Лекция 1. Установка и запуск первой программы

Шаг 1. Установка компилятора

- Переход по ссылке: https://golang.org/dl/
- Установили для своей ОС компилятор GoLang
  Важно : на данном курсе желательно, чтобы у вас была версия компилятра > 1.08

Шаг 2. GoRoot GoPath

Определение : `GOROOT` - это файловый путь, указывающий расположение ***КОМПИЛЯТОРА*** Go.
Определение: `GOPATH` - это файловый путь , указывающий на расположение ***РАБОЧЕГО ОКРУЖЕНИЯ*** (Там где пишем код и мазюкаем проекты). По умолчанию, на курсе мы создали `GOPATH` по адресу `C:\Users\<username>\go`

Шаг 3. Инициализация рабочего окружения

Чтобы создать рабочее окрудение нам надо в `GOPATH` определить 3 диреткории:

- `src` - место, где будут лежать исходники проектов (скрипты .go)
- `bin` - место, где будут лежать скомпилированные бинарники, после выполнения компиляции проектов
- pkg - мсто, где будут жить сторонние пакеты для наших проектов

Шаг 4. Первая программа

В `GOPATH/src` создадим файл `main.go`
со следующей начинкой:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello world!")
}
```

Шаг 5. Запуск и компиляция

Go - компилируемый язык.
Для того, чтобы скомпилировать исполняемый файл, можно выполнить команду

```
go build <path/to/go/file.go>
```

Данная команда создает исполнямый файл по месту ее вызова. Это удобно,когда мы в пылу битвы и хочется посмотреть, скомпилируется ли оно вообще или в случае каких-либо тестов - позволяет на месте все проверять.

Другая команда, которая также позволяет создать исполняемый файл:

```
go install <path/to/go/file.go>
```

Данная команда создает исполняемый файл по пути `GOPATH/bin`.

Третья команда, которая будет часто использоваться на курсе - `go run`

```
go run <path/to/go/file.go>
```

Данная команда делает следующие действия:

- Создает исполняемый файл в временном хранилище
- Запускает этот файл
- И "устарняет его" (зависит ОС)

Для того, чтобы узнать, где это временное хранилище `--work`

Шаг 6. Правильная структуризация рабочего окружения

Очень рекомендую создать следующий путь
`GOPATH/src/github.com/<your_github_username>/<github_repo_name>`

Лекция 2. Декларирование переменных и I/O

Шаг 1. Какая типизация?

В языке Go принята полустрогая статическая типизация.

Шаг 2. Способы декларирования переменных

Декларирование - это процесс связывания имени переменной с типом потенциального значения
При декларировании переменной автоматически происходит ее **инициализация НУЛЕВЫМ ЗНАЧЕНИЕМ ДЛЯ ЭТОГО ТИПА***.
Пример:

```
var age int
fmt.Println("My age is:", age)
```

Будет выведен 0.

Шаг 3. Декларирование и инициализация

Простейший случай единичной инициализации

```
//Декларирование и инициализация пользовательским значением
    var height int = 183
    fmt.Println("My height is:", height)

    //В чем "полустрогость" типзации?
    var uid = 12345
    fmt.Println("My uid:", uid)
    fmt.Printf("%T\n", uid)
```

Короткое присваивание

Оператор := слева от себя требует ***КАК МИНИМУМ ОДНУ НОВУЮ ПЕРЕМЕННУЮ***.

```
//Множественное присваивание через :=
    aArg, bArg := 10, 30
    fmt.Println(aArg, bArg)
    aArg, bArg = 30, 40
    fmt.Println(aArg, bArg)
    // aArg, bArg := 10, 30
    // fmt.Println(aArg, bArg)

    //Исключение из этого правила
    bArg, cArg := 300, 400
    fmt.Println(aArg, bArg, cArg)
```

```
package main

import (
    "fmt"
    "math"
)

func main() {
    //Простейший вывод на косноль. println - это вывод аргумента + '\n'
    fmt.Println("Hello world", "Hello another")
    fmt.Println("Second line")
    //Функция print - простой вывод аргумента
    fmt.Print("First")
    fmt.Print("Second")
    fmt.Print("Third")
    //Форматированный вывод: Printf - стандартный вывод os.Stdout с флагами
форматирования
    fmt.Printf("\nHello, my name is %s\nMy age is %d\n", "Bob", 42)
    ////////////////////////////////
    ////////////////////////////////
    //Декларирование переменных
    var age int
    fmt.Println("My age is:", age)
    age = 32
    fmt.Println("Age after assignment:", age)
```

```
    //Декларирование и инициализация пользовательским значением
    var height int = 183
    fmt.Println("My height is:", height)

    //В чем "полустрогость" типзации? Можно опускать тип переменной
    var uid = 12345
    fmt.Println("My uid:", uid)
    //Декларирование и инициализация переменных одного типа (множественный
случай)
    var firstVar, secondVar = 20, 30
    fmt.Printf("FirstVar:%d SecondVar:%d\n", firstVar, secondVar)
    //Декларирвоание блока переменных
    var (
        personName string = "Bob"
        personAge         = 42
        personUID  int
    )

    fmt.Printf("Name: %s\nAge %d\nUID: %d\n", personName, personAge, personUID)

    //Немного странного
    var a, b = 30, "Vova"
    fmt.Println(a, b)
    a = 300
    //Немного хорошего. Повторное декларирование переменной приводит к ошибке
компиляции
    //var a = 200

    //Короткая декларция (корткое объявление)
    count := 10
    fmt.Println("Count:", count)
    count = count + 1
    fmt.Println("Count:", count)
    //Множественное присваивание через :=
    aArg, bArg := 10, 30
    fmt.Println(aArg, bArg)
    aArg, bArg = 30, 40
    fmt.Println(aArg, bArg)
    // aArg, bArg := 10, 30
    // fmt.Println(aArg, bArg)

    //Исключение из этого правила
    bArg, cArg := 300, 400
    fmt.Println(aArg, bArg, cArg)

    //Пример
    width, length := 20.5, 30.2
    fmt.Printf("Min dimensional of rectangle is : %.2f\n", math.Min(width,
length))

}
```

```go
package main

import (
    "fmt"
    "os"
)

func main() {
    var (
        age  int
        name string
    )

    // fmt.Scan(&age)
    // fmt.Scan(&name)
    fmt.Scan(&age, &name)

    fmt.Printf("My name is: %s\nMy age is : %d\n", name, age)

    //Для ручного использования потока ввода
    fmt.Fscan(os.Stdin, &age)
    fmt.Println("New age:", age)
}
```

