# Golang - wstęp

Adam Pietrzak

19 listopada 2020

# Spis treści

1. Wskaźniki

2. Tablice

3. Struktury

4. Map

# Wskaźniki

# Wskaźniki

GoLang jako język posiada rozróżnienie pomiędzy wskaźnikiem a zwykłym obiektem.

Jak dobrze wiemy, wskaźniki wskazują na element w pamięci. Dzięki temu, możemy optymalizować pamięć i szybkość aplikacji.

Jeżeli chodzi o stos i stertę, GoLang nie mówi jasno, kiedy co trafi gdzie. Kompilator potrafi obiekt zapisany jako wskaźnik trzymać na stosie, aby optymalizować kod.

# Wskaźniki II

```
func a (value *int) {
    // Manage value
}

x := 5
a(&x)
```

Jak widzimy, definicja wskaźnika jest prosta.

Przy wskaźnikach będziemy mówić o dwóch znakach:

- "*": znak gwiazdki będzie rozumiany dwojako, w zależności od użycia:
  - w sytuacji definicji typu, oznaczać będzie, że dana zmienna jest wskaźnikiem: "*int", "*[]int", "*[]*int"
  - odwołania się do wartości, którą reprezentuje dany wskaźnik: "*a = 5"
- "&" który wyciąga adres danej zmiennej

# Wskaźniki II

```
func testFunc(a *int, b *int) {
    *a = *a * *b
    *b = 5
}

a := 12
b := 2

testFunc(&a, &b)

fmt.Println(a, b) // What it prints?

c := &b
c = 4 * a   // ?
```

# Wskaźniki III

Wcześniej często widać zapis:

```
a := 5;

func(&a)
```

Taki zapis dla osób z innych języków będzie jeszcze ok, ale czasami możemy trafić na zapis:

```
func MyFunc() *int {
    a := 5
    return &a
}
```

To już może tworzyć pewne wątpliwości natury wskaźników. Taki zapis w języku “C”/“C++” spowoduje problemy z wartością zmiennej “a”.

Jak dobrze wiemy, wartość ta wyląduje na stosie, a zostanie usunięta wraz z pamięcią funkcji “MyFunc”. W takich sytuacjach kompiler Go jest mądrzejszy i wie, że zmienna “a” nie może zostać usunięta.

# Wskaźniki III - new

Ok, wiemy jak działają wskaźniki, ale jak alokować pamięć ciekawszych typów? Do tego służy słówko kluczowe “new”.

New pozwala nam zaalokować dany typ w pamięci.

```
func a (tmp1 *int){
    //Code
}

i := new(int)    // *int
a(i)
```

Dzięki temu możemy alokować dane bez używania “&”. New przyda nam się także do struktur.

# Tablice

# Tablice

Tablice w GoLang działają podobnie do reszty języków:

- mają stałą wielkość
- wielkość musi być znana podczas kompilacji

```
var a [3]int
b := [8]int{}
c := [2]int{1, 2}
d := [...]int{2, 4, 6, 1}
```

Odwoływanie się do wartości jest takie samo jak w innych językach:

```
a[2]
d[0]
```

Dodatkowo, wyjście poza tablicę rzuca wyjątek.

**Typ string**
String nie jest tablicą jak w innych rozwiązaniach, pomimo, że pod spodem przypomina tablicę. Jest on immutable.

# Tablice II

Wyciągnięcie wielkości tablicy możemy uzyskać za pomocą funkcji: “len”:

```
a := [9]int{}

len(a) == 9 // true
```

Trzeba pamiętać, że tablica nie jest wskaźnikiem na pamięć! Wykonanie kodu:

```
a := [9]int{}
b := a

b[0] = 5
```

spowoduje, że “a[0] != b[0]”.

Dodatkowo, jeżeli typy w środku są porównywalne, tablice można porównywać: “a == b”

**Wartości domyślne**
Domyślnie, tablice są uzupełnione wartościami domyślnymi.

# Slice

Czasami chcemy uzyskać tablice "dynamiczne". W takich sytuacjach Go posiada “slice”. W Go zazwyczaj będziemy mieć doczynienia z “slice”. Są one szybsze i pozwalają na większą swobodę w użyciu.

Podstawowy zapis to:

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

Jak widać, nie różni mocno od tablicy. Dodatkowo istnieje funkcja “make”:

```
a := make([]int, 5, 10) // makes []int{0, 0, 0, 0, 0}
```

która tworzy nowy “slice”. Jako parametry mamy kolejno: “type”, “size”, “capacity”.

