



uma pegadinha em #golang

---

# 3 MODELOS DE VARIÁVEIS

---

*Valores, ponteiros e referências na linguagem Go.*



THE  
DEVELOPER'S  
CONFERENCE

# MODELOS DE VARIÁVEIS EM LINGUAGENS

---

*O que a gente vê por aí*

# MODELOS DE VARIÁVEIS EM ALGUMAS LINGUAGENS

---

	valores	ponteiros	referências
C	✓	✓	
C++	✓	✓	✓
Java	✓		✓
JavaScript			✓
Python			✓
Go	✓	✓	✓

# VARIÁVEIS EM GO

---

*Comportamentos diferentes*

# EM GO, VARIÁVEIS SÃO "CAIXAS"

---

```
i := 3  
i2 := i  
i2++  
fmt.Printf("i\t%#v\ni2\t%#v\n", i, i2)
```

i	3
i2	4

# EM GO, VARIÁVEIS SÃO "CAIXAS"

---

```
i := 3
i2 := i
i2++
fmt.Printf("i\t%#v\ni2\t%#v\n", i, i2)
```

```
i    3
i2   4
```

```
a := [...]int{1, 2, 3}
a2 := a
a2[0]++
fmt.Printf("a\t%#v\na2\t%#v\n", a, a2)
```

```
a    [3]int{1, 2, 3}
a2   [3]int{2, 2, 3}
```

# EM GO, VARIÁVEIS SÃO "CAIXAS"

---

```
i := 3
i2 := i
i2++
fmt.Printf("i\t%#v\ni2\t%#v\n", i, i2)
```

```
i    3
i2   4
```

```
a := [...]int{1, 2, 3}
a2 := a
a2[0]++
fmt.Printf("a\t%#v\na2\t%#v\n", a, a2)
```

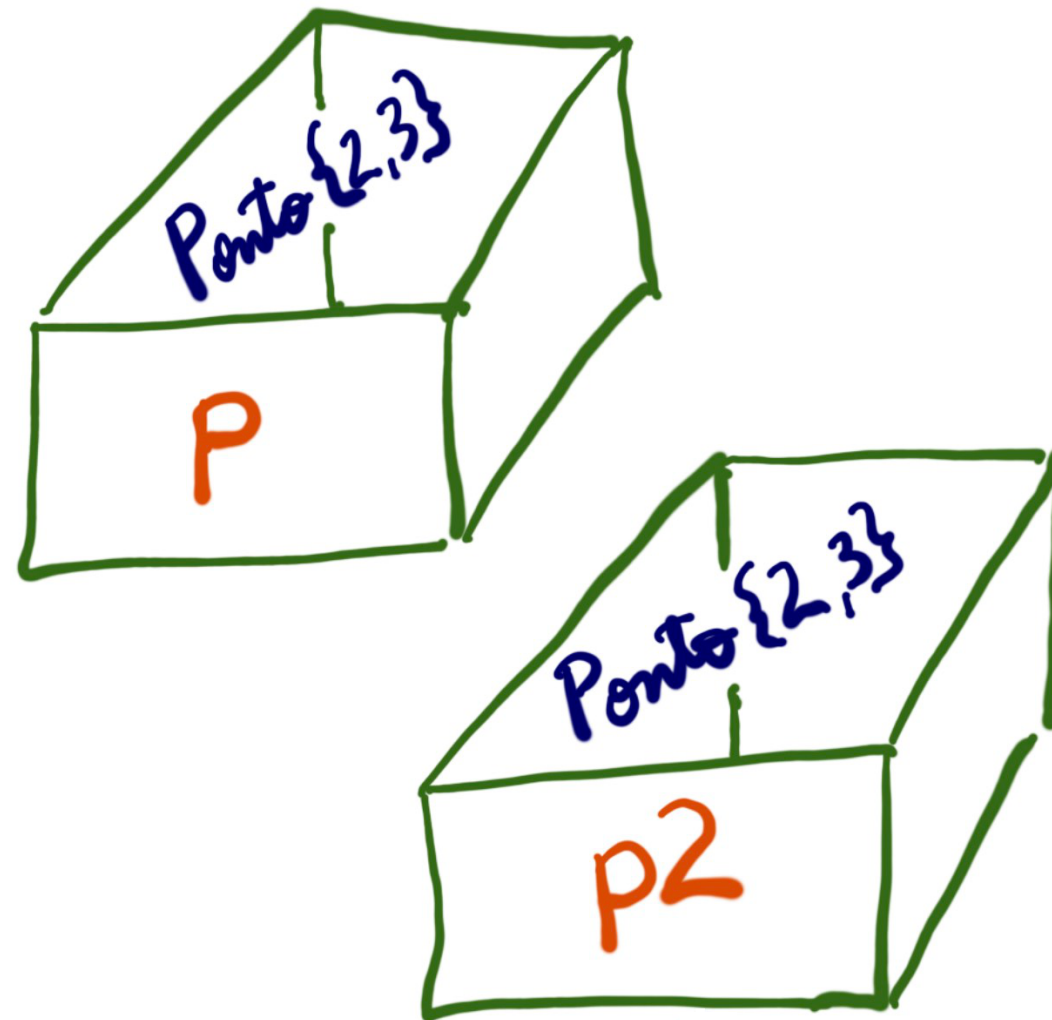
```
a    [3]int{1, 2, 3}
a2   [3]int{2, 2, 3}
```

```
p := Ponto{2, 3}
p2 := p
p2.y++
fmt.Printf("p\t%#v\np2\t%#v\n", p, p2)
```

```
p    main.Ponto{x:2, y:3}
p2   main.Ponto{x:2, y:4}
```

