

# V.02.

## Общий обзор занятия.

## Новый материал - основные вопросы.

- строки изнутри: иммутабельность - причины и следствия; перераспределение памяти при работе со строками
- слайсы изнутри: действия, не связанные с перераспределением памяти; функция copy
- слайсы изнутри: действия, связанные с перераспределением памяти
- Функции, возвращающие указатели на локальную переменную

# Лекция

Для начала углубляемся во внутреннее устройство слайсов и строк. С массивами и struct'ами всё более или менее понятно - они берут себе память при объявлении и после этого всё, что с ними может происходить,- так это может изменяться содержимое этой памяти, сами они перемещаться не могут, так что с точки зрения распределения памяти с ними вообще ничего больше не происходит. Разве что сборщик мусора может освободить от них память, когда они станут не нужны, но это, во-первых, я забегаю вперёд - разговор об областях видимости (scopes) и о времени жизни и сборке мусора ещё будет, а, во-вторых, это для нас совершенно неважно сейчас, ну, может быть освобождает память из-под массива/структур, так его считай, что уже и нет.

Со слайсами всё иначе - они могут изменять свой размер в процессе выполнения программы, и, если эти изменения потребуют увеличения ёмкости (cap), то вполне себе возможно, что слайс переедет в другое место. Мы это уже краем глаза зацепляли ещё в первом занятии, но вот сейчас пришло время разобраться в тамошних парадоксах досконально.

Со строками тоже смешно, но по-другому. Учитывая, что в utf8 коды символов имеют разную длину, изменение строки (именно её образа в оперативной памяти) - вещь опасная и странная, так что в Go сделано вполне себе естественно - строки неизменяемы (иммутабельны, immutable). Так что образы строк в оперативной памяти являются из себя некоторые константы, поэтому разные строковые переменные могут ссылаться на одно и

тоже место в памяти, а любая операция, связанная с изменением значения строковой переменной связана с её перенаправлением в другую область памяти (напомню, что строковая переменная - это адрес и длина кода строки в байтах).

Со строками, вроде, попроще получается, так что начнём с них.

## Строки изнутри. Часть 2.

Строки иммутабельны. Подчеркну, не значения строковых переменных (которое, напомню, есть адрес и длина области памяти, в которой хранится строка, собственно символы), а именно та область памяти, в которой хранятся символы, точнее, их utf8-коды. Ну, а раз строка неизменяема (иммутабельна, immutable), то хранить отдельно её длину и ёмкость смысла нет, ведь это одна и та же величина. Какой смысл делать ёмкость больше длины, если изменять строку всё равно нельзя. Причины иммутабельности состоит в том, что коды символов в utf8 имеют различную длину - от 1 до 4 байтов. Соответственно, мы не можем найти место символа с каким-то определённым номером, не проанализировав всю строку от начала и до этого символа. Так что совершенно естественно мы храним длину строки не в рунах, а в байтах.

Чисто освежить всё это - [пример 01a.go](#).

```
package main

import (
    "fmt"
    "strings"
    "unsafe"
)

type (
    stringHeader struct {
        start    uintptr
        length   uint
    }
)

func main() {
    str:= strings.Repeat("abc",4)
    fmt.Println(str)                                // abcabcabcabc
    fmt.Println(&str)                             // 0xc00003c1c0
```

```

fmt.Println(unsafe.Pointer(&str))           // 0xc00003c1c0
fmt.Println(unsafe.Sizeof(str))             // 16
strarr := (*[2]uintptr)(unsafe.Pointer(&str))
fmt.Println(*strarr)                      // [824634015872 12]
fmt.Printf("%x %d\n", (*strarr)[0], (*strarr)[1]) // c000048080 12
// all together - struct
fmt.Println(*(*stringHeader)(unsafe.Pointer(&str))) // {824634015872 12}
str2:= str
fmt.Println(*(*stringHeader)(unsafe.Pointer(&str2))) // {824634015872 12}
}

```

Но мы же теперь крутые, мы можем залезть в память нетипизованно и просто поменять какой-то байт, обманув тем самым компилятор. Давайте попробуем...

### Пример 01b.go.

```

type (
    stringHeader struct {
        start    unsafe.Pointer
        length   uint
    }
)

func getString(s string) *stringHeader {
    return (*stringHeader)(unsafe.Pointer(&s))
}

func main() {
    str:= "ABCDEFG"
    fmt.Println(str)      // ABCDEFG
    fmt.Println(&str)    // 0xc00003c1c0
    sh:= getString(str)
    fmt.Println(*sh)     // {0x4bf578 7}
    b:= (*byte)(unsafe.Pointer(uintptr((*sh).start)+4))
    fmt.Println(&b, b)  // 0xc000070020 0x4bf57c
    *b = '+'
    // unexpected fault address 0x4bf57c
    // fatal error: fault
    // ...
    fmt.Println(str)
}

```

И блистательно обламываемся. Так значит таки действительно строки неизменяемы? Ну, может быть, но как удаётся отследить, что мы пишем что-то именно внутрь строки, совершенно непонятно. Мы же вроде пишем величину типа `byte`, ну откуда компю знать, что этот адрес попадает внутрь какой-то строки. Не может такого быть.

И тут смотрим, что уж больно адрес этой строки отличается от адресов переменных `str` и `b`. А те друг от друга, конечно, отличаются, но совсем не так сильно. И в этом ключ к решению. Да, константная строка “ABCDEFG” размещается прямо во время компиляции внутри кода программы (точнее говоря, в сегменте данных, но это детям мы говорить не станем - зачем...). Можно сказать, что этот адрес не есть абсолютный адрес, а (что довольно естественно) отсчитывается от начала кода программы (`entry point`), но можно и не говорить.