```go
package main

import (
    "fmt"
    "strings"
    "unicode/utf8"
    "unsafe"
)

func main() {
    //Boolean => default false
    var firstBoolean bool
    fmt.Println(firstBoolean)
    //Boolean operands
    aBoolean, bBoolean := true, true
    fmt.Println("AND:", aBoolean && bBoolean)
    fmt.Println("OR:", aBoolean || bBoolean)
    fmt.Println("NOT:", !aBoolean)

    //Numerics. Integers
    //int8, int16, int32, int64, int
    //uint8, uint16, uint32, uint64, uint
    var a int = 32
    b := 92
    fmt.Println("Value of a:", a, "Value of b:", b, "Sum of a+b:", a+b)
    //Вывод типа через %T форматирование
    fmt.Printf("Type is %T\n", a)
    //Узнаем, сколько байт занимает переемнная типа int
```

```
    fmt.Printf("Type %T size of %d bytes\n", a, unsafe.Sizeof(a))

    //Эксперимент. При использовании короткого объявления - тип для целого числа - int платформо-зависимый
    fmt.Printf("Type %T size of %d bytes\n", b, unsafe.Sizeof(b))

    //Эксперимент 2. Используйте явное приведение типов при необходимости если уверены что не произойдет коллиззии
    var first32 int32 = 12
    var second64 int64 = 13
    fmt.Println(int64(first32) + second64)

    //Эксперимент 3. Если проводятся арифметические операции
    // над int и intX , то обязательно нужно использовать механизм приведения. Т.к. int != int64
    var third64 int64 = 16123414
    var fourthInt int = 156234
    fmt.Println(third64 + int64(fourthInt))
    // + - * / %

    // Аналогичным образом утсроены unit8, uint16, uint32, uint64, uint
    //Numerics. Float
    //float32, float64
    floatFirst, floatSecond := 5.67, 12.54
    fmt.Printf("type of a %T and type of %T\n", floatFirst, floatSecond)
    sum := floatFirst + floatSecond
    sub := floatFirst - floatSecond
    fmt.Println("Sum:", sum, "Sub:", sub)
    fmt.Printf("Sum: %.3f and Sub: %.3f\n", sum, sub)

    //Numeric. Complex
    c1 := complex(5, 7)
    c2 := 12 + 32i
    fmt.Println(c1 + c2)
    fmt.Println(c1 * c2)

    //Strings. Строка - это набор БАЙТ
    name := "Федя"
    lastname := "Pupkin"
    concat := name + " " + lastname
    fmt.Println("Full name:", concat)
    fmt.Println("Length of string :", name, len(name)) // Функция len() возвращает количество элементов в наборе
    fmt.Println("Amount of chars:", name, utf8.RuneCountInString(name))
    //Rune - руна. Это один utf-ный символ.
    //Поиск подстроки в строке
    totalString, subString := "ABCDEDFG", "asd"
    fmt.Println(strings.Contains(totalString, subString))
    //rune -> alias int32
    var sampleRune rune
    var anotherRune rune = 'Q' // Для инициализации руны символьным значением - используйте ''
```

```
    var thirdRune rune = 234
    fmt.Println(sampleRune)
    fmt.Printf("Rune as char %c\n", sampleRune)
    fmt.Printf("Rune as char %c\n", anotherRune)
    fmt.Printf("Rune as char %c\n", thirdRune)
    // "A" < "abcd"
    fmt.Println(strings.Compare("abcd", "a")) // -1 if first < second, 0 if
first == second, 1 if first > second

    var aByte byte // alias uint8
    fmt.Println("Byte:", aByte)
}
```

```
package main

import (
    "fmt"
    "strings"
)

func main() {
    //Классический условный оператор
    //  if condition {
    //      //body
    //  }
    // }
    //  Условный оператор с блоком else
    // if condition {

    // } else {

    // }
    var value int
    fmt.Scan(&value)

    if value%2 == 0 {
        fmt.Println("The number", value, "is even")
    } else {
        fmt.Println("The number", value, "is odd")
    }
    // if condition1 {

    // } else if condition2 {

    // } else if ... {

    // } else {

    // }

    var color string
    fmt.Scan(&color)
```

```
    if strings.Compare(color, "green") == 0 {
        fmt.Println("Color is green")
    } else if strings.Compare(color, "red") == 0 {
        fmt.Println("Color is red")
    } else {
        fmt.Println("Unknown color")
    }

    //Good Инициализация в блоке условного оператора
    //Блок присваивания - только :=
    //Инициализируемая переменная видна ТОЛЬКО внутри области виидимости
условного оператора (в телах if, else if, или else)
    // Но не за его пределами
    if num := 10; num%2 == 0 {
        fmt.Println("EVEN")
    } else {
        fmt.Println("ODD")
    }

    //Ущербно
    /*
        var age int = 10
        if age > 7 {
            fmt.Println("Go to school")
        } //По факту, сюда подставляется ; компилятором, и дальнейший код уже
не имеет связи с предыдущим if
        else {
            fmt.Println("Another case")
        }
    */

    //НЕ ИДЕОМАТИЧНО
    if width := 100; width > 100 {
        fmt.Println("Width > 100")
    } else {
        fmt.Println("Width <= 100")
    }
    //Странное правило номер 1: в Go стараются избегать блоков ELSE

    //Идеоматичность
    if height := 100; height > 100 {
        fmt.Println("height > 100")
        return
    }
    fmt.Println("Height <= 100")
}
```

```
package main

import (
    "fmt"
    "strings"
```

```
)

func main() {
    // for init; condition; post {
    // init - блок инициализации переменных цикла
    // condition - условие (если верно - то тело цикла выполняется, если нет - то цикл завершается)
    // post - изменение переменной цикла (инкрементарное действие, декрементарное действие)
    // }

    for i := 0; i <= 5; i++ {
        fmt.Printf("Current value: %d\n", i)
    }
    //Важный момент : в качестве init может быть использовано выражение присваивания ТОЛЬКО через :=

    //break - команда, прерывающая текущее выполнение тела цикла и передающая управление инструкциям, следующим
    // за циклом
    for i := 0; i <= 15; i++ {
        if i > 12 {
            break
        }
        fmt.Printf("Current value: %d\n", i)
    }
    fmt.Println("After for loop with BREAK")

    //continue - команда, прерывающая текущее выполнение тела цикла и передающая управления СЛЕДУЮЩЕЙ ИТЕРАЦИИ ЦИКЛА
    for i := 0; i <= 20; i++ {
        if i > 10 && i <= 15 {
            continue
        }
        fmt.Printf("Current value: %d\n", i)
    }
    fmt.Println("After for loop with CONTINUE")

