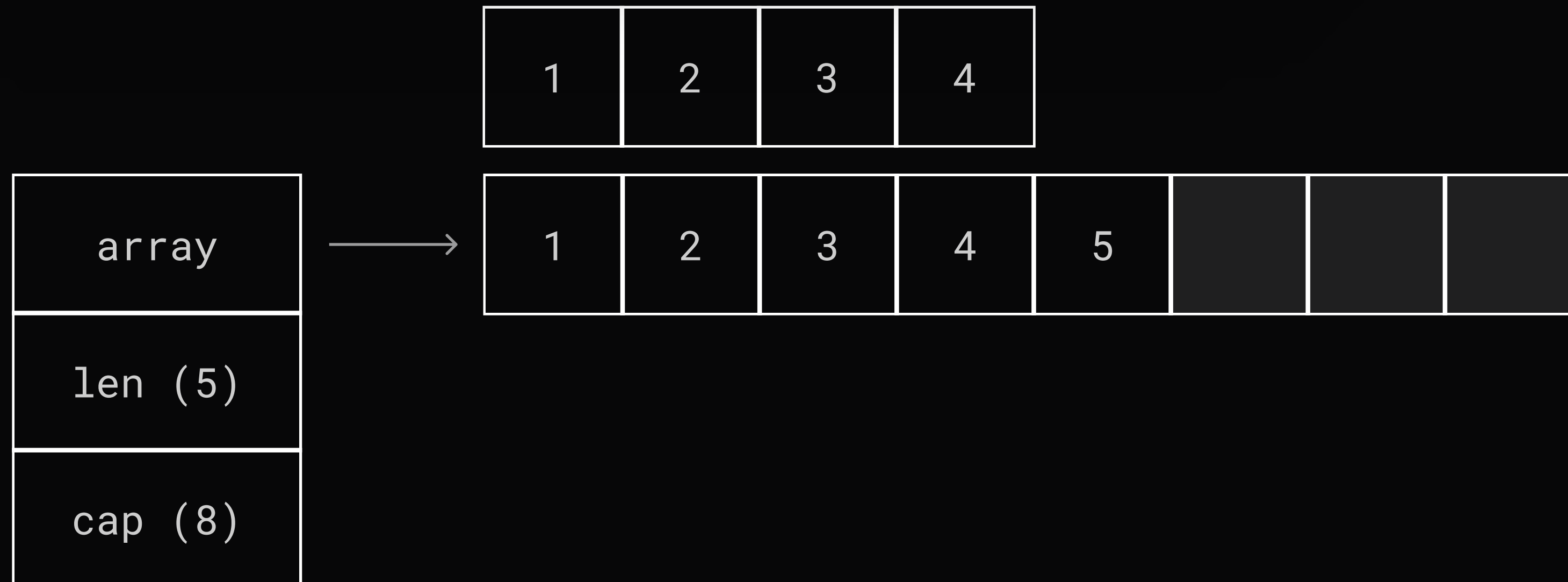
1. Выделяем больше памяти

2. Копируем элементы

3. Добавляем элемент



```
1 data := []int{ 1, 2, 3, 4 }  
2 data = append(data, 5)
```



**КАКОГО РАЗМЕРА ВЫДЕЛЯТЬ
НОВЫЙ МАССИВ?**

Раньше



- 1 Transition from growing 2x for small slices
- 2 to growing 1.25x for large slices. This formula
- 3 gives a smooth-ish transition between the two.

Сейчас



- 1 Instead of growing 2x for < 1024 elements and 1.25x for ≥ 1024 elements,
- 2 use a somewhat smoother formula for the growth factor. Start reducing
- 3 the growth factor after 256 elements, but slowly.

4

5 starting cap	growth factor
6 256	2.0
7 512	1.63
8 1024	1.44
9 2048	1.35
10 4096	1.30

**Емкость среза не уменьшается,
если большая ее часть не используется!**

Terminal: deep_go × + ∨



СОВЕТ

Резервируйте заранее емкость массива, если вам известно его будущее количество элементов

EXAMPLE

Reservation

EXAMPLE

Dirty slice

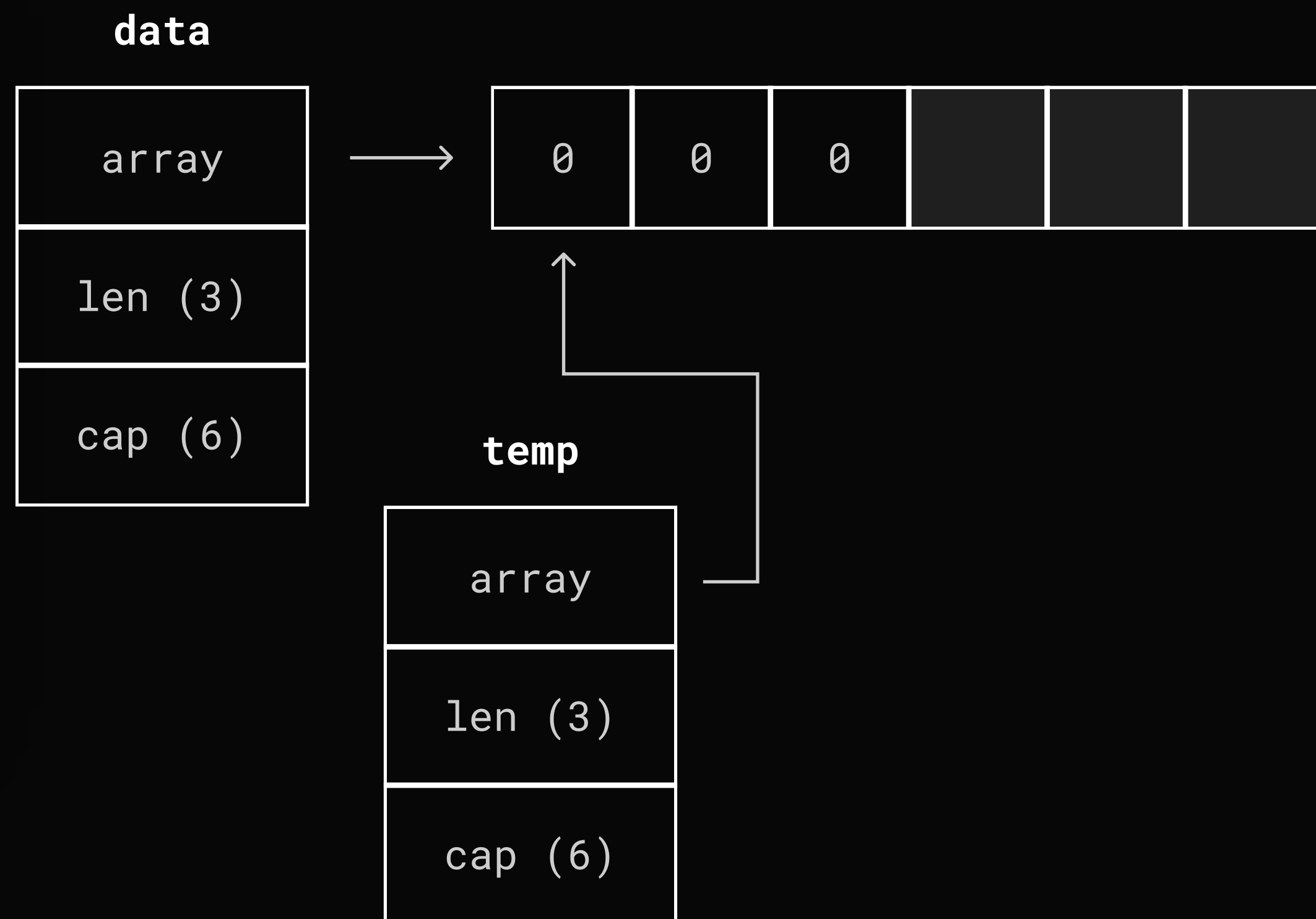
EXAMPLE

Slice type transformation

**ЧТО ПРОИСХОДИТ, КОГДА СРЕЗ
ПЕРЕДАЕТСЯ В ФУНКЦИЮ?**



```
1 func print(temp []int) {  
2     ...  
3 }  
4  
5 func main() {  
6     data := make([]int, 3, 6)  
7     print(data)  
8 }
```