Попробуем так:

### Пример 01c.go.

```
package main

import (
    "fmt"
    "unsafe"
)

type (
    stringHeader struct {
        start    unsafe.Pointer
        length   uint
    }
)

func getString(s string) *stringHeader {
    return (*stringHeader)(unsafe.Pointer(&s))
}

func main() {
    str := "ABCDEFG"
    fmt.Println(str)          // ABCDEFG
    fmt.Println(&str)         // 0xc00003c1c0
    sh := getString(str)
    fmt.Println(*sh)          // {0x4bf579 7}
```

```

str1:= "ABCDEFG"
fmt.Println(str1)      // ABCDEFG
fmt.Println(&str1)      // 0xc00003c1c0
sh1:= getString(str1)
fmt.Println(*sh1)       // {0x4bf579 7}

str1 += "!"
fmt.Println(str1)      // ABCDEFG!
fmt.Println(&str1)      // 0xc00003c1f0
sh1 = getString(str1)
fmt.Println(*sh1)       // {0xc0000480b0 8}
b:= (*byte)(unsafe.Pointer(uintptr((*sh1).start)+4))
fmt.Println(&b, b)      // 0xc000070020 0xc0000480b4
*b = '+'
fmt.Println(str1)      // ABCD+FG!
}

```

Да, конечно, это совсем уж тяжёлым сапогом, совсем на низком уровне, почти у железа мы сделали это - забрались в память бестипово, используя `unsafe.Pointer`. Ну да, теперь мы можем так делать. Другое дело, что злоупотреблять этим нельзя, да и пользоваться крайне осторожно, но такая возможность у нас теперь есть.

И, конечно, тут необходимо знание кодировки `utf8`. Давно это было - во втором семестре, надо бы детям напомнить или загнать их самих вспоминать/разбираться.

Отчётливо видно, что сначала `str1` видит строку, расположенную по “маленькому” адресу, а когда мы выполним с ним некую операцию, конкатенацию в данном случае, он сразу начинает смотреть на строку с “большим” адресом, т.е. выделенную при выполнении программы. Забавно здесь ещё и то, что сначала `str` и `str1` смотрят на одну и ту же область памяти. Так можно именно потому, что строки иммутабельны. Впрочем, такой эффект мы видели уже в примере `01a.go`.

## Слайсы изнутри. Часть 2.

**Объявление/создание слайса. Действия, не связанные с перераспределением памяти. Функция `copy`.**

В общем-то мы уже всё про слайс знаем: мы знаем, что переменная-слайс не хранит в себе данных, а хранит в себе указатель на тот отрезок памяти, где хранятся собственно данные, а также ёмкость этого отрезка памяти (в элементах слайса) и количество задействованных

элементов из этого отрезка памяти - длину. Т.е. слайс никогда не живёт сам по себе, он всегда “нависает” над некоторым отрезком памяти - массивом или отрезком массива. В частности, это объясняет то, что если просто объявить переменную типа слайс, то она оказывается не привязана ни к какому отрезку памяти, и, соответственно, указывает “в никуда”, на nil.

Всё это наглядно показывают примеры [02a.go](#):

```
package main

import (
    "fmt"
    "unsafe"
)

type (
    Slice struct {
        start    unsafe.Pointer
        len      int
        cap      int
    }
)

func printSlice(s []uint) {
    ps := (*Slice)(unsafe.Pointer(&s))
    fmt.Println((*ps).start, (*ps).len, (*ps).cap)
}

func main() {
    var p []uint
    fmt.Println(p)                      // []
    printSlice(p)                      // <nil> 0 0
    p = make([]uint, 2)
    fmt.Println(p)                      // [0 0]
    printSlice(p)                      // 0xc0000540a0 2 2
    p = append(p, 12345)
    fmt.Println(p)                      // [0 0 12345]
    printSlice(p)                      // 0xc000052140 3 4
}
```

и [02b.go](#):

```

package main

import (
    "fmt"
    "unsafe"
)

type (
    Slice struct {
        start    unsafe.Pointer
        len     int
        cap     int
    }
)

func printSlice(s []uint) {
    ps := (*Slice)(unsafe.Pointer(&s))
    fmt.Println((*ps).start, (*ps).len, (*ps).cap)
}

func main() {
    a := [...]uint {0,1,2,3,4,5,6}
    fmt.Println(unsafe.Pointer(&a))      // 0xc000078040
    b := a[2:5]
    fmt.Println(b)                      // [2 3 4]
    fmt.Println(unsafe.Pointer(&a[2]))  // 0xc000078050
    printSlice(b)                     // 0xc000078050 3 5
    c := b[1:2]
    fmt.Println(c)                      // [3]
    printSlice(c)                     // 0xc000078058 1 4
}

```

Во-первых, мы печатаем, кроме самих слайсов, содержимое переменных типа слайс, пользуясь тем, что мы выяснили в прошлый раз. И всё получается довольно естественно. Второй момент, на который здесь следует обратить внимание - это ёмкость слайсов b и c. Да, вполне естественно, их правый край располагается в конце того массива (отрезка памяти), который уже распределён, про который менеджер памяти знает, за которым он следит, который он просто так не выбросит обратно в кучу для повторного использования.

Ещё один пример [02c.go](#)

```

package main

import "fmt"

func square(d []int) {
    for i, x := range d {
        d[i] = x * x
    }
}

func fill(d []int) {
    for i, _ := range d {
        d[i] = i + 1
    }
}

func main() {
    var a [10]int
    fmt.Println(len(a), cap(a), a) // 10 10 [0 0 0 0 0 0 0 0 0 0]
    d := a[2:8]
    fill(d)
    fmt.Println(len(a), cap(a), a) // 10 10 [0 0 1 2 3 4 5 6 0 0]
    fmt.Println(len(d), cap(d), d) // 6 8 [1 2 3 4 5 6]
    square(a[4:7])
    fmt.Println(len(a), cap(a), a) // 10 10 [0 0 1 2 9 16 25 6 0 0]
    fmt.Println(len(d), cap(d), d) // 6 8 [1 2 9 16 25 6]
}

```

ярко и наглядно показывает, как взаимодействуют слайсы и массивы “живущие на общей территории”, а также то, что при передаче слайса в функцию этот слайс изменяется (точнее, изменяются данные слайса, а сам слайс как раз и нет). Почему - понятно, не изменяется содержимое переменной типа слайс - адрес, длина и ёмкость, а данные как раз вполне себе успешно меняются.

Для копирования одного слайса в другой имеется безопасная

## функция copy

```
func copy(dst, src []Type) int
```

которая копирует данные из слайса src в слайс dst. Функция возвращает количество

скопированных элементов, которое есть меньшее из `len(src)` и `len(dst)`. Слайсы могут перекрываться - функция `copy` сначала куда-то запоминает все копируемые данные из слайса-источника, а только потом все их помещает в слайс-приёмник.

