

Очень коротко о семестре.

В принципе, основная линия семестра не меняется: указатели, структуры данных, в частности, линейные структуры; рекурсия, её характерные применения, плюсы и минусы, где стоит разрекурсивливать и т.д. К разговору о структурах данных очень хорошо впишется разговор о построении библиотек. Раньше этот вопрос возникал уже в первом семестре, но там он как-то выглядит несколько преждевременным - до реального построения библиотек дети успевают это вопрос изрядно подзабыть. Здесь же всё получается уместно и логично, также как и разговор о механизмах передачи параметров. Ну, и к разговору об указателях хорошо впишется тщательный разговор о функциональных типах, переменных и параметрах функционального типа. Блок про рекурсию вообще особых изменений, кажется, не требует. А вот с комбинаторикой хочется что-то сделать, убрать её. Комбинаторика использовалась для иллюстрации рекурсивных алгоритмов и соответствующих нерекурсивных алгоритмов. И кажется, что она как-то ломала линию: содержание комбинаторного материала, а содержания там много, всё-таки перетягивает одеяло, сбивает фокус. По крайней мере сейчас кажется, что будет более уместно в качестве серии примеров рекурсии использовать какие-нибудь рекурсивные структуры данных, например, бинарная куча, дерево поиска (в том числе и со счётчиками), возможно, что и наворотами, может быть ещё какие-нибудь кучи. Ну, посмотрим...

V.01.

Общий обзор занятия.

И давайте начнём историю с указателями с адресов вообще, не привязываясь сразу к конструированию структур данных. Это, кажется даст детям немного времени, чтобы освоится с адресными величинами, осознать их смысл, повозиться с синтаксисом, в общем, почувствовать их. Причём мы сразу будем говорить и о типизованных указателях, и о бестиповых, и о "промежуточных" (unsafe pointer), и о преобразованиях pointer'ов. Для этого быстренько разберёмся с package unsafe.

Конструкция строк в Go и наличие в Go слайсов, способы хранения/представления этих данных дают весьма содержательный материал для всяческих разборок с pointer'ами. Также сюда довольно органично вписывается разговор о передаче параметров по значению/по адресу.

Чего не ставим в это занятие: передача параметра по имени - про области видимости (scope) поговорим чуть-чуть позже; про устройство массивов и слайсов говорим в это раз (а также структур и строк, хотя эти структуры я бы бросал в детей для более или менее самостоятельного микроисследования), а вот про преобразования-переезды строк и слайсов при выполнении с ними действий, про перераспределение памяти при таких действиях - в следующий раз.

Ну, само собой, упомянем и о сборке мусора. В самых общих чертах, на уровне, что такое есть. Впрочем, про сборку мусоравольно-невольно придётся упоминать часто и много, ну так и будем упоминать. Подробности и технические детали этого процесса, устройство сборщика мусора обсуждать не будем.

План занятия: новый материал - основные вопросы.

- pointer'ы. Что это такое, зачем и что с этим делать.**

- объявление и получение pointer'a
- константа типа pointer - nil
- создание указателей в run-time - функция new

- указатели (pointer'ы) и функции**

- формальные и фактические параметры
- передача [параметра] по адресу - указатели как параметры функции

- анатомия слайса и всё вокруг**

- package unsafe, unsafe pointer и действия с ними, преобразования указателей
- анатомия массива
- анатомия структуры (struct) - рекомендуется как исследовательское задание на практику
- анатомия слайса
- слайсы и массивы как параметры функций; передача массива по адресу vs передача соответствующего слайса
- анатомия строки (string) - рекомендуется как исследовательское задание на практику

Примечание.

Кажется мне, что тут некоторый перебор по объёму получается. Если так, то не берите в голову - что не влезет, то не влезет - следующее занятие есть

продолжение этого, так что всё, что не вместиится запросто выносится на следующий раз. А там такой материал, что объём очень легко регулируется.

Лекция

pointer'ы. Что это такое, зачем и что с этим делать.

pointer - это адрес. Адрес места в памяти. Если сказать более формально и точно, то pointer - это тип данных, которые хранят адреса, в частности переменная типа pointer может хранить адрес другой переменной (или даже самой себя).

Если переменная a хранит адрес переменной b, то говорят, что переменная a указывает на переменную b.

Ну и что? Мы и так знали, что переменные хранятся в памяти (в оперативной памяти, если точнее), и у каждой, естественно, есть своё место, т.е. есть и адрес. И не менее естественно предположить, что, поскольку оперативная память состоит из байтов, то все они пронумерованы, и адрес - это номер байта, т.е. некоторое целое число без знака. Но зачем это нам нужно? Как это применять? И так всё было замечательно, концепцию переменной мы знаем, чётко её понимаем, и помним, что фишка переменной именно в том, чтобы не возиться с оперативной памятью напрямую, а именно работать с ней, обращаясь по именам переменных, причём у переменных есть типы, которые помогают нам и что-то понять, и предохраняют нас от каких-то ошибок. Ну, посмотрим... Время покажет.

