GeekBrains

Go. Уровень 1

Сложные типы данных: указатели

На этом уроке

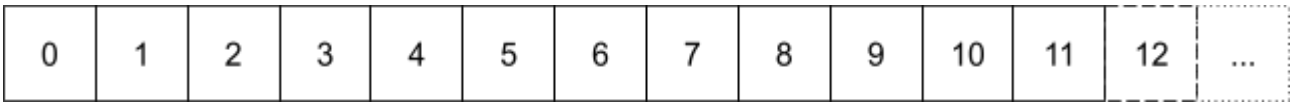
1. Поймём, как устроена память.
2. Узнаем, что такое указатели и для чего они нужны.
3. Научимся работать с указателями в Go.
4. Рассмотрим особенности копирования слайсов и мап.
5. Научимся изменять значение параметра внутри функции.

Оглавление

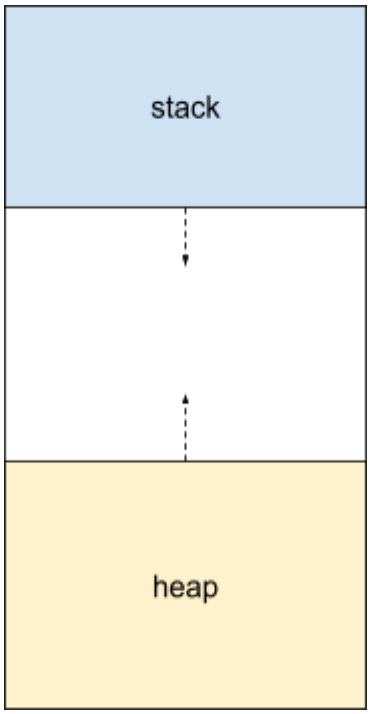
О работе с памятью

Вот и настал момент поговорить про устройство компьютера и память! Концептуально современному компьютеру нужны всего три устройства для работы: процессор, оперативная память и устройства ввода/вывода. В этом уроке нас интересует оперативная память.

Память компьютера можно представить как очень **длинный массив**, **каждая ячейка** которого **пронумерована** и может хранить один байт информации. По номеру ячейки можно получить доступ к любому байту оперативной памяти. Поэтому её называют памятью с произвольным доступом (RAM, Random Access Memory).



Стоит заметить, что компьютерные процессоры устроены таким образом, что доступ к непрерывным кускам памяти быстрее, чем доступ к разрозненным ячейкам.



При запуске программы операционная система вашего компьютера создаёт процесс. Под каждый процесс операционная система выделяет заранее определённое количество памяти. Эта память логически разделяется на стек (stack) и кучу (heap).

Размер стека процесса не может меняться на протяжении его работы. Зато эта память быстрая, так как выделяется непрерывным куском, и хорошо структурирована.