# OS CONTEÚDOS DAS CAIXAS P E P2 SÃO INDEPENDENTES

---



```
p := Ponto{2, 3}
p2 := p
p2.y++
fmt.Printf("p\t%#v\np2\t%#v\n", p, p2)
```

p	main.Ponto{x:2, y:3}
p2	main.Ponto{x:2, y:4}



# O OPERADOR & (ENDEREÇO) DEVOLVE UM PONTEIRO

---

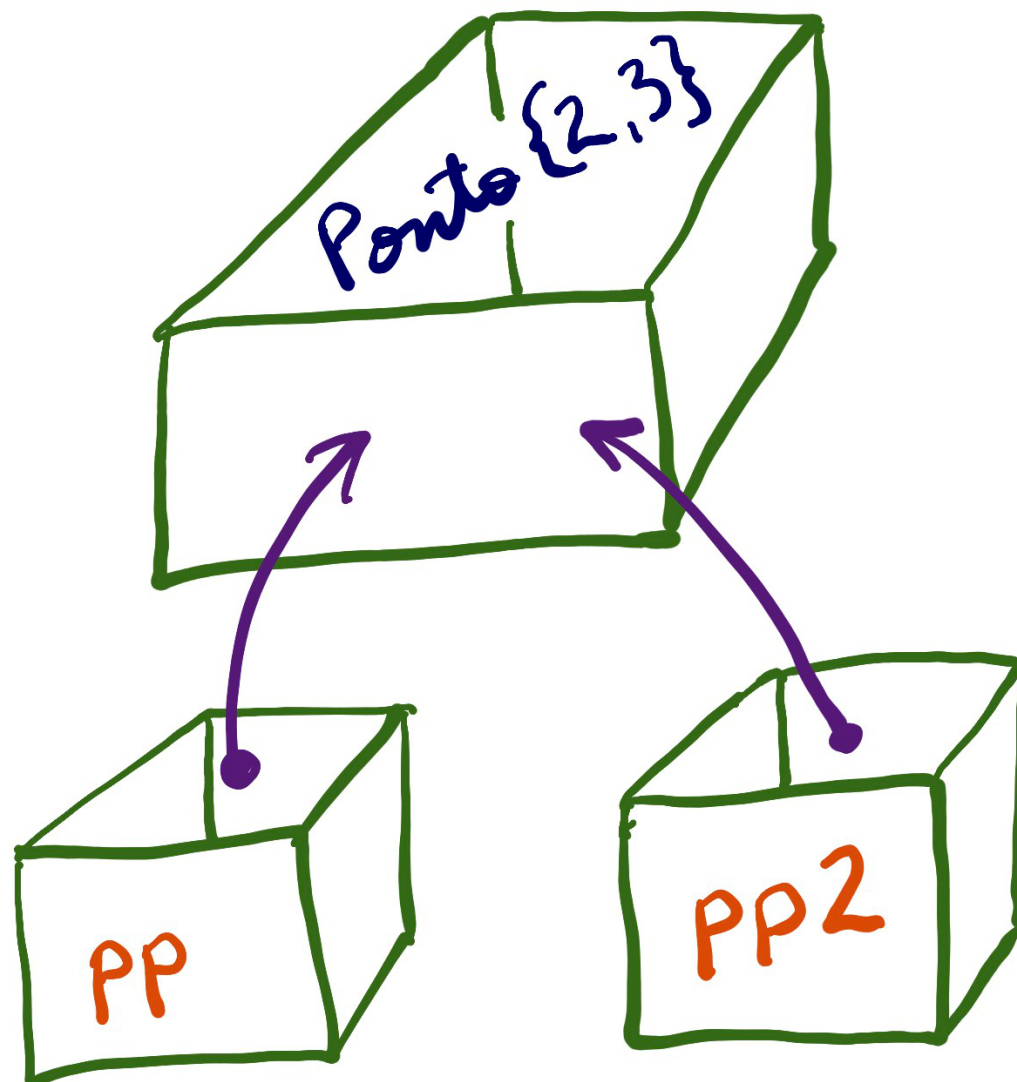
```
pp := &Ponto{2, 3}  
pp2 := pp  
pp2.y++  
fmt.Printf("pp\t%#v\npp2\t%#v\n\n", pp, pp2)
```

```
pp  &main.Ponto{x:2, y:4}  
pp2 &main.Ponto{x:2, y:4}
```