Действие функции `copy` иллюстрирует [пример 03.go](#):

```
package main

import "fmt"

func main() {
    slc0:= make([]int,5)
    fmt.Println(slc0)

    // Creating slices
    slc1:= []string{"one", "two", "three", "four","five", "six", "seven", "eight"}
    var slc2 []string
    slc3:= make([]string, 5)
    slc4:= []string{"eleven", "twelve", "thirteen", "fourteen"}

    // Before copying
    fmt.Println("Slice_1:", slc1)
    fmt.Println("Slice_2:", slc2)
    fmt.Println("Slice_3:", slc3)
    fmt.Println("Slice_4:", slc4)

    // Copying the slices
    copy_1 := copy(slc2, slc1)
    fmt.Println("\nSlice:", slc2)
    fmt.Println("Total number of elements copied:", copy_1)

    copy_2 := copy(slc3, slc1)
    fmt.Println("\nSlice:", slc3)
    fmt.Println("Total number of elements copied:", copy_2)

    copy_3 := copy(slc3, slc4)
    fmt.Println("\nSlice:", slc3)
    fmt.Println("Total number of elements copied:", copy_3)

    copy_4:= copy(slc1, slc4)
    fmt.Println("\nSlice:", slc1)
```

```
fmt.Println("Total number of elements copied:", copy_4)

copy_5:= copy(slc1, slc1[3:])
fmt.Println("\nSlice:", slc1)
fmt.Println("Total number of elements copied:", copy_5)

// [0 0 0 0]
// Slice_1: [one two three four five six seven eight]
// Slice_2: []
// Slice_3: [    ]
// Slice_4: [eleven twelve thirteen fourteen]
//
// Slice: []
// Total number of elements copied: 0
//
// Slice: [one two three four five]
// Total number of elements copied: 5
//
// Slice: [eleven twelve thirteen fourteen five]
// Total number of elements copied: 4
//
// Slice: [eleven twelve thirteen fourteen five six seven eight]
// Total number of elements copied: 4
//
// Slice: [fourteen five six seven eight six seven eight]
// Total number of elements copied: 5
}
```

Прокомментировать это, конечно, надо, но никаких сложностей здесь не видно.

## **Расширяющийся (растущий) слайс - действия, связанные с перераспределением памяти. Механика функции append.**

При увеличивании слайса с помощью функции append может оказаться, что текущей ёмкости слайса, точнее говоря, текущей ёмкости того массива (именованного или анонимного), на который указывает слайс, в котором хранятся собственно данные, недостаточно. И тогда за сценой появляется динамический массив. В чём состоит этот метод? В разных ситуациях он может действовать немного по-разному, но в основном эта техника всегда действует более или менее одинаково. В данном случае происходит следующее:

- создаётся новый массив (выделяется область памяти), достаточный для хранения всех данных - и уже имевшихся, и добавляемых; как правило, память выделяется с запасом, чтобы не слишком часто выполнять такие действия - они недешёвые. Создаваемый массив всегда анонимен, так что обратиться к нему можно только через наш увеличивающийся слайс (ну, или через слайсы, которые будут базироваться на нём).
- данные из старого массива копируются в новый, туда же заносятся и добавляемые данные.
- увеличивающийся слайс изменяет свои внутренности: адрес массива с данными (это и есть только что созданный в п.1 анонимный массив), длину и ёмкость (которая равна размеру только что созданного базового массива).

Размер нового базового массива определяет реализация функции `append`. И зависит он, кроме собственно реализации, от многих факторов: от исходной длины и ёмкости слайса и от размера добавляемых данных - это само собой. Но не только от них. Есть зависимость и от менеджера оперативной памяти (т.е. от операционной системы), и от текущей оперативной ситуации, и от не знаю чего ещё. Но предсказать размер увеличения ёмкости я не возьмусь.

Собственно следующие три примера посвящены иллюстрациям всей этой несложной механики. Собственно всё, что здесь требуется, так это аккуратно пройтись по примерам, разбирая, отслеживая эту самую механику. Я не буду описывать примеры подробно, на них и так всё видно, да они и не сюжетные, а так, чисто технические. Краткие примечания тем не менее оставлю.

### Пример 04.go:

```
package main

import (
    "fmt"
    "unsafe"
)

type (
    Slice struct {
        start unsafe.Pointer
        len   int
        cap   int
    }
)
```

```

func printSlice(s []int) {
    sInfo:= (*(*Slice)(unsafe.Pointer(&s)))
    fmt.Println(sInfo.start, sInfo.len, sInfo.cap)
}

func main() {
    a := [...]int{1, 2, 3, 4, 5, 6, 7, 8}
    fmt.Println("a:", len(a), a)           // a: 8 [1 2 3 4 5 6 7 8]
    fmt.Println(unsafe.Pointer(&a))       // 0xc000078040
    s := a[:5]
    fmt.Println(unsafe.Pointer(&s))       // 0xc000042420
    fmt.Println("s:", len(s), cap(s), s)  // s: 5 8 [1 2 3 4 5]
    printSlice(s)                      // 0xc000078040 5 8
    s2 := s
    fmt.Println(unsafe.Pointer(&s2))     // 0xc000042460
    fmt.Println("s2:", len(s2), cap(s2), s2) // s2: 5 8 [1 2 3 4 5]
    printSlice(s2)                     // 0xc000078040 5 8
    s = append(s, -1, -2, -3, -4)      //
    fmt.Println("a:", len(a), a)           // a: 8 [1 2 3 4 5 6 7 8]
    fmt.Println("s:", len(s), cap(s), s)  // s: 9 16 [1 2 3 4 5 -1 -2 -3 -4]
    printSlice(s)                      // 0xc00007a080 9 16
    s2 = append(s2, 0)                 //
    fmt.Println("a:", len(a), a)           // a: 8 [1 2 3 4 5 0 7 8]
    fmt.Println("s2:", len(s2), cap(s2), s2) // s2: 6 8 [1 2 3 4 5 0]
    printSlice(s2)                     // 0xc000078040 6 8
    s2 = append(s2, -11, 12, 13, 14, 15)
    fmt.Println("a:", len(a), a)           // a: 8 [1 2 3 4 5 0 7 8]
    fmt.Println("s2:", len(s2), cap(s2), s2) // s2: 11 16 [1 2 3 4 5 0 -11 12 13 14
15]
    printSlice(s2)                     // 0xc00007a100 11 16
    s = a[3:6]
    fmt.Println("s:", len(s), cap(s), s)  // s: 3 5 [4 5 0]
    printSlice(s)                      // 0xc000078058 3 5
    s = append(s, 1, 2, 3, 4, 5, 6, 7, 8)
    fmt.Println("s:", len(s), cap(s), s)  // s: 11 12 [4 5 0 1 2 3 4 5 6 7 8]
    printSlice(s)                      // 0xc00003a060 11 12
    s = append(s, 1, 2, 3)
    fmt.Println("s:", len(s), cap(s), s)  // s: 14 24 [4 5 0 1 2 3 4 5 6 7 8 1 2
3]
    printSlice(s)                     // 0xc000086000 14 24