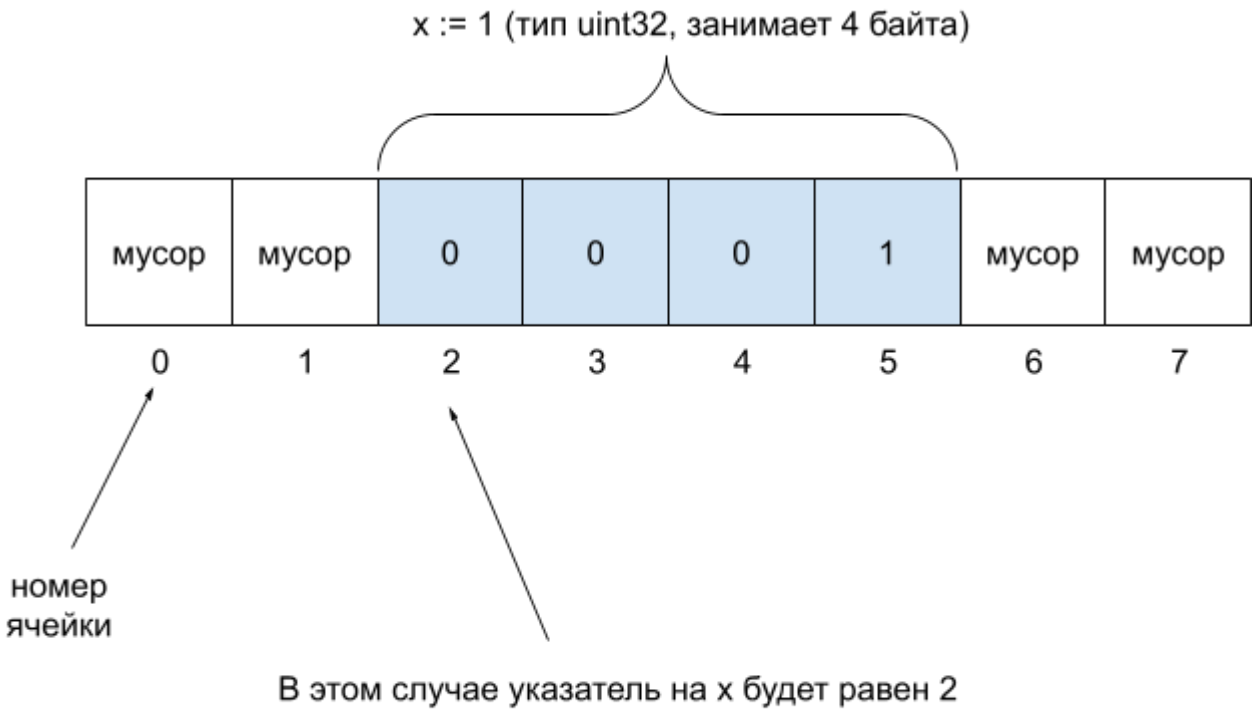
Память в куче не структурирована, но за счёт этого её можно увеличивать во время работы программы.

Из-за того, что количество памяти, выделенной на стек, не меняется во время работы программы, почти все языки программирования используют стек для хранения локальных переменных функций. Это удобно, так как добавление переменной в стек и удаление оттуда — очень быстрая и простая операция.

Можно заметить, что подход, когда каждая ячейка памяти доступна для чтения и записи, небезопасный: любая программа может получить доступ к данным другой программы. Поэтому в 1970-х годах в появилась технология, которая позволяет программе получать доступ только ко своей памяти. Эта технология встроена в процессоры. Её часто называют защищенным режимом (protected mode). Благодаря этому можно быть уверенным, что чужая программа не получит доступ к памяти вашей программы.

Что такое указатель (pointer)?

Указатель в самом примитивном случае — обычное число, которое трактуется как номер ячейки памяти компьютера. То есть, например, есть переменная `x := 1`, значит, в каком-то ряду последовательных ячеек памяти хранятся байты, которые трактуются как число 1. Указатель — это номер первой ячейки памяти этого ряда. Номер ячейки памяти называют адресом.



В этом случае указатель на x будет равен 2

Как работать с указателями?

Для работы с указателем есть два основных оператора: оператор взятия указателя на переменную (или взятия адреса переменной) и оператор разыменования указателя.

В Go эти два оператора выглядят так:

```
package main

import "fmt"

func main() {
	var a uint32 = 1
	var b *uint32 = &a // взятие указателя на a. Тип указателя на тип T — *T
	fmt.Println("b =", b)
	c := *b // взятие значения по указателю
	fmt.Println("c =", c)
}
```

```
> go run app.go
```

```
b = 0xc000016090
c = 1
```

Зачем всё это нужно?

Мы уже знаем, что в Go все значения копируются при присваивании или передаче в функцию. Из-за этого при изменении копии оригинал оставался неизменным. Иногда нужно изменить внешнюю переменную внутри функции, указатели позволяют это сделать.