А пока попробуем просто помотреть, что у нас есть для работы с pointer'ами, они же указатели, они же ссылки.

Объявление и получение pointer'a

Тип `*myType` - это тип адреса величины типа `myType`, т.е. речь пойдёт сейчас о типизованных указателях.

Оператор `&` - это оператор взятия адреса переменной.

Оператор `*` позволяет обратиться к области памяти по её адресу. Адрес при этом должен

быть типизированным - указывать на область памяти, содержащей данные определённого типа, да у нас пока других pointer'ов и нет. Интеллигентно этот оператор называется *разыменование (dereferencing)*.

Их действие показывает пример [01a.go](#).

```
package main

import "fmt"

func main() {
    k:= 12345
    fmt.Printf("Type of k is %T, value of k is %d\n\n", k, k)
    p1:= &k
    fmt.Printf("Type of p1 is %T\n", p1)
    fmt.Println("address of k is", p1)
    fmt.Printf("the number %d is located at address %p\n\n", *p1, p1)
    var p2 *int = &k
    fmt.Printf("Type of p2 is %T\n", p2)
    fmt.Printf("address of k is %p\n", p2)
    fmt.Printf("numerical value of address of k is %x\nor %b or %d\n\n", p2, p2, p2)
}
```

Запускаем, компилируем, получаем на выходе текст:

```
Type of k is int, value of k is 12345

Type of p1 is *int
address of k is 0xc00000e0b0
the number 12345 is located at address 0xc00000e0b0

Type of p2 is *int
address of k is 0xc00000e0b0
numerical value of address of k is c00000e0b0
or 110000000000000000000000000000001110000010110000 or 824633778352
```

Ещё один пример показывает, что обращение к памяти по имени и обращение к памяти по адресу с помощью оператора разыменования эквивалентны.

Пример [01b.go](#).

```
package main

import "fmt"

func main() {
    b := 9999
    a := &b
    fmt.Println("address of b is", a)
    fmt.Println("value of b is", *a)
    *a++
    fmt.Println("new value of b is", b)
}
```

выводит на экран

```
address of b is 0xc00000e0b0
value of b is 9999
new value of b is 10000
```

Nil

В принципе, мы с этим уже сталкивались, но сейчас введём это понятие окончательно и формально. Nil - это нулевое значение pointer'a, это значение неинициализированной переменной этого типа. Пример [01c.go](#)

```
package main

import "fmt"

func main() {
    a := 25
    var b *int
    if b == nil {
        fmt.Println("b is", b)
        fmt.Printf("b is %p or %v\n", b, b)
        b = &a
    }
    fmt.Println("b after initialization is", b)
}
```

даёт такой результат

```
b is <nil>
b is 0x0 or <nil>
b after initialization is 0xc00000e0b0
```

Создание указателей в run-time - функция new

Иными словами, как выделить память (memory allocation) под некоторую величину, не объявляя переменную и не делая := , а во время исполнения программы.

Выделить память под величину типа Type, не объявляя переменной типа Type, можно. И тогда обращаться к этой области памяти можно по адресу, через pointer.

```
func new(Type) *Type
```

Функция new выделяет память для одной величины типа Type и возвращает адрес этой области. Аргумент этой функции - тип, а не величина. Выделенная область памяти инициализируется нулевым значением типа Type.

Действие функции new показывает пример [01d.go](#)

```
package main

import (
    "fmt"
)

func main() {
    someInt := new(int)
    fmt.Printf("someInt value is %d, type is %T, address is %v\n", *someInt, someInt,
    , someInt)
    *someInt = 85
    fmt.Println("New someInt value is", *someInt)
}
```

Вот её результат:

```
someInt value is 0, type is *int, address is 0xc00000e0b0
New someInt value is 85
```

Указатели (pointer'ы) и функции

В принципе давно уже пора поговорить о механизмах передачи параметров. И время пришло. Да, конечно, мельком и очень формально этот вопрос появлялся ещё в первом семестре, но пришла пора поговорить об этом обстоятельно и систематически. И сегодня мы начнём этот разговор. Сравним передачу по значению с передачей по адресу, увидим общее и различия. Но к теме передачи по адресу, кроме сегодняшнего разговора, мы ещё вернёмся немного попозже. Ещё один более или менее популярный вариант - передачу по имени - сегодня затрагивать не будем вообще, но в скором времени поговорим и о нём, а также об областях видимости (scope).