```

```

    s = s[:16]
    fmt.Println("s:", len(s), cap(s), s)      // s: 16 24 [4 5 0 1 2 3 4 5 6 7 8 1 2
3 0 0]
    printSlice(s)                          // 0xc000086000 16 24
}

```

Главное в нём - показать, что если рост слайса не требует перераспределения памяти, то изменяется текущий базовый массив, если же перераспределение памяти происходит, то старый базовый массив не изменяется. По концовке ещё пара примеров, которые показывают, что далеко не всегда ёмкость нового массива получается удвоением ёмкости старого. Единственное, в чём я уверен - новая ёмкость всегда чётна.

Пара наблюдений:

1. Писать так `s = s[:cap(s)]` можно, даже если длина слайса при этом увеличивается, т.е. `len(s) < cap(s)`. Если не происходит перераспределения памяти, то мы можем двигать правую границу слайса как угодно в рамках “базового массива” этого слайса.
2. Если мы заранее хотя бы приблизительно знаем, сколько элементов мы запихаем в слайс, то лучше создавать его так:

```
go s := make([]Type, 0, ожидаемая длина слайса)
```

И тогда вставка нового элемента с помощью `append` не будет требовать перераспределения памяти (разве сто в конце, если мы не угадаем с примерным ожидаемым размером и будет выполняться существенно быстрее,

**Пример 05.go:**

```

package main

import "fmt"

func f(d []int) {
    for i, _ := range d {
        d[i] = i + 1
    }
    d = append(d, 1, 2, 3, 4, 5)
    fmt.Println(len(d), cap(d), d)
}

func main() {
    var a [10]int
    fmt.Println(len(a), cap(a), a)      // 10 10 [0 0 0 0 0 0 0 0 0 0]
    c := a[2:4]
}

```

```

f(c)                      // 7 8 [1 2 1 2 3 4 5]
fmt.Println(len(a), cap(a), a) // 10 10 [0 0 1 2 1 2 3 4 5 0]
fmt.Println(len(c), cap(c), c) // 2 8 [1 2]
cc := []int{11, 12, 13, 14}
fmt.Println(len(cc), cap(cc), cc) // 4 4 [11 12 13 14]
f(cc)                      // 9 10 [1 2 3 4 1 2 3 4 5]
fmt.Println(len(cc), cap(cc), cc) // 4 4 [1 2 3 4]
}

```

Пример показывает, что происходит со слайсом и его базовым массивом внутри функции с параметром-слайсом. Формальный параметр может внутри функции меняться как угодно, соответствующий фактический параметр не меняется. Но! Может меняться содержимое базового массива - напомню ещё раз, что переменная-слайс содержит адрес базового массива, длину и ёмкость. А может и не меняться, если, например, append внутри вызываемой функции требует перераспределять память.

Очень хорошо было бы изменить функцию `func f(d []int)` на функцию, получающую слайс по адресу: `func f(d *([]int))`. Хорошее **упражнение на практику** получается: изменить заголовок функции, исправить функцию, чтобы она работала, а уже потом посмотреть, как именно работает получившаяся функция и объяснить себе, почему именно так.

И, наконец **пример 06.go** имеет единственную цель - показать как прихотливо может изменяться ёмкость слайса при append'ах:

```

package main

import (
    "fmt"
    "unsafe"
)

type (
    Slice struct {
        start unsafe.Pointer
        len   int
        cap   int
    }
)

func sliceInfo(s []int) Slice {

```

```
    return *(*Slice)(unsafe.Pointer(&s))
}

func main() {
    b := make([]int, 0, 1)
    fmt.Println(b)           // []
    fmt.Println(sliceInfo(b)) // {0xc000048058 0 1}
    b = append(b, 7)
    fmt.Println(b)           // [7]
    fmt.Println(sliceInfo(b)) // {0xc000048058 1 1}
    b = append(b, 7)
    fmt.Println(b)           // [7 7]
    fmt.Println(sliceInfo(b)) // {0xc0000480b0 2 2}
    b = append(b, 7)
    fmt.Println(b)           // [7 7 7]
    fmt.Println(sliceInfo(b)) // {0xc0000460e0 3 4}
    fmt.Println()
    c := cap(b)
    for i := 0; i < 1000000; i++ {
        b = append(b, i)           // {0xc000078040 5 8}
        if cap(b) != c {           // {0xc00007a080 9 16}
            fmt.Println(sliceInfo(b)) // {0xc000034100 17 32}
            c = cap(b)             // {0xc000086000 33 64}
        }
        // {0xc000088000 65 128}
        // {0xc00008a000 129 256}
        // {0xc00008c000 257 512}
        // {0xc00008e000 513 1024}
        // {0xc000090000 1025 1280}
        // {0xc00009a000 1281 1696}
        // {0xc0000a4000 1697 2304}
        // {0xc0000b6000 2305 3072}
        // {0xc0000bc000 3073 4096}
        // {0xc0000c4000 4097 5120}
        // {0xc0000ce000 5121 7168}
        // {0xc0000dc000 7169 9216}
        // {0xc0000ee000 9217 12288}
        // {0xc000106000 12289 15360}
        // {0xc000124000 15361 19456}
        // {0xc00014a000 19457 24576}
        // {0xc00017a000 24577 30720}
        // {0xc0001b6000 30721 38912}
    }
}
```

```
// {0xc000202000 38913 49152}
// {0xc000262000 49153 61440}
// {0xc0002da000 61441 76800}
// {0xc000370000 76801 96256}
// {0xc000082000 96257 120832}
// {0xc00016e000 120833 151552}
// {0xc00044e000 151553 189440}
// {0xc0005c0000 189441 237568}
// {0xc000082000 237569 296960}
// {0xc00044e000 296961 371712}
// {0xc000082000 371713 464896}
// {0xc00044e000 464897 581632}
// {0xc0008be000 581633 727040}
// {0xc000e4a000 727041 909312}
// {0xc00044e000 909313 1136640}

}
```

Ещё один забавный момент, о котором уже упоминалось в прошлый раз, но теперь окончательно

## Функции, возвращающие указатели на локальную переменную.

Да, функция может возвращать адрес локальной переменной, и тогда локальная переменная не умирает, компилятор Go это замечает и оставляет такие локальные переменные жить. Да, сборщик мусора замечает, что такая переменная ещё нужна и не убивает её.