Например, в этом коде мы считываем пользовательский ввод в переменную с помощью функции `fmt.Scan`.

```
package main

import (
   "fmt"
   "os"
)

func main() {
   var a uint32 // заготавливаем переменную для считывания

      // fmt.Scan позволяет считать данные из стандартного потока ввода в
переменную.
   // На вход требует указатель на эту переменную.
   _, err := fmt.Scan(&a) // считываем ввод в переменную
   if err != nil {
            fmt.Println(err)
            os.Exit(1)
   }

   fmt.Println("your input is", a)
}
```

```
> go run app.go
23
your input is 23
```

Помимо дополнительной функциональности, указатели помогают оптимизировать код. Допустим, у нас есть структура с 5 полями типа uint64.

```
type ExampleStruct struct {
   field1 uint64
   field2 uint64
   field3 uint64
   field4 uint64
   field5 uint64
}
```

Структура скопируется, если передать её в функцию. То есть компьютеру придётся скопировать 5 чисел.

```
func exampleFunc(s ExampleStruct) {
   // ...
}
```

Однако мы помним, что указатель — обычное число. Поэтому при передаче в функцию указателя на структуру копируется только указатель, то есть только одно число.

```
func exampleFunc(s *ExampleStruct) {
   // ...
}
```

То же самое справедливо для массивов.

Нулевой указатель

Как мы помним, в Go у каждого типа есть нулевое значение. Нулевое значение любого указателя — `nil`. Можно проследить связь с функциональными типами, слайсами, мапами и их копированием.

Особенность нулевого указателя в том, что его нельзя разыменовать. Разыменование нулевого указателя приводит к панике.

```
package main

import "fmt"

func main() {
   var a *uint32 = nil
   fmt.Println("value of *a:", *a)
}
```

```
> go run app.go
```

```
panic: runtime error: invalid memory address or nil pointer dereference
[signal SIGSEGV: segmentation violation code=0x1 addr=0x0 pc=0x109cf33]

goroutine 1 [running]:
main.main()
      /Users/dedefer/Desktop/dev/golang/app.go:7 +0x23
exit status 2
```

Особенности указателей для разных типов данных

Структуры

Методы, привязанные к указателю на тип

В разговоре про структуры и методы мы говорили, что метод можно привязать к любому собственному типу данных. Так вот, указатель на тип — это тоже тип, поэтому можно привязать метод, например, к указателю на структуру.

```go
package main

import "fmt"

type User struct {
   Name    string
   Surname string
   Phone   string
}

// привязываем метод к типу *User
func (u *User) SetPhone(phone string) error {
   if err := validatePhone(phone); err != nil {
       return err
   }
   u.Phone = phone
   return nil
}

func validatePhone(phone string) error {
   // логика валидации
   return nil
}

func main() {
   user := User{Name: "Иван", Surname: "Иванов"}
   if err := user.SetPhone("+79165230808"); err != nil {
```

```
        panic(err)
    }
    fmt.Printf("user: %+v\n", user)
}
```

```
> go run app.go
user: {Name:Иван Surname:Иванов Phone:+79165230808}
```

В этом примере важно заметить, что:

1. Методы, привязанные к указателю на структуру, можно использовать для изменения структуры.
2. При вызове метода, привязанного к указателю на структуру, копируется только указатель, то есть число, а не три строковых значения.
3. Оператор `.` для доступа к методам структуры позволяет использовать методы указателя на структуру. Если бы такого поведения оператора `.` не было, пришлось бы писать `(&user).SetPhone(...)`.

Аналогично можно определять методы не только для указателей на структуры, но и для указателей на другие типы данных.

```
package main

import (
    "errors"
    "fmt"
    "math"
)

type Uint32NoOverflow uint32

func (i *Uint32NoOverflow) Inc() error {
    if *i == math.MaxUint32 {
        return errors.New("Uint32Overflow")
    }
    *i++
    return nil
}

func (i Uint32NoOverflow) Get() uint32 { return uint32(i) }

func main() {
    a := Uint32NoOverflow(math.MaxUint32 - 2)
    for i := 0; i < 100; i++ {
        fmt.Println(a.Get())
        if err := a.Inc(); err != nil {
            fmt.Println(err)
            break
        }
    }
```

```
}
```

```
> go run app.go
4294967293
4294967294
4294967295
Uint32Overflow
```

На практике чаще всего методы используют именно со структурами. В таком случае методы почти всегда определяют именно для указателя на структуру, чтобы уменьшить затраты на копирование больших структур и облегчить внесение изменений в код.

