

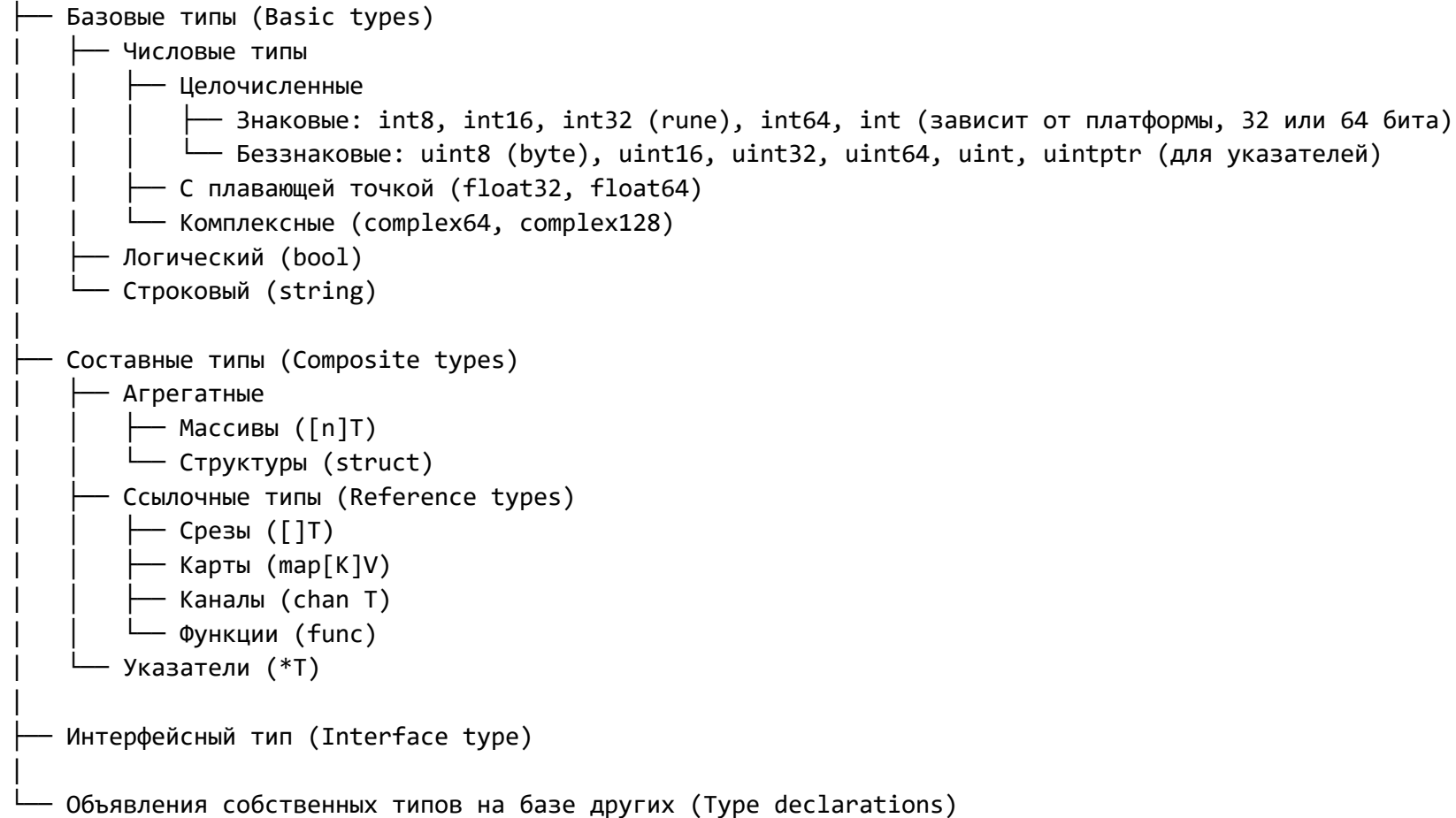
# Программирование на языке Go

## Лекция 6

Go

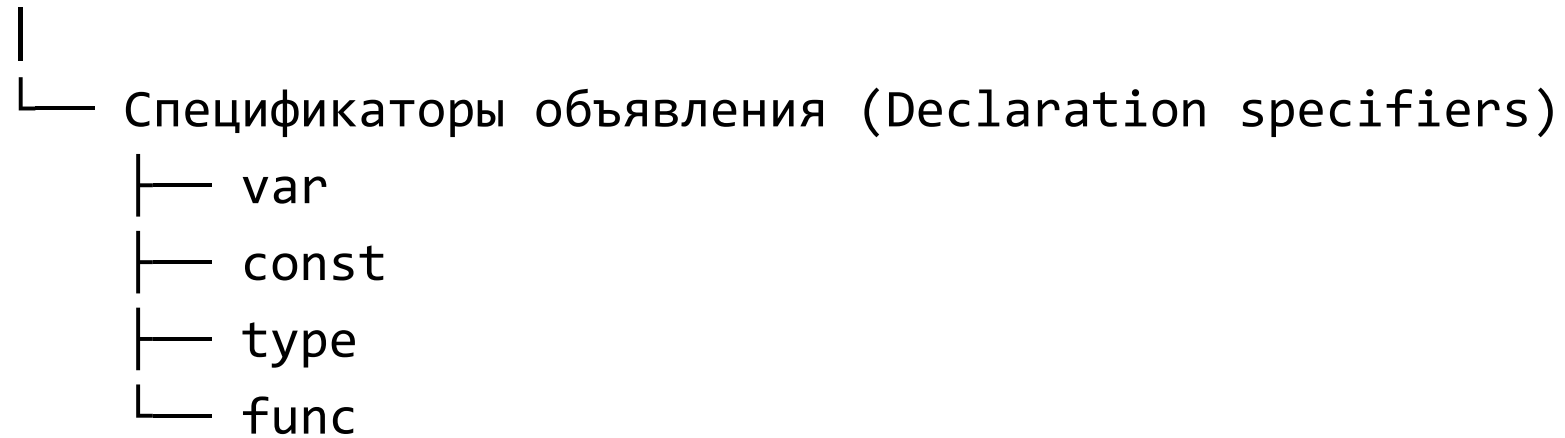
# Типы данных Go

## Типы данных Go



# Спецификаторы типа в Go

Спецификаторы и модификаторы в Go



Переменные

# Переменные

```
// Явное объявление с типом и инициализатором
var age int = 1
var name string = "John"
var score, data int = 10, 20 // OK
var score, data int = 10, "20" // Ошибка
var score int = 10, data string = "20" // Ошибка

// Явное объявление с типом, без инициализатора (гарантируется дефолтное значение)
var age int // 0
var name string // ""
var score, data int // OK
var score int, data string // Ошибка

// Явное объявление без типа, но с инициализатором (тип выводится компилятором)
var age = 1
var name = "John"
var score, data = 100, "2.5" // OK
var score = 100, data = "2.5" // Ошибка

// Короткое объявление (тип выводится компилятором). Только для локальных переменных
score := 95 // int
pi := 3.14 // float64
score, data := 100, "2.5" // data будет создано со значением 2.5, score перезаписано на 100
score, data := 200, "3.5" // Ошибка. Все переменные уже существуют
```

# Переменные – групповое объявление

```
var (  
    pi = 3.14  
    _2pi, _3pi = 2 * pi, 3 * pi  
    score int = 1  
    data string  
    //err := 1 // Ошибка  
)
```

# Константы

# Константы

```
// Типизированные константы  
const Pi float64 = 3.14159 // явный тип  
const Prefix string = "app_"
```

```
// Нетипизированные константы  
const Pi = 3.14159  
const MaxUsers = 1000  
const AppName = "MyApp"
```

```
// Групповое объявление  
const (  
    StatusOK = 200  
    StatusNotFound = 404  
    StatusError = 500  
)
```



## Константы – только compile time

```
const (  
    Width  = 1920  
    Height = 1080  
    Pixels = Width * Height           // ДА: Вычисляется на этапе компиляции  
    alsoValid = len("constant string") // ДА: len от строки - константа  
)
```

```
var dynamicVar = 30  
const (  
    invalid = 10 + dynamicVar           // НЕТ: переменные времени выполнения  
    alsoInvalid = len(someVar)          // НЕТ: len от переменной  
)
```