    //Вложенные циклы и лейблы
    for i := 0; i < 10; i++ {
        for j := 0; j <= i; j++ {
            fmt.Print("*")
        }
        fmt.Println()
    }
    fmt.Println("По идее выше треугольник")

    //Иногда бывает плохо. С лейблами по лучше. Лейблы - это синтаксический сахар
outer:
    for i := 0; i <= 2; i++ {
        for j := 1; j <= 3; j++ {
```

```
            fmt.Printf("i:%d and j:%d and sum i+j=%d\n", i, j, i+j)
            if i == j {
                break outer // Хочу чтобы вообще все циклы (внешние тоже
остановились)
            }
        }
    }
    //Модификации цикла for.
    //1. Классчиеский цикл while do
    var loopVar int = 0
    // while (loopVar < 10){
    //  ....
    //  loopVar++
    // }
    for loopVar < 10 {
        fmt.Printf("In while like loop %d\n", loopVar)
        loopVar++
    }
    //2. Классический бесконечный цикл
    var password string
outer2:
    for {
        fmt.Print("Insert password: ")
        fmt.Scan(&password)
        if strings.Contains(password, "1234") {
            fmt.Println("Weak password . Try again")
        } else {
            fmt.Println("Password Accepted")
            break outer2
        }
    }

    //3. Цикл с множественными переменными цикла
    for x, y := 0, 1; x <= 10 && y <= 12; x, y = x+1, y+2 {
        fmt.Printf("%d + %d = %d\n", x, y, x+y)
    }
}
```

```
package main

import "fmt"

func main() {
    //Switch!
    var price int
    fmt.Scan(&price)
    //В switch-case запрещены дублирующиеся условия в case"ах
    switch price {
    case 100:
        fmt.Println("First case")
    case 110:
        fmt.Println("Second case")
```

```
    case 120:
        fmt.Println("Third case")
    case 130:
        fmt.Println("Another case")
    default:
        //Отрабатывает только в том случае, если не один из выше перечисленных
кейсов - не сработал
        fmt.Println("Default case")

    }

    //Case с множеством вариантов
    var ageGroup string = "q" //Возрастнаые группы : "a", "b", "c", "d", "e"
    switch ageGroup {
    case "a", "b", "c":
        fmt.Println("AgeGroup 10-40")
    case "d", "e":
        fmt.Println("AgeGroup 50-70")
    default:
        fmt.Println("You are too yong/old")
    }

    // Case с выражениями
    var age int
    fmt.Scan(&age)

    switch {
    case age <= 18:
        fmt.Println("Too yong")
    case age > 18 && age <= 30:
        fmt.Println("Second case")
    case age > 30 && age <= 100:
        fmt.Println("Too old")
    default:
        fmt.Println("Who are you")
    }

    //Case с проваливаниями. Проваливания выполняют ДАЖЕ ЛОЖНЫЕ КЕЙСЫ
    //В момент выполнения fallthroug у следующего кейса не проверяется условие,
    //а сразу выполняется тело
    var number int
    fmt.Scan(&number)
outer:
    switch {
    case number < 100:
        fmt.Printf("%d is less then 100\n", number)
        if number%2 == 0 {
            break outer
        }
        fallthrough
    case number > 200:
        fmt.Printf("%d GREATER then 200\n", number)
```

```
        fallthrough
    case number > 1000:
        fmt.Printf("%d GREATER then 1000\n", number)
        fallthrough
    default:
        fmt.Printf("%d DEFAULT\n", number)
    }

    //Гадость с терминацией цикла for из switchv
    var i int
uberloop:
    for {
        fmt.Scan(&i)
        switch {
        case i%2 == 0:
            fmt.Printf("Value is %d and it's even\n", i)
            break uberloop
        }
    }

    fmt.Println("END")
}
```

```
package main

import "fmt"

func main() {
    //Массивы. Основа
    //1. Определение массива.
    //Создадим массив под хранение 5-ти целочисленных элементов
    var arr [5]int // При инициализации массива важно передать информацию -
сколько элементов в нем будет
    fmt.Println("This is my array:", arr)
    //2. Определение элементов массива (после предварительной инициализации)
    // Необходимо обратиться к элементу массива через синтаксис arr[i] = elem
    arr[0] = 10
    arr[1] = 20
    arr[3] = -500
    arr[4] = 720
    fmt.Println("After elemtns init:", arr)
    //3. Определние массива с указанием элементов на месте
    // Если при инициализации количество элементов меньше номинальной длины
массива
    // то недостающие элементы инициализируются нулями
    newArr := [5]int{10, 20, 30}
    fmt.Println("Short declaration and init:", newArr)
    //4. Создание массива через инициализацию переменных
    arrWithValues := [...]int{10, 20, 30, 40}
    fmt.Println("Arr declaration with [...]:", arrWithValues, "Length:",
len(arrWithValues))
    arrWithValues[0] = 10000
```

```
    fmt.Println("Arr declaration with [...]:", arrWithValues, "Length:",
len(arrWithValues))
    //5. Массив - это набор ЗНАЧЕНИЙ. То есть при всех манипуляциях - массив
копируется (жестко, на уровне компилятора)
    first := [...]int{1, 2, 3}
    second := first
    second[0] = 10000
    fmt.Println("First arr:", first)
    fmt.Println("Second arr:", second)
    //6. Массив и его размер - это две составляющие одного типа (Размер массив
- часть типа)
    // var aArr [5]int
    // var bArr [6]int
    // aArr[0] = 100
    // bArr = aArr
    // 7. Итерирование по массиву
    floatArr := [...]float64{12.5, 13.5, 15.2, 10.0, 12.0}
    for i := 0; i < len(floatArr); i++ {
        fmt.Printf("%d element of arr is %.2f\n", i, floatArr[i])
    }
    // 8. Итерирование по массиву через оператор range
    var sum float64
    for id, val := range floatArr {
        fmt.Printf("%d element of arr is %.2f\n", id, val)
        sum += val
    }
    fmt.Println("Total sum is:", sum)
    // 9. Игнорирование id в range based for цикле
    for _, val := range floatArr {
        fmt.Printf("%.2f value WO id\n", val)
    }
    // 10. Многомерные массивы
    words := [2][2]string{
        {"Bob", "Alice"},
        {"Victor", "Jo"},
    }
    fmt.Println("Multidimensional array:", words)
    // 11. Итерирование по многомерному массиву
    for _, val1 := range words {
        for _, val2 := range val1 {
            fmt.Printf("%s ", val2)
        }
        fmt.Println()
    }