Пример [07.go](#)

```
package main

import "fmt"

func hello() *string {
    s := "hello, world"
    return &s
}

func main() {
    fmt.Println(*hello()) // hello, world
```

```
hi := hello()
fmt.Printf("Тип переменной hi %T,\n", hi)           // Тип переменной hi *string,
fmt.Printf("она содержит указатель на строку \"%s\"\n", *hi)
                                                // она содержит указатель на строку "hello, world"
}
```

## Задачи и упражнения

Сегодня говорили в основном о механике работы со слайсами. Хотелось бы осознать, какие варианты лучше, какие хуже, почему и т.д.

И поэтому я предлагаю дать детям несколько простых сюжетов. Пусть дети их реализуют. Некоторые варианты реализации я здесь опишу-напишу-набросаю, но пусть бы дети постарались что-нибудь ещё придумать. И при этом пусть реализуют действия как функции и засекают скорость работы соответствующих функций. В следующий раз поговорим о benchmark'ах, а сегодня пусть засекают время по-простому с помощью package time - используя функции time.Now() и time.Since.

Итак, сюжеты. Во всех предполагается, что это только для затравки, что дети придумают ещё варианты.

### 1. вычеркнуть i-й элемент слайса, сохраняя порядок следования оставшихся элементов

Варианты реализации:

- a. `s = append(s[:i], s[i+1]...)`
- b. `for k:=i; k < len(s)-2; k++ {  
 s[k] = s[k+1]  
}  
s = s[:len(s)-1]`
- c. аналог b, но вместо цикла для копирования элементов использовать функцию copy
- d. `for k:= i; k > 0; k-- {  
 s[k] = s[k-1]  
}  
s = s[2:]`
- e. аналог d, но вместо цикла для копирования элементов использовать функцию copy
- f. комбинация b. и d. - если  $i < \text{len}(s)/2$ , то c, иначе b
- g. аналог f, но вместо цикла для копирования элементов использовать функцию copy

## 2. Вставить x на i-е место.

Варианты реализации:

- a. `s = append( append(s[:i], x), s[i:]...)`
- b. `s = append(s, s[len(s)-1])`  
`for k = len(s)-2; k>i; k-- {`  
 `s[k] = s[k-1]`  
`}`  
`s[i] = x`
- c. `s = append(s, s[len(s)-1])`  
`copy (s[i:len(s)-2], s[i+1]...)`  
`s[i] = x`

## 3. Написать свою функцию copy. Сравнить по скорости не только реализации между собой, но и с функцией copy

- a. с помощью цикла, копируя элементы по одному
- b. если копируем более короткий слайс в более длинный, то собирать результат, как сцепку (с помощью append) - слайс-источник + хвост от слайса-приёмника: иначе просто присваиваем приёмнику начальный отрезок слайса-источника.

## 4. Вычеркнуть i-й элемент, не обязательно сохраняя порядок следования оставшихся элементов.

- a. `s[i] = s[len(s)-1]`  
`s = s[:len(s)-1]`
- b. `s[i] = s[0]`  
`s = s[1:]`

## 5. Сдвинуть циклически на k позиций влево.

пример для k=3: 1 2 3 4 5 6 7 8 9 -> 4 5 6 7 8 9 1 2 3

- a. `s = append(s[k:], s[:k]...)`
- b. `buff := make([]int, k, k)`  
`copy(buff, s)`  
`copy(s, s[k:])`  
`copy(s[len(s)-k:], buff)`

## 6. Сдвинуть циклически на k позиций влево, не заводя дополнительных массивов/слайсов.

Разумеется речь не идёт о том, чтобы к раз исполнять сдвиг на одну позицию, используя одну дополнительную переменную - и так понятно, что тормоза. Речь идёт о тех трёх способах, которые разбирали (возможно не все) в третьем семестре. Я привожу их в приложения. Все способы исполняют сдвиг на месте - не привлекая для этого дополнительных слайсов/массивов. Кроме того, это приложение вместе со всеми дополнительными файлами лежит в каталоге `shift`. Можно весь его давать детям. Хотя там приводятся готовые коды - лучше бы детям самим их написать.

## Приложение. Три решения задачи о циклическом сдвиге массива.

---

Суть задачи формулируется просто: на вход подается одномерный массив длины  $N$  и число  $K$  ( $0 < K < N$ ); требуется сместить все элементы на  $K$  позиций влево, причём массив считаем "зацикленным" - перед начальным элементом массива стоит последний. При этом результат должен получиться на том же самом месте. Например, если массив `[1, 2, 3, 4, 5, 6, 7, 8]` сдвинуть циклически на 3 позиции влево, то получим массив `[4, 5, 6, 7, 8, 1, 2, 3]`.

Задача прекрасна сама по себе, но приятно, что она имеет очевидный практический смысл - ведь нам очень часто необходимо переставить более или менее значительный по размеру кусок данных в другое место, да хотя бы переместить кусок текста в текстовом редакторе - а ведь циклический сдвиг как раз и состоит в том, что мы меняем местами два последовательных куска массива данных.