// Функции при объявлении констант использовать нельзя (только некоторые)

# Что может быть константой

// Допустимые типы для констант:

```
const (  
    num      = 42           // целочисленная  
    float    = 3.14         // вещественная  
    str       = "hello"     // строковая  
    boolean  = true         // булева  
    rune     = 'A'          // руна  
    complex  = 1 + 2i       // комплексная  
)
```

// Недопустимые типы:

```
// - Массивы  
// - Срезы  
// - Карты  
// - Структуры  
// - Указатели  
// - Функции  
// - Каналы  
// - Интерфейсы
```

// Константы могут быть созданы только отдельно. Константных параметров, полей и т.д. нет

# Нетипизированные константы

```
const exactValue = 1234567890123456789 // Может быть больше int64!  
const exactValue2 = exactValue * exactValue // OK
```

```
var i int = exactValue // Ошибка компиляции если не влезает  
var i64 int64 = exactValue // Ошибка компиляции если не влезает  
var f64 float64 = exactValue // Может потерять точность
```

```
// У каждой нетипизированной константы есть базовый тип  
// Нетипизированные константы могут подстраиваться под тип, если он совпадает с базовым
```

```
type MyInt int
```

```
const val = 1  
var i = val // OK  
var mi MyInt = i // Ошибка. Разные типы  
var mi MyInt = val // OK
```

# Спецификаторы хранения

# Спецификаторы хранения

В go нет спецификаторов хранения, т.к. в них нет необходимости. Среда исполнения go сама определяет где создавать переменную на стеке или в куче. За удаление переменных отвечает сборщиком мусора.

# type

Ключевое слово `type` используется для объявления типов:

- `type` Имя `struct` { ... } - создание структуры
- `type` Имя существующий\_тип - создание нового типа
- `type` Имя = существующий\_тип - создание псевдонима (Go 1.9+)
- `type` Имя `interface` { ... } - создание интерфейса

## Составные типы

# Статические массивы

Объявление статического массива

```
var name [/* размер */] /* тип */
```

// Примеры объявления

```
var scores [5]int // Массив из 5 целых чисел
```

```
var temperatures [24]float64 // Массив из 24 чисел с плавающей точкой
```

```
var name [50]string // Массив из 50 строк
```

// Размер под статические массивы выделяется менеджером памяти

// для небольших на стеке, для больших в куче



## Массивы переменной длины (VLA) – в go не может быть

```
var size = 10;  
var array [size]int // Ошибка
```

```
const size = 10  
var array [size]int // OK
```

```
var array [10]int // OK
```

# Инициализация статических массивов

// Полная инициализация

```
var numbers [5]int = [5]int{1, 2, 3, 4, 5}  
numbers2 := [5]int{1, 2, 3, 4, 5}
```

// Частичная инициализация (остальные элементы - zero value)

```
partial := [5]int{1, 2} // [1, 2, 0, 0, 0]
```

// Автоматическое определение размера

```
auto_size := [...]int{1, 2, 3, 4} // Размер = 4
```

// Инициализация по индексам

```
sparse := [5]int{0: 100, 3: 400} // [100, 0, 0, 400, 0]
```

// Инициализация нулями

```
zeros    := [5]int{}    // [0, 0, 0, 0, 0]  
zeros2   := [5]int{0}   // [0, 0, 0, 0, 0]  
var zeros2 [5]int       // [0, 0, 0, 0, 0]
```

## Доступ к элементам статического массива

```
arr := [5]int{10, 20, 30, 40, 50}
```

```
// Чтение значений
```

```
first := arr[0]    // 10 (первый элемент)
```

```
last  := arr[4]    // 50 (последний элемент)
```

```
// Запись значений
```

```
arr[1] = 25        // Теперь arr[1] = 25
```

```
arr[3] = arr[2] * 2 // arr[3] = 60
```

```
// Выход за границы массива
```

```
// arr[5] = 100      // Ошибка
```

```
// arr[-1] = 100     // Ошибка
```

## Определение размера статического массива

```
arr := [...]int{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}  
var size int = len(arr)      // 10  
var max_size int = cap(arr) // 10
```

// Размер - часть типа

```
var a [3]int
```

```
var b [3]int
```

```
var c [5]int
```

```
a = b // ОК копирование
```

```
c = a // Ошибка. Разные типы
```

# Сравнение статических массивов

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

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

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

```
fmt.Println(a == b) // true
```

```
fmt.Println(a == c) // false
```

```
fmt.Println(a > c)  // ОШИБКА компиляции
```

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

```
fmt.Println(a == x) // ОШИБКА компиляции. Массивы разных размеров нельзя сравнивать
```

# Обход статического массива

```
numbers := [...]int{10, 20, 30, 40, 50}

// По индексу (для изменения значений использовать этот цикл)
for i := 0; i < len(numbers); i++ {
    fmt.Printf("Index %d: %d\n", i, numbers[i])
}

// Range loop (получаем копии)
for index, value := range numbers {
    fmt.Printf("Index %d: %d\n", index, value)
}

// Только значения
for _, fruit := range numbers {
    fmt.Println(fruit)
}

// Только индексы
for i := range numbers {
    fmt.Println(i)
}
```

# Особенности захвата переменной в цикле

```
package main
import "fmt"

func main(){
    values := [...]int{1, 2, 3}
    funcs := make([]func() int, 0)

    for _, val := range values {
        funcs = append(funcs, func()int {
            return val    // Это может вывести не то, что вы ожидаете!
        })
    }

    values[2] = 10
    for _, val := range funcs {
        fmt.Println(val()) // Каждый раз будет напечатано 3
    }
}
```

// Исправление циклов for в Go 1.22: <https://go.dev/blog/loopvar-preview>

## Передача статического массива в функцию по значению

// Статические массивы передаются по ЗНАЧЕНИЮ (копируются)

```
func modifyArray(arr [3]int) {  
    arr[0] = 100 // изменяется копия  
    fmt.Println(arr) // [100, 2, 3]  
}
```

```
func main() {  
    numbers := [3]int{1, 2, 3}  
    modifyArray(numbers)  
    fmt.Println(numbers) // [1, 2, 3] - оригинал не изменился  
}
```



## Передача статического массива в функцию по указателю

// Передача по указателю, чтобы избежать копирования

```
func modifyArrayByPointer(arr *[3]int) {  
    arr[0] = 100 // синтаксис такой же!  
    // Эквивалентно: (*arr)[0] = 100  
}
```

```
func main() {  
    numbers := [3]int{1, 2, 3}  
    modifyArrayByPointer(&numbers)  
    fmt.Println(numbers) // [100, 2, 3]  
}
```

## Передача статического массива в функцию - дженерик

```
func modifyArrayByPointer[N int](arr *[N]int) {  
    // Нельзя  
}
```

# Возврат статического массива из функции по значению

```
package main

import "fmt"

func newArray() [3]int {
    arr := [3]int{1, 2, 3}
    return arr
}

func main() {
    dst := newArray()
    fmt.Println(dst) // [1, 2, 3]
}

// dst получает копию локального массива arr
// Скорее всего они оба будут созданы на стеке
```

# Возврат статического массива из функции по указателю

```
package main
```

```
import "fmt"
```

```
func newArray() *[3]int {  
    arr := [3]int{1, 2, 3}  
    return &arr  
}
```

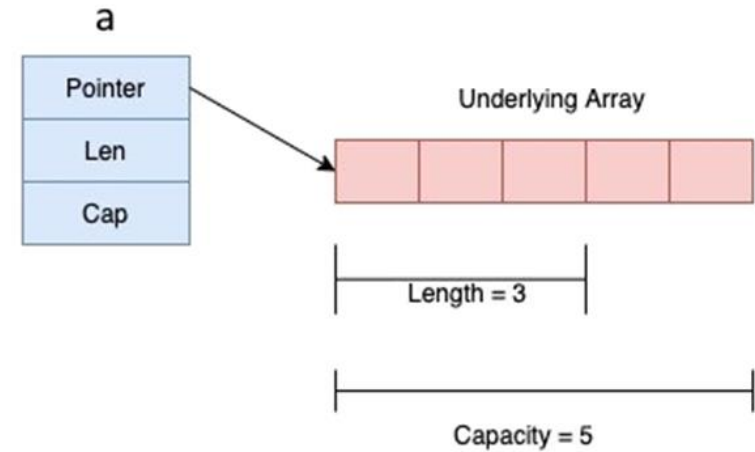
```
func main() {  
    dst := newArray()  
    fmt.Println(*dst) // [1, 2, 3]  
}
```

```
// dst получает адрес массива arr (безопасно, т.к. менеджер памяти рулит процессом)  
// массив arr скорее всего будет создан в куче
```