}
```

```
package main

import "fmt"

func main() {
```

```go
    //1. Слайсы (они же - срезы)
    // Слайс - это динамическая обвязка над массивом.
    startArr := [4]int{10, 20, 30, 40}
    var startSlice []int = startArr[0:2] // Слайс инициализируется пустыми квадратными скобками
    fmt.Println("Slice[0:2]:", startSlice)
    // Создали слайс, основываясь уже на существующем массиве

    //2. Создание слайса без явной инициализации массива
    secondSlice := []int{15, 20, 30, 40}
    fmt.Println("SecondSlice:", secondSlice)

    //3. Измнение элементов среза
    originArr := [...]int{30, 40, 50, 60, 70, 80}
    firstSlice := originArr[1:4] // Это набор ссылок на элементы нижележащего массива
    for i, _ := range firstSlice {
        firstSlice[i]++
    }
    fmt.Println("OriginArr:", originArr)
    fmt.Println("FirstSlice:", firstSlice)

    //4. Один массив и два производных среза
    fSlice := originArr[:]
    sSlice := originArr[2:5]

    fmt.Println("Before modifications: Arr:", originArr, "fSlice:", fSlice, "sSlice:", sSlice)
    fSlice[3]++
    sSlice[1]++
    fmt.Println("After modifications: Arr:", originArr, "fSlice:", fSlice, "sSlice:", sSlice)

    //5. Срез как встроенный тип
    // type slice struct {
    //  Length int
    //  Capacity int
    //  ZeroElement *byte
    // }

    //6. Длина и емкость слайса
    wordsSilce := []string{"one", "two", "three"}
    fmt.Println("slice:", wordsSilce, "Length:", len(wordsSilce), "Capacity:", cap(wordsSilce))
    wordsSilce = append(wordsSilce, "four")
    fmt.Println("slice:", wordsSilce, "Length:", len(wordsSilce), "Capacity:", cap(wordsSilce))
    // Capacity (cap) или ёмкость слайса - это значение, показывающее СКОЛЬКО ЭЛЕМЕНТОВ В ПРИНЦИПЕ
    // можно добавить в срез БЕЗ ВЫДЕЛЕНИЯ ДОПОЛНИТЕЛЬНОЙ ПАМЯТИ ПОД НИЖЕЛЕЖАЩИЙ МАССИВ.
```

```
    // Допустим у нас есть срез на 3 элемента (инициализировали без явного
указания массива)
    // Компилятор при создании этого среза СНАЧАЛА создал массив ровно на 3
элемента
    // После этого компилятор вернул адрес, где этот масив живет
    // Срез запомнил этот адрес и теперь ссылается на него
    // Потом
    // Начинаем деформировать слайс (увеличим длину /увеличим количество
элементов)
    // Проблема - в нижележащем массиве (на котором основан слайс) все 3
ячейки. Что делать?
    // Компилятор ищет в памяти место для массива размера 3*2 (в общем случае
n*2, где n - первоначальный размер)
    // После того как место найдено (в нашем случае найдено место для 6
элементов), в это место копируются
    // старые 3 элемента на свои позиции. На 4-ую позицию мы добавляем новый
элемент
    // После этого компилятор возвращает нашему слайсу новый адрес в памяти,
где находится массив под 6 элементов.

    //Емкость всегда будет изменять как n*2.
    numerics := []int{1, 2}
    for i := 0; i < 200; i++ {
        if i%5 == 0 {
            fmt.Println("Current len:", len(numerics), "Current cap:",
cap(numerics))
        }
        numerics = append(numerics, i)
    }

    //Важно: после выделения памяти под новый массив, ссылки со старым будут
перенсены в новый
    // Пример
    numArr := [2]int{1, 2}
    numSlice := numArr[:]

    numSlice = append(numSlice, 3) // В этот момент numSlice больше не
ссылается на numArr
    numSlice[0] = 10
    fmt.Println(numArr)
    fmt.Println(numSlice)

    // 7. Как создавать слайсы наиболее эффективно?
    // make() - это функция, позволяющая более детально создавать срезы
    sl := make([]int, 10, 15)
    // []int - тип коллекции
    // 10 - длина
    // 15 - емкость
    //Сначала инициализируется arr = [15]int
    //Затем по нему делается срез arr[0:10]
    //После чего он возаращается
    fmt.Println(sl)
```

```
    // 8. Добавление элементов в СРЕЗ
    myWords := []string{"one", "two", "three"}
    fmt.Println("myWords:", myWords)
    anotherSlice := []string{"four", "five", "six"}
    myWords = append(myWords, anotherSlice...)
    myWords = append(myWords, "seven", "eight")
    fmt.Println("myWords:", myWords)
    //9. Многомерный срез
    mSlice := [][]int{
        {1, 2},
        {3, 4, 5, 6},
        {10, 20, 30},
        {},
    }
    fmt.Println(mSlice)

}
```

```
package main

import (
    "fmt"
    "unicode/utf8"
)

func main() {
    name := "Hello world"
    fmt.Println(name)

    //1. Строка - это байтовый слайс со своими особенносятми при обращении
    //к нижелажащему массиву
    word := "Тестовая строка"
    fmt.Printf("String %s\n", word)
    // Какие значения байт сейчас находятся в слайсе word?
    fmt.Printf("Bytes: ")
    for i := 0; i < len(word); i++ {
        fmt.Printf("%x ", word[i]) //%x - формат представления 16-ти ричного
байта
    }
    fmt.Println()
    // Каким образом получать доступ к отдельно стоящим символам?
    fmt.Printf("Characters: ")
    for i := 0; i < len(word); i++ {
        fmt.Printf("%c ", word[i]) //%c - формат представления символа
    }
    fmt.Println()
    //2. Строки в Go - хранятся как наборы UTF-8символов. Каждый символ, вообще
говоря, может занимать
    // больше чем 1 байт

    //3. Руна (Rune) - это стандартный встроенный тип в Go (alias над int32),
позволяющий хранить
```

```
    //единый неделимый UTF символ ВНЕ ЗАВИСИМОСТИ ОТ ТОГО сколько байт он
занимает
    fmt.Printf("Runes: ")
    runeSlice := []rune(word) // Преобразование слайса байт к слайсу рун
[]byte(sliceRune)
    for i := 0; i < len(runeSlice); i++ {
        fmt.Printf("%c ", runeSlice[i])
    }
    fmt.Println()
    //4. Итерирование по строке с использованием рун
    for id, runeVal := range word { // id - это индекс байта, с КОТОРОГО
НАЧИНАЕТСЯ РУНА runeVal
        fmt.Printf("%c starts at postion %d\n", runeVal, id)
    }