# CAIXAS PP E PP2 TÊM PONTEIROS PARA A MESMA CAIXA

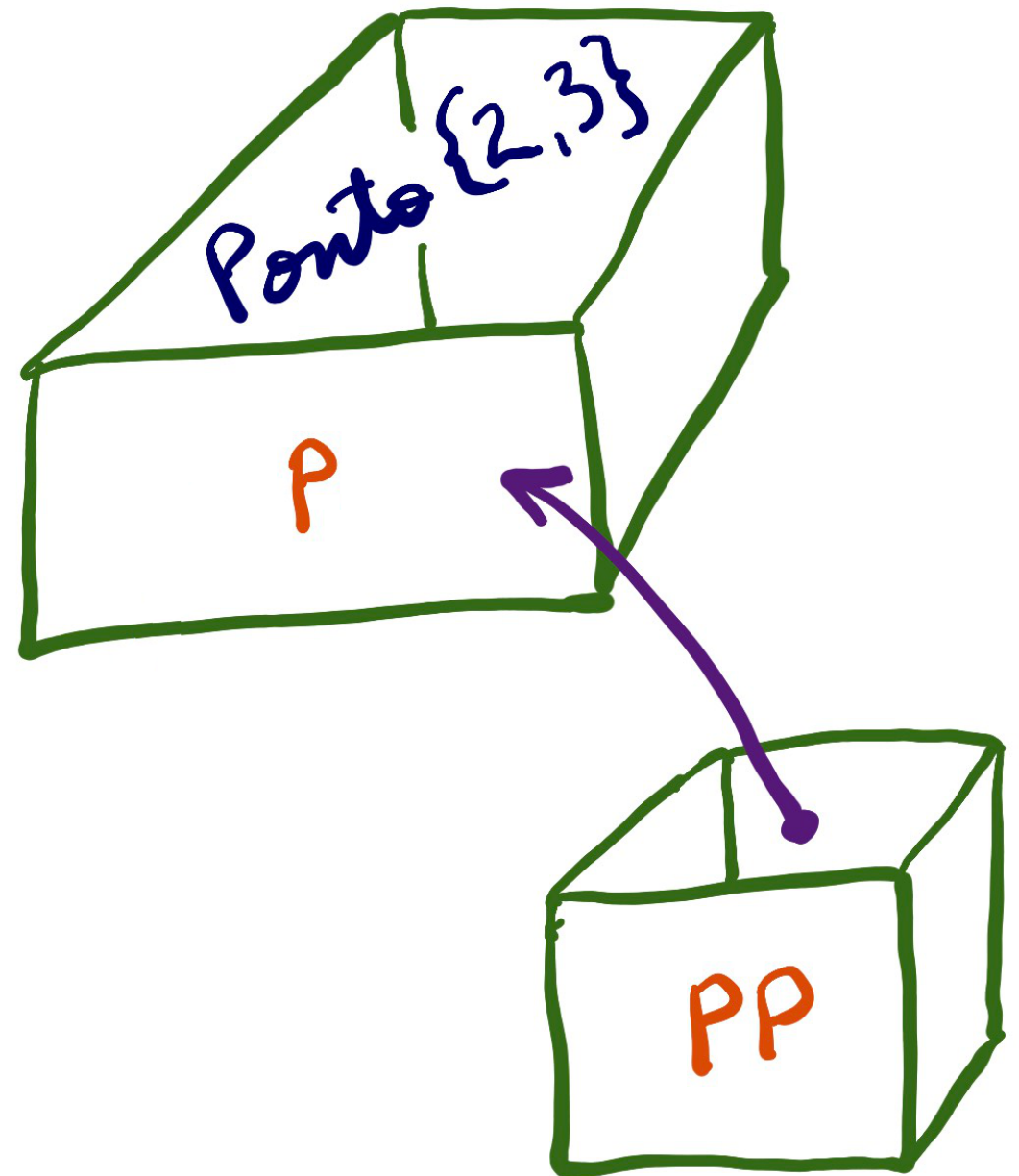
```
pp := &Ponto{2, 3}  
pp2 := pp  
pp2.y++  
fmt.Printf("pp\t%#v\npp2\t%#v\n\n", pp, pp2)
```

```
pp &main.Ponto{x:2, y:4}  
pp2 &main.Ponto{x:2, y:4}
```