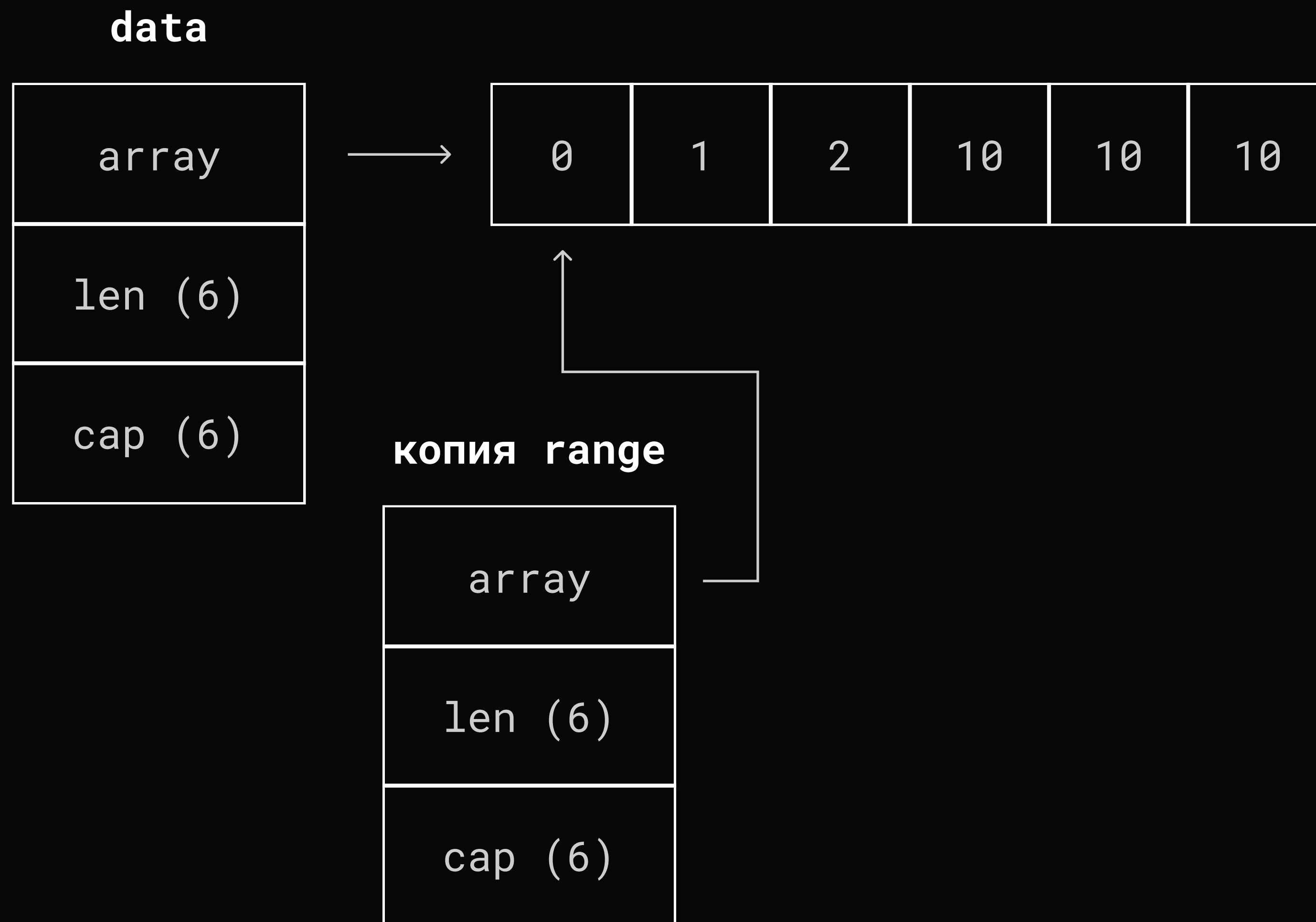


КАК ИТЕРИРОВАТЬСЯ ПО СРЕЗАМ?

EXAMPLE

Endless range

При использовании `range`, указываемое выражение
вычисляется только один раз перед началом цикла!



Предоставленное выражение копируется во временную переменную,
а затем range выполняет итерации над этой переменной

EXAMPLE

Different range by array

**Нужно быть осторожным, когда итерируетесь
с использованием range по большим массивам (не срезам)**

EXAMPLE

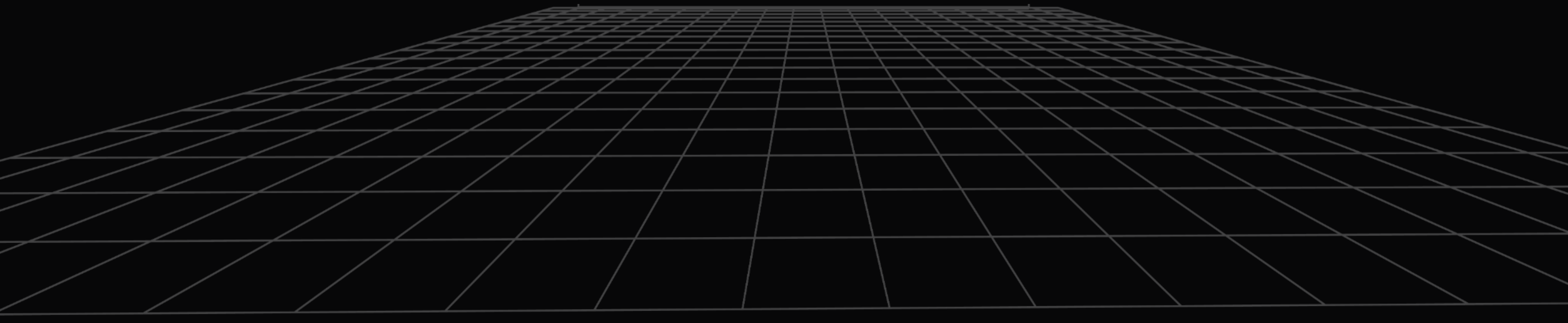
Iteration optimization

EXAMPLE

Range task

FAQ

Срезы

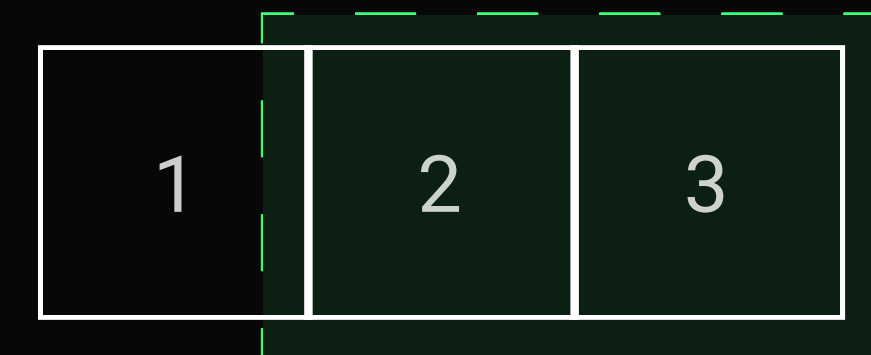
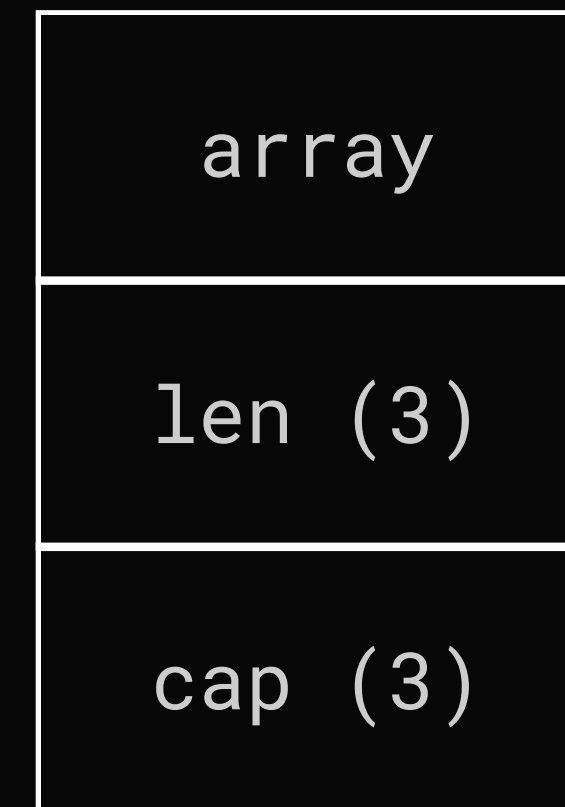


ПОЛУЧЕНИЕ ПОДСРЕЗОВ (НАРЕЗКА СРЕЗОВ)

