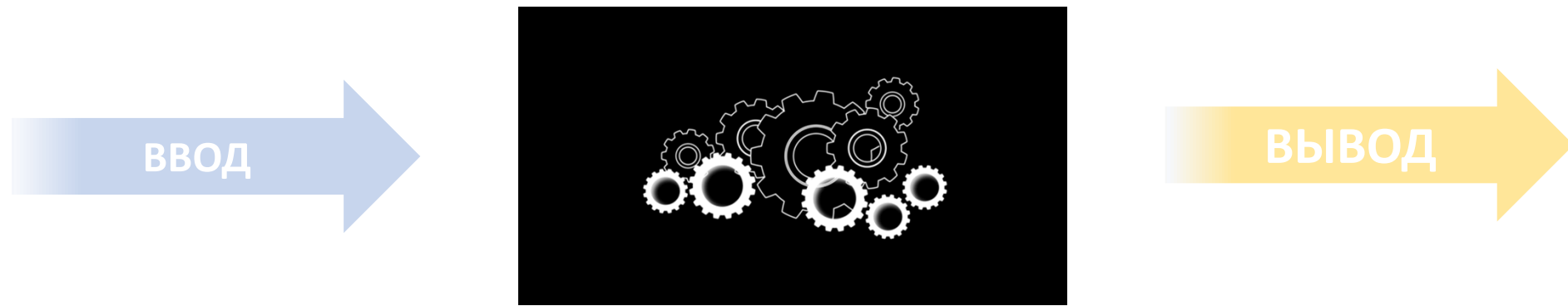


Алгоритмизация и программирование

Лекция 3 (Go)

Модель программы



* МОНОЛИТ С ВВОДОМ И ВЫВОДОМ ТОЛЬКО В КОНСОЛЬ

Простой код

```
package main
import "fmt"

func main(){
    var a, b int
    fmt.Scan(&a, &b)
    c := a + b
    fmt.Println(c)
}
```

Код побольше

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    fmt.Println("Введите числа (введите 'stop' для завершения ввода):")

    var numbers []int
    for {
        var input string
        fmt.Print("> ")
        fmt.Scanln(&input)

        if input == "stop" {
            break
        }

        num, err := strconv.Atoi(input)
        if err != nil {
            fmt.Println("Некорректный ввод. Попробуйте еще раз.")
            continue
        }

        numbers = append(numbers, num)
    }

    total := 0
    for _, num := range numbers {
        total += num
    }
    average := float64(total) / float64(len(numbers))

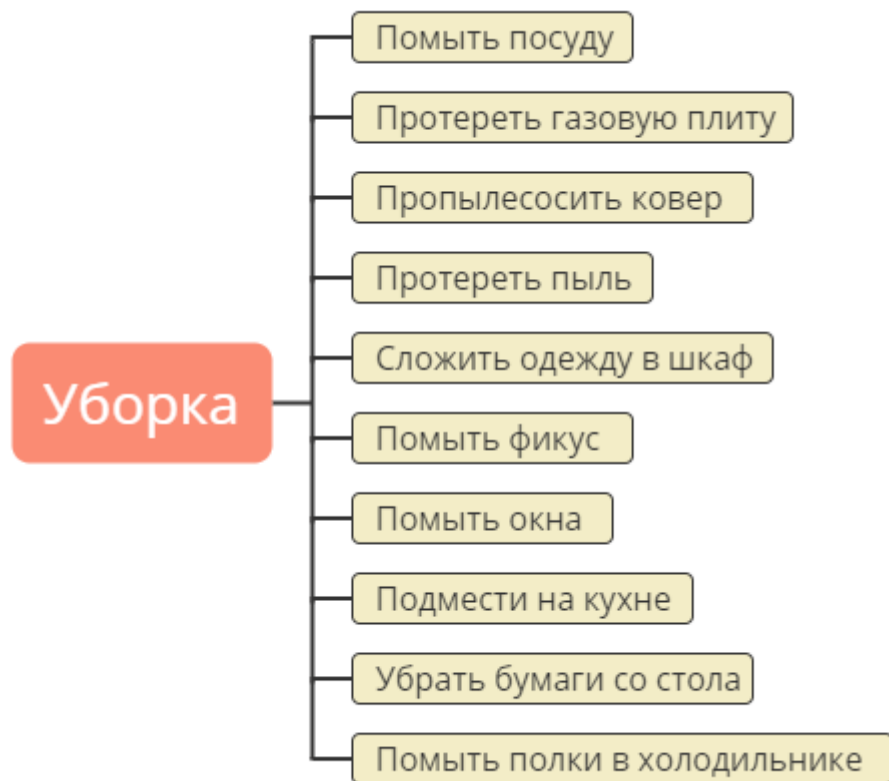
    fmt.Println("Среднее значение:", average)
}
```

Декомпозиция

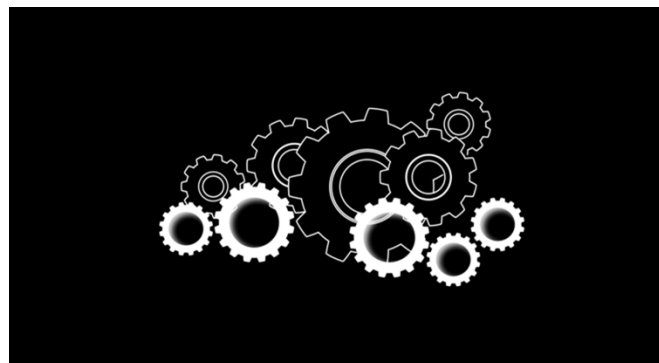
Декомпозицией называют разделение задачи на отдельные небольшие шаги — подзадачи



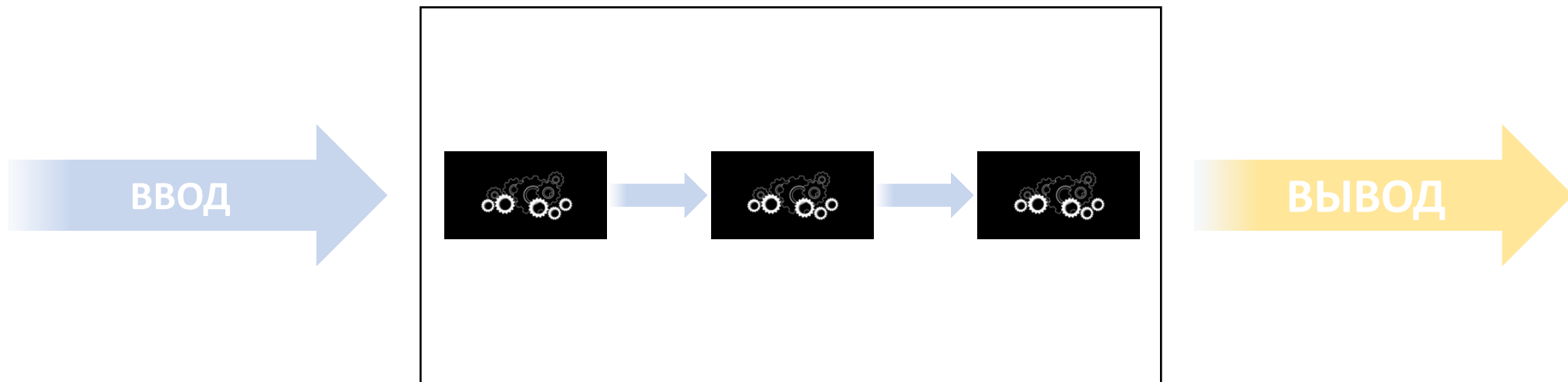
Декомпозиция



Подпрограмма



Программа



Код побольше + декомпозиция