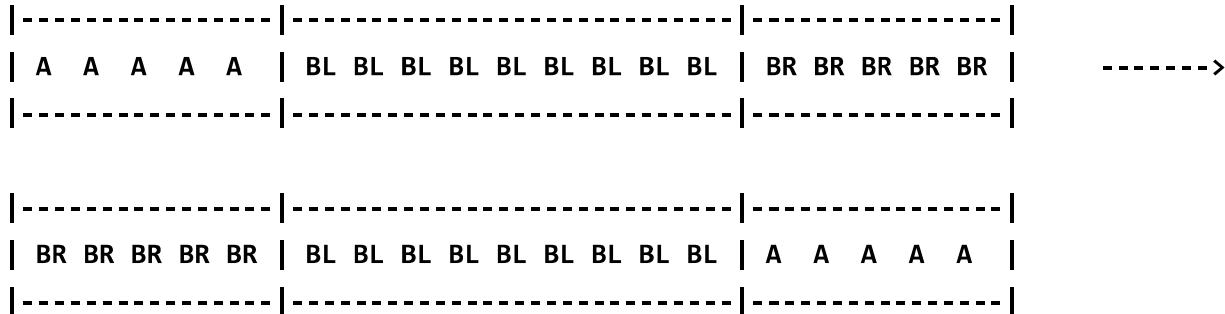
Задача классическая, так что о ней в сети понаписано много где, естественно в большинстве своём понаписано всякого дерьяма, так что я лучше дам [нормальный линк](#). На этом линке более или менее подробно изложен материал [главы 2 книги \\*Дж. Бентли. Жемчужины программирования](#). Во второй главе, помимо задачи о циклическом сдвиге, речь идёт и о половинном делении, и довольно много говорится о пользе сортировки (но не об алгоритмах сортировки - о них подробно написано в другом месте этой же книги). О сортировке мы сегодня не говорим, но у меня не повернулась рука резать эту главу, уж больно хороша!

Я тоже буду, в основном, придерживаться вышеупомянутой книги, но изложу алгоритмы поподробнее и изменю порядок изложения.

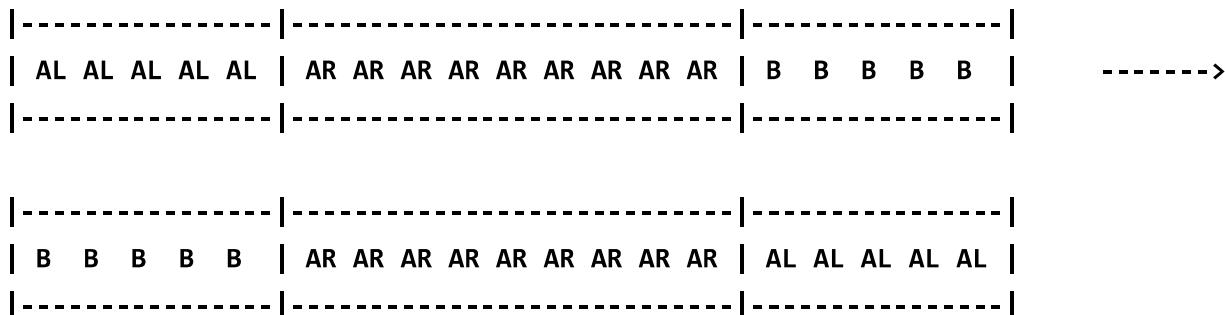
### 1. сдвиг через перестановку блоков

Цитата из Бентли: “Можно предложить и другой алгоритм, который возникает из рассмотрения задачи с другой точки зрения. Циклический сдвиг массива  $x$  сводится фактически к замене  $AB$  на  $BA$ , где  $A$  — первые  $K$  элементов массива, а  $B$  — оставшиеся элементы. Предположим, что  $A$  короче  $B$ . Разобьем  $B$  на  $B_{left}$  и  $B_{right}$ , где  $B_{right}$  содержит  $K$  элементов (столько же, сколько и  $A$ ). Поменяем местами  $A$  и  $B_{right}$ , получим  $B_{right}B_{left}A$ . При этом  $A$  окажется в конце массива — там, где и полагается. Поэтому можно сосредоточиться на перестановке  $B_{right}$  и  $B_{left}$ . Эта задача сводится к начальной, поэтому алгоритм можно вызывать рекурсивно. Программа, реализующая этот алгоритм, будет достаточно красивой, но она требует аккуратного написания кода, а оценить ее эффективность непросто”.

Эта цитата всё-таки требует некоторых комментариев. Во-первых, рекурсивный вызов в данном случае совершенно ни к чему — процесс чудеснейшим и естественнейшим образом реализуется циклом. Во-вторых, сложность анализа эффективности преувеличена катастрофически — там всё совсем просто, но об этом чуть позже. В-третьих, ничего не сказано об окончании процесса, а он закончится тогда, когда  $A$  и  $B$  будут иметь одинаковую длину. Этот момент непременно настанет, об этом чуть позже — в анализе алгоритма. А пока я нарисую схемку, иллюстрирующую вышеописанный подход:



А вот что будет в случае, если часть  $A$  длиннее части  $B$ :



После чего остаётся переставить местами  $AL$  и  $AR$ .