# Слайсы (динамические массивы)

// Слайс устроен так:

```
type slice struct {  
    Pointer uintptr // Указатель на стат массив  
    Len    int  
    Cap    int  
}
```



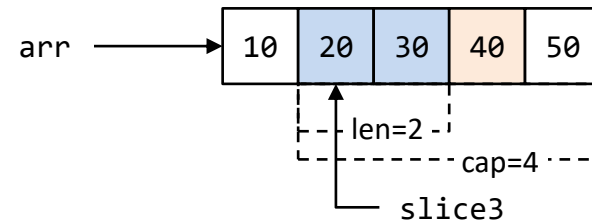
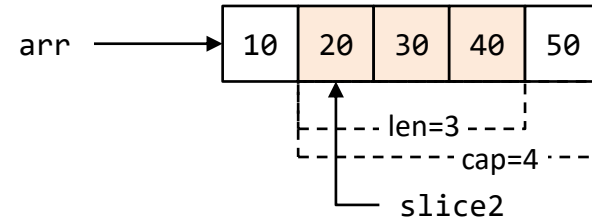
# Создание слайса

```
// Литерал слайса
slice1 := []int{1, 2, 3, 4, 5}
fmt.Println(len(slice1), cap(slice1)) // 5 5

// Создание из массива
arr := [5]int{10, 20, 30, 40, 50}
slice2 := arr[1:4] // элементы с 1 по 3
fmt.Println(len(slice2), cap(slice2)) // 3 4

// Создание из слайса
slice3 := slice2[0:2] // элементы с 0 по 1
fmt.Println(len(slice3), cap(slice3)) // 2 4

// Использование make()
slice3 := make([]int, 3)
slice4 := make([]int, 3, 10)
fmt.Println(len(slice3), cap(slice3)) // 3 3
fmt.Println(len(slice4), cap(slice4)) // 3 10
```



# Создание слайса

```
// Пустой слайс
var slice5 []int           // nil слайс (указатель в структуре слайса равен nil)
slice6 := []int{}          // пустой, но не nil (указатель в структуре слайса указывает на базовый
адрес)
fmt.Println(slice5 == nil) // true
fmt.Println(slice6 == nil) // false
```

# Операции со слайсами – append

```
var slice []int
```

```
// Добавление одного элемента
```

```
slice = append(slice, 1)
```

```
slice = append(slice, 2)
```

```
// Добавление нескольких элементов
```

```
slice = append(slice, 3, 4, 5)
```

```
// Добавление другого слайса (используем ...)
```

```
anotherSlice := []int{6, 7, 8}
```

```
slice = append(slice, anotherSlice...)
```

```
fmt.Println(slice) // [1 2 3 4 5 6 7 8]
```

```
// Если при добавлении len == cap, то:
```

```
// - создаётся новый массив;
```

```
// - элементы копируются в новый массив и указатель слайса перемещается на него (размер нового массива в 2 раза больше старого с понижением коэффициента роста до 1.25);
```

```
// - старый массив может быть удалён сборщиком мусора (если не используется).
```



# Операции со слайсами – copy

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

```
// Неправильное копирование (обе переменные указывают на один массив)
```

```
wrongCopy := source
```

```
wrongCopy[0] = 100
```

```
fmt.Println("source:", source) // [100 2 3 4 5]
```

```
fmt.Println("copy:", wrongCopy) // [100 2 3 4 5]
```

```
// Правильное копирование
```

```
properCopy := make([]int, len(source))
```

```
n := copy(properCopy, source)
```

```
properCopy[0] = 200
```

```
fmt.Println("source:", source) // [1 2 3 4 5]
```

```
fmt.Println("copy:", properCopy) // [200 2 3 4 5]
```

```
fmt.Println("Скопировано элементов:", n) // 5
```

```
// Частичное копирование
```

```
small := make([]int, 3)
```

```
copy(small, source) // скопирует только первые 3 элемента
```

```
fmt.Println("copy:", small) // [1 2 3]
```

# Операции со слайсами – slicing

```
slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

// Базовый синтаксис: slice[low:high]
sub1 := slice[2:5] // элементы с индексом 2, 3, 4
fmt.Println("slice[2:5]:", sub1) // [2 3 4]

// Опускание границ
sub2 := slice[:3] // от начала до индекса 2
sub3 := slice[7:] // от индекса 7 до конца
sub4 := slice[:]  // полная копия (но тот же базовый массив!)

fmt.Println("slice[:3]:", sub2) // [0 1 2]
fmt.Println("slice[7:]:", sub3) // [7 8 9]
fmt.Println("slice[:]:", sub4)  // [0 1 2 3 4 5 6 7 8 9]

// С указанием емкости: slice[low:high:src_max] НЕ шаг!
slice[2:5:4] // Ошибка. src_max должен быть >= конечному индексу
sub5 := slice[2:5:5] // len=3, cap=3 (5-2)
sub6 := slice[2:5:7] // len=3, cap=5 (7-2)
fmt.Printf("slice[2:5:5]: , len(sub5), cap(sub5)) // [2 3 4] 3 3
fmt.Printf("slice[2:5:7]: , len(sub6), cap(sub6)) // [2 3 4] 3 5
```

# Операции со слайсами – slicing

```
slice := []int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

// Базовый синтаксис: slice[low:high]
sub1 := slice[2:5] // элементы с индексом 2, 3, 4
fmt.Println("slice[2:5]:", sub1) // [2 3 4]

// Опускание границ
sub2 := slice[:3] // от начала до индекса 2
sub3 := slice[7:] // от индекса 7 до конца
sub4 := slice[:] // полная копия (но тот же базовый массив!)

fmt.Println("slice[:3]:", sub2) // [0 1 2]
fmt.Println("slice[7:]:", sub3) // [7 8 9]
fmt.Println("slice[:]:", sub4) // [0 1 2 3 4 5 6 7 8 9]

// С указанием емкости: slice[low:high:src_max] НЕ шаг!
slice[2:5:4] // Ошибка. src_max должен быть >= конечному индексу
sub5 := slice[2:5:5] // len=3, cap=3 (5-2)
sub6 := slice[2:5:7] // len=3, cap=5 (7-2)
fmt.Printf("slice[2:5:5]: , len(sub5), cap(sub5)) // [2 3 4] 3 3
fmt.Printf("slice[2:5:7]: , len(sub6), cap(sub6)) // [2 3 4] 3 5
```

## Операции со слайсами – удаление элементов

```
// Массив
arr := [...]int{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}

// Полный слайс
slice := arr[:]

// Удаляем n-й элемент
n := 5
slice = append(slice[:n], slice[n+1:]...)

fmt.Println(slice) // [0 1 2 3 4 5 6 7 8 9]
fmt.Println(arr)   // [0 1 2 3 4 6 7 8 9 9]
```

# Утечка памяти через слайс

```
func processData() []byte{  
    data := make([]byte, 0, 1000000) // 1MB  
  
    // ... заполняем данными ...  
  
    // Возвращаем маленький срез, но базовый массив остается в памяти  
    return data[:10] // УТЕЧКА! весь 1MB массив остается  
}
```

# Передача слайсов в функцию по значению

```
func foo(arg []int){  
    // arg копия слайса src (т.е. копия структуры), но указывает на тот же массив  
    fmt.Print(arg)  
}  
  
func main(){  
    src := make([]int, 3, 5)  
    foo(src)  
}
```

# Передача слайсов в функцию по значению

```
package main

import "fmt"

func change(abc []int) {
    //
    for i := range abc {
        abc[i] = 4
    }
    fmt.Println(abc)
}

func main() {
    abc := []int{1, 2, 3}
    change(abc)
    fmt.Println(abc)
}
```

```
[4 4 4]
[4 4 4]
```

```
package main

import "fmt"

func change(abc []int) {
    abc = []int{4, 4, 4}
    fmt.Println(abc)
}

func main() {
    abc := []int{1, 2, 3}
    change(abc)
    fmt.Println(abc)
}
```

```
[4 4 4]
[1 2 3]
```