**make - ostatni parametr**
Ostatni parametr“make” nie jest wymagany. Wtedy “cap” == “size”.

# Dowolna ilość argumentów

Wydawało by się, że "make" jest wyjątkiem w języku. Pozwala on na pominięcie jednego z argumentów. Jak sprawdzimy za pomocą IDE, to definicja make wygląda tak:

```
func make(t Type, size ...int) T {
    // Code
}
```

Tak jest w teorii, jednak Go pilnuje, aby ilość argumentów była poprawna. Jest to wyjątek, nie do uzyskania przez programistę.

**The language designers have basically said:**
We need function overloading to design the language we want, so make, range and so on are essentially overloaded, but if you want function overloading to design the API you want, well, that's tough.

# Slice II

Slice daje nam kolejne funkcje, które pomogą nam nimi zarządzać:

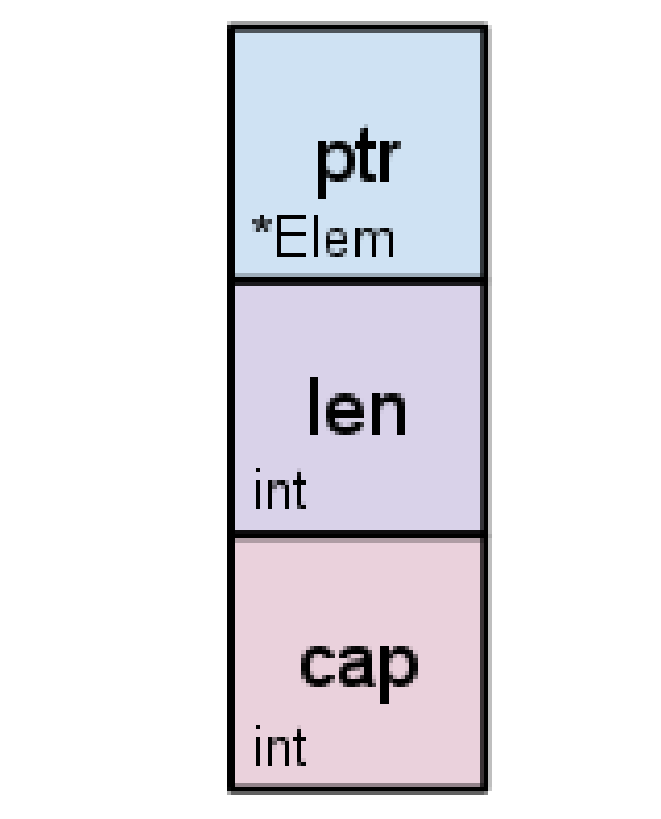
- “cap(slice)” - pojemność slice
- “append(slice, ...value) newSlice” - dodawanie na koniec slice

Slice są zbudowane w oparciu o tablice. Jeżeli przekroczymy ich pojemność, wymagane będzie przeniesienie tablicy pod spodem.

Slice dają nam większe możliwości i poprawę performance. Lecz mogą także powodować nowe problemy.

# Slice III

Slice to tak na prawdę obiekt z trzema polami:




- ptr wskazuje początek tablicy
- len jej długość
- cap jej pojemność

# Slice III

Dzięki takiemu zabiegowi, slice dają nam ciekawe możliwości indeksowania:

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

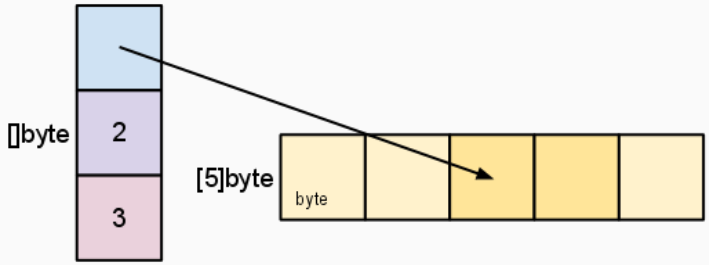
Powyższe indeksowanie różni się od tablic. Tutaj nie mamy kopii tablicy, pracujemy wciąż na oryginale. Z tego powodu:

```
a := []int{2, 3, 4}
b := a[1:]
b[0] = 4
fmt.Println(a)
```

Wypisze nam: “2, 4, 4”.

# Slice IV

Dzięki temu, że slice jest obiektem wskazującym na tablicę, Go nie musi kopiować zawartości, tylko sam ten obiekt. Gdyby spojrzeć jak to wygląda pod spodem, to dostaniemy:




Widać, że wyciągnięcie “kawałka” slice powoduje, że zmienia sie tylko obiekt definiujący slice.