```
package main

import (
    "fmt"
    "strconv"
)

func main() {
    numbers := readNumbersFromTerminal()
    average := calculateAverage(numbers)
    fmt.Println("Среднее значение:", average)
}
```

```
func readNumbersFromTerminal() []int {
    var numbers []int

    fmt.Println("Введите числа (введите 'stop' для завершения ввода):")

    for {
        var input string
        fmt.Print("> ")
        fmt.Scanln(&input)

        if input == "stop" {
            break
        }

        num, err := strconv.Atoi(input)
        if err != nil {
            fmt.Println("Некорректный ввод. Попробуйте еще раз.")
            continue
        }

        numbers = append(numbers, num)
    }

    return numbers
}
```

```
func calculateAverage(numbers []int) float64 {
    total := 0
    for _, num := range numbers {
        total += num
    }
    average := float64(total) / float64(len(numbers))
    return average
}
```

Виды подпрограмм

Функция — это подпрограмма которая выполняет указанную последовательность действий и возвращает результат. Вызов функции может использоваться в других выражениях или в качестве правой части присваивания.

Процедура — это подпрограмма которая выполняет указанную последовательность действий и НЕ возвращает результат. Вызов процедуры нельзя использовать в выражениях или в качестве правой части присваивания.

Плюсы разделения на подпрограммы

- Уменьшение сложности:
 - Структура основной программы стала проще для понимания;
 - Каждая отдельная подпрограмма тоже обладает небольшой сложностью;
 - Разработка происходит небольшими, законченными этапами;
 - Проще покрыть текстами.
- Повторное использование;
- Совместная работа над решением

Магическое число семь плюс-минус два («кошелёк Миллера») — закономерность, обнаруженная американским учёным-психологом Джорджем Миллером, согласно которой кратковременная человеческая память, как правило, не может запомнить и повторить более 7 ± 2 элементов.

Термины

Идентификатор

Идентификатор (identifier — опознаватель) — нечто, позволяющее отличить один объект от других, то есть идентифицировать.

В реальном мире в качестве идентификаторов мы, обычно, используем **слова**.

Сам по себе идентификатор ничего не означает. Я придумал слово: «Джиоптирум». У него нет значения, но сам по себе он **уникальный**.

В программировании идентификатор (имя) — произвольная, уникальная, последовательность символов которая начинается с буквы или символа подчёркивания. В именах не используются пробелы и спец. символы. В Go регистр имеет значение.

```
person_name  
personName  
PersonName
```

Объявление (англ. declaration)

Объявление создаёт идентификатор, описывает, но не создает, программную сущность и устанавливает связь между ними.

Объявление используется, чтобы уведомить компилятор о существовании программной сущности **без её создания**, т.к. по не некоторым причинам она создаётся в другом месте.

Пусть «Джиоптирум» - это мой ноутбук. Это объявление. Теперь использование слова «Джиоптирум» в предложении будет аналогично использованию слов «мой ноутбук».

В программировании объявление связывает идентификатор с программной сущностью (переменной, типом, функцией, и т.д.)

В Go чистые объявления существуют только для функций и используются когда функция реализована вне Go. В этом случае у функции не указывается тело.

Определение (англ. definition)

Определение создаёт идентификатор, программную сущность и устанавливает связь между ними.

Во время определения **создаётся** переменная, функция, класс, и т.д., а также описывается их свойства и поведение.

Определение включает в себя объявление и создание сущности.

В сленге часто используют слово "объявление" даже в тех случаях когда код - это "определение".

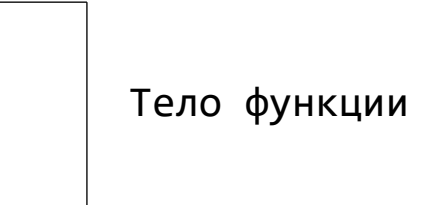
```
var personName string // идентификатор personName – переменная
type myInt int         // идентификатор myInt - тип
func main(){           // идентификатор main – функция
}
```

Определение функции

Прежде чем функцией можно будет пользоваться её нужно определить в коде. Функция не может быть определена внутри другой функции или типа.

Общая структура определения:

```
func идентификатор_функции (список_параметров) (список_результатов){  
    выполняемые_операторы  
}
```



Тело функции

```
func min(a int, b int) (int){  
    if a < b {  
        return a  
    }  
    return b  
}
```


Сигнатура функции

Часть объявления функции которая используется компилятором для того, чтобы однозначно отличить одну функцию от другой. Функции с одинаковой сигнатурой с точки зрения компилятора не различимы и считаются переопределением.

Обычно в сигнатуру входит: название функции и типы параметров, но в зависимости от языка понятие сигнатуры может меняться.

```
func min(a int, b int) (int){  
    if a < b {  
        return a  
    }  
    return b  
}
```

```
(int, int) (int)
```

```
// Сигнатура функции min согласно спецификации Go
```

Тип функции

Функции в Go имеют определённый тип, так же как и переменные. Тип функции можно получить из заголовка если отбросить все идентификаторы указанные в ней:

```
func min(a int, b int) (int) // заголовок функции
func(int, int) (int)         // тип функции
```

Если создать переменную или параметр такого типа, то ей можно будет присвоить любую подходящую функцию и, в дальнейшем обращаться к этой функции при помощи идентификатора переменной/параметра.

```
func min(a int, b int) (int){
    if a < b {
        return a
    }
    return b
}
```

```
var other_min func(int, int) (int) = min
other_min(5, 10) // Результат: 5
```

Параметры функции

Через параметры функция получает входные данные. С точки зрения функции, параметры это её локальные переменные. Параметры указываются в скобках после имени функции. Для каждого параметра указывается имя и тип (как для обычной переменной, но без слова `var`).

Параметры разделяются запятыми. Если несколько параметров одного типа идут подряд, то тип можно указать 1 раз после последнего.

```
func add(a int, b float64) (float64)
```

```
func min(a int, b int) (int)
```

```
func max(a, b int) (int)
```

Неопределенное количество параметров

Иногда в функцию нужно передать неопределённое количество значений (0, 5, 100, и т.д.) . Язык Go разрешает это делать, но только для значений одного типа.

Чтобы показать, функции можно передать любое количество аргументов, перед типом параметра нужно поставить троеточие:

```
func sum(a ...int) (int)
```

Такой тип параметра можно комбинировать с обычными, но параметр с троеточием обязательно должен быть в конце списка:

```
func equalAnyOf(val int, a ...int) (bool)
```

В теле функции параметр с троеточием доступен в виде среза.

Неопределенное количество параметров

При вызове мы можем передать в функцию `sum` разное количество чисел

```
sum()  
sum(1, 2)  
sum(1, 2, 3, 4)
```

Несмотря на то, что внутри функции аргументы будут доступны в виде среза, сам срез в обычном виде передать нельзя

```
var nums = []int{5, 6, 7, 2, 3}  
sum(nums)
```

Можно, если дописать троеточие после аргумента:

```
sum(nums...)
```

Возвращаемые значения

В Go функции могут возвращать результат или не возвращать ничего.

Если функция не возвращает ничего, то список возвращаемых значений не указывается:

```
func doSomething(a int, b int)
```

Если функция возвращает что-либо одно, то указывается тип возвращаемого значения в круглых скобках или без них:

```
func sum(a, b int) (int)
```

```
func sum(a, b int) int
```

Если функция возвращает более одного результата, то указываются тип всех возвращаемых значений, обязательно в круглых скобках:

```
func div(a, b int) (int, error)
```

Возвращаемые значения

```
func doSomething(a int, b int)
```

В теле функции можно, но не обязательно, использовать оператор `return` без указания значения. Используется для досрочного выхода из функции.

```
func sum(a, b int) int
```

В теле функции обязательно использовать оператор `return` со значением совпадающим с типом возвращаемого значения указанного в заголовке.

```
func div(a, b int) (int, error)
```

В теле функции обязательно использовать оператор `return`, при этом количество и типы значений должны совпадать с количеством и типами указанными в заголовке.

Оператор `return`, в теле функции, можно использовать произвольное количество раз.

Именованные возвращаемые значения

Если функция возвращает что-либо, то перед типом возвращаемого результата можно указать имя. Это имя будет доступно в теле функции как обычная локальная переменная.

Список именованных возвращаемых значений обязательно записывается в скобках:

```
func sum(a, b int) (res int)
func div(a, b int) (res int, err error)
```

Чтобы вернуть результат из функции объявленной таким образом можно:

- воспользоваться обычным синтаксисом возврата `return` со значениями. В этом случае именно эти значения и будут результатом функции.
- воспользоваться `return` без значений. В этом случае результат функции будет определяться последним значением, которое было присвоено переменным использующимся для возврата. При этом `return` нельзя указывать в блоках в которых имена выходных переменных сокрыты ("затенены").