# Передача слайсов в функцию

```
package main

import "fmt"

func change(abc []int) {
    abc = append(abc, 4)
    for i := range abc {
        abc[i] = 4
    }
    fmt.Println(abc)
}

func main() {
    abc := []int{1, 2, 3}
    change(abc)
    fmt.Println(abc)
}
```

```
[4 4 4 4]
[1 2 3]
```

```
package main

import "fmt"

func change(abc []int) {
    abc = append(abc, 4)
    for i := range abc {
        abc[i] = 4
    }
    fmt.Println(abc)
}

func main() {
    abc := []int{1, 2, 3}
    abc = append(abc, 4)
    change(abc)
    fmt.Println(abc)
}
```

```
[4 4 4 4 4]
[4 4 4 4]
```



# Передача слайсов в функцию по указателю

```
package main

import "fmt"

func change(abc *[]int) {
    *abc = []int{4, 4, 4}
    fmt.Println(*abc) // [4, 4, 4]
}

func main() {
    abc := []int{1, 2, 3}
    change(&abc)
    fmt.Println(abc) // [4, 4, 4]
}

// Как правило от передачи слайсов по указателю нет особенной пользы
```

# Возврат слайсов из функции по значению

```
package main
```

```
import "fmt"
```

```
func nweArray() []int {  
    arr := [3]int{1, 2, 3}  
    return arr[:]  
}
```

```
func main() {  
    dst := nweArray()  
    fmt.Println(dst) // [1, 2, 3]  
}
```

```
// Безопасно. Массив будет жить, пока на него ссылается слайс
```

# Возврат слайсов из функции по указателю

```
package main

import "fmt"

func nweArray() *[]int {
    arr := []int{1, 2, 3}
    slise := arr[:]
    return &slise
}

func main() {
    dst := nweArray()
    fmt.Println(*dst) // [1, 2, 3]
}

// Безопасно. Массив будет жить, пока на него ссылается слайс
// Не имеет практической пользы
```

# Многомерные статические массивы

```
// Объявление 3x4 матрицы
var matrix [3][4]int

// Инициализация
matrix = [3][4]int{
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12},
}

// Трехмерный массив (куб 2x3x4)
cube := [2][3][4]int{
    {
        {1,2,3,4}, {5,6,7,8}, {9,10,11,12},
    },
    {
        {13,14,15,16}, {17,18,19,20}, {21,22,23,24},
    },
}

// Автоматический подсчет размера (только для первого индекса)
autoMatrix := [...][2]int{ // [3][2]int
    {1, 2},
    {3, 4},
    {5, 6},
}

// Доступ к элементам
value := matrix[1][2] // 7
matrix[0][0] = 100    // изменяем элемент
```

# Размещение многомерного статического массива в памяти

// Двумерный массив (матрица 3x4)

```
matrix = [3][4]int{  
    {1, 2, 3, 4},  
    {5, 6, 7, 8},  
    {9, 10, 11, 12},  
}
```

// Элементы расположены в памяти

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

// Трехмерный массив (куб 2x3x4)

```
cube := [2][3][4]int{  
    {  
        {1,2,3,4}, {5,6,7,8}, {9,10,11,12},  
    },  
    {  
        {13,14,15,16}, {17,18,19,20}, {21,22,23,24},  
    },  
}
```

// Элементы расположены в памяти

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24

# Многомерные слайсы

```
// Создание 2D слайса
rows, cols := 3, 4
matrix := make([][]int, rows)
for i := range matrix {
    matrix[i] = make([]int, cols)
    for j := range matrix[i] {
        matrix[i][j] = i*10 + j
    }
}

fmt.Println("Матрица 3x4:")
for _, row := range matrix {
    fmt.Println(row)
}

// Слайс слайсов (нерегулярная матрица)
jagged := [][]int{
    {1},
    {2, 3},
    {4, 5, 6},
}

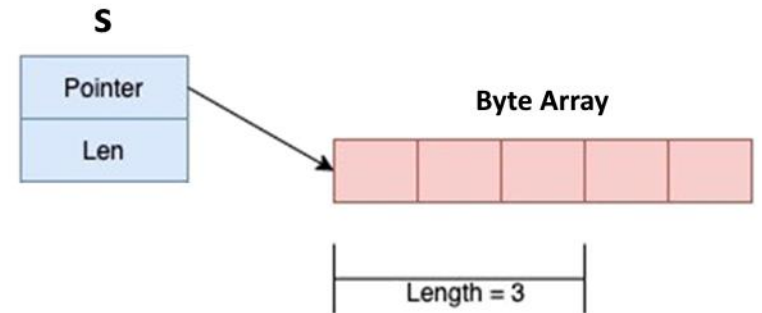
fmt.Println("\nРваный массив:")
for _, row := range jagged {
    fmt.Println(row)
}
```

Строки

# Строка

// Строка устроена так:

```
type string struct {  
    Pointer uintptr // Указатель на массив байт  
    Len    int  
}
```





# Создание и изменение строки

```
// Создаём строку
var str string // Пустая строка
var str1 = "Hello"
str2 := "World"

fmt.Printf("Длина строки (%s): %d\n", str2, len(str2)) // 5

str2[0] = 'h' // ОШИБКА: строка - это неизменяемая последовательность байт
```

## Строка -> Массив байт

```
str := "Hello"

// Преобразование в массив байт фиксированного размера
var byteArr [5]byte
copy(byteArr[:], str) // Копируем содержимое строки в массив
fmt.Printf("Массив байт: %v\n", byteArr) // [72 101 108 108 111]

// Автоматическое преобразование в срез байт
byteSlice := []byte(str)
fmt.Printf("Срез байт: %v\n", byteSlice) // [72 101 108 108 111]

// Изменяем срез байт
byteSlice[0] = 104 // 'h'
fmt.Printf("Измененный срез: %v\n", byteSlice) // [104 101 108 108 111]
fmt.Printf("Оригинальная строка: %s\n", str) // "Hello" - не изменилась
```

## Массив байт -> Строка

```
// Из массива байт
```

```
byteArray := []byte{65, 66, 67} // 'A', 'B', 'C'
```

```
str1 := string(byteArray[:])
```

```
fmt.Println(str1) // "ABC"
```

```
// Из среза байт
```

```
byteSlice := []byte{72, 101, 108, 108, 111} // "Hello"
```

```
str2 := string(byteSlice)
```

```
fmt.Println(str2) // "Hello"
```

```
// Прямое создание из массива (неявное преобразование)
```

```
str3 := string(byteArray[:])
```

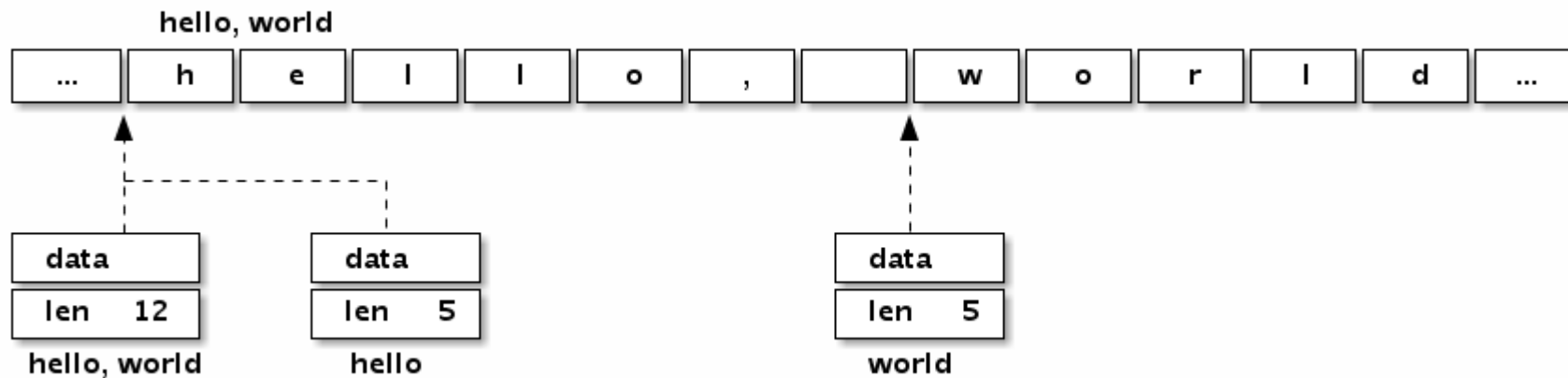
```
fmt.Println(str3) // "ABC"
```

# Итерирование по массиву байт строки

```
str := "Hello"  
  
// Доступ к отдельным байтам  
// Функция len возвращает длину в байтах!  
for i := 0; i < len(str); i++ {  
    fmt.Printf("str[%d] = %d (%c)\n", i, str[i], str[i])  
}
```