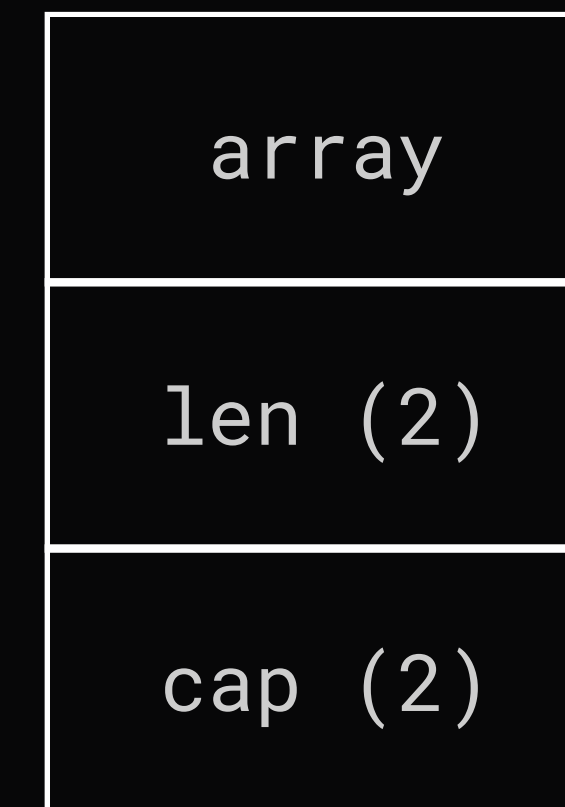


```
1 data := []int { 1, 2, 3 }  
2 temp := data[1:3]
```

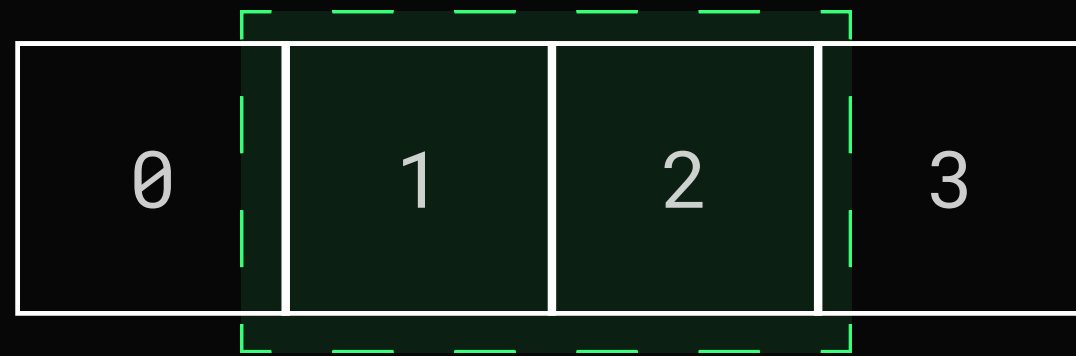
data



temp = data[1:3]

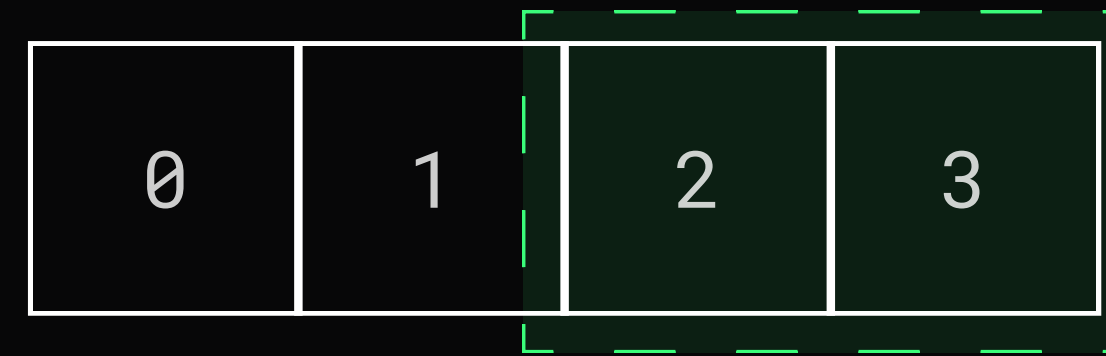


```
temp = data[1:3]
```



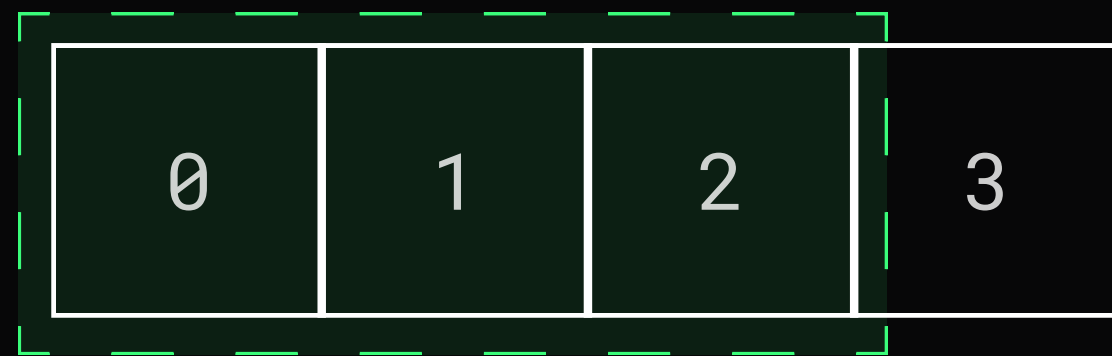
temp

```
temp = data[2:]
```



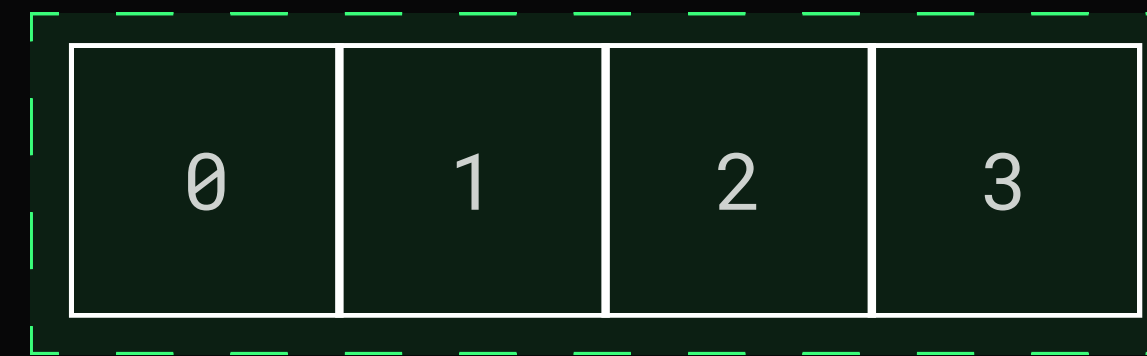
temp

```
temp = data[:3]
```



temp

```
temp = data[:] // => temp = data
```



temp

Получается полуинтервал $[x; y)$, открытый слева
и замкнутый справа (как в Python нельзя
использовать отрицательные числа)

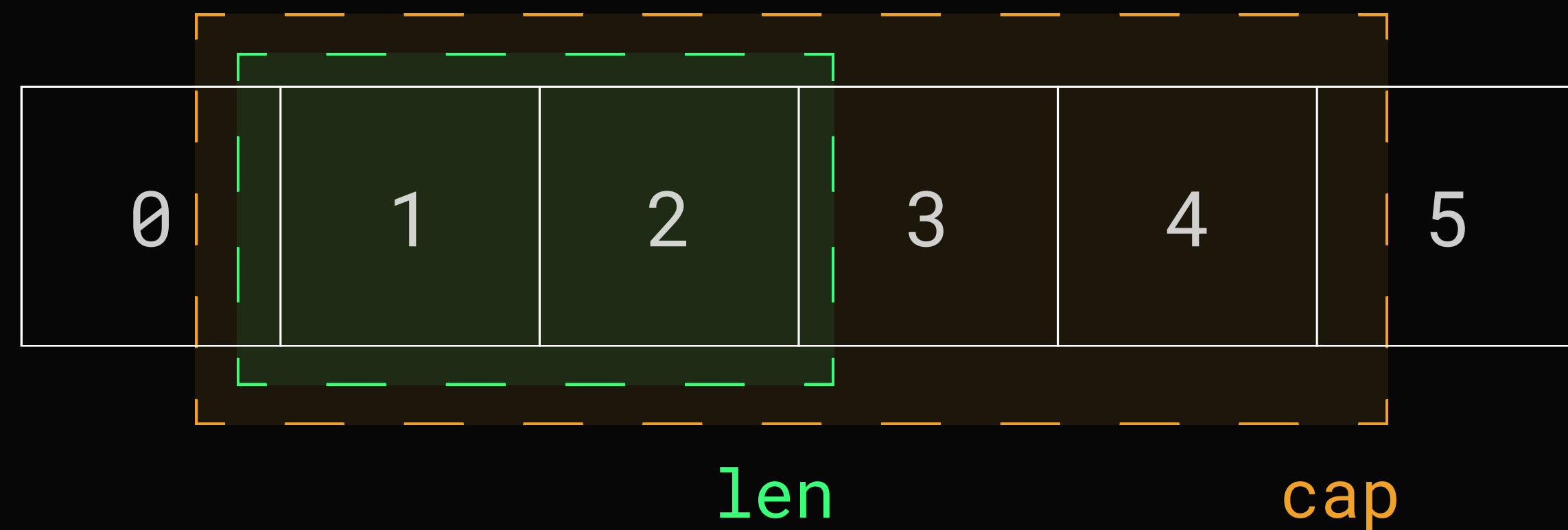
EXAMPLE

Slicing operations

Длина производного среза может быть больше,
чем у базового среза, но емкость никогда не будет
больше, чем у базового среза