Но к делу. Что делать, если мы хотим написать функцию, которая получает переменную и как-то её изменяет, причём результат этой обработки должен отражаться в вызывающей функции? Упростим себе жизнь, введя несколько важных понятий. Конечно, это всего лишь термины, но они полезны для формулировок своих соображений, и, соответственно, важны для дальнейшего понимания - песни о том, что как сформулируешь в голове, так и напишешь в коде, вечны; исполнять их надо при любом случае.

Формальными параметрами называются переменные описанные в заголовке функции и используемые только внутри функции. Формальные параметры указывают, с какими параметрами следует обращаться к этой функции - их количество, последовательность, типы. Порядок следования, количество, имена и типы параметров могут быть любыми. Список формальных параметров обеспечивает связь функции с вызывающей её функцией. Через него в функцию передаются исходные данные и, возможно, возвращается результат.

Фактическими параметрами называются параметры, которые задаются при вызове функции. Фактические параметры должны совпадать с формальными по количеству, по порядку следования, по типу.

Эти два понятия чрезвычайно выразительны, помогают понять суть явления вызова функции как способа организации программ, технологии разработки программы. Так что на этом месте надо бы чуть-чуть приостановиться, вникнуть в смысл слов. Действительно, фактический параметр - это то, что при вызове реально существует, то, чем мы оперируем именно сейчас, в момент вызова, во время разработки функции, написания кода функции. Формальный параметр в это время для нас - это чистая форма, в которую мы погружаем наши текущие данные и передаем их на обработку, которую и исполняет вызываемая функция. В ней они обретают плоть и смысл, а для нас сейчас - это просто упаковка, которую мы ставим наши данные и пихаем их в какой-то механизм с целью что-то получить

на выходе из него. Что именно там происходит нас не особенно волнует, нас интересует только результат, который мы получим на выходе. Конечно, мы рассчитываем, что результат отвечает спецификации вызываемой функции, что она делает именно то, на что мы рассчитываем, судя по её описаниям/обещаниям.

Существуют разные способы принятия/возврата/обработки параметров (брать адрес параметра или его значение - паскалевские var-параметры или параметры-значения; разрешать или запрещать изменять его внутри вызываемой функции - var- и const-параметры в Pascal, const-параметры в C++ или Java и т.д.), но это для нас совершенно неважно и сейчас, и в ближайшем будущем, так что это так, для полноты картины.

А важно для нас сейчас, особенно с технологической точки зрения, то, что формальные параметры существуют только внутри функции, когда функция прекращает свою работу, когда происходит выход из функции, её формальные параметры исчезают, растворяются, их больше нет, они прекращают своё существование. И то же самое относится к **локальным переменным** - переменным, объявленным внутри функции - они живы, пока работает функция. **Внимание:** последняя фраза в Go неверна - функция может возвращать адрес локальной переменной, и тогда локальная переменная не умирает, компилятор Go это замечает и оставляет такие локальные переменные жить. Пример будет ниже. Подобное поведение - есть специфика Go, другие компиляторы так не делают (не скажу 100%-но за все, но по крайней мере в C/C++ и Pascal такое поведение точно носит неопределённый характер). Суть же локальных переменных (исключая только что описанный особый случай) в том, что они являются служебными переменными, необходимыми функции в её реализации.

В Go (как и, например, в C/C++) передача параметров происходит только по значению: фактический параметр передаёт своё значение соответствующему формальному параметру, а тот это значение, натурально, получает, инициализируется этим значением в начале выполнения функции при её вызове.

Указатели как параметры функции, или Передача параметра по адресу

И теперь ответим на вопрос, поставленный в начале этого раздела: что делать, если мы хотим написать функцию, которая получает переменную и как-то её изменяет, причём результат этой обработки должен отражаться в вызывающей функции? Только сначала переформулируем его во вновь обретённых терминах: что делать, если мы хотим чтобы изменения формального параметра отражались и соответственно изменяли фактический параметр? Ответ совсем простой, хоть и не совсем очевидный: передавать адрес

фактического параметра, который мы хотим изменить в результате действия функции. В самом деле, мы передаём функции всегда только значение, других вариантов в Go нет (так что о них можно даже и не упоминать, хотя можно и упомянуть, но так, к слову, для общего развития). А хотим мы чтобы некоторая наша переменная изменилась. Что есть переменная? Переменная есть область памяти. Так что мы хотим, чтобы изменилась некоторая область памяти. Как именно - это уже вопрос реализации, это сделает функция, которую мы вызываем, а вот какую именно область памяти менять - это мы должны той функции сообщить, передать. Т.е. при вызове надо указывать какую именно область памяти надо изменить, т.е. адрес этой области памяти - адрес переменной. Всё логично и естественно.

Характерный пример такого вызова показывает пример [02.go](#)

```
package main

import "fmt"

func inc(val *int, delta int) {
    *val += delta
}

func main() {
    a := 12345
    fmt.Println("a =", a)      // a = 12345
    inc(&a, 5)
    fmt.Println("a =", a)      // a = 12350
    b := &a
    inc(b, -3)
    fmt.Println("a =", a)      // a = 12347
}
```

Действие функции `inc` понятно без слов, добавлять тут нечего.