```
func sum(a, b int) (res int){
    return a + b
}
```

```
func sum(a, b int) (res int){
    res = a + b
    return
}
```


Вызов функции

Вызов функции возможен только в теле другой функции. Для этого нужно указать имя функции и в круглых скобках список аргументов, в том же количестве, в том же порядке и тех же типов, что и параметры в сигнатуре.

В качестве аргументов можно использовать как переменные, так и литералы (значения):

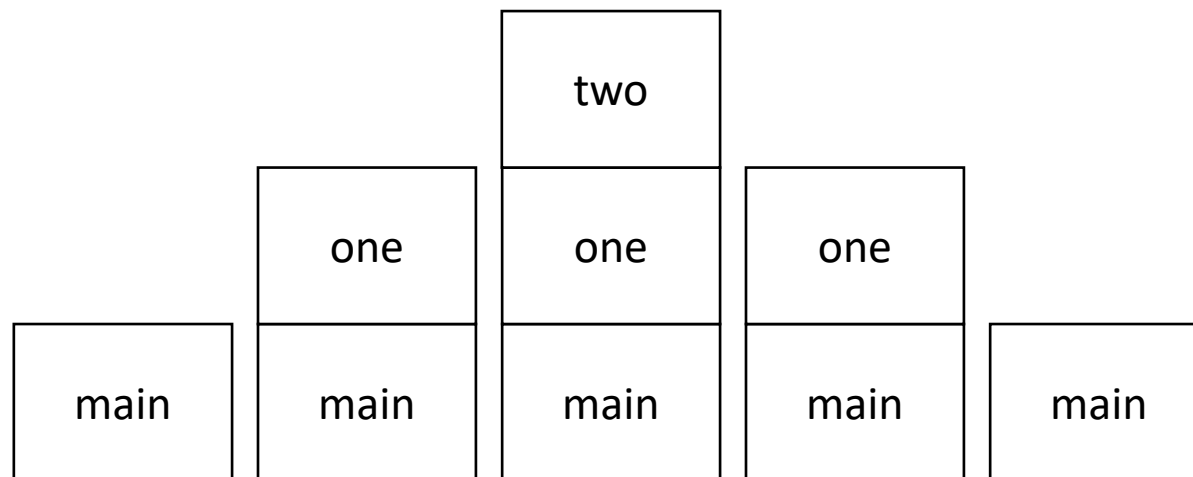
```
func sum(a, b int) int
```

```
one, two := 1, 2  
sum(one, two)    // Вызов  
three := sum(1, 2) // Вызов
```

Стек вызовов

При вызове функции `one` исполнение функции `main` приостанавливается и ожидает завершения функции `one`. Функция `one` вызывая функцию `two` тоже приостанавливается и ожидает завершения функции `two`. Т.е. функции как бы складываются в стопку (стек) и в этой стопке исполняется только та функция которая лежит на вершине.

```
func two(){  
    fmt.Println("two")  
}  
  
func one(){  
    fmt.Println("one")  
    two()  
}  
  
func main(){  
    fmt.Println("main")  
    one()  
}
```



Стек-трейс (stack trace)

Стек-трейс (stack trace) – это информация о последовательности вызовов функций и методов. Стек-трейс полезен при поиске ошибок в коде и автоматически выводится в терминал при падении программы.

```
func two(a, b int){  
    fmt.Println(a / b) // тут ошибка  
}  
  
func one(){  
    fmt.Println("one")  
    two(1, 0)  
}  
  
func main(){  
    fmt.Println("main")  
    one()  
}
```

```
panic: runtime error: integer  
divide by zero  
  
goroutine 1 [running]:  
main.two(...)   
    /home/main.go:12  
main.one()  
    /home/main.go:16 +0x7c  
main.main()  
    /home/main.go:20 +0x7b
```

Вызов функции

В Go функцию можно вызвать:

- Немедленно. Вызываем функцию обычным образом. При этом поток исполнения «перепрыгивает» из вызывающей функции в вызываемую и после её завершения возвращается обратно.

```
sum(1, 2)
```

- Отложенный вызов. Перед вызовом функции указываем ключевое слово `defer`. При этом поток исполнения вызывающей функции продолжает выполнять свои команды игнорируя вызываемую функцию. Перед тем, как вызывающая функция завершится (любым путём) будут вызваны все функции отмеченные как `defer`, при этом они будут исполняться в обратном порядке.

```
defer sum(1, 2)
```

- В параллельном потоке (горутина). Перед вызовом функции указываем ключевое слово `go`. Вызывающая функция продолжает работать игнорируя вызываемую функцию. При этом для вызываемой функции создаётся отдельный «поток» и она начинает работать параллельно с вызывающей.

```
go sum(1, 2)
```

Вызов функции

- Немедленный вызов:

```
func other(){  
    fmt.Println("other")  
}  
  
func main(){  
    fmt.Println("main start")  
    other()  
    fmt.Println("main stop")  
}
```

```
main start  
other  
main stop
```

- Отложенный вызов:

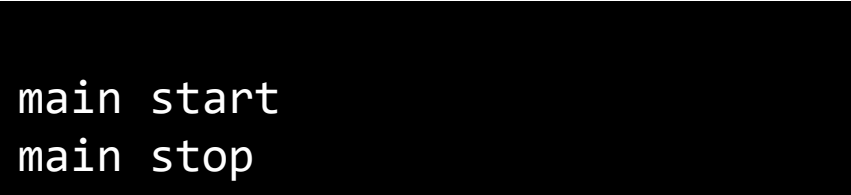
```
func other(){  
    fmt.Println("other")  
}  
  
func main(){  
    fmt.Println("main start")  
    defer other()  
    fmt.Println("main stop")  
}
```

```
main start  
main stop  
other
```

Вызов функции

- В параллельном потоке (горутина):

```
func other(){  
    fmt.Println("other")  
}  
  
func main(){  
    fmt.Println("main start")  
    go other()  
    fmt.Println("main stop")  
}
```