ИЗМЕНЕНИЕ ЕМКОСТИ

```
temp = data[1:3:4]
```

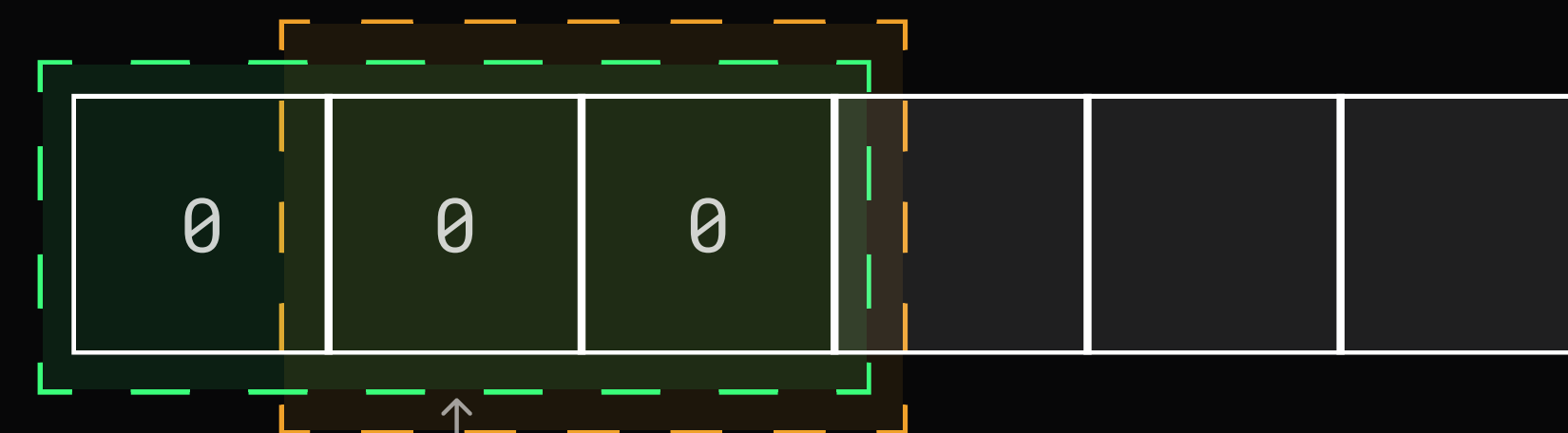
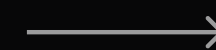




```
1 data1 := make([]int, 3, 6)
2 data2 := data1[1:3]
```

data1

array
len (3)
cap (6)



data2

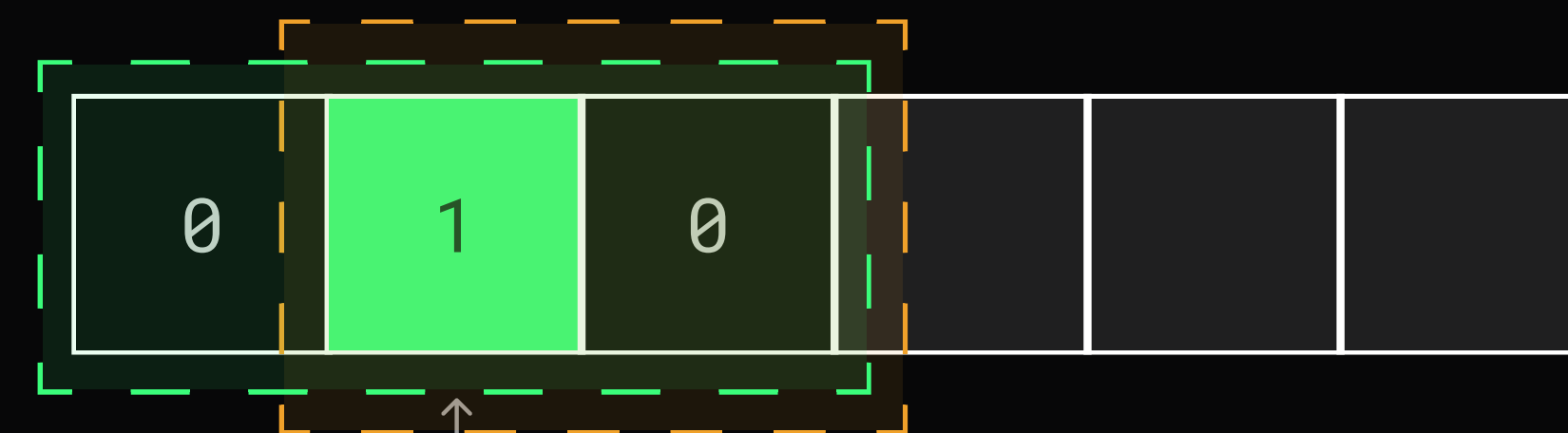
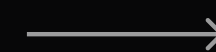
array
len (2)
cap (5)



```
1 data1[1] = 1
```

data1

array
len (3)
cap (6)

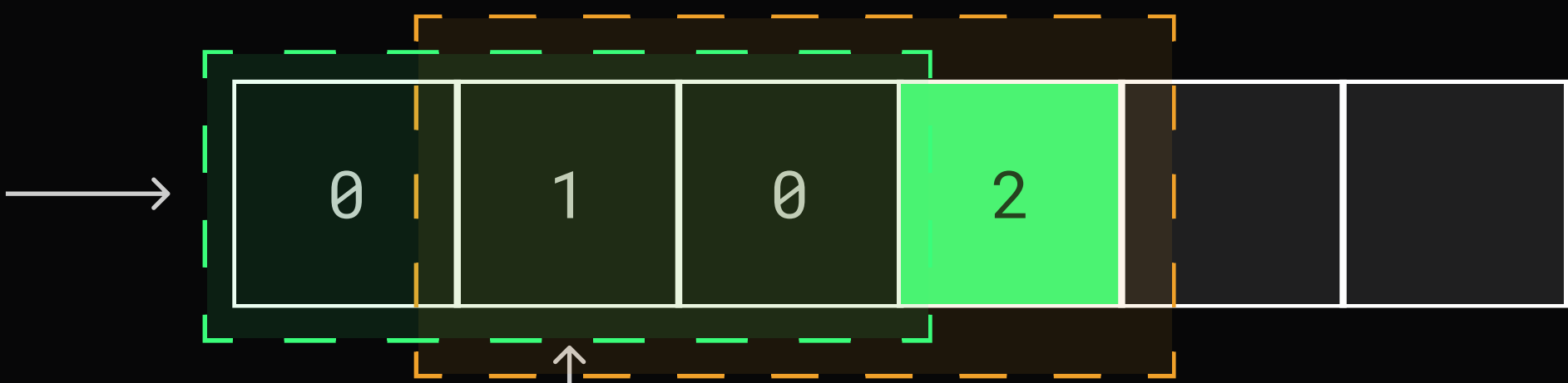


data2

array
len (2)
cap (5)

```
1 data2 = append(data2, 2)
```

data1		
array		
len (3)		
cap (6)		



data2		
array		
len (3)		
cap (5)		

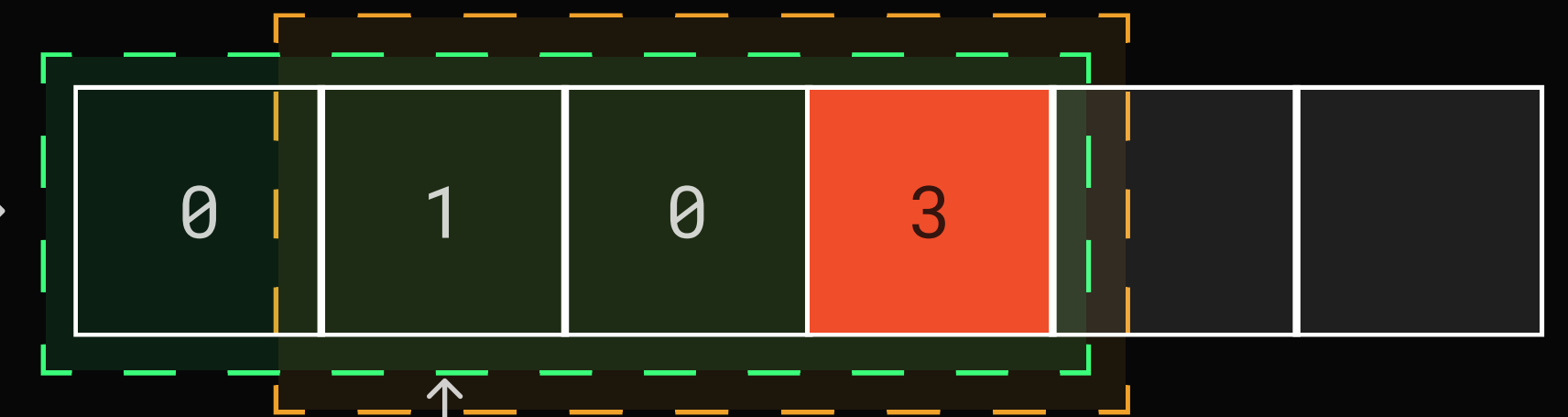
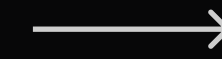
data1 = [0,1,0]
data2 = [1,0,2]



```
1 data1 = append(data1, 3)
```

data1

array
len (4)
cap (6)



data2

array
len (3)
cap (5)

data1 = [0,1,0,3]

data2 = [1,0,3]

При изменении длины/емкости одного среза,
изменения никак не отражаются на другом срезе!