# Создание новой строки как слайса байт

```
str := "hello, world"  
str2 := str[:5]  // hello  
str3 := str[7:]  // world
```



```
str := "Hello, 世界"  
str2 := str[7:9]  // ??  
str3 := str[7:10] // 世  
// Такой синтаксис вырезает последовательности байт, но не символов  
// 1 символ может быть закодирован последовательностью из нескольких байт
```

## Строка --> Слайс рун --> Строка

```
str := "Hello, 世界"
```

```
runes := []rune(str)
```

```
fmt.Printf("Срез рун: %v\n", runes) // [72 101 108 108 111 44 32 19990 30028]
```

```
fmt.Printf("Длина в рунах: %d\n", len(runes)) // 9
```

```
// Обратно в строку
```

```
newStr := string(runes)
```

```
fmt.Println(newStr) // "Hello, 世界"
```

```
// Изменение через руны
```

```
runes[7] = 'М'
```

```
runes[8] = 'И'
```

```
modifiedStr := string(runes)
```

```
fmt.Println(modifiedStr) // "Hello, МИ"
```

# Итерирование по рунам строки

```
package main

import "fmt"
import "unicode/utf8"

func main(){
    fmt.Println(utf8.RuneCountInString("мир"), len("мир")) // 3 6
    fmt.Println(utf8.RuneCountInString("世界"), len("世界")) // 2 6

    str := "Hello, 世界"
    for _, r := range str {
        fmt.Printf("%c ", r) // H e l l o ,   世 界
    }
}
```

# Сборка строки из кусочков

```
package main

import "fmt"
import "strings"

func main(){
    runes := []rune{'П', 'р', 'и', 'в', 'е', 'т'}

    str1 := ""
    for i := 0; i < len(runes); i++ {
        str1 = str1 + string(runes[i])
    }

    // Быстрее
    builder := strings.Builder{}
    for i := 0; i < len(runes); i++ {
        builder.WriteRune(runes[i])
    }

    str2 := builder.String()
    fmt.Print(str1, str2)
}
```



# Структуры

## Постановка задачи

- Хранить в программе описание характеристик некоторого объекта

## Решение I

```
var aliceBirthYear int
var aliceBirthMonth int
var aliceBirthDay int
var aliceHeight float64
var aliceWeight float64

var bobBirthYear int
var bobBirthMonth int
var bobBirthDay int
var double bobHeight float64
var double bobWeight float64
```

## Решение I - Проблемы

- Для каждого человека нужно создавать по пять отдельных переменных – **долго, могут быть опечатки**
- Чтобы передать в функцию, нужно перечислить все аргументы – **можно перепутать порядок**

```
print(aliceBirthYear, aliceBirthMonth,  
      aliceBirthDay, aliceHeight, aliceWeight);
```

- Как вернуть из функции?

## Решение II - Структуры

```
type human struct{    // Свой тип данных
    aliceBirthYear int
    aliceBirthMonth int
    aliceBirthDay int
    aliceHeight float64
    aliceWeight float64
}

var alice, bob human // Создаём переменные
```

## Решение II - Структуры

```
var alice, bob struct {  
    aliceBirthYear  int  
    aliceBirthMonth int  
    aliceBirthDay   int  
    aliceHeight     float64  
    aliceWeight     float64  
}
```

## Решение II - Структуры

```
alice := struct {  
    aliceBirthYear  int  
    aliceBirthMonth int  
    aliceBirthDay   int  
    aliceHeight     float64  
    aliceWeight     float64  
}{  
    // Тут обязательно значения  
}
```

# Где можно объявлять структуры?

- Внутри функций

```
func foo() {  
    type num struct {  
        i int  
    }  
}
```
- Вне функций

```
type num struct {  
    i int  
}  
func foo() {  
}
```
- Внутри других структур

```
type num struct {  
    j struct {  
        val int  
    }  
    i int  
}
```



## Что может быть членом структуры?

Если можно создать переменную этого типа, то это может быть членом структуры

Например:

- Примитивные типы: `int`, `float64`, `byte` ...
- Другие структуры;
- Массивы;
- Строки;
- ...

# Имена полей

```
type DataOut struct {  
    Year  int  
    Month int  
    Day   int  
}
```

```
type DataIn struct {  
    year  int  
    month int  
    day   int  
}
```

Поля структуры `DataIn` видны только в этом пакете, т.е. их можно назвать приватными полями.  
Поля структуры `DataOut` видны за пределами пакета, т.е. их можно назвать публичными полями.

## Как работать со структурой

```
type Data struct {  
    Year  int  
    Month int  
    Day   int  
}
```

```
var now Data  
now.Year = 2018  
now.Day  = 9  
now.Month = 11
```

## Как работать со структурой (указатель)

```
type Data struct {  
    Year  int  
    Month int  
    Day   int  
}
```

```
var now *Data = new(Data)  
now.Year = 2018  
now.Day = 9  
now.Month = 11
```

## Как работать со структурой (указатель II)

```
type Data struct {  
    Year  int  
    Month int  
    Day   int  
}
```

```
var now *Data = &Data{}  
now.Year = 2018  
now.Day = 9  
now.Month = 11
```

## Как работать со структурой

```
now.Year = now.Year + 1 // 2019
```

```
fmt.Print(now.Day) // 9
```

```
now.Month = now.Day + now.Year // 2028
```

```
var p *int = &now.Month
```

# Инициализация структуры

```
type Employee struct {  
    id    int  
    age   int  
    wage  float64  
}
```

// Значения по умолчанию 0. Поменять нельзя

```
var Kate Employee  
var frank = Employee{}  
var joe = Employee{1, 32, 60000.0} // Нужны все!!!  
var john = Employee{id: 2, wage: 60000.0}
```

# Инициализация слайса структур

```
type location struct {  
    name string  
    lat  float64  
    long float64  
}
```

```
locations := []location{  
    {name: "Bradbury Landing", lat: -4.5895, long: 137.4417},  
    {name: "Columbia", lat: -14.5684, long: 175.472636},  
    {name: "Challenger", lat: -1.9462, long: 354.4734},  
}
```



## Присваивание значений структурам I

```
type Employee struct {  
    id int  
    age int  
    wage float64  
}
```

```
var joe Employee  
joe.id = 1  
joe.age = 32  
joe.wage = 60000.0
```

## Присваивание значений структурам II

```
type Employee struct {  
    id    int  
    age   int  
    wage  float64  
}
```

```
var joe, mike = Employee{1, 20, 3.0}, Employee{}  
mike = joe // Копирование значений joe в mike  
joe = Employee{2, 22, 6.3}
```

## Передача структуры как параметр в функцию

```
type Employee struct {  
    id    int  
    age   int  
    wage  float64  
}
```

```
func printInformation(employee Employee) {  
    fmt.Println("ID: ", employee.id)  
    fmt.Println("Age: ", employee.age)  
    fmt.Println("Wage: ", employee.wage)  
}
```