```
main start  
main stop
```

В этом примере функция `main` завершилась раньше, чем функция `other` успела вывести сообщение на экран. Это возможно, т.к. после вызова, функция `other` получила свой собственный поток исполнения команд, а функция `main` осталась в своём потоке.

В этом примере вывод мог бы быть и другим. Например здесь сообщение «other» будет каждый раз между разными цифрами:

```
func main(){  
    go other()  
    for i:=0; i<1000; i++ {  
        fmt.Println(i)  
    }  
}
```

Передача аргументов в функцию

В Go данные в функцию можно передать:

- по значению. В этом случае функция получает **копии** передаваемых **данных**. Следовательно вызываемая функция не имеет доступа к данным из вызывающей функции.

Стоит отметить, что происходит не глубокое копирование, т.е. если передать структуру, которая хранит указатель на данные, то сама структура будет скопирована, а данные на которые она указывает нет. Для слайсов, словарей и каналов также не происходит копирования данных, т.к. эти три типа являются ссылочными, т.е. это указатели, которые сам язык автоматически разыменовывает при обращении к ним.

- по указателю. В этом случае функция получает **копии адресов** передаваемых данных. Зная адрес размещения оригинальных данных вызываемая функция может прочитать и изменить данные вызывающей функции.

Чтобы передать данные по указателю в качестве параметра функции используются указатели, а в качестве аргумента – адреса данных

```
func swap(a, b int){  
    var t int = a  
    a = b  
    b = t  
}  
  
func main(){  
    first, second := 10, 20  
    fmt.Println(first, second)  
    swap(first, second)  
    fmt.Println(first, second)  
}
```

10 20
10 20

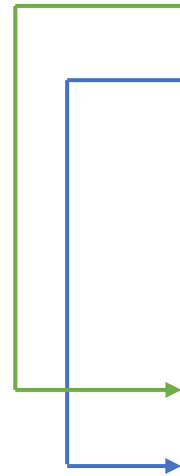
```
func swap(a, b *int){  
    var t int = *a  
    *a = *b  
    *b = t  
}  
  
func main(){  
    first, second := 10, 20  
    fmt.Println(first, second)  
    swap(&first, &second)  
    fmt.Println(first, second)  
}
```

10 20
20 10

Переменные в памяти

Состояние переменных в памяти сразу после вызова функции `swap`

```
func swap(a, b *int){  
    var t int = *a  
    *a = *b  
    *b = t  
}  
  
func main(){  
    first, second := 10, 20  
    fmt.Println(first, second)  
    swap(&first, &second)  
    fmt.Println(first, second)  
}
```



Имя	Тип	Значение	Адрес
...			
b	*int	0xc000074018	0xc00007e028
a	*int	0xc000074010	0xc00007e020
...			
t	int	0	0xc000074038
...			
second	int	20	0xc000074018
first	int	10	0xc000074010
...			

Указатели

Переменная-указатель предназначена для хранения адреса в памяти. Чтобы создать переменную-указатель нужно перед типом данных поставить символ звёздочка:

```
var pointer *тип_данных
```

```
var pointer1 *int      // Адрес int-а
```

```
var pointer2 *float64  // Адрес float64
```

```
var pointer3 *[3]int    // Адрес массива из 3х int-ов
```

Переменная-указатель по умолчанию имеет специальное значение `nil`. Это говорит о том, что этот указатель не хранит адрес (никуда не указывает).

```
pointer1 == nil  // true
```

Указатели – получение адреса

Чтобы сохранить в переменную адрес его сначала нужно получить. Для этого используется оператор взятия адреса – амперсанд (&). Его нужно написать перед именем объекта адрес которого нам нужен (например переменной) или получить как результат работы функции.

```
var value int = 10
var pointer *int = nil
pointer = &value           // Получаем адрес переменной value
```

Положить в переменную что-то кроме адреса нельзя

```
var pointer2 *int = value    // Ошибка! pointer2 ждёт адрес, а не значение
```

Не у всех программных объектов есть адреса

```
var pointer3 *int = &5       // Ошибка! У 5 нет адреса в памяти
```

Совместимы только адреса объектов одного типа

```
var valuef float64 = 10.0
var pointer4 *int = &valuef  // Ошибка! Не тот тип адреса
```

Указатели – доступ к значению через адрес

Зная адрес можно получить доступ к данным расположенным по этому адресу. Через адрес можно как читать данные, так и изменять их. Для получения доступа используется оператор разыменования (звёздочка) который нужно написать перед **переменной-указателем**:

```
var value int = 10
var pointer *int = &value // Получаем адрес переменной value
fmt.Println(*pointer)      // Читаем данные из value через её адрес

*pointer = 20              // Меняем данные в value через её адрес
fmt.Println(value)         // Проверяем
```

Не имеет значение каким образом получен адрес, если к нему применить оператор разыменования, то мы получим то, на что указывает адрес. Если к результату применить оператор взятия адреса, то мы снова получим адрес и т.д.