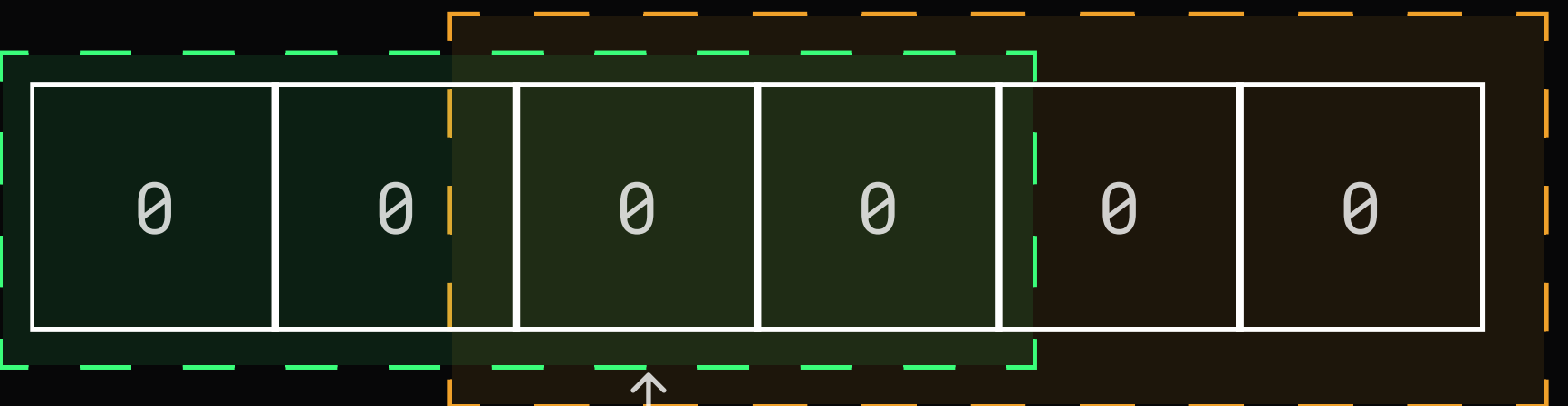
ЧТО ПРОИЗОЙДЕТ В ЭТОЙ СИТУАЦИИ?



```
1 data1 := make([]int, 4, 6)
2 data2 := data1[2:6]
3 // data2 = append(data2, 1)
```

data1

array
len (4)
cap (6)



data2

array
len (4)
cap (4)

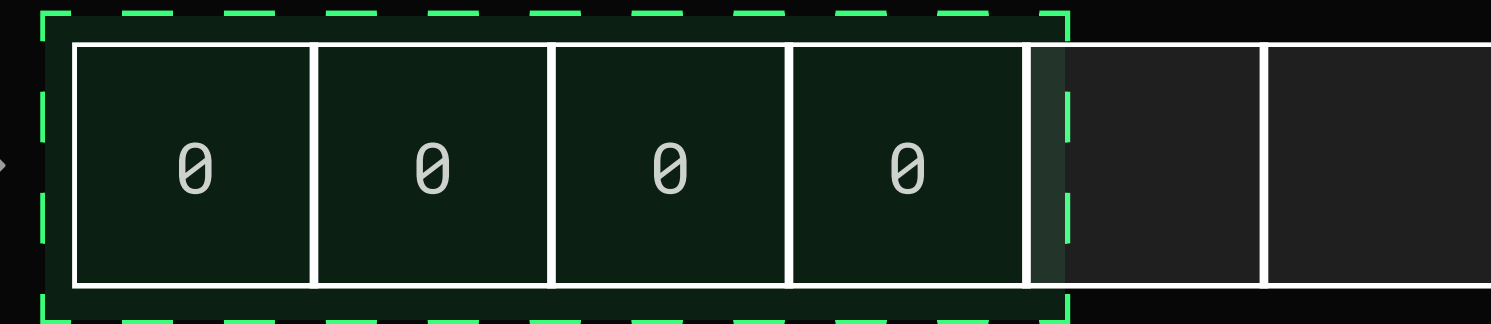
НОВАЯ ЕМКОСТЬ МАССИВА СТАЛА РАВНА 8, А НЕ 12



```
1 data1 := make([]int, 4, 6)
2 data2 := data1[2:6]
3 data2 = append(data2, 1)
```

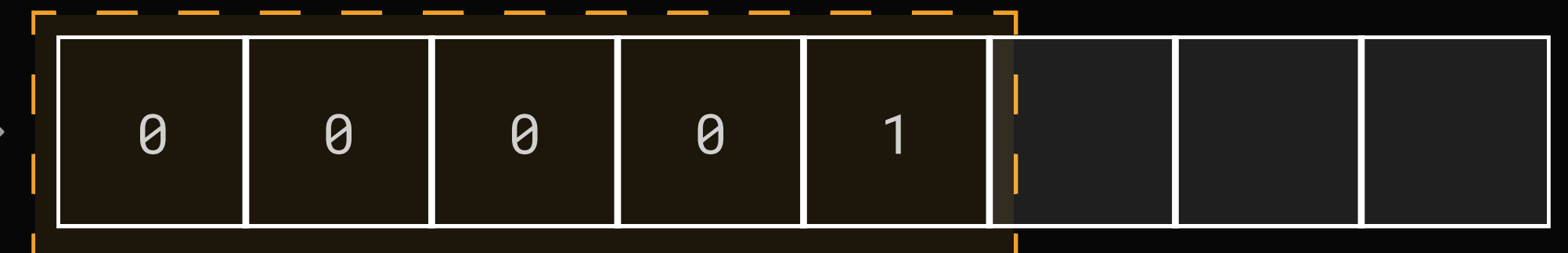
data1

array
len (4)
cap (6)



data2

array
len (5)
cap (8)

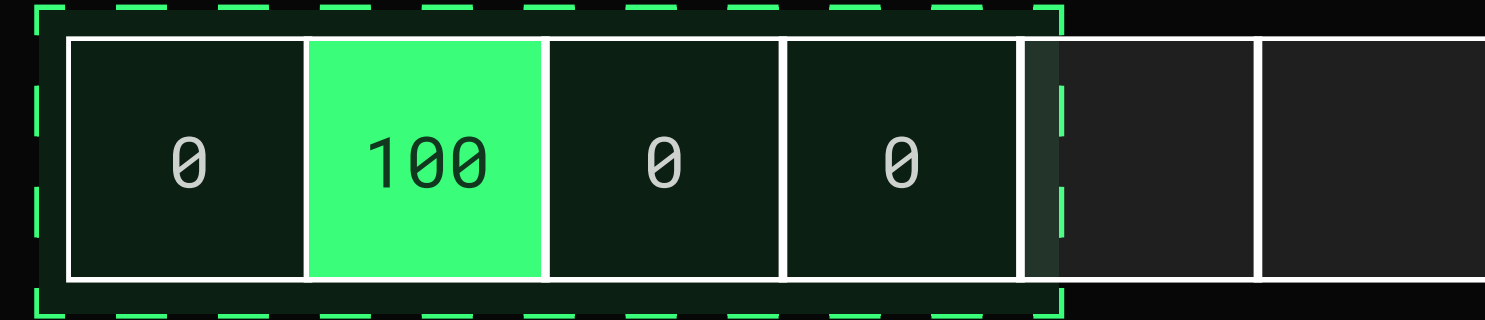
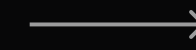




```
1 data1 := make([]int, 4, 6)
2 data2 := data1[2:6]
3 data2 = append(data2, 1)
4 data1[1] = 100
```

data1

array
len (4)
cap (6)



data2

array
len (5)
cap (8)