# SINTAXE DE PONTEIROS: &X, PX, \*PX

```
type Ponto struct {  
    x, y float64  
}  
  
func main() {  
    p := Ponto{2, 3}  
    fmt.Printf("p\t%#v\n\n", p)  
  
    → var pp *Ponto  
  
    → pp = new(Ponto)  
    fmt.Printf("pp\t%#v\n", pp)  
    fmt.Printf("\t(%p)\n\n", pp)  
  
    → pp = &p  
    fmt.Printf("pp\t%#v\n", pp)  
    fmt.Printf("\t(%p)\n\n", pp)  
  
    fmt.Printf("*pp\t%#v\n\n", *pp)  
}
```



# FORMATO %P MOSTRA O PONTEIRO EM SI, NÃO SEU ALVO

```
type Ponto struct {  
    x, y float64  
}  
  
func main() {  
    p := Ponto{2, 3}  
    fmt.Printf("p\t%#v\n\n", p)  
  
    var pp *Ponto  
  
    pp = new(Ponto)  
    fmt.Printf("pp\t%#v\n", pp)  
    fmt.Printf("\t(%p)\n\n", pp)  
  
    pp = &p  
    fmt.Printf("pp\t%#v\n", pp)  
    fmt.Printf("\t(%p)\n\n", pp)  
  
    fmt.Printf("*pp\t%#v\n\n", *pp)  
}
```

p main.Ponto{x:2, y:3}

pp &main.Ponto{x:0, y:0}  
(0xc0000140c0)

pp &main.Ponto{x:2, y:3}  
(0xc000014080)

\*pp main.Ponto{x:2, y:3}

# MAS O QUE HÁ NESSAS "CAIXAS"?

---

```
s := []int{1, 2, 3}
s2 := s
s2[0]++
fmt.Printf("s\t%#v\ns2\t%#v\n\n", s, s2)
```

```
s    []int{2, 2, 3}
s2   []int{2, 2, 3}
```

```
m := map[byte]int{1: 1, 2: 2, 3: 3}
m2 := m
m2[3]++
fmt.Printf("m\t%#v\nm2\t%#v\n\n", m, m2)
```

```
m    map[uint8]int{0x2:2, 0x3:4, 0x1:1}
m2   map[uint8]int{0x1:1, 0x2:2, 0x3:4}
```