```
fmt.Println(&*&value)      // Можно так. Это тоже самое, что и просто value
```

Здесь операторы применяются справа налево (<--) начиная от имени переменной.

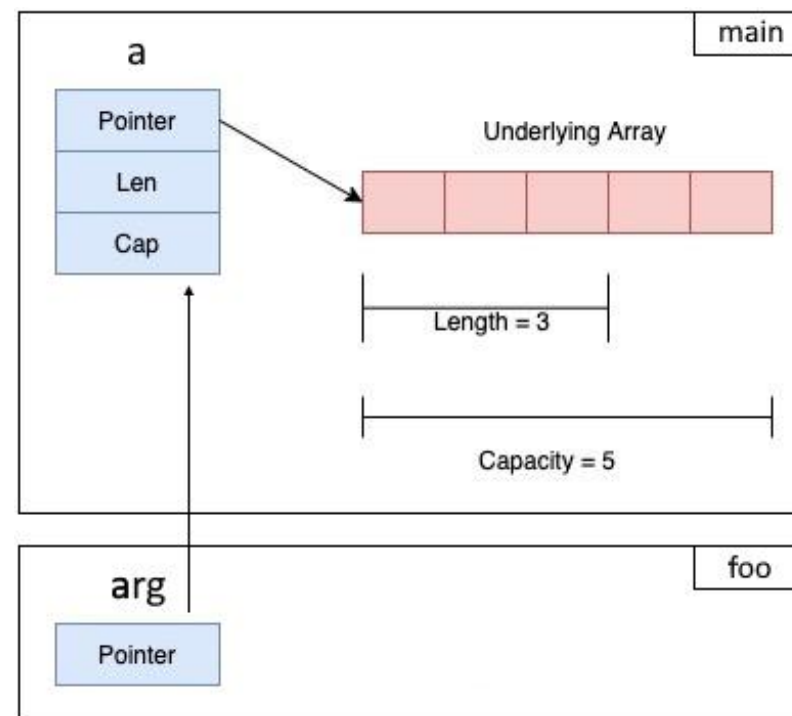
Передача слайсов в функцию

Слайс устроен так:

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

При передаче слайса по значению в функцию получается:

```
func foo(arg []int){  
    fmt.Print(arg)  
}  
  
func main(){  
    a := make([]int, 3, 5)  
    foo(a)  
}
```



Передача слайсов в функцию

```
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 = 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]
```

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)
    abc = append(abc, 4)
    fmt.Println(abc)
}
```

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

```
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]
```

Возврат результата по указателю

В Go данные из функции можно возвращать:

- по значению. В этом случае вызывающая функция получит **копию данных** которые были в вызываемой функции.
- по указателю. В этом случае вызывающая функция получает **копии адресов** объектов созданных или переданных в вызываемую функцию. В Go возврат результата по указателю безопасен, т.к. за удаление объектов отвечает сборщик мусора (GC) и сами объекты создаются с учётом того будут ли они чисто локальными по отношению к функции или их передадут наружу как результат.

```
func createBord() [3][3]int{  
    var bord = [3][3]int{{1,1,1},  
                        {1,1,1},  
                        {1,1,1}}  
  
    return bord  
}  
  
func main(){  
    fmt.Println(createBord())  
}
```

```
[[1 1 1] [1 1 1] [1 1 1]]
```

```
func createBord() *[3][3]int{  
    var bord = [3][3]int{{1,1,1},  
                        {1,1,1},  
                        {1,1,1}}  
  
    return &bord  
}  
  
func main(){  
    fmt.Println(*createBord())  
}
```

```
[[1 1 1] [1 1 1] [1 1 1]]
```

Область видимости (scope)

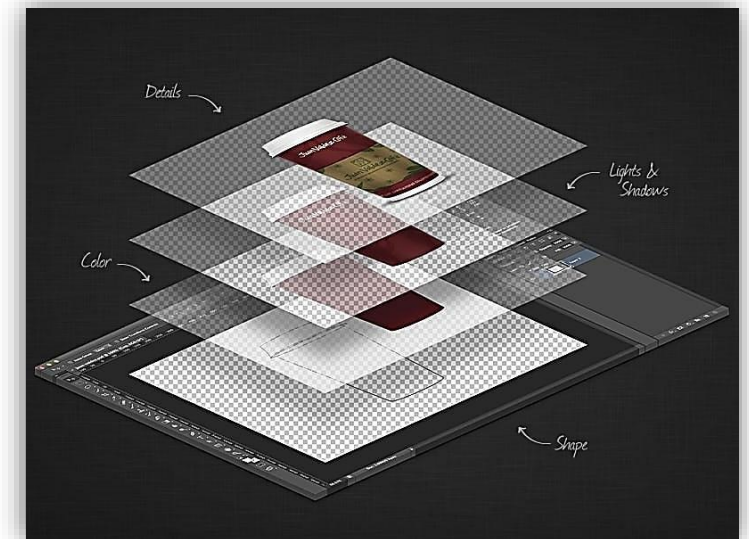
Область видимости (scope)

Область видимости — **часть программы** (кода), в пределах которой идентификатор, объявленный как имя некоторой программной сущности (обычно — переменной, типа данных или функции), остаётся связанным с этой сущностью, то есть позволяет посредством себя обратиться к ней.

Говорят, что идентификатор объекта «виден» в определённом месте программы, если в данном месте по нему можно обратиться к данному объекту. За пределами области видимости тот же самый идентификатор может быть связан с другой переменной или функцией, либо быть свободным (не связанным ни с какой из них). Область видимости может, но не обязана совпадать с областью существования объекта, с которым связано имя.

В пределах одной области видимости идентификатор может быть связан только с одной программной сущностью. Добавление идентификатора в область видимости происходит через объявление/определение. Таким образом, в одной области видимости может быть только одно определение.

Области видимости входят друг в друга и **составляют иерархию**, от локальной области видимости, ограниченную функцией (или даже её частью), до глобальной, идентификаторы которой доступны во всей программе. Также в зависимости от правил конкретного языка программирования области видимости могут быть реализованы двумя способами: лексически (статически) или динамически



Shadowing

Говорят, что идентификатор скрывает "затеняет" другой идентификатор, если он переопределяет его в более конкретной области видимости, т.е. ближе по иерархии к точке, в которой указано обращение к идентификатору.