    //5. Создание строки из слайса байт
    myByteSlice := []byte{0x40, 0x41, 0x42, 0x43} // Исходное представление
байтов
    myStr := string(myByteSlice)
    fmt.Println(myStr)

    myDecimalByteSlice := []byte{100, 101, 102, 103} // Синтаксический сахар -
можно использовать десятичное представление байтов
    myDecimalStr := string(myDecimalByteSlice)
    fmt.Println(myDecimalStr)

    //6. Создание строки из слайса рун
    // Руны как hex
    runeHexSlice := []rune{0x45, 0x46, 0x47, 0x48}
    myStrFromRune := string(runeHexSlice)
    fmt.Println("From Runes(hex):", myStrFromRune)
    // Руны как литералы
    runeLiteralSlice := []rune{'V', 'a', 's', 'y', 'a'} // '' - таким образом
обозначается руна
    myStrFromRuneLiterals := string(runeLiteralSlice)
    fmt.Println("From Runes(literals):", myStrFromRuneLiterals)

    fmt.Printf("%s and %T\n", myStrFromRuneLiterals, myStrFromRuneLiterals)

    //7. Длина и емкость строки
    // Длина len() - количество байт в слайсе
    fmt.Println("Length of Вася:", len("Вася"), "bytes")
    // Длина RuneCounter - количество !рун!
    fmt.Println("Length of Вася:", utf8.RuneCountInString("Вася"), "runes")
    // Вычисление емкости строки - бессмысленно, т.к. строка базовый тип
    fmt.Println(cap([]rune("Вася")))

    //8. Сравнение строк == и !=. Начиная с go 1.6
    word1, word2 := "Вася", "Петя"
    if word1 == word2 {
        fmt.Println("Equal")
    } else {
```

```
        fmt.Println("Not equal")
    }

    //9. Конкатенация
    word3 := word1 + word2
    fmt.Println(word3)

    //10. Строитель строк (java -> StringBuiler)
    firstName := "Alex"
    secondName := "Johnson"
    fullName := fmt.Sprintf("%s #### %s", firstName, secondName)
    fmt.Println("FullName:", fullName)

    //11. Строки не изменяемые
    // fullName[0] = "Q"

    //12. А слайсы изменяемые :)
    fullNameSlice := []rune(fullName)
    fullNameSlice[0] = 'F'
    fullName = string(fullNameSlice)
    fmt.Println("String mutation:", fullName)

    //13. Сравнение рун
    if 'Q' == 'Q' {
        fmt.Println("Runes equal")
    } else {
        fmt.Println("Runes not equal")
    }

    //14. Где живут полезные методы работы со строками?
    // import "strings"

}
```

```go
package main

import (
    "bufio"
    "fmt"
    "os"
    "strconv"
)

func main() {
    var name string
    input := bufio.NewScanner(os.Stdin)
    if input.Scan() { // Команда захвата потока ввода и сохранения в буфер (захват идет до символа окончания строки)
        name = input.Text() // Команда возвращения элементов, помещенных в буфер (отдаст string)
    }
    fmt.Println(name)

    fmt.Println("For loop started:")
    for {
        if input.Scan() {
            result := input.Text()
            if result == "" {
                break
            }
            fmt.Println("Input string is:", result)
        }
    }

    //Преоброазование строкового литерала к чему-нибудь числовому
    numStr := "10"
    numInt, _ := strconv.Atoi(numStr) // Atoi - Anything to Int (именно int)
    fmt.Printf("%d is %T\n", numInt, numInt)

    numInt64, _ := strconv.ParseInt(numStr, 10, 64)
    numFloat32, _ := strconv.ParseFloat(numStr, 32) // Но это 64-разрядное число будет без проблем ГАРАНТИРОВАНО ПРЕВОДИТЬСЯ К 32
    fmt.Println(numInt64, numFloat32)
    fmt.Printf("%.3f and %T\n", numFloat32, float32(numFloat32))

}
```

```go
package main

import "fmt"

func main() {
    //1. Map - это набор пар ключ:значение. Инициализация пустой мапы
    mapper := make(map[string]int)
    fmt.Println("Empty map:", mapper)
```

```
    //2. Добавление пар в существующую мапу
    mapper["Alice"] = 24
    mapper["Bob"] = 25
    fmt.Println("Mapper after adding pairs:", mapper)

    //3. Инициализация мапы с указанием пар
    newMapper := map[string]int{
        "Alice": 1000,
        "Bob":   1000,
    }
    newMapper["Jo"] = 3000
    fmt.Println("New Mapper:", newMapper)

    //4. Что может быть включом в мапе?
    //4.1 ВАЖНО: Мапа НЕ УПОРЯДОЧЕНА В GO
    //4.2 КЛЮЧОМ В МАПЕ МОЖЕТ БЫТЬ ТОЛЬКО СРАВНИМЫЙ ТИП (==, !=)

    //5. Нулевое значение для map
    // var mapZeroValue map[string]int // mapZeroValue == nil
    // mapZeroValue["Alice"] = 12

    //6. Получение элементов из map
    //6.1 Получение элемента, который представлен в мапе
    testPerson := "Alice"
    fmt.Println("Salary of testPerson:", newMapper[testPerson])
    //6.2 Получение элемента, который НЕ представлен в мапе
    testPerson = "Derek"
    fmt.Println("Salary of new testPerson:", newMapper[testPerson]) // При
обращении по несуществующему ключу - новая пара не добавляется
    fmt.Println(newMapper)

    //7. Проверка вхождения ключа
    employee := map[string]int{
        "Den":   0,
        "Alice": 0,
        "Bob":   0,
    }

    //7.1 При обращении по ключу - возвращается 2 значения
    if value, ok := employee["Den"]; ok {
        fmt.Println("Den and value:", value)
    } else {
        fmt.Println("Den does not exists in map")
    }

    if value, ok := employee["Jo"]; ok {
        fmt.Println("Jo and value:", value)
    } else {
        fmt.Println("Jo does not exists in map")
    }

    //8. Перебор элементов мапы
```

```
    fmt.Println("============================")
    for key, value := range employee {
        fmt.Printf("%s and value %d\n", key, value)
    }

    //9. Как удалять пары
    //9.1 Удаление существующей пары
    fmt.Println("Before deleting:", employee)
    delete(employee, "Den")
    fmt.Println("After first deleting:", employee)