**ЧТО БУДЕТ, ЕСЛИ ПОЛУЧИТЬ
СРЕЗ ИЗ МАССИВА?**



```
1 var array = [...]int{ 1, 2, 3, 4 }  
2 slice := array[1:3]
```

array



slice



**Можно сделать обратное преобразование и получить
указатель на массив или массив из среза**

EXAMPLE

Slices to arrays

EXAMPLE

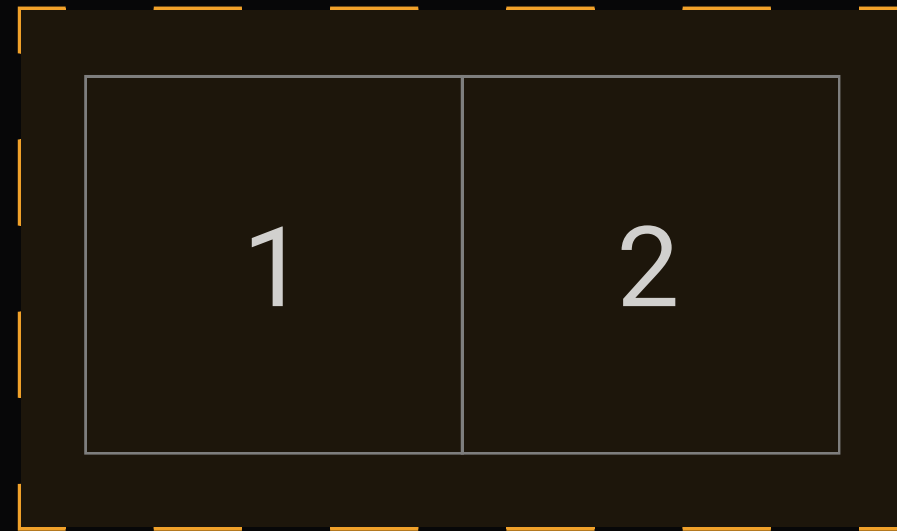
Slices to array pointer

EXAMPLE

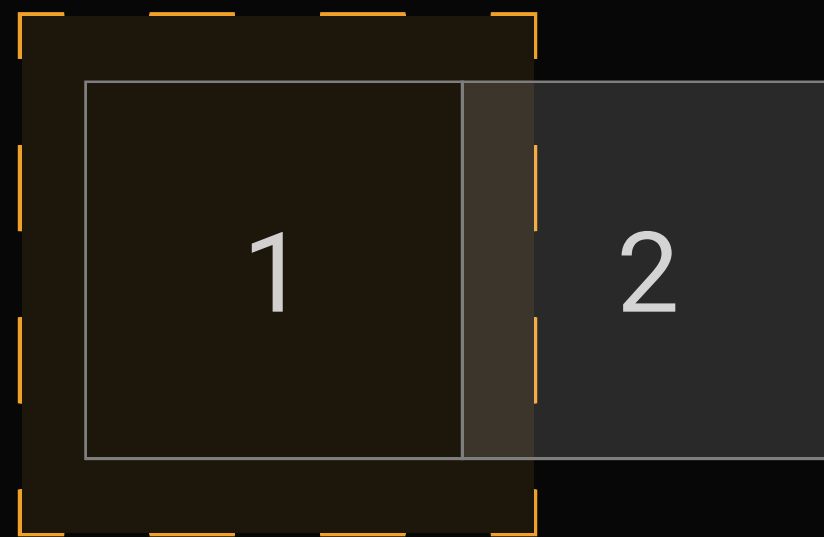
Iteration by pointer to array

**КАК УДАЛИТЬ ЭЛЕМЕНТ
ИЗ КОНЦА СРЕЗА?**

1.



2.



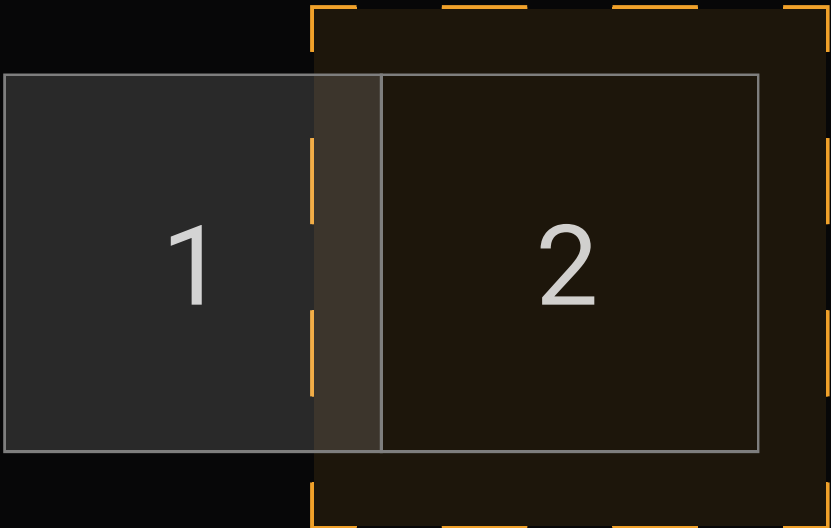
```
1 data := []int{ 1, 2 }  
2 data = data[:len(data) - 1]
```

**КАК УДАЛИТЬ ЭЛЕМЕНТ
ИЗ НАЧАЛА СРЕЗА?**

1.

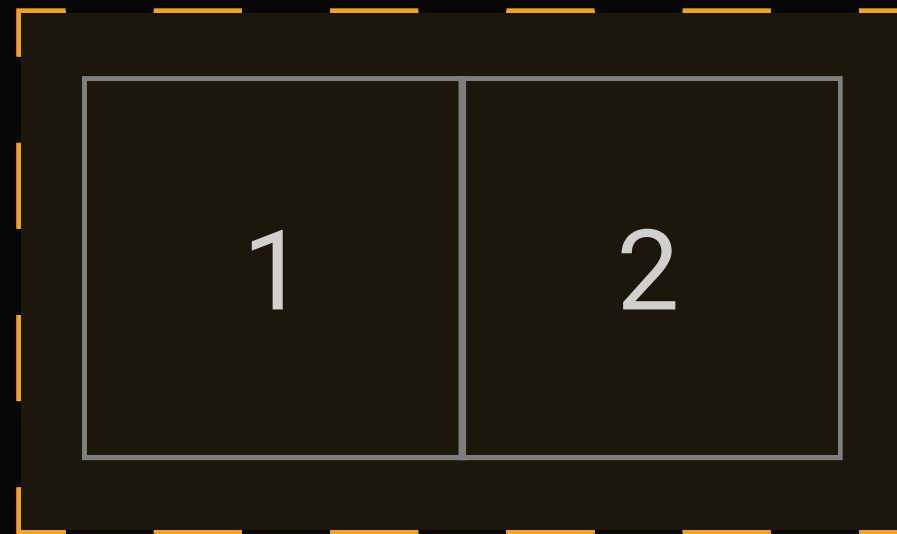


2.

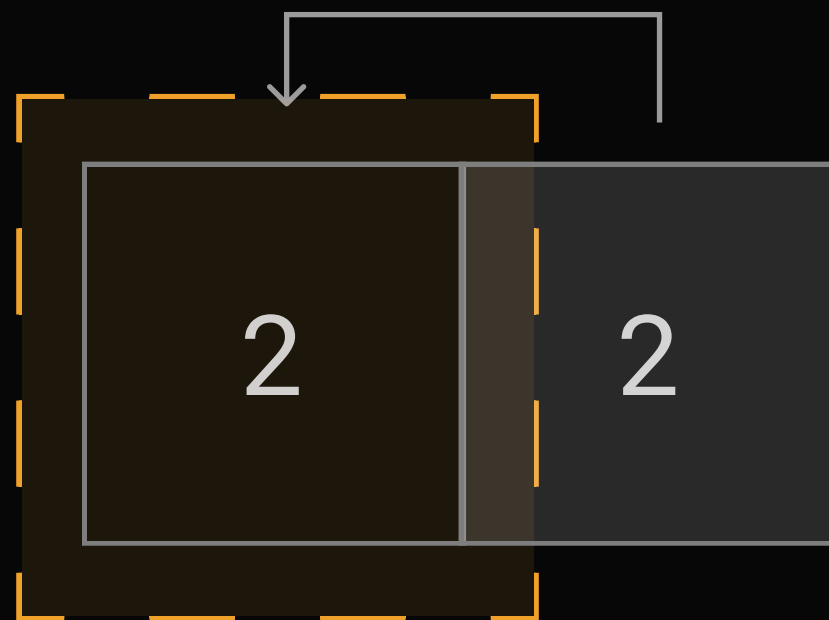


```
1 data := []int{ 1, 2 }  
2 data = data[1:]
```

1.



2.