Оператор `.` умеет работать и в обратную сторону.

```
package main

import "fmt"

type User struct {
   Name    string
   Surname string
}

// метод GetFullName привязан к структуре, а не к указателю
func (u User) GetFullName() string {
   return u.Name + " " + u.Surname
}


func main() {

   // user имеет тип *User,
   // так как  используется конструкция &User{...}.
   // То есть мы создаём структуру и
   // сразу же используем оператор взятия адреса на неё.
   user := &User{Name: "Иван", Surname: "Иванов"}

   userFullName := user.GetFullName()

   fmt.Printf("userFullName: %s\n", userFullName)
}
```

```
> go run app.go
userFullName: Иван Иванов
```

В этом случае оператор `.` работает как `(*user).GetFullName()`.

Расположение структур в памяти и связь с указателями

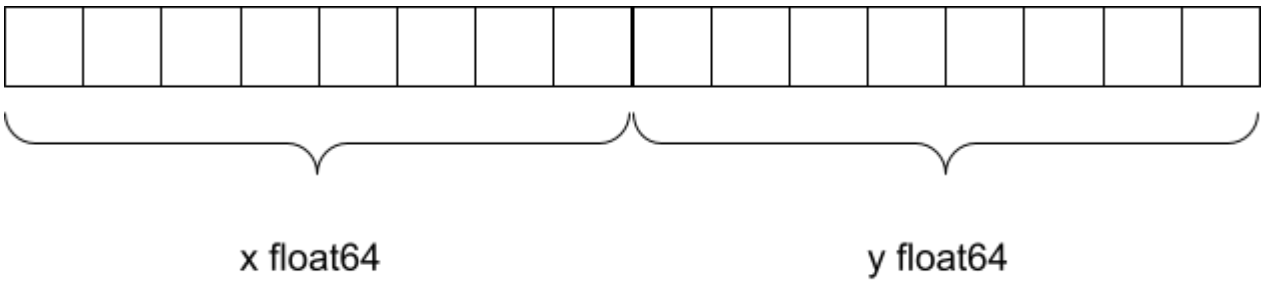
Теперь углубимся в расположение структур в памяти.
В предыдущем уроке мы говорили, что структура в Go занимает количество памяти, равное сумме полей.

Вспомним структуру `Point` из предыдущего урока.

```
type Point struct {
   x float64
   y float64
}
```

Мы говорили, что эта структура занимает 16 байт в памяти, так как каждое поле имеет тип float64, который занимает по 8 байт.

В памяти эта структура выглядит как неразрывный блок из 16 байт.



Мы говорили, что нельзя делать рекурсивные структуры.

Рассмотрим структуру одного элемента двусвязного списка.

```
type Node struct {
   value        int64
   previousNode Node
   nextNode     Node
}
```

Эта структура вызовет ошибку компиляции, так как для неё невозможно подсчитать количество требуемой памяти. `Node` в этом случае хранит внутри себя поле типа `Node`, которое также должно хранить в себе поле типа `Node`, и так далее. В связи с этим размер структуры потенциально бесконечен.

Теперь вспомним, что указатель на любой тип — число. То есть все указатели имеют определённый размер.

```
type Node struct {
   value        int64
   previousNode *Node
   nextNode     *Node
}
```

Такая структура не вызовет ошибку компиляции, так как она по сути хранит 3 числа, а значит её размер можно точно определить.

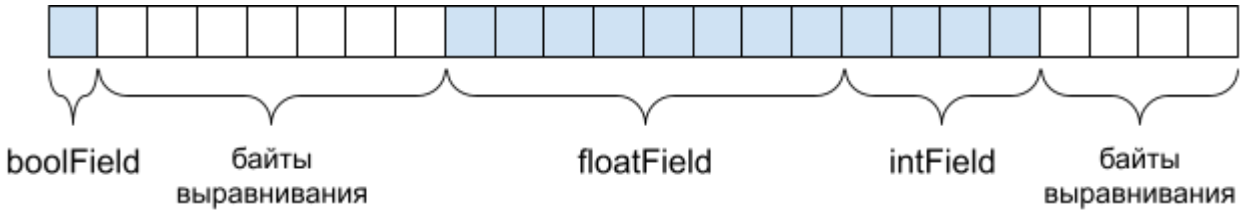
Продвинутый материал

В связи с устройством процессоров невыгодно загружать из памяти произвольное количество байт. Процессор загружает данные из памяти блоками определённого размера.