    //9.2 Удаление не существующей пары
    if _, ok := employee["Anna"]; ok {
        delete(employee, "Anna") // ОЧЕНЬ ДОРОГАЯ ОПЕРАЦИЯ
    }

    fmt.Println("After second deleting:", employee)

    //10. Количество пар == длина мапы
    fmt.Println("Pair amount in map:", len(employee))

    //11. Мапа (как и слайс) ссылочный тип
    words := map[string]string{
        "One": "Один",
        "Two": "Два",
    }

    newWords := words
    newWords["Three"] = "Три"
    delete(newWords, "One")
    fmt.Println("words map:", words)
    fmt.Println("newWords map:", newWords)

    //12 . Сравнение мап
    //12.1 Сравнение массивов (массив можно использовать как ключ в мапе)
    if [3]int{1, 2, 3} == [3]int{1, 2, 3} {
        fmt.Println("Equal")
    } else {
        fmt.Println("Not equal")
    }
    //12.2 Сравнение слайсов. Слайсы (из-за того что тип ссылочный - можно
сравнить на равенство только с nil)
    // if []int{1, 2, 3} == []int{1, 2, 3} {
    //  fmt.Println()
    // }

    //12.3 Сравнение мап.  Мапа (из-за того, что тип ссылочный - можно
сравнивать только с nil)
    aMap := map[string]int{
        "a": 1,
    }
    var bMap map[string]int
```

```
    if aMap == nil {
        fmt.Println("Zero value map")
    }

    if bMap == nil {
        fmt.Println("Zero value of map bMap")
    }

    //13. Грустное следствие
    // Если мапа/слайс являются составляющими какой-либо структуры - структура
автоматически не сравнима

}
```

```
package main

import "fmt"

//1. Явная функция - локально-определенный блок кода , имеющий имя (ЯВНОЕ
ОПРЕДЕЛЕНИЕ)
// Функцию необходимо определить + Функциию необходимо вызвать
//2. Сигнатура функций и их определение
// func functionName(params type) typeReturnValue {
//  //body
// }

func main() {
    fmt.Println("Hello world")
    //4. Вызов простейшей функции
    res := add(10, 20)
    fmt.Println("Result of add(10, 20):", res)
    fmt.Println("Result of mult(1, 2, 3, 4):", mult(1, 2, 3, 4))
    per, area := rectangleParameters(10.5, 10.5)
    newPer, _ := rectangleParameters(10, 10)
    fmt.Println("Area of rect(10.5, 10.5):", area)
    fmt.Println("Perimeter of rect(10.5, 10.5):", per)
    fmt.Println("NewPer:", newPer)
    secondPer, secondArea := namedReturn(10, 20)
    fmt.Println(secondArea, secondPer)
    emptyReutrn(10)
    helloVariadic(10, 20, 30, 40, 50, 60, 60)
    helloVariadic()
    someStrings(2, 3)
    sum1 := sumVariadic(10, 30, 40, 50, 60)
    sliceNumber := []int{10, 20, 30}
    sum2 := sumVariadic(sliceNumber...)
    fmt.Println(sum1, sum2)

    fmt.Println(sumSlice([]int{30, 40, 50, 60, 80, 90, 100}))
    fmt.Println(sumSlice(sliceNumber))
```

```
    //12. Анонимная функция (синтаксис)
    anon := func(a, b int) int {
        return a + b
    }

    fmt.Println("Anon:", anon(20, 30))
    fmt.Println("BigFunction(10, 20):", bigFunction(10, 20))

}

//13. Анонимная функция внутри явной
func bigFunction(aArg, bArg int) int {
    return func(a, b int) int { return a + b + 1 }(aArg, bArg)
}

//3. Простейшая функция (определить функцию можно как до момента ее вызова в
функции main,
// так и в любом месте пакета, главное чтобы она была определена в принципе
где-нибудь)
func add(a int, b int) int {
    result := a + b
    return result
}

//4. Функция с однотипными параметрами
func mult(a, b, c, d int) int {
    result := a * b * c * d
    return result
}

//5. Возврат больше чем одного значения (returnType1, returnType2.......)
func rectangleParameters(length, width float64) (float64, float64) {
    var perimeter = 2 * (length + width)
    var area = length * width

    return perimeter, area
}

//6. Именованный возврат значений
func namedReturn(a, b int) (perimeter int32, area int) {
    perimeter = int32(2 * (a + b))
    area = a * b
    return // Не нужно указывать возвращаемые переменные
}

//7. При вызове оператора return функцию прекращает свое выполнение и
возвращает что-то
func funcWithReturn(a, b int) (int, bool) {
    if a > b {
        return a - b, true
    }
```

```
    if a == b {
        return a, true
    }

    return 0, false
}

//8. Что вернется в случае, если return в принципе не указан (или он пустой)
func emptyReutrn(a int) {
    fmt.Println("I'M emptyReturn with parameter:", a)
}

//9. Variadic parameters (континуальные параметры)
func helloVariadic(a ...int) {
    fmt.Printf("value %v and type %T\n", a, a)
}

//10. Смешение параметров с variadic (
//  1. Континуальыний параметр всегда самый последний
//  2. Variadic параметр - на всю функцию один (для удобочитаемости)
// )
func someStrings(a, b int, words ...string) {
    fmt.Println("Parameter:", a)
    fmt.Println("Parameter:", b)
    var result string
    for _, word := range words {
        result += word
    }
    fmt.Println("Result concat:", result)
}

//11. Передача слайса или использование variadic parameters?
func sumVariadic(nums ...int) int {
    var sum int
    for _, val := range nums {
        sum += val
    }
    return sum
}

func sumSlice(nums []int) int {
    var sum int
    for _, val := range nums {
        sum += val
    }
    return sum
}
```

```
package main

import "fmt"
```

```
//1. Явные функции (в принципе любая функция в Go) - является
//экземлпяром 1-го уровня (функцию можно присваивать в перменную, ее можно
передавать в
//качестве параметра и возвращать из других функций)

//2. Возврат функции в качестве значения
func calcAndReturnValidFunc(command string) func(a, b int) int {
    if command == "addition" {
        return func(a, b int) int { return a + b }
    } else if command == "substraction" {
        return func(a, b int) int { return a - b }
    } else {
        return func(a, b int) int { return a * b }
    }
}