Ну, и теперь самый интересный момент сегодняшнего занятия - применим вновь полученные средства к анализу внутренней структуры слайсов, займёмся анатомией слайса.

Но сначала займёмся анатомией массивов - это необходимо, ведь слайсы “живут” на массивах, а для этого, для дотошного влезания внутрь памяти, нам потребуется ещё некоторые средства. И их нам предоставляет package `unsafe`. Он позволяет нам работать с

адресами более вольно (но и, разумеется, требует особой осторожности в применении). Собственно название этого пакета, этой библиотеки и происходит от термина `unsafe pointer` - адрес, не связанный всякими условностями с типизацией областей памяти.

package unsafe

На всякий случай поставлю здесь ссылку на package unsafe

Главное, ради чего создан package unsafe, - это тип Pointer

```
type Pointer *ArbitraryType
```

ArbitraryType - это только обозначение произвольного типа, необходим он только для описания. Реально такого типа нет, в том числе он не объявлен и в package unsafe. Так что **Pointer** - это указатель на выражение любого типа, иначе говоря, **Pointer** - это бестиповы́й указатель, адрес вообще. Т.е. Pointer - это то общее, что есть во всех указательных типах, т.е. адрес есть, а тип величины, на который он указывает, нет.

Имеется 4 специфических операции, доступных только с применением типа Pointer:

- значение указателя любого типа может быть преобразовано в величину типа Pointer.
- Pointer может быть преобразован в указатель любого типа.
- uintptr может быть преобразован в Pointer.
- Pointer может быть конверирован в uintptr.

Таким образом Pointer позволяет программе преодолеть ограничения системы типов и читать/писать любую память. Это делать можно, но делать это следует с великой осторожностью.

В мануале по package unsafe перечислены все варианты возможных применений таких преобразований. Конечно, можно их прочитать прямо там, но приведу некоторые и здесь.

1. Преобразование указателя T_1 в T_2 .

Предполагается, что T_2 занимает памяти не больше, чем T_1 .

Пример применения показывает реализация известной нам из II семестра функции

Float64bits

```
func Float64bits(f float64) uint64 {
    return (*uint64)(unsafe.Pointer(&f))
}
```

2. Преобразование Pointer в uintptr.

Тут никаких проблем нет - просто получаем числовое значение адреса.

```
var x ArbitraryType
fmt.Printf("%p %x %d\n", &x, &x, &x)
fmt.Println(uintptr(unsafe.Pointer(&x)))
```

А вот преобразование uintptr в Pointer, вообще говоря, допустимо сильно не всегда. uintptr - это число, а не ссылка (адрес). И дело в том, что uintptr содержит просто число, и даже если он содержит реальный адрес реального объекта, менеджер памяти не сможет отслеживать этот адрес - ни предотвращать использование этой памяти другим объектом, ни отслеживать возможность освободить эту область памяти. Тут стоит хотя бы немножко пояснить детям, что такое сборщик мусора (garbage collector), этот момент всплывает регулярно, мы с ним столкнёмся буквально в продолжении этого занятия, например, когда будем говорить об устройстве слайсов, так что особо сурово на это упирать вот прямо сейчас ни к чему - оно само ещё проявится совершенно естественным образом.

В мануале приводятся несколько допустимых преобразований uintptr в Pointer, я приведу одно.

3. Преобразование Pointer в uintptr и обратно с арифметикой.

Если *p* указывает на какой-то уже существующий объект, под который менеджер памяти уже распределил память, то можно обращаться к частям этого объекта, используя адресную арифметику (которой, повторю, напрямую в Go, в отличие от C, нет). Как правило это используется для доступа к полю структуры или элементу массива. Вот примеры из мануала:

```
// equivalent to f := unsafe.Pointer(&s.f)
f := unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + unsafe.Offsetof(s.f))

// equivalent to e := unsafe.Pointer(&x[i])
e := unsafe.Pointer(uintptr(unsafe.Pointer(&x[0])) + i*unsafe.Sizeof(x[0]))
```

Про функции unsafe.Sizeof и unsafe.Offsetof чуть ниже.

Ну, и для полноты картины процитирую ещё раз мануал, в котором приводятся примеры недопустимых операций.