Т.к. в процессе связывания выбирается идентификатор из ближайшей области видимости, одноимённая переменная становится, как бы, на время невидимой (скрытой в тени).

Go

```
func main(){  
    x := 0  
    for x:=0; x<3; x++ {  
        fmt.Println(x)  
    }  
  
    fmt.Println(x)  
}
```

0
1
2
0

Python

```
x = 0  
  
def outer():  
    x = 1  
    print("outer:", x)  
  
outer()  
print("global:", x)
```

1
0

Связывание идентификатора (binding)

Связывание идентификатора — **процесс** определения программного объекта, доступ к которому даёт идентификатор в конкретном месте программы и в конкретный момент её выполнения.

- **лексическое (статическое) связывание** (англ. *lexical (static) binding*), **лексическая область видимости**, или **лексический контекст** (англ. *lexical scope*): поиск объявлений подходящих для связывания происходит только с учётом размещения идентификатора в коде.
- **динамическое связывание** (англ. *dynamic binding*), или **динамическая область видимости**, или **динамический контекст** (англ. *dynamic scope*): поиск объявлений подходящих для связывания происходит с учётом истории выполнения кода.

C++

```
void print_x(){  
    std::cout << x;  
}  
  
int x = 10;  
int main(){  
    print_x();  
}
```

Ошибка
компиляции

Python

```
def print_x():  
    print(x)  
  
x = 10  
def main():  
    print_x()  
  
main()
```

10

Пустой идентификатор (blank identifier)

Пустой идентификатор представлен символом подчеркивания `_`. Он служит анонимным заполнителем вместо обычного (непустого) идентификатора и имеет особое значение в объявлениях, в качестве операнда и в операторах присваивания.

Пустой идентификатор может использоваться как любой другой идентификатор **в объявлении**, но для него не выполняется привязка и, следовательно, объявлять его не нужно.

Т.к. привязка для пустого идентификатора не выполняется, то и использование его в обычных выражениях нельзя.

Используется, когда функция или выражение возвращает больше значений, чем вам реально нужно для игнорирования лишних.

```
func main() {  
    fd, err := os.Open("test.go")  
    if err != nil {  
        log.Fatal(err)  
    }  
}
```

Ошибка fd объявлен, но не используется

```
func main() {  
    _, err := os.Open("test.go")  
    if err != nil {  
        log.Fatal(err)  
    }  
}
```

OK

Начало области видимости

В зависимости от языка программирования точка **начала** действия объявления внутри области видимости может определяться по разному:

- Начало действия от точки объявления и ниже по коду. В этом случае в процессе связывания просматриваются объявления текущей области видимости от точки обращения к идентификатору и выше по коду. Если объявлений не обнаружено, то поиск продолжается в следующей по иерархии области видимости.
- Идентификатор действует сразу во всей области видимости независимо от того, в каком её месте находится объявление. В этом случае в процессе связывания просматриваются вся текущей области видимости и если объявление найдено, то происходит связывание, а если не найдено, то поиск продолжается в следующей по иерархии области видимости

Go

```
var scope = "global"
func f() {
    fmt.Println(scope)
    var scope = "local"
    fmt.Println(scope)
}
```

global
local

JavaScript

```
var scope = "global";
function f() {
    alert(scope);
    var scope = "local";
    alert(scope);
}
```

undefined
local

Блочная область видимости

Ещё один нюанс в семантике лексической области видимости — наличие или отсутствие так называемой «блочной видимости», то есть возможности объявить локальную переменную не только внутри функции, процедуры или модуля, но и внутри отдельного блока команд

- В Go блок операторов образует локальную область видимости, и объявляемая внутри цикла переменная `x` — это новая переменная, областью видимости которой является только тело цикла.
- В JavaScript нет блочной области видимости (в версиях, предшествующих ES6), а повторное объявление локальной переменной работает просто как обычное присваивание.

Go

```
func f() {  
    var x = 3  
    fmt.Println(x)  
    for i := 10; i < 30; i += 10 {  
        var x = i  
        fmt.Println(x)  
    }  
    fmt.Println(x)  
}
```

3
10
20
3

JavaScript

```
function f() {  
    var x = 3;  
    alert(x);  
    for (var i = 10; i < 30; i += 10) {  
        var x = i;  
        alert(x);  
    }  
    alert(x);  
}
```

3
10
20
20

Блоки в Go

Universe block. Стартовый блок в иерархии. Содержит весь код из всех файлов используемых в программе. Содержит [предварительно объявленные идентификаторы](#) типов, констант, функций.

Package block. У каждого пакета есть собственный блок. Он содержит весь исходный код из всех файлов которые относятся к одному пакету. Файлы как правило размещают в одной папке, но это не обязательно.

The file block. Каждый файл содержит файловый блок, содержащий весь исходный код в этом файле.

The control structure block. Считается, что каждый оператор "if", "for" и "switch" находится в своем собственном неявном блоке. Блок начинается перед ключевым словом и заканчивается после последней закрывающей фигурной скобки.

Clause block. Каждое предложение "case" в операторе "switch" или "select" действует как неявный блок.

Блоки могут быть вложенными и создают отдельные области видимости.

Области видимости в Go

- Областью действия предварительно объявленного идентификатора является Universe block .
- Идентификаторы, обозначающего константу, тип, переменную или функцию (но не метод), объявленную на верхнем уровне (вне какой-либо функции) видны во всех файлах одного пакета.
- Область видимости имени импортируемого пакета - это файловый блок файла, содержащего объявление импорта.
- Областью видимости идентификатора, обозначающего приемник метода, параметр функции или переменную возврата, является тело функции.
- Область видимости идентификатора, обозначающего параметр типа функции или объявленного приёмником метода, начинается после имени функции и заканчивается в конце тела функции.
- Область видимости идентификатора, обозначающего параметр типа, начинается после имени типа и заканчивается в конце спецификации типа.
- Область видимости идентификатора константы или переменной, объявленный внутри функции начинается в конце объявления и действует до конца функции или блока.
- Область видимости идентификатора типа, объявленного внутри функции. Начинается с идентификатора в объявлении типа и действует до конца функции или блока.