И теперь код

```
//shift_1a.go
package main

import "fmt"

func swap(x []int, start1 int, start2 int, length int) {
    i1, i2 := start1, start2
    for i := 0; i < length; i++ {
        x[i1], x[i2] = x[i2], x[i1]
        i1++
        i2++
    }
}

func shift(x []int, left0 int, right0 int, rightK int) {
    // x - изменяемый массив
    // left0 - начало левого фрагмента
    // right0 - начало правого фрагмента
    // rightK - конец правого фрагмента
    for {
        lengthL := right0 - left0 // длина левой части
        lengthR := rightK - right0 + 1 // длина правой части
        if lengthL < lengthR { // левая часть короче правой
            swap(x, left0, rightK - lengthL + 1, lengthL)
            rightK = rightK - lengthL
        } else
        if lengthL > lengthR { // левая часть длиннее правой
            swap(x, left0, right0, lengthR)
            left0 = left0 + lengthR
        } else {
            // длины частей равны
            swap(x, left0, right0, lengthL)
            break
        }
    }
}

func main() {
    var (
```

```

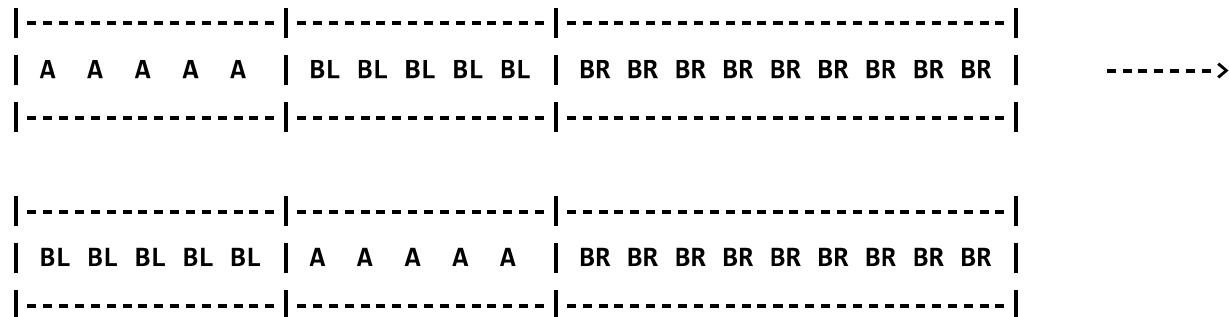
n, k int
)

fmt.Println("Введите длину массива: ")
fmt.Scanln(&n)
fmt.Println("Введите величину сдвига: ")
fmt.Scanln(&k)

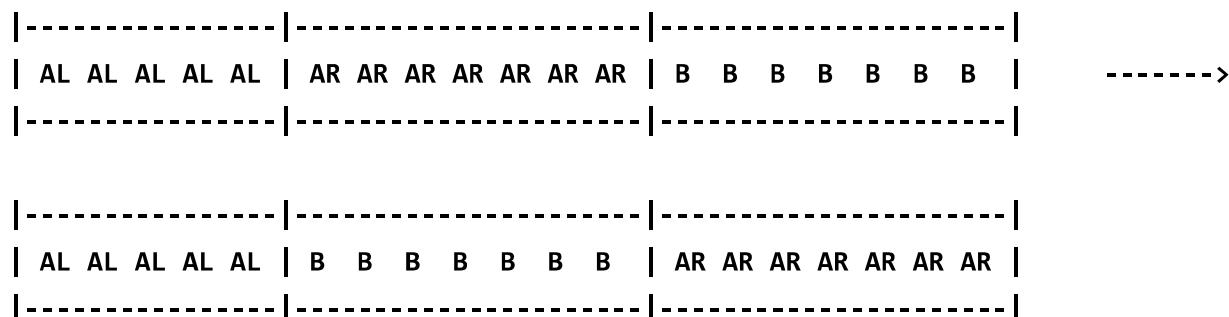
x:= make([]int, n, n)
for i:= 0; i < n; i++ {
    x[i] = i+1
}
fmt.Println(x)
shift(x, 0, k, n-1)
fmt.Println(x)
}

```

Рассмотрим ещё один вариант того же самого подхода. В первом варианте мы меняли местами крайние части одинаковой длины. Попробуем теперь переставлять крайнюю часть со средней частью. Не буду расписывать словами, приведу схемы:



В результате часть BL встала на своё место, а нам остаётся переставить части A и BR. А вот что будет, если часть A длиннее части B:



и остаётся перетавить AL и B.

[Код](#), разумеется почти не изменился:

```
//shift_1b.go
package main

import "fmt"

func swap(x []int, start1 int, start2 int, length int) {
    i1, i2 := start1, start2
    for i := 0; i < length; i++ {
        x[i1], x[i2] = x[i2], x[i1]
        i1++
        i2++
    }
}

func shift(x []int, left0 int, right0 int, rightK int) {
    // x - изменяемый массив
    // left0 - начало левого фрагмента
    // right0 - начало правого фрагмента
    // rightK - конец правого фрагмента
    for {
        lengthL := right0 - left0 // длина левой части
        lengthR := rightK - right0 + 1 // длина правой части
        if lengthL < lengthR { // левая часть короче правой
            swap(x, left0, right0, lengthL) // !!!
            left0, right0 = right0, rightK - lengthL + 1 // !!!
        } else
        if lengthL > lengthR { // левая часть длиннее правой
            swap(x, left0, right0, lengthR)
            right0, rightK = left0 + lengthR, right0-1 // !!!
        } else {
            // длины частей равны
            swap(x, left0, right0, lengthL)
            break
        }
    }
}
```

```

func main() {
    var (
        n, k int
    )

    fmt.Print("Введите длину массива: ")
    fmt.Scanln(&n)
    fmt.Print("Введите величину сдвига: ")
    fmt.Scanln(&k)
    x := make([]int, n, n)

    for i := 0; i < n; i++ {
        x[i] = i+1
    }
    fmt.Println(x)
    shift(x, 0, k, n-1)
    fmt.Println(x)
}

```

Отличия в коде минимальные - изменились только три строки, они отмечены комментарием // !!!

*Анализ алгоритма перестановки блоков.* А анализ прост до изумления. Каждый раз, когда мы меняем местами два элемента (в функции swap) один из них становится на свое окончательное место и после этого уже никогда не двигается. Так что весь алгоритм потребует не более  $N$  операций обмена двух элементов. Точнее говоря, даже меньше, поскольку при последнем вызове функции swap (когда длины частей равны) при каждом обмене на место становятся сразу оба элемента.

А теперь я приведу ещё один алгоритм, который у Бентли называется “Алгоритм #3: переворотами”.

## 2. Циклический сдвиг массива переворотами

Не буду ничего придумывать в данном случае, а просто процитирую Бентли:  
начало цитаты

Задача кажется сложной, пока вас не осенит озарение («ага!»): итак, нужно преобразовать массив AB в BA. Предположим, что у нас есть функция reverse, переставляющая элементы некоторой части массива в противоположном порядке. В исходном состоянии массив имеет вид AB. Вызвав эту функцию для первой части, получим A<sub>r</sub>B (прим. редактора: A<sub>r</sub> - это

модифицированная часть A, к которой применили функцию перестановки reverse). Затем вызовем ее для второй части: получим  $A_rB_r$ . Затем вызовем функцию для всего массива, что даст  $(A_rB_r)_r$ , а это в точности соответствует BA. Посмотрим, как будет такая функция действовать на массив abcdefgh, который нужно сдвинуть влево на три элемента: псевдокод: [Сдвиг через функцию перестановки reverse](#)

```
1. reverse(0, k-1) /* cba|defgh */
2. reverse(k, n-1) /* cba|hgfed */
3. reverse(0, n-1) /* defgh|abc */
```

Дуг Макилрой (Doug McIlroy) предложил наглядную иллюстрацию циклического сдвига массива из десяти элементов вверх на пять позиций (рис. 2.3); начальное положение: обе руки ладонями к себе, левая над правой: [картинка находится в файле](#)

Код, использующий функцию переворота, оказывается эффективным и малотребовательным к памяти, и настолько короток и прост, что при его реализации сложно ошибиться.