- в отличие от C в Go нельзя переводить указатель за пределы распределённой области памяти, даже устанавливая указатель на конец этой области

```
// INVALID: end points outside allocated space.  
var s thing  
end = unsafe.Pointer(uintptr(unsafe.Pointer(&s)) + unsafe.Sizeof(s))  
  
// INVALID: end points outside allocated space.  
b := make([]byte, n)  
end = unsafe.Pointer(uintptr(unsafe.Pointer(&b[0])) + uintptr(n))
```

- Оба преобразования - и туда, и обратно - должны содержаться в одном выражении, в котором допускается арифметика

```
// INVALID: uintptr cannot be stored in variable  
// before conversion back to Pointer.  
u := uintptr(p)  
p = unsafe.Pointer(u + offset)
```

- Указатель должен указывать на уже существующий в памяти объект, поэтому операции с nil недопустимы.

```
// INVALID: conversion of nil pointer  
u := unsafe.Pointer(nil)  
p := unsafe.Pointer(uintptr(u) + offset)
```

Функций в package unsafe всего три, все они имеют то или иное отношение к последующему или предыдущему, так что пусть здесь будет их описание.

```
func Sizeof(x ArbitraryType) uintptr
```

Говоря по-простому, функция `Sizeof` возвращает размер переменной. Однако, в качестве аргумента функция `Sizeof` может использовать любое выражение, а какой же размер у выражения... Поэтому, и я приведу здесь эту формулировку из мануала как пример строгости, точности и корректности, функция `Sizeof` берёт просто выражение `x` произвольного типа и возвращает размер в байтах гипотетической переменной `v`, как если бы `v` была объявлена посредством `var v = x` (*takes an expression x of any type and returns the size in bytes of a hypothetical variable v as if v was declared via var v = x*). При этом это именно размер памяти, которую занимает эта переменная, а не размер памяти, на которую она потенциально ссылается. Так, например, слайс может быть многомегабайтным, а переменная типа слайс (и мы это скоро увидим) всё равно занимает 24 байта.

```
func Alignof(x ArbitraryType) uintptr
```

Выравнивание. В том же духе: объявляем (гипотетически) переменную v: var v = x . Адрес переменной v должен “выровнен” - должен делиться на некоторое число, которое эта функция и возвращает. Да, переменные в оперативной памяти идут не (всегда) вплотную одна к другой, между ними могут быть зазоры. Да, вроде бы на это уходит впустую какой-то объём памяти, но он невелик, а окупаются эти движения тем, что так эффективнее компилировать, а, самое главное, за счёт этого может существенно вырасти скорость выполнения действий.

```
func Offsetof(x ArbitraryType) uintptr
```

В данном случае ArbitraryType - это обязательно какой-то структурный тип (struct), а x - это обязательно какое-то поле этого struct'a. Функция возвращает, с какого байта, считая от адреса (начала хранения) структуры, располагается это поля. И да, поля в структурах тоже на обязательно идут вплотную друг к другу.

Действие последних двух функций [показывает пример 05.go](#). Текст этого примера приводится в данном доке ниже.

Массивы изнутри (Arrays internal).

Ну, с массивами всё довольно естественно и несложно. Взглянем на пример [03.go](#)

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    a := [...]uint {0,1,2,3,4,5,6,7,8,9}
    fmt.Println(a, "length =", len(a), "capacity =", cap(a))
    fmt.Printf("array a. address: %p size: %d\n", &a, unsafe.Sizeof(a))
    fmt.Printf("    a[0]. address: %p size: %d\n", &a[0], unsafe.Sizeof(a[0]))
    fmt.Printf("    a[2]. address: %p size: %d\n", &a[2], unsafe.Sizeof(a[2]))
    fmt.Println("    a[2] =", *(*int)(unsafe.Pointer((uintptr(unsafe.Pointer(&a)) + unsafe.Sizeof(a[0]) + unsafe.Sizeof(a[1])))))
    fmt.Printf("    a[9]. address: %p size: %d\n", &a[9], unsafe.Sizeof(a[9]))
    b := a
    fmt.Printf("array b. address: %p size: %d\n", &b, unsafe.Sizeof(b))
```

```
b[0] = 100
fmt.Println("    a:", a)
fmt.Println("    b:", b)
}
```

Программа в результате выдаёт:

```
[0 1 2 3 4 5 6 7 8 9] length = 10 capacity = 10
array a. address: 0xc0000122d0 size: 80
    a[0]. address: 0xc0000122d0 size: 8
    a[2]. address: 0xc0000122e0 size: 8
    a[2] = 2
    a[9]. address: 0xc000012318 size: 8
array b. address: 0xc000086050 size: 80
    a: [0 1 2 3 4 5 6 7 8 9]
    b: [100 1 2 3 4 5 6 7 8 9]
```

Всё видно совершенно отчётливо: массив а послушно получил память на 10 элементов, переменная-массив а - это адрес начала массива = адрес нулевого элемента массива, элементы массива идут подряд один за другим, имея, разумеется, одинаковый размер - ведь все они одного типа. Ну, и мы уже и раньше знали, что массив - штука стабильная, место получает сразу при запуске программы и потом его уже не меняет, и размер его по ходу выполнения программы не меняется, обращаться мимо имеющихся элементов массива невозможно. Новый массив, хотя бы и при копировании, создаётся на новом месте и живёт, соответственно, своей собственной жизнью. В общем, всё просто и понятно.

Структуры изнутри (Structs internal)

Важное замечание. Я бы этот пункт не стал рассказывать явно, может быть вкратце обговорил, но совсем без деталей. **А детальную проработку оставил детям на практику**, пусть проведут небольшое исследование, будет интересно.

А со структурами всё точно также. Отличие только в том, что к элементам массивов обращаемся по индексам, а к полям структур обращаемся по именам полей (квалификаторам). Да, можно показать детям пример [05.go](#)