Области видимости в Go

Метки используются в операторах "break", "continue" и "goto".

В отличие от других идентификаторов, метки не ограничены областью блока и не конфликтуют с идентификаторами, которые не являются метками. Область видимости метки - это тело функции, в которой она объявлена исключая тело любой вложенной функции.

Запрещено определять метку, которая никогда не используется.
Запрещено "перепрыгивать" с помощью метки через объявления **новых** переменных. Т.е. перепрыгивать назад можно.

RowLoop:

```
for y, row := range rows {  
    for x, data := range row {  
        if data == endOfRow {  
            continue RowLoop  
        }  
        row[x] = data + bias(x, y)  
    }  
}
```

OuterLoop:

```
for i = 0; i < n; i++ {  
    for j = 0; j < m; j++ {  
        switch a[i][j] {  
        case nil:  
            state = Error  
            break OuterLoop  
        case item:  
            state = Found  
            break OuterLoop  
        }  
    }  
}
```

Анонимные функции

Анонимные функции

Анонимные функции - это функции, которым не назначен идентификатор. Они отличаются от обычных функций также тем, что они могут определяться внутри других функций и также могут иметь доступ к контексту выполнения.

```
func main() {  
    f := func(x, y int) int{ return x + y }  
  
    fmt.Println(f(3, 4))    // 7  
    fmt.Println(f(6, 7))    // 13  
}
```

Анонимная функция как аргумент функции

Очень удобно использовать анонимные функции в качестве аргументов других функций:

```
package main
```

```
import "fmt"
```

```
func action(n1 int, n2 int, operation func(int, int) int){  
    result := operation(n1, n2)  
    fmt.Println(result)  
}
```

```
func main() {  
    action(10, 25, func (x int, y int) int { return x + y })    // 35  
    action(5, 6, func (x int, y int) int { return x * y })      // 30  
}
```

Анонимная функция как результат функции

Анонимная функция может быть результатом другой функции:

```
package main

import "fmt"

func selectFn(n int) (func(int, int) int){
    if n==1 {
        return func(x int, y int) int{ return x + y}
    }else if n==2{
        return func(x int, y int) int{ return x - y}
    }else{
        return func(x int, y int) int{ return x * y}
    }
}

func main() {
    f := selectFn(1)
    fmt.Println(f(2, 3))    // 5
    fmt.Println(f(4, 5))    // 9
    fmt.Println(f(7, 6))    // 13
}
```

Доступ к окружению (замыкания)

Преимуществом анонимных функций является то, что они имеют доступ к окружению, в котором они определяются.

Замыкание (англ. closure) в программировании — функция первого класса, в теле которой присутствуют ссылки на переменные, объявленные вне тела этой функции в окружающем коде и не являющиеся её параметрами. Говоря другим языком, замыкание — функция, которая ссылается на свободные переменные в своей области видимости.

Замыкание — это особый вид функции. Она определена в теле другой функции и создаётся каждый раз во время её выполнения. Синтаксически это выглядит как функция, находящаяся целиком в теле другой функции. При этом вложенная внутренняя функция содержит ссылки на локальные переменные внешней функции. Каждый раз при выполнении внешней функции происходит создание нового экземпляра внутренней функции, с новыми ссылками на переменные внешней функции.

Нелокальные переменные — с точки зрения функции это переменные которые не являются её локальными переменными, параметрами, переменными возврата, глобальными переменными.

Go

```
func add(x int) (func(int) int){
    return func(y int) int{
        z := x + y
        fmt.Printf("%d+%d=%d", x, y, z)
        return z
    }
}

add_3 := add(3)
add_3(6)
```

add_3(6)
вернёт 9 и
выведет в
терминал
3+6=9

JavaScript

```
const add = function(x) {
    return function(y) {
        const z = x + y;
        console.log(x + '+' + y + '=' + z);
        return z;
    };
};

const add_3 = add(3);

add_3(6);
```

add_3(6)
вернёт 9 и
выведет в
терминал
3+6=9

Классификация функций

Функцию называют функцией **первого класса**, если она является объектом первого класса.

Объект называют «объектом первого класса», если он:

- может быть сохранен в переменной или структурах данных;
- может быть передан в функцию как аргумент;
- может быть возвращен из функции как результат;
- может быть создан во время выполнения программы;
- не зависит от именования.

Функция высшего порядка — в программировании функция, принимающая в качестве аргументов другие функции или возвращающая другую функцию в качестве результата. Основная идея состоит в том, что функции имеют тот же статус, что и другие объекты данных.

Чистая функция — это функция, которая:

- является детерминированной. Функция является детерминированной, если для одного и того же набора входных значений она возвращает одинаковый результат. Т.е. функция не зависит не от чего, кроме своих параметров и не имеет внутреннего состояния.
- не обладает побочными эффектами. В императивных языках некоторые функции в процессе выполнения своих вычислений могут модифицировать значения глобальных переменных, осуществлять операции ввода-вывода, реагировать на исключительные ситуации, вызывая их обработчики. Такие функции называются **функциями с побочными эффектами**. Другим видом побочных эффектов является модификация переданных в функцию параметров (переменных), когда в процессе вычисления выходного значения функции изменяется и значение входного параметра.