# Slice V

Można by powiedzieć, że z wielkimi możliwościami przychodzi wielka odpowiedzialność. Tutaj w zasadzie pasuje powiedzieć, że przychodzą wielkie memory leaki.

GoLang pracuje w oparciu o garbage collector. Można tutaj zrobić bardzo niemiłą rzecz, czyli:

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

return b
```

Kod wygląda ok, lecz generuje pewnego rodzaju memory leaka.

Tablica pod spodem, jest trzymana w całości tak długo, jak istnieje przynajmniej jeden slice. Nie ważne, że ma on 1 element, a tablica 1000, GC nie wyrzuci zbędnych 999.

# Slice VI

Aby poradzić sobie z tym problemem, powstała funkcja “copy”:

```
a := []int{2, 3, 4}
b := make([]int, len(a[1:]))
copy(b, a[1:])
return b
```

**Zamiana array na slice**
Najłatwiejszy sposób zamiany tablicy na slice jest: “myArray[:]”. Utworzy on nowy slice, który w dodatku nie kopiuje tablicy pod spodem, lecz jej używa.

**Uwaga!**
Pomiędzy funkcjami najlepiej przekazywać slice, zamiast tablicy. Tablica jest przekazywana przez wartość, czyli jej kopia. Przy dużych tablicach może to chwile potrwać. Dlatego lepiej przekazać slice.

**Uwaga 2!**
Jeżeli mamy funkcję “func a(b int[])”, to nie przyjmie ona argumentu: “[3]int1, 2, 3”

# Struktury

# Struktury

Struktura to definicja własnego wyglądu obiektu. Zawiera ona zbiór pól, jakie struktura ma zawierać.

```
type MyStruct struct {
    FieldA int
    FieldB string
}
```

Warto pamiętać, że nazwy pól struktury także poddają się zasadzie prywatności ze względu na wielkość pierwszej litery.

# Struktury II

Podstawowe poperacje na strukturze:

```
type MyStruct struct {
    FieldA int
    FieldB string
}

myValue := MyStruct {
    FieldA: "A",
    FieldB: "B",
}

myValue2 := &MyStruct {
    "A",
    "B",
}

myValue == *myValue2 // true
myValue.A = "A1"
myValue2.B = "B"
```

# Struktury III

Należy pamiętać, że podawanie pól w strukturze jest opcjonalne przy tworzeniu. Najlepiej używać funkcji tworzących struktury.

Przekazywanie struktury jako argument funkcji powoduje zrobienie jej kopii. Przy dużych strukturach zaleca się wskaźniki.

# Struktury IV

Struktury nie mogą po sobie dziedziczyć. Jako symulację tworzy się pole anonimowe:

```
type MyStruct struct {
    FieldA int
    FieldB string
}

type ExtendedStruct struct {
    MyStruct
    FieldC string
}

a := ExtendedStruct{
    MyStruct{
        FieldA: 87,
    },
    "C",
}

a.FieldB == a.FieldC
```

# Struktury V

Istnieje możliwość zdefiniowania struktury, która będzie posiadać własne metody. Metody w GoLangu przyjmują "receivera", który może być wskaźnikiem.

```
type MyStruct struct {
    FieldA int
    FieldB string
}

func (m *MyStruct) test () {
}

func (m MyStruct) test2 () {
}
```

Receivery, które nie są wskaźnikami, nie mogą modyfikować struktury.

Zazwyczaj Go będzie dbać o poprawne tworzenie wskaźników w takiej strukturze.

Polecam ten URL jeżeli nie wiesz który typ wybrać

# Mapa

# Map

Mapa pozwala na zapisywanie wartości typu “klucz: wartość”. Pod spodem jest zaimplementowana hashmap.

W mapie jako klucz może być dowolny typ, który da się porównać.

Mapę zapisujemy jako:

```
var m map[string]int
```

Mapa jest wskaźnikiem, dlatego też powyższy zapis spowoduje utworzenie zmiennej z wartością nil. Aby przypisać jej wartość, można użyć “make”

```
m = make(map[string]int)
```

# Map II

Mapę można szybciej inicjalizować:

```
m := map[string]int{}

k := map[string]int{
    "a": 20
}
```

Operacje na mapie są proste:

```
m["b"] = 50
a := m["a"] // if not exists, returns zero value

c, ok := m["c"]
```

# Map II

Z mapy można usuwać elementy za pomocą funkcji “delete”

```
k := map[string]int{
    "a": 20
}

delete(k, "a")
```

Oraz po mapie można iterować:

```
for key, value := range k {

}
```