//3. Функция как парметр в другой функции
func recieveFuncAndReturnValue(f func(a, b int) int) int {
    var intVarA, intVarB int
    intVarA = 100
    intVarB = 200

    return f(intVarA, intVarB)
}

func add(a, b int) int {
    return a + b
}
func main() {

    var command string
    command = "substraction"
    res := calcAndReturnValidFunc(command)
    fmt.Println("Result with command :", command, "value:", res(10, 20))
    fmt.Println(res(30, 40))

    //4. Тип функции
    fmt.Printf("Type of func is %T\n", res)
    fmt.Printf("Type of calcAndReturnValidFunc is %T\n",
calcAndReturnValidFunc)
    //5. Тип функции в Go определяется как входными парамтерами, так и
выходными

    fmt.Println("recieveFuncAndReturnValue(add):",
recieveFuncAndReturnValue(add))
    fmt.Println(recieveFuncAndReturnValue(func(a, b int) int {
        return a*a + b*b + 2*a*b
    }))

}
```

```
package main
```

```go
import (
    "fmt"
    "math"
)

//1. Константы - это неизменяемые переменные, которые служат для:
//  1) Более строго опнимания кода
//  2) Для того, чтобы случайно не поменять значение (предполагается что
значение константны не изменно)
//  3) Для удобных преобразований

const (
    MAIN_PORT = "8001"
)

func main() {
    //2. Объвление одной константы
    const a = 10
    fmt.Println(a)
    //3. Объявление блока констант с областью видимости внутри функции main
    const (
        ipAddress string = "127.127.00.03"
        port             = "8000"
        dbName           = "postgres"
    )
    fmt.Println("ipAddress value:", ipAddress)
    fmt.Println(checkPortIsValid())

    //4. Константу никак нельзя поменять в ходе работы программы
    // const b = 200
    // b = 30

    //5. Значения констант ДОЛЖНЫ БЫТЬ ИЗВЕСТНЫ на момент компиляции
    var sqrt = math.Sqrt(25)
    //const sqrt = math.Sqrt(25) //Нельзя присвоить в константу что-либо, что
является результатом вызова функции, метода
    fmt.Println("Var sqrt:", sqrt)

    //6. Типизированные и нетипизированные константы
    const ADMIN_EMAIL string = "admin@admin.com" // Указание типа - повышение
читабельности кода

    //7. Нетипизирвоанные константы и их профит
    //При использовании нетипизированных констант мы разрешаем компилятору
    //исопльзовать неявное приведение типов в момент присваивания значений
константа в перменные
    const NUMERIC = 10
    var numInt8 int8 = NUMERIC
    var numInt32 int32 = NUMERIC
    var numInt64 int64 = NUMERIC
    var numFloat32 float32 = NUMERIC
```

```
    var numComplex complex64 = NUMERIC

    fmt.Printf("numInt8 value %v type %T\n", numInt8, numInt8)
    fmt.Printf("%v + %v is %v\n", numInt8, NUMERIC, numInt8+NUMERIC)
    fmt.Printf("numInt32 value %v type %T\n", numInt32, numInt32)
    fmt.Printf("numInt64 value %v type %T\n", numInt64, numInt64)
    fmt.Printf("numFloat32 value %v type %T\n", numFloat32, numFloat32)
    fmt.Printf("numComplex value %v type %T\n", numComplex, numComplex)
    //8. Константы в Go зашиваются в момент компиляции в RUNTIME программы и не
выбрасываются до ее окончания

}

func checkPortIsValid() bool {
    if MAIN_PORT == "8001" {
        return true
    }
    return false
}
```

```
package main

import "fmt"

//1. Указатели - переменная, хранящая в качестве значения - адрес в памяти
другой переменной

func main() {

    //2. Определение указателя на что-то
    var variable int = 30
    var pointer *int = &variable //&.... - операция взятия адреса в памяти
    //Выше у нас создан указатель на переменную variable
    //В pointer лежит 18293xcd000132 - это место в памяти, где хранится int
значение 30
    fmt.Printf("Type of pointer %T\n", pointer)
    fmt.Printf("Value of pointer %v\n", pointer)

    //3. А какое нулевое значение для указатели?
    var zeroPointer *int //zeroValue имеет значение nil (это указатель в
никуда)
    fmt.Printf("Type %T and value %v\n", zeroPointer, zeroPointer)
    if zeroPointer == nil {
        zeroPointer = &variable
        fmt.Printf("After initializatoin type %T and value %v\n", zeroPointer,
zeroPointer)
    }

    //4. Разыменование указателя (получение значения): *pointer - возвращает
значение, хранимое по адресу
    var numericValue int = 32
    pointerToNumeric := &numericValue
```

```
    fmt.Printf("Value in numericValue is %v\nAddress is %v\n",
*pointerToNumeric, pointerToNumeric)

    //5. Создание указателей на нулевые занчения типов
    // var zeroVar int
    // var zeroPoint *int = &zeroVar
    zeroPoint := new(int) // Создает под капотом zeroValue для int, и
возвращает адрес, где этот 0 хранится
    fmt.Printf("Value in *zeroPointer %v\nAddress is %v\n", *zeroPoint,
zeroPoint)

    //6. Изменение значения хранимого по адресу через указатель
    zeroPointerToInt := new(int)
    fmt.Printf("Addres is %v and Value in zeroPointerToInt is %v\n",
zeroPointerToInt, *zeroPointerToInt)
    *zeroPointerToInt += 40
    fmt.Printf("Addres is %v and New Value in zeroPointerToInt is %v\n",
zeroPointerToInt, *zeroPointerToInt)

    b := 345
    a := &b
    c := &b
    *a++
    *c += 100
    fmt.Println(b)
    //7. Указательная арфиметика ОТСУТСТВУЕТ ПОЛНОСТЬЮ
    // У вас на руках адрес одной ячейки - вы можете через этот адрес
продвинуться в другие ячейки
    //

    //8. Передача указателей в функции
    // Колоссальный прирост производительности засчет того, что передается не
значение (которые должно копироваться)
    // а передается лишь адрес в памяти, за которым уже хранится какое-то
значение
    sample := 1
    //samplePointer := &sample

    fmt.Println("Origin value of sample:", sample)
    changeParam(&sample)
    fmt.Println("After changing sample is:", sample)

    //9. Возврат поинтера из функции (В C++ результат работы такого механизма -
неопределен)
    ptr1 := returnPointer()
    ptr2 := returnPointer()
    fmt.Printf("Ptr1: %T and address %v and value %v\n", ptr1, ptr1, *ptr1)
    fmt.Printf("Ptr2: %T and address %v and value %v\n", ptr2, ptr2, *ptr2)

}
```

```go
//9.1  Инициализация функции, возвращающей указатель
func returnPointer() *int {
    var numeric int = 321
    return &numeric //В момент возврата Go перемещает данную переменную в кучу
}