# ESSES SÃO EXEMPLOS DE “ALIASING”

---

Aliasing é literalmente “apelidamento”:  
ocorre quando uma coisa tem vários nomes ou apelidos.

```
pp := &Ponto{2, 3}  
pp2 := pp  
pp2.y++
```

```
pp  &main.Ponto{x:2, y:4}  
pp2 &main.Ponto{x:2, y:4}
```

```
s := []int{1, 2, 3}  
s2 := s  
s2[0]++
```

```
s  []int{2, 2, 3}  
s2 []int{2, 2, 3}
```

```
m := map[byte]int{1: 1, 2: 2, 3: 3}  
m2 := m  
m2[3]++
```

```
m  map[uint8]int{0x2:2, 0x3:4, 0x1:1}  
m2 map[uint8]int{0x1:1, 0x2:2, 0x3:4}
```

# SEMÂNTICA DE VALORES

---

## Semântica de valores:

- Variáveis são áreas de memória que contém os bits representando os dados em si.
- Não ocorre aliasing.
- Atribuição faz cópia dos dados.
- Parâmetros recebidos por funções são cópias dos argumentos passados.  
A função pode alterar sua cópia, mas não tem como alterar os dados do cliente (quem a invocou).

# SEMÂNTICA DE VALORES X SEMÂNTICA DE REFERÊNCIAS

---

## Semântica de valores:

- Variáveis são áreas de memória que contém os bits representando os dados em si.
- Não ocorre aliasing.
- Atribuição faz cópia dos dados.
- Parâmetros recebidos por funções são cópias dos argumentos passados.  
A função pode alterar sua cópia, mas não tem como alterar os dados do cliente (quem a invocou).

## Semântica de referências:

- Variáveis contém apenas referências ou ponteiros que apontam para os dados alocados em outra parte da memória.
- Pode ocorrer aliasing: mais de uma referência/ponteiro indicando o mesmo dado.
- Atribuição faz cópia da referência ou ponteiro; os dados são compartilhados.
- Parâmetros recebidos por funções são referências/ponteiros para os dados do cliente, que podem ser alterados pela função.



# UMA FORMA DE ENTENDER

---

Go adota a semântica de valores em todos os casos, mas em alguns casos o valor é um ponteiro ou uma estrutura que têm referências (ponteiros ocultos).

# AS PEGADINHAS

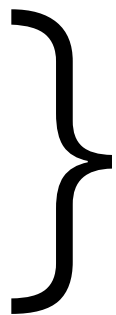
---

Referências em Go são implícitas: são embutidas em structs que você só pode inspecionar usando o pacote **unsafe** 🦴.

Ponteiros têm sintaxe explícita (&x, \*p) mas valores com referências **não têm sintaxe explícita**.

Somente 3 tipos nativos mutáveis usam referências:

- slice
- map
- channel



As únicas estruturas de dados construídas com **make()**



**Magic!**

Você não pode criar seus próprios tipos com referências.

A photograph of three people in a meeting, overlaid with a green tint. A woman on the left is pointing at a document. A man in the center is leaning over the table, looking at the document. A man on the right is sitting at a laptop, smiling. The background shows a wall with many sticky notes.

ThoughtWorks®

# ANATOMIA DE SLICES

---

Examinando um tipo de referência por dentro.

# ANALISADOR DE SLICE

Inspirado em  
post de Bill  
Kennedy:  
“Understanding  
slices”

[tgo.li/2QjwoR3](https://tgo.li/2QjwoR3)

Código-fonte  
deste exemplo:

[tgo.li/2L7xNEQ](https://tgo.li/2L7xNEQ)

```
package main

import (
    "fmt"
    "unsafe"
)

func InspectSlice(intSlice []int) {

    fmt.Println("intSlice:")
    fmt.Printf("\t%#v\n\n", intSlice)

    // Get slicePtr of slice structure
    slicePtr := unsafe.Pointer(&intSlice)
    ptrSize := unsafe.Sizeof(slicePtr)

    // Compute addresses of len and cap
    lenAddr := uintptr(slicePtr) + ptrSize
    capAddr := uintptr(slicePtr) + (ptrSize * 2)

    // Create pointers to len and cap
    lenPtr := (*int)(unsafe.Pointer(lenAddr))
    capPtr := (*int)(unsafe.Pointer(capAddr))

    // Get pointer to underlying array
    // How to do this without hardcoding the array size?
    arrayPtr := (*[100]int)(unsafe.Pointer((*uintptr)(slicePtr)))

    fmt.Println("intSlice:")

    // Not using %T on next line to show expected data array size
    // fmt.Printf("\t%p: data %T = %p\n", slicePtr, arrayPtr, arrayPtr)
    fmt.Printf("\t%p: data *[%d]int = %p\n", slicePtr, *capPtr, arrayPtr)

    fmt.Printf("\t%p: len %T = %d\n", lenPtr, *lenPtr, *lenPtr)

    fmt.Printf("\t%p: cap %T = %d\n", capPtr, *capPtr, *capPtr)

    fmt.Println("data:")

    for index := 0; index < *capPtr; index++ {
        fmt.Printf("\t%p: [%d] %T = %d\n",
            &(*arrayPtr)[index], index, (*arrayPtr)[index], (*arrayPtr)[index])
    }
}

func main() {
    intSlice := make([]int, 3, 5)
    intSlice[0] = 11
    intSlice[1] = 12
    intSlice[2] = 13

    InspectSlice(intSlice)

    for _, n := range []int{140, 150, 160} {
        intSlice = append(intSlice, n)
        InspectSlice(intSlice)
    }
}
```

```
intSlice:
    []int{11, 12, 13}

intSlice:
    @0xc00000a060: data *[5]int = 0xc000072030
    @0xc00000a068: len int = 3
    @0xc00000a070: cap int = 5
data:
    @0xc000072030: [0] int = 11
    @0xc000072038: [1] int = 12
    @0xc000072040: [2] int = 13
    @0xc000072048: [3] int = 0
    @0xc000072050: [4] int = 0

intSlice:
    []int{11, 12, 13, 140}

intSlice:
    @0xc00000a0a0: data *[5]int = 0xc000072030
    @0xc00000a0a8: len int = 4
    @0xc00000a0b0: cap int = 5
data:
    @0xc000072030: [0] int = 11
    @0xc000072038: [1] int = 12
    @0xc000072040: [2] int = 13
    @0xc000072048: [3] int = 140
    @0xc000072050: [4] int = 0

intSlice:
    []int{11, 12, 13, 140, 150}

intSlice:
    @0xc00000a0e0: data *[5]int = 0xc000072030
    @0xc00000a0e8: len int = 5
    @0xc00000a0f0: cap int = 5
data:
    @0xc000072030: [0] int = 11
    @0xc000072038: [1] int = 12
    @0xc000072040: [2] int = 13
    @0xc000072048: [3] int = 140
    @0xc000072050: [4] int = 150

intSlice:
    []int{11, 12, 13, 140, 150, 160}

intSlice:
    @0xc00000a120: data *[10]int = 0xc0000180f0
    @0xc00000a128: len int = 6
    @0xc00000a130: cap int = 10
data:
    @0xc0000180f0: [0] int = 11
    @0xc0000180f8: [1] int = 12
    @0xc000018100: [2] int = 13
    @0xc000018108: [3] int = 140
    @0xc000018110: [4] int = 150
    @0xc000018118: [5] int = 160
    @0xc000018120: [6] int = 0
    @0xc000018128: [7] int = 0
    @0xc000018130: [8] int = 0
    @0xc000018138: [9] int = 0

intSlice:
    []int{11, 12, 13, 140, 150, 160, 170}

intSlice:
    @0xc00000a160: data *[10]int = 0xc0000180f0
    @0xc00000a168: len int = 7
    @0xc00000a170: cap int = 10
data:
    @0xc0000180f0: [0] int = 11
    @0xc0000180f8: [1] int = 12
    @0xc000018100: [2] int = 13
    @0xc000018108: [3] int = 140
    @0xc000018110: [4] int = 150
    @0xc000018118: [5] int = 160
    @0xc000018120: [6] int = 170
    @0xc000018128: [7] int = 0
    @0xc000018130: [8] int = 0
    @0xc000018138: [9] int = 0
```

# ANALISADOR DE SLICE: MAIN

---

```
func main() {  
    intSlice := make([]int, 3, 5)  
    intSlice[0] = 11  
    intSlice[1] = 12  
    intSlice[2] = 13  
  
    InspectSlice(intSlice)  
  
    for _, n := range []int{140, 150, 160} {  
        intSlice = append(intSlice, n)  
        InspectSlice(intSlice)  
    }  
}
```

Código-fonte  
deste exemplo:

[tgo.li/2L7xNEQ](https://tgo.li/2L7xNEQ)