Компиляторы знают об этой особенности, поэтому применяют оптимизацию под названием выравнивание данных (data alignment). Разберемся на примере.

```
type ExampleStruct struct {
   boolField  bool    // 1 байт
   floatField float64 // 8 байт
   intField   int32   // 4 байта
}
```

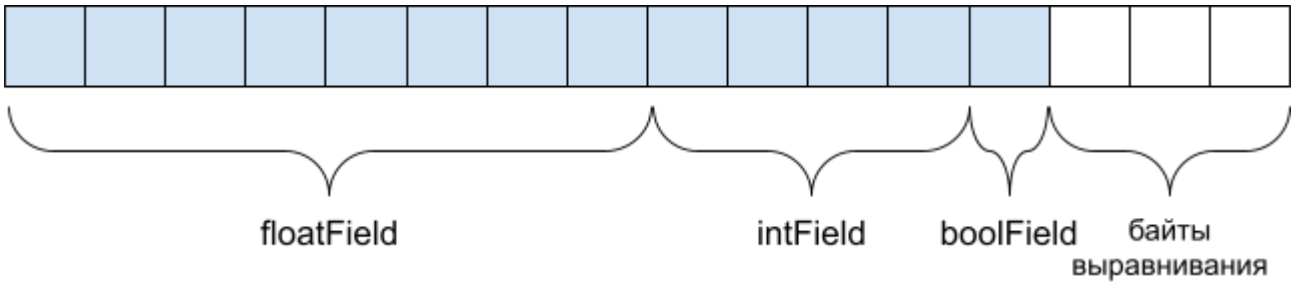
Казалось бы, эта структура должна занимать 1 + 8 + 4 = 13 байт памяти, однако компилятор выровнит поля так:



Из-за этого структура на самом деле будет занимать 24 байта.

Можно реорганизовать порядок полей в структуре и уменьшить её размер до 16 байт.

```
type ExampleStruct struct {
   floatField float64 // 8 байт
   intField   int32   // 4 байта
   boolField  bool    // 1 байт
}
```

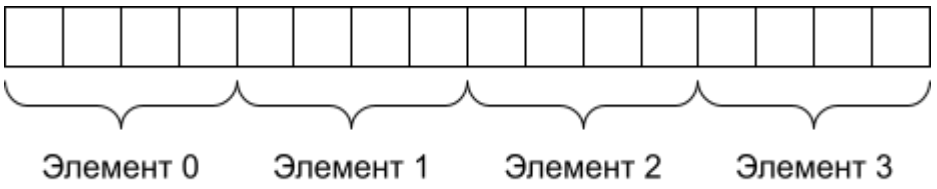


Это поведение компилятора может поменяться. Например, компилятор Rust умеет перемешивать поля самостоятельно, чтобы уменьшить количество байт выравнивания. Вероятно, в Go тоже появится эта фича.

Массивы

Мы уже говорили, что массивы располагаются в памяти неразрывным куском.

Допустим, мы создали переменную типа `[4]int32`. Вот так он будет выглядеть в памяти:



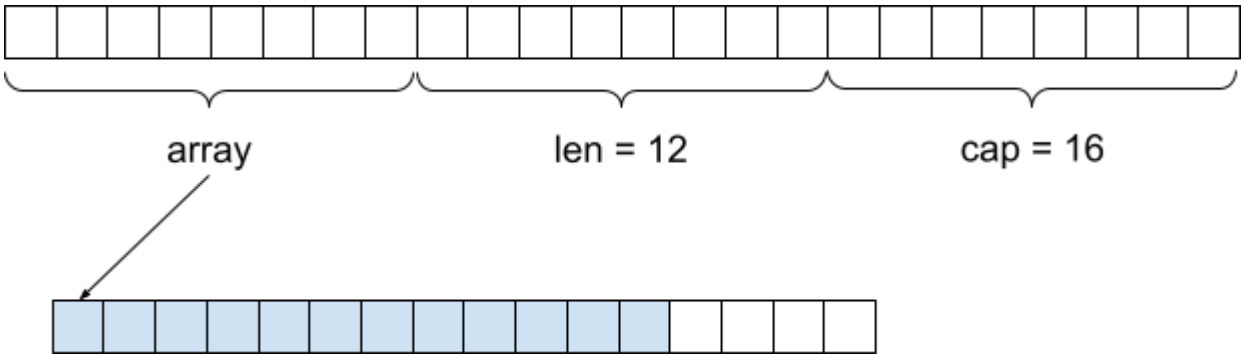
Нет ничего проще для процессора, чем обработка данных из массива.

Слайсы

Наконец, мы готовы понять, почему слайс так странно копируется. К счастью большая часть Go написана на Go, в том числе и слайсы. На официальном сайте мы можем увидеть, как реализован слайс.

```
type slice struct {
   array unsafe.Pointer
   len   int
   cap   int
}
```

Слайс — это обычная структура, состоящая из трёх полей. `array unsafe.Pointer` — указатель на нижележащий кусок памяти. То есть можно считать это настоящим массивом. `len` и `cap` — это длина и вместимость этого куска памяти `array`. Сам же кусок памяти, на который указывает array, располагается точно так же, как массив из предыдущего раздела.



Именно поэтому при копировании и передаче слайса в функцию его содержимое можно изменять. Копируется только указатель на кусок памяти, а не она сама.

Именно поэтому нулевое значение слайса — `nil`.

Именно поэтому `append` возвращает новый слайс. Указатель `array` изменится, если `cap` не хватит для вставляемых элементов.

Мапы

Мапа очень похожа на слайс с точки зрения устройства. В рассматриваемом примере уберём большинство полей, так как они неинтересны в рамках урока. Ссылка на код.

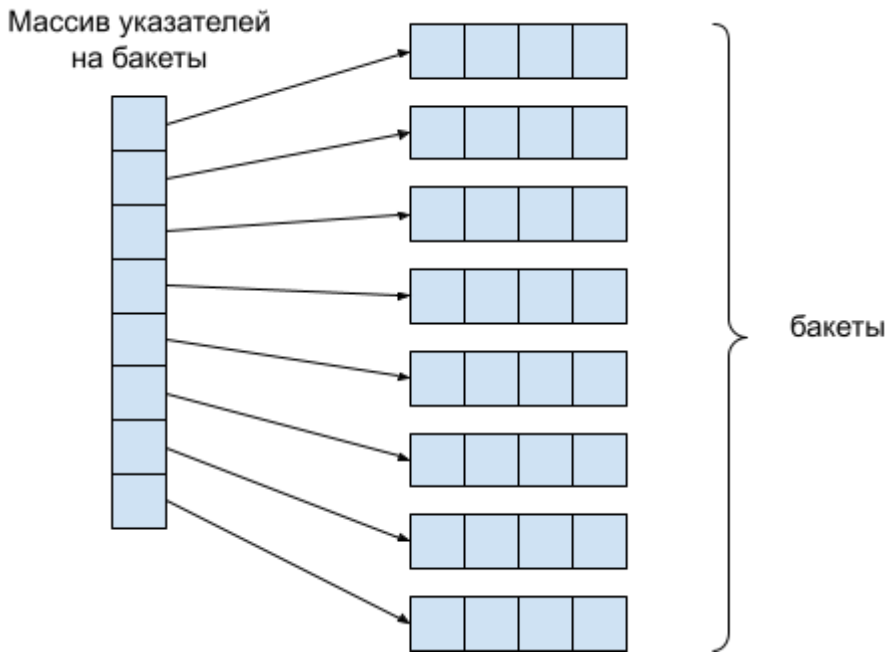
```
type hmap struct {
   count int   // количество занятых ячеек
   B     uint8 // 2^B - всего ячеек (почти cap)

   buckets unsafe.Pointer // массив бакетов

   // другие вспомогательные поля
}
```

Структура очень похожа на слайс, однако, вместо обычного массива мы видим указатель на массив бакетов `buckets`. Мапа в Go реализована в виде хеш-таблицы с разрешением коллизий методом цепочек.