```
package main

import (
```

```

"fmt"
"unsafe"
}

type tag struct {
    order     byte
    article   *string
    size      int
    active    bool
}

func main() {
    s := "sample struct"
    v := tag{11, &s, 123456, true}
    fmt.Println("v:", v)
    fmt.Printf("v. address: %p size: %d\n", &v, unsafe.Sizeof(v))
    fmt.Println(" order: size =", unsafe.Sizeof(v.order), "offset= ", unsafe.Offsetof(v.order))
    fmt.Println("article: size =", unsafe.Sizeof(v.article), "offset= ", unsafe.Offsetof(v.article))
    fmt.Println(" size: size =", unsafe.Sizeof(v.size), "offset= ", unsafe.Offsetof(v.size))
    fmt.Println(" active: size =", unsafe.Sizeof(v.active), "offset= ", unsafe.Offsetof(v.active))
    w := v
    w.order++
    fmt.Println("v:", v, "address =", unsafe.Pointer(&v))
    fmt.Println("w:", w, "address =", unsafe.Pointer(&w))
}

```

Вот результаты:

```

v: {11 0xc00002c1f0 123456 true}
v. address: 0xc0000044c0 size: 32
    order: size = 1 offset= 0
    article: size = 8 offset= 8
    size: size = 8 offset= 16
    active: size = 1 offset= 24
v: {11 0xc00002c1f0 123456 true} address = 0xc0000044c0
w: {12 0xc00002c1f0 123456 true} address = 0xc000004520

```

Всё то же самое: структура получает необходимую память, поля идут одно за другим, хотя здесь есть очень важная разница - поля выравниваются, т.е. они идут одно за другим, но не подряд, между ними возможны "дыры". Но, как и массив, структура - стабильный объект, место под него выделяется при запуске программы и потом его уже он не двигается, размер его по ходу выполнения программы не меняется. Новая структура, хотя бы и при копировании, создаётся на новом месте и живёт, соответственно, своей собственной жизнью.

Но думается, что вполне себе этот вопрос можно отдать детям на практику для исследования. Пусть сами сформулируют выводы, пусть постараются сформулировать какие-то гипотезы и попытаются их доказать (или опровергнуть). Пусть попробуют в качестве полей использовать не только скалярные величины разной длины (1, 2, 4 или 8), но и более сложные - массивы или другие структуры. Это даст возможность, в частности, поэкспериментировать с большими полями - полями, размеры которых больше 8. Только пока не надо использовать поля типа слайсов или строк - с ними мы пока не разобрались. Хотя сделать это в ближней перспективе - очень неплохо бы.

А вот со слайсами всё сложнее. Прежде всего, они могут изменять свой размер в процессе выполнения программы, т.е. память под слайс может перераспределяться. Что же из себя представляют слайсы, а они - чрезвычайно важный, полезный и часто применимый тип данных, давайте разберёмся.

Анатомия слайсов (Slices internal).

Давайте создадим слайс и посмотрим, что у него внутри: пример [04a.go](#):

```
package main

import (
    "fmt"
    "unsafe"
)

func main() {
    a := [...]uint{0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    b := a[2:7]
    fmt.Println(b, len(b), cap(b))           // [2 3 4 5 6] 5 8
    fmt.Printf("%d %x %x\n", b[0], &b[0], &a[2]) // 2 c000122100 c000122100
    fmt.Printf("%d %x %x\n", b[1], &b[1], &a[3]) // 3 c000122108 c000122108
```