Б. Керниган и П. Дж. Плоджер пользовались именно этим методом для перемещения строк в текстовом редакторе в своей книге (B. Kernighan, P. J. Plauger, Software Tools in Pascal, 1981). Керниган пишет, что эта функция заработала правильно с первого же запуска, тогда как их предыдущая версия, использовавшая связный список, содержала несколько ошибок. Этот же код используется в некоторых текстовых редакторах, включая тот, в котором я впервые набрал настоящую главу. Кен Томпсон (Ken Thompson) написал этот редактор с функцией reverse в 1971 году, и он утверждает, что она уже тогда была легендарной...”  
*конец цитаты*

Реализацию писать не стану - всё совсем просто и ясно, так что оставляем её в качестве **упражнения**.

### 3. последовательный сдвиг по одному элементу

Рассмотрим алгоритм из много цитировавшейся книги Дж. Бентли “Жемчужины программирования” (п. 2.3). В изложении считается, что нумерация массива начинается с 0.  
*начало цитаты*

Алгоритм #1: последовательный обмен

Одним из вариантов решения будет введение дополнительной переменной. Элемент  $x_0$  помещается во временную переменную  $t$ , затем  $x_k$  помещается в  $x_0, x_{2*k}$  — в  $x_k$  и так далее (перебираются все элементы массива  $x$  с индексом по модулю  $n$ ). *примечание от меня: я*

боюсь, что здесь какая-то типографская накладка, явно имеется в виду “с индексом  $rk$ ,  $r = 0, 1, 2, \dots$  по модулю  $n$ ”), пока мы не возвращаемся к элементу  $x_0$ , вместо которого записывается содержимое переменной  $t$ , после чего процесс завершается. Если  $i = 3$ , а  $n = 12$ , этот этап проходит следующим образом: см. [рисунок](#)

Если при этом не были переставлены все имеющиеся элементы, процедура повторяется, начиная с  $x[1]$  и так далее, до достижения конечного результата.  
конец цитаты

Есть и ещё один [рисунок](#), который иллюстрирует процесс для  $N=15$ ,  $K=3$ .

Попробую изложить этот алгоритм поподробнее, заодно и поймём, откуда у Бентли в коде (я его здесь не привожу, но в материалах к предыдущему занятию есть полный текст соответствующей главы из книги) появляется слово `gcd`.

Давайте начнём вышеописанный процесс для  $N=15$ ,  $K=3$ . Тогда 3-й элемент перейдёт на 0-е место, 6-й - на 3-е, 9-й - на 6-е, 12-й - на 9-е, и, наконец, 0-й - на 12-е. Переехали на свои новые места только 5 элементов. Повторяем процесс, начиная с элемента на 1-ом месте. В результате последовательно встанут на свои новые места 4-й, 7-й, 10-й, 13, и 1-й элементы. Наконец, совершив такой последовательный обмен, начиная со 2-го элемента, поставим на свои новые места 5-й, 8-й, 11-й, 14-й и 2-й элементы.

Интересное наблюдение: каждый раз (в данном примере каждый из трёх раз) процесс последовательного обмена останавливается на том элементе, с которого он начался. Почему так происходит, почему мы никогда не натыкаемся на уже обработанный элемент из другого внутреннего цикла? Ответ на этот вопрос легко виден из приводимого рисунка. В каждую ячейку ведут ровно два отрезка: по одному из них в эту ячейку “приехал новый жилец”, по другому - “уехал старый”. Понятно (просто из задачи), что других отрезков ни в какую ячейку не входит-выходит. И вот мы выходим из какой-то ячейки. Каждый раз мы входим в новую ячейку и выходим из неё, зайти в уже посещённую ячейку мы не можем - это будет третий отрезок, ведущий в неё. С другой стороны процесс должен закончиться - мы на каждом шаге посещаем новую ячейку, а они в конце концов закончатся. Но закончить рисование нашей ломаной линии мы можем только одним единственным способом - вернуться в начальную вершину, в ту, из которой начался последовательный обмен.

Итак, первый цикл закончен, на рисунке ему соответствует замкнутая ломаная 1-4-7-10-13-1. Если в массиве остались еще не отмеченные ячейки, то среди них обязательно будет и ячейка номер 2. Иначе, если наш цикл привел нас из первой ячейки во вторую, то он, прежде, чем замкнуться, должен привести нас в третью, из третьей - в четвертую, и т.д., т.е. свободных ячеек не останется. В самом деле, просто повернём кружок так, чтобы двойка

встала на место 1. Тогда ломаная будет вести нас из двойки в тройку. Ну, и т.д. Начинаем второй цикл со второй ячейки. Если второй цикл после себя оставит непомеченные ячейки, то выполним третий цикл, который, конечно же, начнется с третьей ячейки, и т.д.

И как долго и т.д.? А пока не переставим всё. Можем просто подсчитывать количество передвинутых элементов и заканчивать тогда, когда переставим N элементов. [Этот вариант реализован в программе](#):

```
// shift_3.go
package main

import "fmt"

func shift (x []int, k int)  {
    var (
        tmp, current, next int
        count int = 0
    )

    for start := 0; count < len(x); start++  {
        tmp = x[start]
        current = start
        for  {
            next = (current + k)%len(x)
            if next == start  {
                x[current] = tmp
                count++
                break
            }
            x[current] = x[next]
            count++
            current = next
        }
    }
}

func main()  {
    var (
        n, k int
    )
}
```

```
fmt.Println("Введите длину массива: ")
fmt.Scanln(&n)
fmt.Println("Введите величину сдвига: ")
fmt.Scanln(&k)

x:= make([]int, n, n)
for i:= 0; i < n; i++ {
    x[i] = i+1
}
fmt.Println(x)
shift(x, k)
fmt.Println(x)
}
```