# ANALISADOR DE SLICE: INSPECT SLICE

---

Código-fonte  
deste exemplo:  
[tgo.li/2L7xNEQ](https://tgo.li/2L7xNEQ)

```
func InspectSlice(intSlice []int) {

    fmt.Println("intSlice:")
    fmt.Printf("\t%#v\n\n", intSlice)

    // Get slicePtr of slice structure
    slicePtr := unsafe.Pointer(&intSlice)
    ptrSize := unsafe.Sizeof(slicePtr)

    // Compute addresses of len and cap
    lenAddr := uintptr(slicePtr) + ptrSize
    capAddr := uintptr(slicePtr) + (ptrSize * 2)

    // Create pointers to len and cap
    lenPtr := (*int)(unsafe.Pointer(lenAddr))
    capPtr := (*int)(unsafe.Pointer(capAddr))

    // Get pointer to underlying array
    arrayPtr := (*[100]int)(unsafe.Pointer((*uintptr)(slicePtr)))

    fmt.Println("intSlice:")

    fmt.Printf("\t%p: data *[%d]int = %p\n", slicePtr, *capPtr, arrayPtr)
    fmt.Printf("\t%p: len %T = %d\n", lenPtr, *lenPtr, *lenPtr)
    fmt.Printf("\t%p: cap %T = %d\n", capPtr, *capPtr, *capPtr)

    fmt.Println("data:")
    for index := 0; index < *capPtr; index++ {
        fmt.Printf("\t%p: [%d] %T = %d\n",
            &(*arrayPtr)[index], index, (*arrayPtr)[index], (*arrayPtr)[index])
    }
}
```

# ANALISADOR DE SLICE: MAIN

```
func main() {  
    intSlice := make([]int, 3, 5)  
    intSlice[0] = 11  
    intSlice[1] = 12  
    intSlice[2] = 13
```

Slice é um  
struct com  
três campos:  
**data, len, cap**

→ InspectSlice(intSlice)

```
for _, n := range []int{140, 150, 160} {  
    intSlice = append(intSlice, n)  
    InspectSlice(intSlice)  
}
```

}

*Array subjacente  
(underlying array)*

[tgo.li/2L7xNEQ](https://tgo.li/2L7xNEQ)

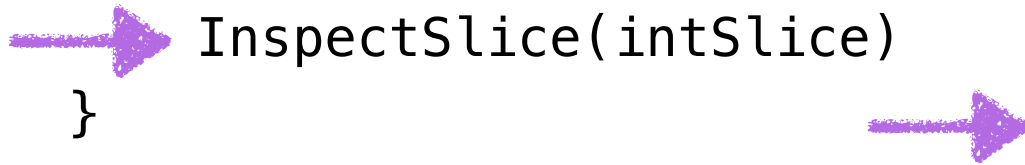
intSlice:  
[]int{11, 12, 13}

intSlice:  
@0xc00000a060: data \*[5]int = 0xc000072030  
@0xc00000a068: len int = 3  
@0xc00000a070: cap int = 5

data:  
@0xc000072030: [0] int = 11  
@0xc000072038: [1] int = 12  
@0xc000072040: [2] int = 13  
@0xc000072048: [3] int = 0  
@0xc000072050: [4] int = 0

# ANALISADOR DE SLICE: MAIN

```
func main() {  
    intSlice := make([]int, 3, 5)  
    intSlice[0] = 11  
    intSlice[1] = 12  
    intSlice[2] = 13  
  
    InspectSlice(intSlice)  
  
    for _, n := range []int{140, 150, 160} {  
        intSlice = append(intSlice, n)  
        InspectSlice(intSlice)  
    }  
}
```



```
intSlice:  
    []int{11, 12, 13, 140}  
  
intSlice:  
    @0xc00000a0a0: data *[5]int = 0xc000072030  
    @0xc00000a0a8: len int = 4  
    @0xc00000a0b0: cap int = 5  
data:  
    @0xc000072030: [0] int = 11  
    @0xc000072038: [1] int = 12  
    @0xc000072040: [2] int = 13  
    @0xc000072048: [3] int = 140  
    @0xc000072050: [4] int = 0
```