```

    fmt.Printf("%d %x %x\n", b[2], &b[2], &a[4]) // 4 c000122110 c000122110
    a[5] = 97
    fmt.Println(b)                                // [2 3 4 97 6]
    fmt.Println(&b)                                // &[2 3 4 97 6]
    fmt.Printf("%p\n", &b)                          // 0xc0000044c0
    fmt.Println(unsafe.Pointer(&b))                // 0xc0000044c0
}

```

Ну, что, начало нормальное, слайс `b` “базируется” на массиве `a`, начиная с `a[2]`, в слайсе 5 элементов, а его ёмкость равна 8 - действительно от `a[2]` до конца массива (до `a[9]`) как раз 8 элементов. Элементы слайса `b` расположены как раз там, где лежат соответствующие им элементы массива `a`: `b[0]` и `a[2]`, `b[1]` и `a[3]`, `b[2]` и `a[4]`, понятно, что и остальные элементы слайса расположены на “правильных” местах. Но сам слайс - переменная `b` - где-то же должен быть. Подчеркну, это очень важный момент: элементы слайса `b` расположены там же, где и соответствующие элементы массива `a`, собственно они и есть элементы массива `a`, изменение любого элемента слайса сказывается на соответствующем элементе массива и наоборот, всё это мы давно знаем. Но где-то же хранится специфическая информация о слайсе `b`, ведь откуда-то программа знает, что он начинается со второго элемента массива и заканчивается шестым (напомню, именно, шестым, а не седьмым) элементом массива. Посмотрим. Первая попытка (`fmt.Println(&b)`) неудачна, выводится какая-то фигня, но мы легко её обойдём и даже двумя способами. И мы видим, что да, собственно слайс `b`, специфическая информация о нём располагается совсем в другом месте.

Что же там находится, на этом самом месте? Конечно, можно почитать описания, но давайте попробуем исследуем всё “честно”, своими руками. Пример [04b.go](#) - это и есть то самое исследование. Давайте посмотрим на него.

```

package main

import (
    "fmt"
    "unsafe"
)

type (
    sliceHeader struct {
        start  uintptr
        len    int
        cap    int
    }
)

```

```

}

func main() {
    a := [...]uint {0,1,2,3,4,5,6,7,8,9}
    b := a[2:7]
    fmt.Println(unsafe.Pointer(&b)) // 0xc0000044c0
    fmt.Println(unsafe.Sizeof(b)) // 24
    barr := (*[3]int)(unsafe.Pointer(&b))
    fmt.Println(*barr) // [824633795296 5 8]
    // #0
    fmt.Printf("%x %p %p\n", (*barr)[0], &b[0], &a[2]) // c0000122e0 0xc0000122e0 0xc0000122e0
    // #1
    fmt.Printf("%d %d\n", (*barr)[1], len(b)) // 5 5
    // #2
    fmt.Printf("%d %d\n", (*barr)[2], cap(b)) // 8 8
    // all together - struct
    fmt.Println(*(*sliceHeader)(unsafe.Pointer(&b))) // {824633795296 5 8}
}

```

Для начала выводим, где расположен именно слайс b, переменная b, это мы уже делали, но пусть будет ещё раз, чисто для связи с предыдущим примером. А дальше посмотрим на размер переменной b. И размер этот - 24 байта. Интересно. Не 40 байтов, которые занимают 5 int'ов, а 24. Ну, это мы уже тоже знаем, переменная b располагается на там, где переменная a. Но именно 24, давайте посмотрим, что же лежит в этих 24 байтах. 24 - 3 по 8, 3 раза по 8 байтов, три int'a. Так... Что же в них лежит. Расположим массив из трёх int'ов на месте слайса b и посмотрим, что у него внутри. Получаем [824633795296 5 8]. Первое число - это что-то очень большое, скорее всего, это какой-то адрес, сейчас разберёмся, что это за адрес. А вот 5 и 8 - это явно len и cap - длина и ёмкость слайса.

Но вернёмся к адресу. Вполне естественно предположить, что там хранится адрес начала слайса. Проверим это. Да, это оно! Всё срослось. И теперь уже с чистой совестью заведём структурный тип slice с соответствующими названиями полей, а не с безликими индексами, как у массива barr.

Массивы и слайсы как параметры функций.

Здесь мы просто рассмотрим примеры трёх ситуаций.

Передача массива по значению

Пример [06a.go](#)

```
package main

import "fmt"

func process(arr [3]int) {
    fmt.Println(arr)      // [2020 2021 2022]
    for i, _ := range arr {
        arr[i] *= 2
    }
    fmt.Println(arr)      // [4040 4042 4044]
}

func main() {
    a := [3]int{2020, 2021, 2022}
    fmt.Println(a)        // [2020 2021 2022]
    process(a)
    fmt.Println(a)        // [2020 2021 2022]
}
```

Передача массива по адресу

Пример [06b.go](#)

```
package main

import "fmt"

func process(arr *[3]int) {
    fmt.Println(*arr)    // [2020 2021 2022]
    for i, _ := range *arr {
        (*arr)[i] *= 2
    }
    fmt.Println(*arr)    // [4040 4042 4044]
}
```

```
func main() {
    a := [3]int{2020, 2021, 2022}
    fmt.Println(a)          // [2020 2021 2022]
    process(&a)
    fmt.Println(a)          // [4040 4042 4044]
}
```

Передача слайса

Пример [06c.go](#)