## Передача структуры как параметр в функцию (указатель)

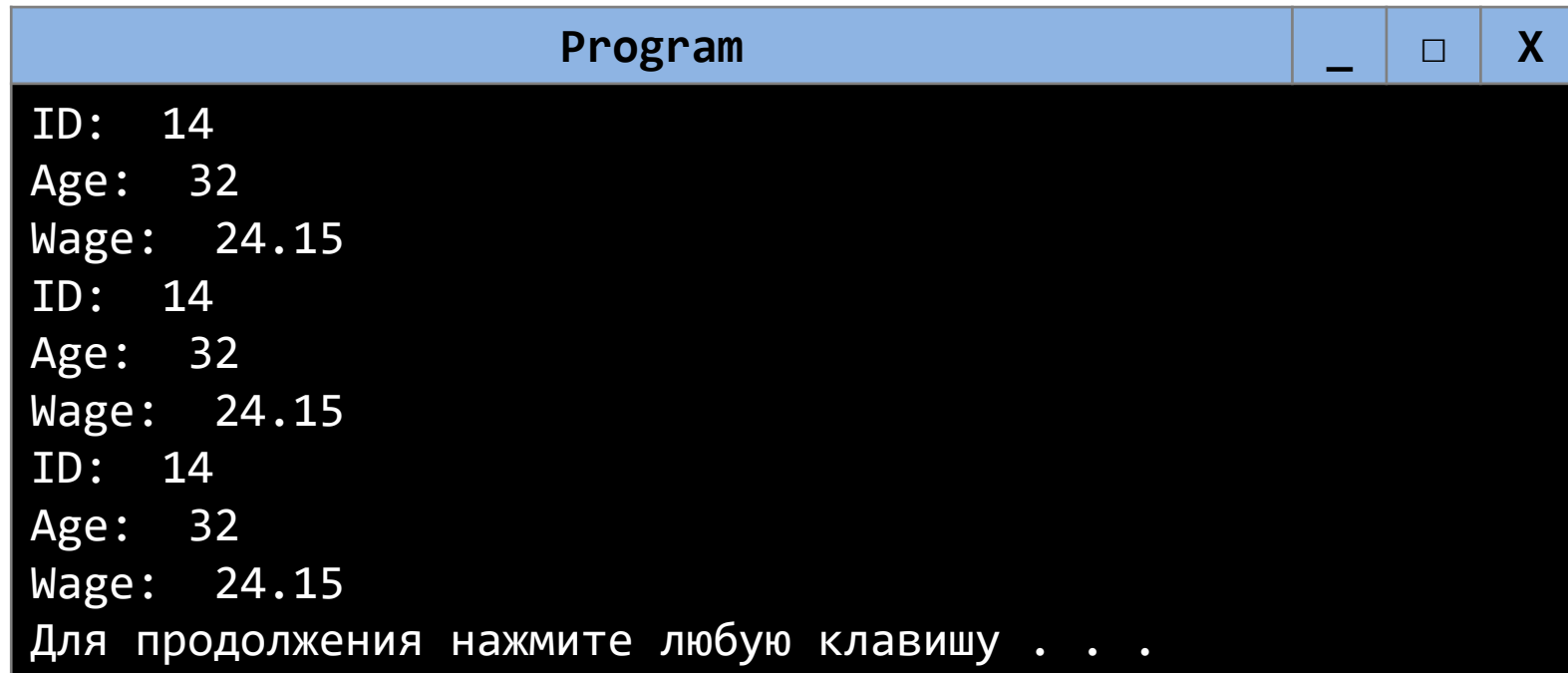
```
type Employee struct {  
    id    int  
    age   int  
    wage  float64  
}
```

```
func printInformationPtr(employee *Employee) {  
    fmt.Println("ID: ", employee.id)  
    fmt.Println("Age: ", employee.age)  
    fmt.Println("Wage: ", employee.wage)  
}
```

## Передача структуры как параметр в функцию

```
func main() {  
    var john = Employee{14, 32, 24.15}  
  
    printInformation(john)  
    printInformation(Employee{14, 32, 24.15})  
    printInformationPtr(&john)  
}
```

## Передача структуры как параметр в функцию



```
Program
ID: 14
Age: 32
Wage: 24.15
ID: 14
Age: 32
Wage: 24.15
ID: 14
Age: 32
Wage: 24.15
Для продолжения нажмите любую клавишу . . .
```

## Возврат структур из функций

```
type Point3d struct {  
    x, y, z float64  
}  
  
func getZeroPoint() Point3d {  
    return Point3d{0.0, 0.0, 0.0}  
}  
  
func main() {  
    zero := getZeroPoint()  
    fmt.Print(zero)  
}
```

# Метод String

```
type Point3d struct {  
    x, y, z float64  
}  
  
func (b Point3d) String() string {  
    return "Point3d"  
}  
  
func main() {  
    var zero Point3d  
    fmt.Print(zero)  
}
```



Дополнительные сведения

# Разные типы

```
type Point3d struct {  
    x, y, z float64  
}
```

```
type Vector3d struct {  
    x, y, z float64  
}
```

```
p := Point3d{ 0.0, 0.0, 0.0 }
```

```
var v Vector3d
```

```
v = p // Ошибка. У v и p разные типы, но можно преобразовать
```

# Преобразование структур I

```
type Point3d struct {  
    x, y, z float64  
}
```

```
type Vector3d struct {  
    x, y, z float64  
}
```

```
p := Point3d{ 0.0, 0.0, 0.0 }
```

```
var v Vector3d
```

```
v = Vector3d(p)  // ОК. Одинаковый базовый тип
```

## Преобразование структур II

```
type Point3d struct {  
    x    float64    `json:"x"`  
    y, z float64  
}
```

```
type Vector3d struct {  
    x    float64  
    y, z float64  
}
```

```
p := Point3d{ 0.0, 0.0, 0.0 }
```

```
var v Vector3d
```

```
v = Vector3d(p)  // ОК. Одинаковый базовый тип
```

## Преобразование структур III

```
type Point3d struct {  
    x, y, z float64  
}
```

```
type Vector3d struct {  
    x, y, z, u float64  
}
```

```
p := Point3d{ 0.0, 0.0, 0.0 }
```

```
var v Vector3d
```

```
v = Vector3d(p) // Ошибка. Разный базовый тип
```

# Вложенные структуры

```
type Employee struct {  
    id    int16  
    age   int32  
    wage  float64  
}  
  
type Company struct {  
    CEO           Employee // CEO – это структура  
    numberOfEmployees int  
}  
  
var myCompany = Company{Employee{1, 42, 60000.0}, 5}  
fmt.Print(myCompany.CEO.id)
```

# Встроенные поля

```
type Person struct {
    Name string
    Age  int
}

type Employee struct {
    Person // Имя типа или указатель на тип. Без повторов
    Position string
}

func main(){
    e := Employee{
        Person: Person{
            Name: "John",
            Age: 25,
        },
        Position: "boss",
    }

    fmt.Println(e)
    fmt.Println(e.Name, e.Age, e.Position) // OK
    fmt.Println(e.Person.Name, e.Person.Age, e.Position) // OK
}
```

# Инициализация слайса и карты анонимных структур

```
employees := []struct {  
    ID    int  
    Name  string  
}{  
    {1, "Alice"},  
    {2, "Bob"},  
}
```

```
employeeMap := map[int]struct {  
    Name  string  
    Age   int  
}{  
    1: {"Alice", 30},  
    2: {"Bob", 40},  
}
```



# Размер структуры и выравнивание I

```
type Point3d struct {  
    x int16    // Sizeof: 2  
    y int32    // Sizeof: 4  
    z float64  // Sizeof: 8  
}
```

```
var zero Point3d  
fmt.Print(unsafe.Sizeof(zero)) // 16  != ( 2 + 4 + 8 )
```

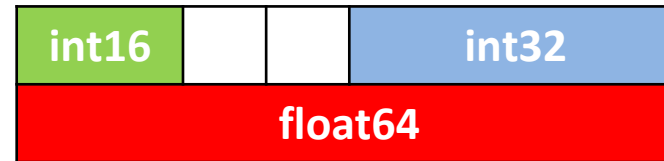
## Размер структуры и выравнивание II

```
type Point3d struct {  
    x int16    // Sizeof: 2  
    z float64  // Sizeof: 8  
    y int32    // Sizeof: 4  
}
```

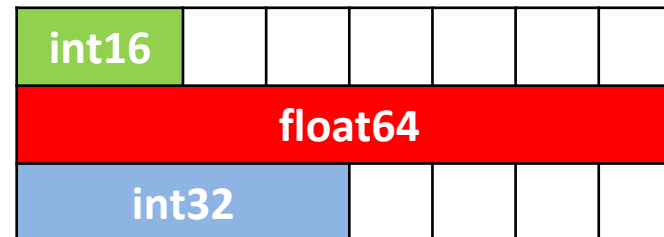
```
var zero Point3d  
fmt.Print(unsafe.Sizeof(zero)) // 24  != ( 2 + 4 + 8 )
```

## Размер структуры и выравнивание II

```
type Point3d struct {  
    x int16  
    y int32  
    z float64  
}
```



```
type Point3d struct {  
    x int16  
    z float64  
    y int32  
}
```



Объединения

## Что такое объединение?

В Go нет типа данных "объединение" или аналога. Можно найти реализацию аналога (`variant`) в пакетах разработанных сторонними разработчиками.