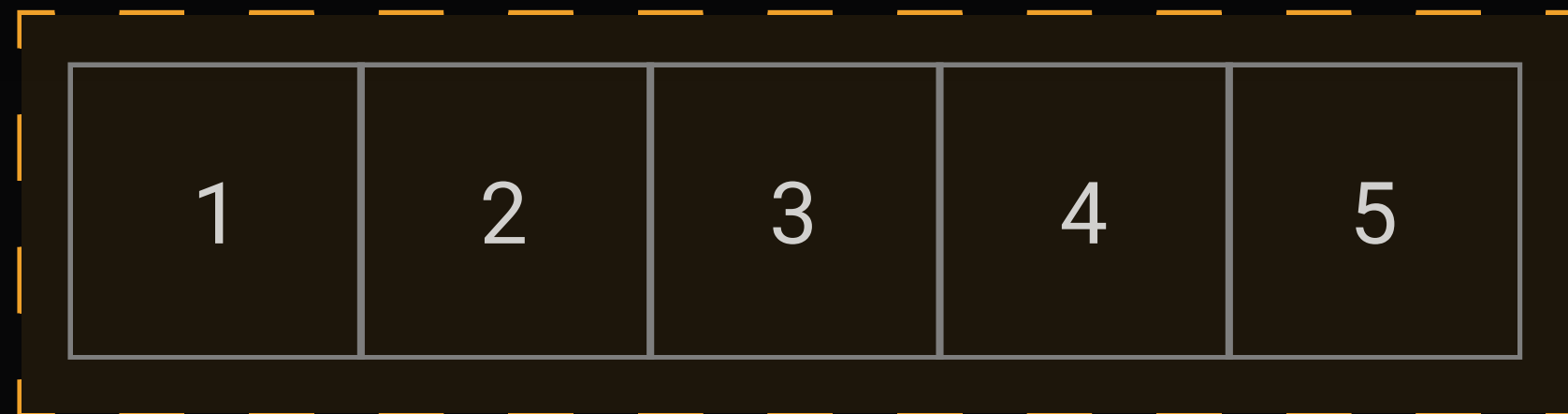
```
1 data := []int{1, 2}
2 data = append(data[:0], data[1:]...)
```

Таким способом можно удалить и из середины среза



```
1 data := []int{1, 2, 3, 4, 5}
2 data = append(data[:2], data[3:]...)
```

1.



2.



Terminal: deep_go × + ∨



СОВЕТ

Лучше стараться избегать удалений
из начала/середины срезов

МОЖНО ПИСАТЬ БОЛЕЕ ПОНЯТНЫЙ КОД



```
1 data := []int{0, 1, 2, 3, 4, 5}
2 fmt.Println(data, len(data), cap(data)) // [0 1 2 3 4 5] 6 6
3 data = slices.Delete(data, 1, 3)
4 fmt.Println(data, len(data), cap(data)) // [0 3 4 5] 4 6
```

КАК ЖИТЬ В GO БЕЗ СТЕКОВ И ОЧЕРЕДЕЙ?



```
1 // Pop front
2 data, front = data[1:], data[0]
3 // Pop back
4 data, back = data[:len(data)-1], data[len(data)-1]
5 // Push front
6 data = append([]int{ value }, data...)
7 // Push back
8 data = append(data, value)
```

**КАК УВЕЛИЧИТЬ
РАЗМЕР СРЕЗА?**

EXAMPLE

Resize slice

**КАК РЕАЛЛОЦИРОВАТЬ
ЕМКОСТЬ СРЕЗА?**

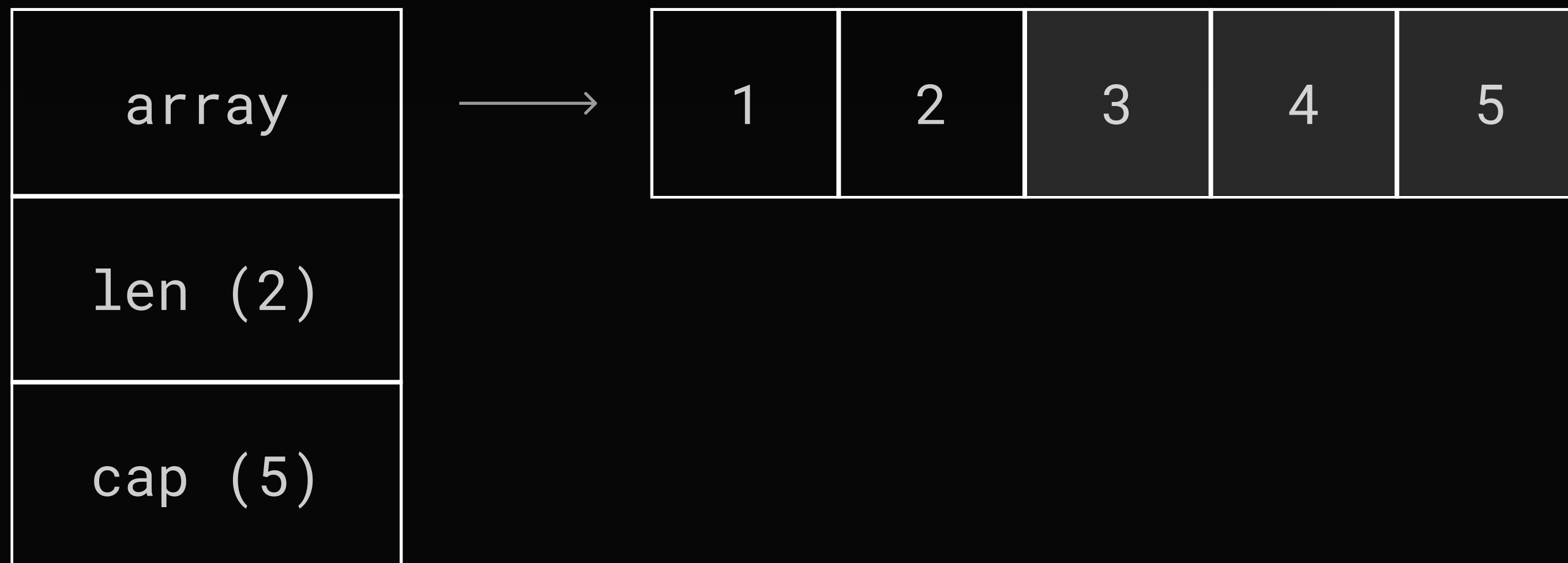
EXAMPLE

Clip

Поскольку меньший срез использует тот же массив,
сборщик мусора не удалит массив или часть массива



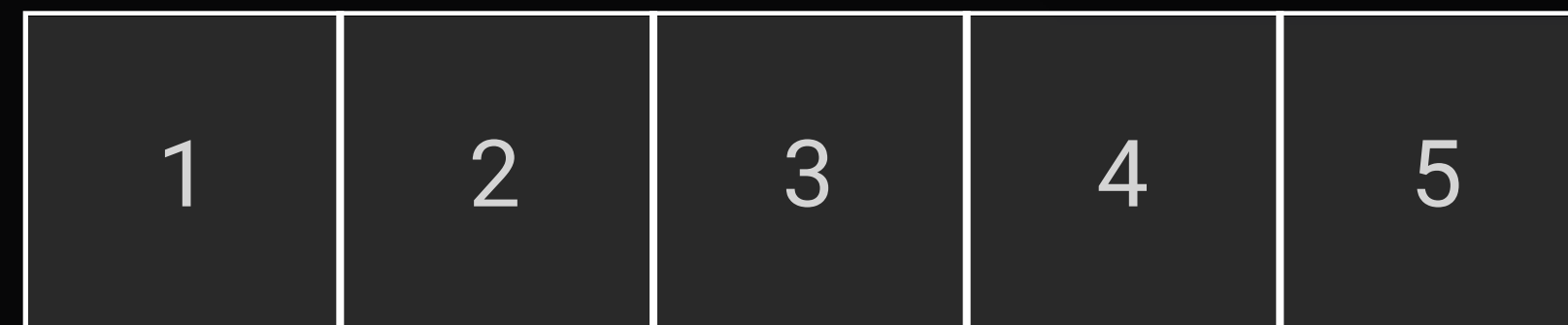
```
1 data := []int{ 1, 2, 3, 4, 5 }  
2 data = data[:2]
```



Придется перевыделять память
и копировать данные,
но работать будет!



```
1 data := []int{1, 2, 3, 4, 5}
2 data = append([]int(nil), data[:2]...)
```