//8.1 Определдение фукнции, принимающей параметр как указатель
func changeParam(val *int) {
    *val += 100
}
```

```go
package main

import "fmt"

//1. Указатели на массивы. Почему так делать не надо
func mutation(arr *[3]int) {
    // (*arr)[1] = 909
    // (*arr)[2] = 100000
    //Можно написать и так, т.к. Go сам разыменует указатель на массив (из-за того,что функци принимает *arr)
    arr[1] = 909
    arr[2] = 10000
}

//2. Используйте лучше слайсы (это идеоматично с точки зрения Go)
func mutationSlice(sls []int) {
    sls[1] = 909
    sls[2] = 10000
}

func main() {
    arr := [3]int{1, 2, 3}
    fmt.Println("Arr before mutation:", arr)
    mutation(&arr)
    fmt.Println("Arr after mutation:", arr)

    newArr := [3]int{1, 2, 4}
    fmt.Println("newArr before mutationSlice:", newArr)
    mutationSlice(newArr[:])
    fmt.Println("newArr after mutationSlcie:", newArr)
}
```

```go
package main

import "fmt"

//1. Структура - заименованный набор полей (состояний), определяющий новый тип данных.

//2. Определение структуры (явное определение)
type Student struct {
```

```
    firstName string
    lastName  string
    age       int
}

//3. Если имеется ряд состояний одного типа, можно сделать так
type AnotherStudent struct {
    firstName, lastName, groupName string
    age, courseNumber              int
}

//11. Структура с анонимными полями
type Human struct {
    firstName string
    lastName  string
    string
    int
    bool
}

func PrintStudent(std Student) {
    fmt.Println("==================")
    fmt.Println("FirstName:", std.firstName)
    fmt.Println("LastName:", std.lastName)
    fmt.Println("Age:", std.age)
}

func main() {

    //4. Создание представителей структуры
    stud1 := Student{
        firstName: "Fedya",
        age:       21,
        lastName:  "Petrov",
    }
    PrintStudent(stud1)
    stud2 := Student{"Petya", "Ivanov", 19} // Порядок указания свойств - такой
же как в структуре
    PrintStudent(stud2)

    //5. Что если не все поля структуры определить?
    stud3 := Student{
        firstName: "Vasya",
    }
    PrintStudent(stud3)

    //6. Анонимные структуры (структура без имени)
    anonStudent := struct {
        age           int
        groupID       int
        proffesorName string
    }{
```

```
        age:           23,
        groupID:       2,
        proffesorName: "Alexeev",
    }
    fmt.Println("AnonStudent:", anonStudent)

    //7. Доступ к состояниям и их модфикация
    studVova := Student{"Vova", "Ivanov", 19}
    fmt.Println("firstName:", studVova.firstName)
    fmt.Println("lastName:", studVova.lastName)
    fmt.Println("age:", studVova.age)
    studVova.age += 2
    fmt.Println("new age:", studVova.age)

    //8. Инициализация пустой структуры
    emptyStudent1 := Student{}
    var emptyStudent2 Student
    PrintStudent(emptyStudent1)
    PrintStudent(emptyStudent2)

    //9. Указатели на экземпляры структур
    studPointer := &Student{
        firstName: "Igor",
        lastName:  "Sidorov",
        age:       22,
    }
    fmt.Println("Value studPointer:", studPointer)
    secondPointer := studPointer
    (*secondPointer).age += 20
    fmt.Println("Value afterPointerModify:", studPointer)
    studPointerNew := new(Student)
    fmt.Println(studPointerNew)

    //10. Работа с доступ к полям структур через указатель
    fmt.Println("Age via (*...).age:", (*studPointer).age)
    fmt.Println("Age via .age:", studPointer.age) //Неявно происходит
разыменование указателя studpointer и запрос соотв поля

    //12. Создание экземпляра с анонимными полями структуры
    human := &Human{
        firstName: "Bob",
        lastName:  "Johnson",
        string:    "Additional Info",
        int:       -1,
        bool:      true,
    }

    fmt.Println(human)
    fmt.Println("Anon field string:", human.string)
}
```

```
package main
```

```go
import "fmt"

//1. Вложенные структуры (вложение структур). Это использование одной
структуры, как тип поля
//в другйо структуре
type University struct {
    age       int
    yearBased int
    infoShort string
    infoLong  string
}

type Student struct {
    firstName  string
    lastName   string
    university University
}

//4. Встроенные структуры (когда мы добавляем поля одной структуры к другой)
type Professor struct {
    firstName string
    lastName  string
    age       int
    greatWork string
    //papers     map[string]string - добавление этого поля делает структуру
несравнимой
    University // В этом месте происходит добавление всех полей структуры Uni в
Professor
}

func main() {
    //2. Создание экземпляров структур с вложением
    stud := Student{
        firstName: "Fedya",
        lastName:  "Petrov",
        university: University{
            yearBased: 1991,
            infoShort: "cool University",
            infoLong:  "very cool University",
        },
    }

    //3. Получение доступа к вложенным полям структур
    fmt.Println("FirstName:", stud.firstName)
    fmt.Println("LastName:", stud.lastName)
    fmt.Println("Year based Uni:", stud.university.yearBased)
    fmt.Println("Long info:", stud.university.infoLong)

    //5. Создание экземпляра с встраиванием структур
    prof := Professor{
        firstName: "Anatoly",
```

```
        lastName:  "Smirnov",
        age:       125,
        greatWork: "Ultimate C programming",
        University: University{
            yearBased: 1734,
            infoShort: "short Info",
            age:       2021 - 1734,
        },
    }
    //6. Обращение к состояниям с встроенной структурой
    fmt.Println("FirstName:", prof.firstName)
    fmt.Println("Year based:", prof.yearBased)
    fmt.Println("Info Short:", prof.infoShort)
    fmt.Println("Age:", prof.University.age) //prof.age - получим доступ к полю ВЫШЕЛЕЖАЩЕЙ СТРУКТУРЫ

    //7. Сравнение экземпляров ==
    //При сравнении экзмеляров происходит сравнение всех их полей друг с другом
    profLeft := Professor{}
    profRight := Professor{}

    fmt.Println(profLeft == profRight)
    //8. Если ХОТЯ БЫ ОДНО ИЗ ПОЛЕЙ СТРУКТУР - НЕ СРАВНИМО - то и вся структура несравнима
}
```

<!-- 07 Lec18 -->

<!-- markdown-pdf -o summary.pdf summary.md -->