# Перечисления

# Что такое перечисление

Перечисление – это пользовательский тип данных, определяющий набор целочисленных констант.

Зачем нужен:

- Сделать код более читабельным путём замены «магических чисел» на элементы перечисления;  
Пример: `return 0` `return SUCCESS`
- Как дополнительный контроль, защищающий от случайных, автоматических преобразований типов.

# Что такое перечисление

В Go нет типа "перечисление", но можно создать аналог



# Объявление I

```
type Weekday int
```

```
const (  
    Sunday      Weekday = iota + 1 // 1  
    Monday      // 2  
    Tuesday     // 3  
    Wednesday   // 4  
    Thursday    // 5  
    Friday      // 6  
    Saturday    // 7  
)
```

## Объявление II

```
type Weekday int
```

```
const (  
    Sunday      Weekday = iota + 1 // 1  
    Monday           // 2  
    Tuesday         // 3  
    Wednesday Weekday = 1          // 1  
    Thursday        // 1  
    Friday          // 1  
    Saturday        // 1  
)
```

# Переменные

```
type Direction int

const (
    North Direction = iota // 0
    East                 // 1
    South                // 2
    West                 // 3
)

var a Direction
var b Direction = 0
var c Direction = North
```

## Ввод / Вывод

```
type Direction int
```

```
const (  
    North Direction = iota // 0  
    East             // 1  
    South            // 2  
    West             // 3  
)
```

```
var a Direction = North  
fmt.Println(a) // 0  
fmt.Scan(&a)    // ВВОДИТЬ нужно число
```

# Ввод / Вывод

```
type Direction int

const (
    North Direction = iota // 0
    East                  // 1
    South                 // 2
    West                  // 3
)

func (d Direction) String() string {
    return [...]string{"North", "East", "South", "West"}[d]
}

func (d Direction) EnumIndex() int {
    return int(d)
}

func main() {
    var d Direction = West
    fmt.Println(d)           // West
    fmt.Println(d.String())  // West
    fmt.Println(d.EnumIndex()) // 4
}
```

# Операции

Для перечислений работают все операции базового типа, но при этом тип результата остаётся новым:

```
var d Direction = West
var i int = d + 1          // Ошибка. Результат выражения НЕ int

var d Direction = West
var i Direction = d + 1    // Ошибки нет
```

Переменной перечисляемого типа можно присвоить только перечислитель соответствующего типа:

```
var d Direction = West
var e Direction = NewWest // Ошибка. Значение как у West, но другой тип
var f Direction = Direction(NewWest) // Нормально
```

# Перечисления и функции

```
type Direction int

const (
    North Direction = 1
    South Direction = -1
    East  Direction = -2
    West  Direction = 2
)

func move(xpos int, ypos int, d Direction) {
    switch d {
    case North, South:
        xpos += int(d)
    case East, West:
        ypos += int(d) / 2
    }
}

move(0, 0, West)
move(0, 0, 1)
var dir int = 1
move(0, 0, dir) // Ошибка
```

# Интерфейсы



# Интерфейсы

Обычная переменная может сохранять только значения своего типа:

```
var a int = 1    // ОК  
var b int = "1"  // Ошибка
```

Интерфейсная переменная (или параметр функции) может сохранять значения типов удовлетворяющих указанным в интерфейсе требованиям. Если у типа есть, то что нужно интерфейсу, то того достаточно (утиная типизация), но можно и больше:

```
var a any = 1    // ОК  
var b any = "1"  // ОК
```

# Интерфейс

```
package main

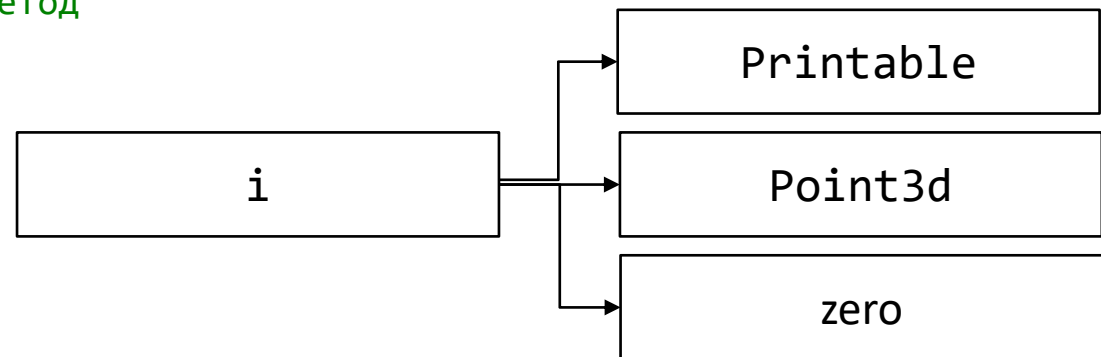
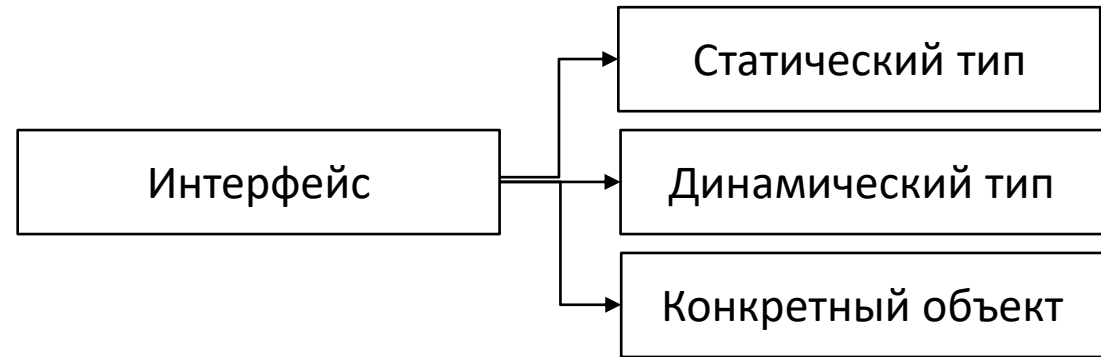
import "fmt"

type Point3d struct {
    x, y, z float64
}

func (b Point3d) String() string {
    return "Point3d"
}

type Printable interface{
    String() string // Требуем, чтобы был этот метод
}

func main() {
    var zero Point3d
    var i Printable = zero
    fmt.Print(i)
}
```



# Проверка соответствия статическому типу

```
type Printable interface{
    String() string
}

func main() {
    data := 42
    var i Printable = data // Ошибка. У int нет метода "String() string"
    fmt.Print(i)
}
```

# Пустой интерфейс

```
type myAny interface{  
}  
  
func main() {  
    var i myAny = 42           // OK  
    var j any = 3.14          // OK  
    var k interface{} = "Hello" // OK  
    fmt.Print(i, j, k)  
}
```

# Пустой интерфейс

```
type Point3d struct {  
    x, y, z float64  
}  
  
func (b Point3d) String() string {  
    return "Point3d"  
}  
  
type Line3d struct {  
    a, b Point3d  
}  
  
func (b Line3d) String() string {  
    return "Line3d"  
}  
  
type Printable interface{  
    String() string  
}  
  
func main() {  
    var point Point3d  
    var line Line3d  
  
    var i Printable = point // Динамический тип Point3d  
    var j Printable = line  // Динамический тип Line3d  
  
    fmt.Print(i, j) // Point3d Line3d  
}
```

# Сравнение интерфейсов

```
type myString string

func main() {
    var i any = "hello"
    fmt.Println(i == "hello") // (string, 'hello') == (string, 'hello')

    var str2 myString = "hello"
    var k any = str2
    fmt.Println(k == "hello") // (myString, 'hello') == (string, 'hello')

    var pStr *string = nil
    var j any = pStr
    fmt.Println(j == nil)      // (*string, nil) == (nil, nil)
}

// Интерфейс хранит два значения (указатель на динамический тип, указатель на объект)
// Сравняются не только значения, но и динамические типы
```

Словарь

(карта, мэп, ассоциативный массив)

## Что такое словарь

**Словарь** – структура данных, как и массив, предназначен для хранения набора данных, но вместо одиночных значений словарь хранит пары – ключ и значение.