Условно хеш-таблицу этого типа можно представить так:



Каждый бакет соответствует определённому хешу. Так как бакетов ограниченное количество, а вариаций ключей бесконечно много, то разные ключи могут иметь одинаковые хеши.

Ситуация, когда два разных ключа имеют одинаковый хеш, называется коллизией хеша. Чтобы разрешить коллизию Go кладет пары ключ-значение в один бакет.
Алгоритм поиска по ключу в такой хеш-таблице примерно такой:

1. Находим хеш от ключа.
2. По хешу находим бакет, в котором должно находиться значение по этому хешу.
3. Проверяем все пары «ключ-значение» в бакете на равенство с ключом.

Первые два шага всегда занимают одно и то же время, так как это обычные арифметические операции. Последний шаг — проход по небольшому массиву (обычно длина < 10 элементов). Таким образом, поиск в мапе по ключу занимает константное время O(1).

Замыкания

Рассмотрим пример из прошлого урока с `defer`, циклом и замыканием.

```
package main

func main() {
    for i := 0; i < 5; i++ {
        defer func() {
            println(i)
        }()
    }
}
```

```
> go run app.go
```

```
5
5
5
5
5
```

Такое происходит из-за того, что все переменные, захватываемые замыканием, захватываются по указателю. То есть замыкание можно представить как функцию + список указателей на захваченные переменные из внешней области видимости.

Заключение

В этом уроке мы познакомились с концепцией указателей и устройством встроенных типов данных в Go. Указатели в Go применяются повсеместно. К сожалению, понять указатели до конца можно только на практике, так что желаем вам удачи в работе над практическим заданием.

Главное в этом уроке:

- Указатель — это обычное число.
- Указатель трактуется как номер ячейки памяти.
- Для работы с указателями есть операторы взятия адреса (&) и разыменования (*).
- Существует нулевой указатель nil.
- Разыменование нулевого указателя приводит к панике.
- Функции могут изменять параметры, переданные по указателям.
- Методы можно привязывать к указателям на типы, такие методы могут изменять целевую переменную.
- Странности копирования слайсов и мап объясняются использованием указателей в их устройстве.
- Странности поведения defer и замыкания объясняется захватом внешних переменных по указателю.

Практическое задание

1. Проанализируйте задания предыдущих уроков.
    a. В каких случаях необходима была явная передача указателя в качестве входных параметров и возвращаемых результатов или в качестве приёмника в методах?
    b. В каких случаях мы фактически имеем дело с указателями при передаче параметров, хотя явно их не указываем?
2. Для арифметического умножения и разыменования указателей в Go используется один и тот же символ — оператор (*). Как вы думаете, как компилятор Go понимает, в каких случаях в выражении имеется в виду умножение, а в каких — разыменование указателя?

Список литературы

1. Книга «Современные операционные системы» (Таненбаум Э. С., Бос Х.) — Глава 3. Управление памятью.

2. Про выделение памяти на стеке и на куче в Go: https://golang.org/doc/faq#stack_or_heap
3. Understanding Allocations: the Stack and the Heap: Understanding Allocations: the Stack and the Heap — GopherCon SG 2019.
4. Illustrative Exploration of Go Memory Allocator: Illustrative Exploration of Go Memory Allocator / Ankur Anand.
5. Статья про размещение структур в памяти: How to organize the go struct, in order to save memory.
6. Зачем в Go амперсанд и звёздочка: Зачем в Go амперсанд и звёздочка (& и *)?.
7. Две статьи про указатели в блоге Дейва Чени: 3 thoughts on Pointers in Go и Understand Go pointers in less than 800 words or your money back.
8. Go FAQ: Pointers https://golang.org/doc/faq#Pointers
9. Ссылка на код мап: Source file src/runtime/map.go.
10. Ссылка на код слайсов: Source file src/runtime/slice.go