```
package main

import "fmt"

func process(arr []int) {
    fmt.Println(arr)      // [2020 2021 2022]
    for i, _ := range arr {
        arr[i] *= 2
    }
    fmt.Println(arr)      // [4040 4042 4044]
}

func main() {
    a := [3]int{2020, 2021, 2022}
    fmt.Println(a)          // [2020 2021 2022]
    process(a[:])
    fmt.Println(a)          // [4040 4042 4044]
}
```

Хорошо видно, что передача слайса практически эквивалентна передаче массива по адресу. Причины этого понятны: ведь слайс - это, фактически, и есть адрес массива. И если нам надо просто изменить элементы массива, то можем передавать именно соответствующий слайс, а не адрес массива. Более того, принято считать, что передавать слайс для преобразования массива более присуще Go. Да так и есть, действительно, пример 06c.go выглядит чище, более естественно. Другое дело, что если мы начнём так изменять слайс внутри функции, что он будет изменять свой размер, точнее, увеличивать свою ёмкость (cap), то массив может и не измениться (и не изменится), но об этом в следующий раз...

И последний момент, который я бы тоже отправил **на практику** - вопрос о внутреннем устройстве строк (string).

Строки изнутри (Strings internal)

Повторю **важное замечание**, сделанное в связи с устройством структур: Я бы этот пункт не стал рассказывать явно, может быть вкратце обговорил, но совсем без деталей. А детальную проработку оставил детям на практику, пусть проведут небольшое исследование, будет интересно.

Пример [07.go](#)

```
package main

import (
    "fmt"
    "unsafe"
)

type (
    stringHeader struct {
        start    uintptr
        length   uint
    }
)

func main() {
    str := "ABCD"
    str += ".abcd"
    fmt.Println(str)                                // ABCD.abcd
    fmt.Println(&str)                               // 0xc00002c1f0
    fmt.Println(unsafe.Pointer(&str))               // 0xc00002c1f0
    fmt.Println(unsafe.Sizeof(str))                  // 16
    strarr := (*[2]uintptr)(unsafe.Pointer(&str))   // &[824633778352 9]
    fmt.Println(*strarr)                            // c00000e0b0 9
    fmt.Printf("%x %d\n", (*strarr)[0], (*strarr)[1])
    *(*byte)(unsafe.Pointer((*strarr)[0]+4)) = '+'
    fmt.Println(str)                                // ABCD+abcd
    // all together - struct
    fmt.Println(*(*stringHeader)(unsafe.Pointer(&str))) // {824633778352 9}
```

```
str2:= str
fmt.Println(*(*stringHeader)(unsafe.Pointer(&str2)))// {824633778352 9}
}
```

Кратко сформулирую некоторые основные моменты.

1. Основной момент на сегодня: переменная типа `string` - это пара (адрес строки, длина строки в байтах). Да, хорошо бы тут поиграться с нелатинскими (русскими/латышскими/другими) буквами, чей utf8-код занимает 2-3 байта, чтобы убедиться в том, что хранится длина строки именно в байтах, а не в символах, что, кстати, совершенно естественно - не будем же мы для определения длины отрезка памяти пробегать по нему, считая байты по длинам utf8-кодов.
2. Очень важный момент здесь то, что строка `str` преобразуется в процессе выполнения программы. Это заставляет строку переместиться куда-то в кучу (heap). Тут фишка в том, что строки неизменяемы (`immutable`), и различные строковые переменные могут указывать на одно и то же место в памяти. Это, кстати, иллюстрируют последние две строки программы-примера. А если убрать оператор `str += ".abcd"`, то переменная `str` будет ссылаться на строковую константу "ABCD", а та расположена где-то в коде программы, и попытка обратиться туда через `unsafe.Pointer` завершится жёстким отказом. Как это объяснить детям прямо сейчас? Ну, слегка объяснить, не торопясь. Пусть немного поупираются, хотя особо морить их не надо, давать советы типа "попробуй сделать вот так, авось получится" можно и нужно, но не забегая впереди паровоза. А подробно мы поговорим об этих явлениях в следующий раз.

Практика.

- Два задания: разобраться с внутренним устройством структур и с внутренним устройством строк - уже описаны выше, даже примеры приведены. Давать их детям или не давать (примеры, а не задания) - по обстоятельствам. Скорее давать, но не сразу. Может быть по концовке занятия, для домашних штудий и дальнейших исследований. Но тут уж смотрим, как оно пойдёт.
- Очень уместно будет дать детям какие-нибудь задания на работу со структурами, слайсами, мапами, с чтением данных из файлов, возможно, что из текстовых, с последующим парсингом данных. Первое занятие в семестре - надо давать всякое разное для повторения. Много, например, таких заданий во втором семестре в планах к занятиям 8, 9, 10, материалы к заданиям - в каталоге `tasks` в материалах ко второму семестру.