В словаре ключи не могут дублироваться, а значения могут. Как правило элементы словаря не упорядочены (но это зависит от реализации), поэтому словарь относится к ассоциативным типам данных.

По сути переменная-словарь в go – это **указатель** на структуру с данными словаря. Поэтому поведения словаря отличается от поведения слайса (слайс – это структура).



## Как определить словарь

```
// имя := make(map[тип ключа] тип значения, длина)
```

```
dict1 := make(map[string]string, 3)
```

```
dict2 := make(map[string]string)
```

```
var dict3 map[string]string // Нельзя использовать - nil
```

```
dict3 = map[string]string{} // Теперь можно – пустой map
```

Тип значения может быть любым.

Тип ключа должен иметь неизменный хэш.

# Инициализация элементов

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":      "Sammy",
        "animal":    "shark",
        "color":     "blue",
        "location":  "ocean",
    }
    fmt.Println(sammy)
}
```

# Обращение к элементам

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":    "Sammy",
        "animal":  "shark",
        "color":   "blue",
        "location": "ocean",
    }
    sammy["cost"] = "10000$"

    fmt.Println(sammy["animal"]) // shark
    fmt.Println(sammy["color"])  // blue
    fmt.Println(sammy["location"]) // ocean
    fmt.Println(sammy["cost"])   // "10000$"
}
```

# Перебор элементов

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":      "Sammy",
        "animal":    "shark",
        "color":     "blue",
        "location":  "ocean",
    }

    for key, value := range sammy {
        fmt.Printf("%q is the key for the value %q\n", key, value)
    }
}
```

# Удаление элементов

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":    "Sammy",
        "animal":  "shark",
        "color":   "blue",
        "location": "ocean",
    }

    delete(sammy, "location") // Удаляем ключ "location"

    for key, value := range sammy {
        fmt.Printf("%q is the key for the value %q\n", key, value)
    }
}
```

# Размер словаря

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":      "Sammy",
        "animal":    "shark",
        "color":     "blue",
        "location":  "ocean",
    }

    fmt.Print(len(sammy))
}
```

# Проверка наличия элемента

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":    "Sammy",
        "animal":  "shark",
        "color":   "blue",
        "location": "ocean",
    }

    cost, ok := sammy["cost"]
    if ok {
        fmt.Printf("Sammy has a cost of %s\n", cost)
    } else {
        fmt.Println("Sammy cost was not found")
    }
}
```

# Копирование словаря

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":    "Sammy",
        "animal":  "shark",
        "color":   "blue",
        "location": "ocean",
    }

    newSammy := sammy           // Указывают на один словарь
    newSammy["cost"] = "10000$" // Меняются оба

    fmt.Print(sammy, newSammy)
}
```

Если нужна копия, то нужно копировать вручную, например в цикле.



# Указатель на значение в словаре

```
package main

import "fmt"

func main() {
    sammy := map[string]string{
        "name":      "Sammy",
        "animal":    "shark",
        "color":     "blue",
        "location": "ocean",
    }

    // Словари не упорядочены и могут перестроиться
    // при добавлении/удалении элементов, поэтому
    // так нельзя
    var color *string = sammy["color"]
}
```

# Передача словаря в функцию

```
package main

import "fmt"

func setCost(obj map[string]string) {
    obj["cost"] = "10000$"
}

func main() {
    sammy := map[string]string{
        "name":    "Sammy",
        "animal":  "shark",
        "color":   "blue",
        "location": "ocean",
    }

    setCost(sammy) // Изменится
    fmt.Print(sammy)
}
```

Множество

## Что такое множество

**Множество** – структура данных, которая является реализацией математического объекта множество.

Множество хранит элементы в единственном числе, без дублей. Как правило элементы множества не упорядочены (но это зависит от реализации), поэтому множество относится к ассоциативным типам данных.

В Go нет типа данных "множество", но его аналог можно реализовать при помощи словаря

# Множество

```
set := make(map[string]bool)
set["apple"] = true    // Добавляем элементы
set["orange"] = true
set["mango"] = true
fmt.Println(set)

delete(set, "apple")   // Удаляем элемент
fmt.Println(set)

_, ok := set["mango"] // Проверяем присутствие
fmt.Println("mango in set:", ok)
```

Каналы

# Каналы

Каналы в Go (Golang) – это механизм для обмена данными между горутинами (легковесными потоками). Они обеспечивают синхронизацию и предотвращают состояние гонки.

Рассмотрим позже, когда будем изучать многопоточное программирование

Информация о типах



# Размер типа - аналог sizeof

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    var i int
    var f float64
    var s string
    var arr [10]int

    fmt.Println("Size of int:", unsafe.Sizeof(i))           // 8 байт (на 64-бит)
    fmt.Println("Size of float64:", unsafe.Sizeof(f))       // 8 байт
    fmt.Println("Size of string:", unsafe.Sizeof(s))        // 16 байт (data + len)
    fmt.Println("Size of [10]int:", unsafe.Sizeof(arr))     // 80 байт

    // Для слайсов - размер заголовка
    var slice []int
    fmt.Println("Size of slice header:", unsafe.Sizeof(slice)) // 24 байта
}
```

# Базовая информация о типах

```
package main

import (
    "fmt"
    "reflect"
)

func main() {
    var i int = 42
    var s string = "hello"
    var f float64 = 3.14

    // Тип значения
    fmt.Println("Type of i:", reflect.TypeOf(i))    // int
    fmt.Println("Type of s:", reflect.TypeOf(s))    // string
    fmt.Println("Type of f:", reflect.TypeOf(f))    // float64

    // Kind (категория типа)
    fmt.Println("Kind of i:", reflect.TypeOf(i).Kind()) // int
    fmt.Println("Kind of s:", reflect.TypeOf(s).Kind()) // string

    // Имя типа
    fmt.Println("Name of int:", reflect.TypeOf(i).Name()) // int
}
```

# Сравнение типов

```
package main

import (
    "fmt"
    "reflect"
)

type MyInt int

func main() {
    var i int = 42
    var mi MyInt = 42

    t1 := reflect.TypeOf(i)
    t2 := reflect.TypeOf(mi)

    fmt.Println("t1 == t2:", t1 == t2) // false
    fmt.Println("t1.Kind() == t2.Kind():", t1.Kind() == t2.Kind()) // true

    // Проверка присваиваемости
    fmt.Println("t1 assignable to t2:", t1.AssignableTo(t2)) // false
    fmt.Println("t2 assignable to t1:", t2.AssignableTo(t1)) // false
}
```

# Type assertions (утверждения типа)

```
package main

import "fmt"

func process(value interface{}) {
    // Type assertion - извлечение конкретного типа из interface{}
    if str, ok := value.(string); ok {
        fmt.Println("It's a string:", str, "length:", len(str))
    } else if num, ok := value.(int); ok {
        fmt.Println("It's an int:", num, "doubled:", num*2)
    } else {
        fmt.Printf("Unknown type: %T\n", value)
    }
}

func main() {
    process("hello")
    process(42)
    process(3.14)
}
```

# Type switches

```
package main

import "fmt"

func describe(value interface{}) {
    switch v := value.(type) {
    case string:
        fmt.Printf("String: %q (length: %d)\n", v, len(v))
    case int:
        fmt.Printf("Integer: %d (double: %d)\n", v, v*2)
    case float64:
        fmt.Printf("Float: %f (square: %f)\n", v, v*v)
    case bool:
        fmt.Printf("Boolean: %t (inverted: %t)\n", v, !v)
    default:
        fmt.Printf("Unknown type: %T\n", v)
    }
}

func main() {
    describe("golang")
    describe(100)
    describe(2.5)
    describe(true)
    describe([]int{1, 2, 3})
}
```

# Преобразование типов

# Преобразование типов

<https://go.dev/ref/spec#Conversions>