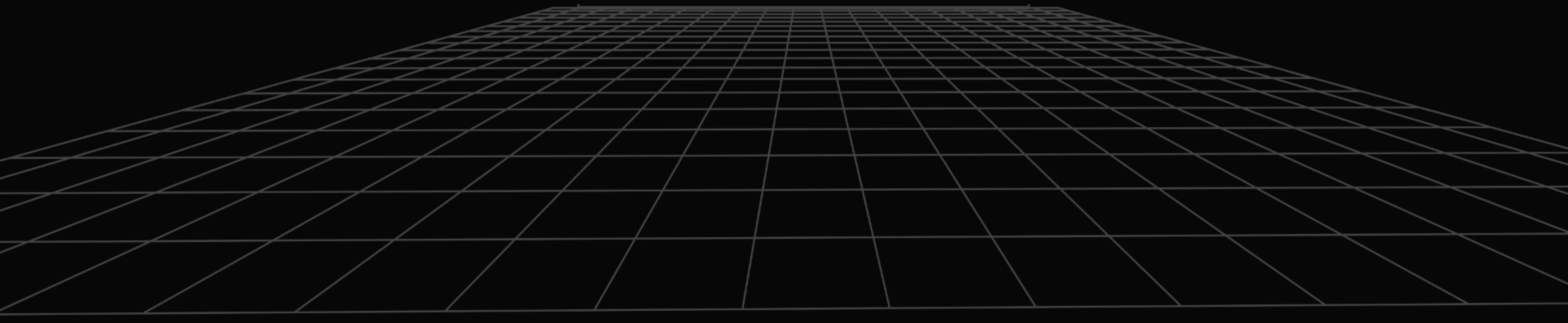


EXAMPLE

Slicing task

FAQ

Получение подсезов



ПЕРЕРЫВ 5 МИНУТ

ТОНКОСТИ СРЕЗОВ

GO АЛЛОЦИРУЕТ МАССИВ СРЕЗА В СТЕКЕ, ЕСЛИ ЕГО РАЗМЕР МЕНЬШЕ $\leq$ 64 КБ

но не забываем про escape analysis



```
1 var slice = make([]byte, N)
```

EXAMPLE

Slices allocations

Когда происходит реаллокация,
новый массив аллоцируется в куче

EXAMPLE

Slice increasing

**КАК ПРОАЛЛОЦИРОВАТЬ МАССИВ
СРЕЗА БОЛЬШЕ, ЧЕМ 64КБ В СТЕКЕ?**

МОЖНО АЛЛОЦИРОВАТЬ МАССИВ НА СТЕКЕ

который будет не более 10 МБ,
чтобы затем получить его срез



```
1 var array [N]byte
2 var slice = array[:]
```

Почему у `io.Reader` такая сигнатура?



```
1 func (T) Read(b []byte) (n int, err error)
```

А не такая?



```
1 func (T) Read(n int) (b []byte, err error)
```

EXAMPLE

Reader comparison

КАК ОЧИСТИТЬ СРЕЗ?

EXAMPLE

Slice clearing

EXAMPLE

Memclr

**В ЧЕМ РАЗНИЦА
ПУСТЫХ И НУЛЕВЫХ СРЕЗОВ?**

ПУСТЫЕ И НУЛЕВЫЕ СРЕЗЫ

- Срез считается **пустым**, если его длина равна **0**
- Срез считается **нулевым**, если его значение равно **nil**

EXAMPLE

Empty and nil slices

**ЗАЧЕМ РАЗЛИЧАТЬ
ПУСТЫЕ И НУЛЕВЫЕ СРЕЗЫ?**

EXAMPLE

Check length

Terminal: deep_go × + ∨



СОВЕТ

Проверяйте длину среза – это лучший способ
для проверки пустых и нулевых срезов



```
1 var data1 []float32
2 var customer1 := customer {
3     ID: 100,
4     Operations: data1,
5 }
6
7 var data2 := make([]float32, 0)
8 var customer2 := customer {
9     ID: 200,
10    Operations: data2,
11 }
```



```
1 {"ID":100,"Operations":null}
2 {"ID":200,"Operations":[]}
```

МОЖНО ЛИ СРАВНИВАТЬ СРЕЗЫ?

EXAMPLE

Slices comparison

EXAMPLE

Slices comparison recursive

ОСОБЕННОСТИ

- `reflect.DeepEqual()` замечает различие между пустыми и нулевыми срезами
- `reflect.DeepEqual()` лучше использовать в контексте тестирования из-за ее скорости работы

EXAMPLE

Slices comparison effective

КАК СКОПИРОВАТЬ ДАННЫЕ СРЕЗА? (DEEP COPY)

EXAMPLE

Copy slice

COPY()

Число элементов, скопированных в другой слайс,
определяется **минимумом между:**

- длиной первого среза
- длиной второго среза

EXAMPLE

Copy implementation

АЛЬТЕРНАТИВНЫЕ СПОСОБЫ



```
1 src := []int{ 0, 1, 2 }  
2 dst := append([]int(nil), src...)
```



```
1 src := []int{ 0, 1, 2 }  
2 dst := slices.Clone(src)
```

ПОЧЕМУ ФУНКЦИЯ `APPEND()`
ВОЗВРАЩАЕТ СРЕЗ?

EXAMPLE

Slice with function 1

EXAMPLE

Slice with function 2

Terminal: deep_go × + ∨



СОВЕТ

Возвращайте срез из функции, если вы добавляете в него элементы — не зря же `append()` возвращает срез



Передавать по указателю — ну такое себе

**КАКИЕ ЕЩЕ МОГУТ БЫТЬ ПРОБЛЕМЫ
С ФУНКЦИЕЙ APPEND()?**

EXAMPLE

Dangerous append

Terminal: deep_go × + ∨



СОВЕТ

Нужно присваивать результат `append()` той же переменной
(лучше написать лишнюю строчку)

КАК РЕАЛИЗОВАТЬ
ФУНКЦИЮ APPEND()?

EXAMPLE

Append implementation

**МОГУТ ЛИ БЫТЬ УТЕЧКИ ПАМЯТИ
ПРИ РАБОТЕ СО СРЕЗАМИ?**

EXAMPLE

Big slice incorrect

EXAMPLE

Big slice correct

EXAMPLE

Pointer to slice element

EXAMPLE

Slice of slices

Terminal: deep_go × + ∨



СОВЕТ

Будьте осторожны с получением среза
от больших срезов и взятием указателя
на элемент среза — **иногда лучше копировать**

**КАКИЕ ЕСТЬ НЮАНСЫ ПРИ ОБРАЩЕНИИ
К ЭЛЕМЕНТАМ МАССИВОВ И СРЕЗОВ?**

Компилятор Go вставляет следующий код перед
каждым обращением к элементу слайса `a[i]`:



```
if i < 0 || i >= len(a) {  
    panic("index out of bounds")  
  
}
```

Since Go toolchain v1.7, the standard Go compiler has started to support BCE (bounds check elimination).

BCE can avoid some unnecessary bounds checks, so that the standard Go compiler could generate more efficient programs.

EXAMPLE

BCE presence

EXAMPLE

BCE absence

EXAMPLE

BCE optimization 1

EXAMPLE

BCE optimization 2

EXAMPLE

BCE optimization 3

EXAMPLE

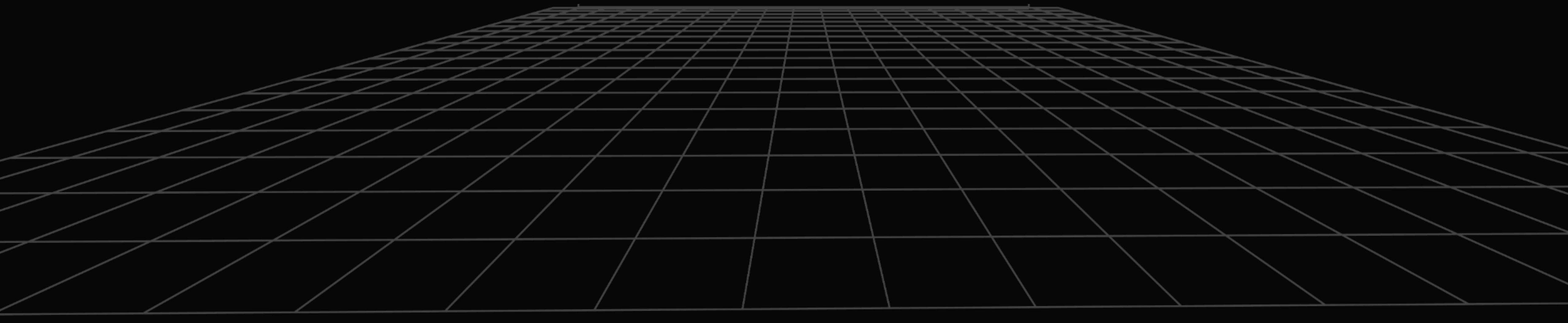
Without bound checks

EXAMPLE

Collection optimization

FAQ

Тонкости срезов



ПОЖАЛУЙСТА, ЗАПОЛНИ ОПРОС О ЗАНЯТИИ

Ссылка в чате и в группе участников