# ANALISADOR DE SLICE: MAIN

```
func main() {  
    intSlice := make([]int, 3, 5)  
    intSlice[0] = 11  
    intSlice[1] = 12  
    intSlice[2] = 13  
    → InspectSlice(intSlice)  
  
    for _, n := range []int{140, 141, 142, 143, 144}  
        intSlice = append(intSlice, n)  
        → InspectSlice(intSlice)  
    }  
}
```

[tgo.li/2L7xNEQ](https://tgo.li/2L7xNEQ)

```
intSlice:  
    []int{11, 12, 13}  
  
intSlice:  
    @0xc00000a060: data *[5]int = 0xc000072030  
    @0xc00000a068: len int = 3  
    @0xc00000a070: cap int = 5  
data:  
    @0xc000072030: [0] int = 11  
    @0xc000072038: [1] int = 12  
    @0xc000072040: [2] int = 13  
    @0xc000072048: [3] int = 0  
    @0xc000072050: [4] int = 0  
  
intSlice:  
    []int{11, 12, 13, 140}  
  
intSlice:  
    @0xc00000a0a0: data *[5]int = 0xc000072030  
    @0xc00000a0a8: len int = 4  
    @0xc00000a0b0: cap int = 5  
data:  
    @0xc000072030: [0] int = 11  
    @0xc000072038: [1] int = 12  
    @0xc000072040: [2] int = 13  
    @0xc000072048: [3] int = 140  
    @0xc000072050: [4] int = 0
```

# ANALISADOR DE SLICE

Ao fazer **append**, quando a capacidade inicial é ultrapassada, um novo array subjacente é criado com o dobro da capacidade, e o conteúdo anterior é copiado para lá.

Código-fonte deste exemplo:

[tgo.li/2L7xNEQ](https://tgo.li/2L7xNEQ)

```
intSlice:
  []int{11, 12, 13, 140, 150}

intSlice:
  @0xc00000a0e0: data *[5]int = 0xc000072030
  @0xc00000a0e8: len int = 5
  @0xc00000a0f0: cap int = 5
data:
  @0xc000072030: [0] int = 11
  @0xc000072038: [1] int = 12
  @0xc000072040: [2] int = 13
  @0xc000072048: [3] int = 140
  @0xc000072050: [4] int = 150

intSlice:
  []int{11, 12, 13, 140, 150, 160}

intSlice:
  @0xc00000a120: data *[10]int = 0xc0000180f0
  @0xc00000a128: len int = 6
  @0xc00000a130: cap int = 10
data:
  @0xc0000180f0: [0] int = 11
  @0xc0000180f8: [1] int = 12
  @0xc000018100: [2] int = 13
  @0xc000018108: [3] int = 140
  @0xc000018110: [4] int = 150
  @0xc000018118: [5] int = 160
  @0xc000018120: [6] int = 0
  @0xc000018128: [7] int = 0
  @0xc000018130: [8] int = 0
  @0xc000018138: [9] int = 0
```

# UMA FORMA DE ENTENDER

---

Go adota a semântica de valores em todos os casos, mas em alguns casos o valor é uma referência ou ponteiro.

# CONCLUSÃO

---

# ANOTHER WAY TO DO A BIG TITLE OR SECTION DIVIDER

---

*Add a subhead if you want.*



ThoughtWorks®

# FULL BLEED PHOTO WITH TEXT OVERLAY

---

Place the text wherever the photo subjects dictates. You can delete this subhead and the keyline above.

# YOUR BASIC, HARD-WORKING CONTENT SLIDE

---

## A content headline

Did you know hitting tab at the beginning of a line demotes it from a headline (default) to body copy (what you see here)?

Shift-tab makes a headline again.

And back down to body by hitting tab again.

- Tab one more time to start bullets
- Enter begins the next bullet
  - Tab again for sub-bullets
    - And again for tiny bullets

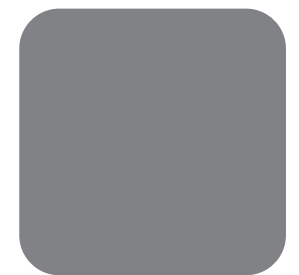
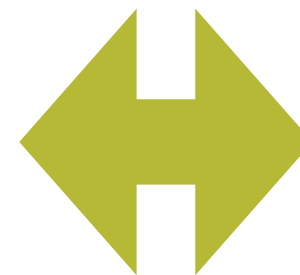
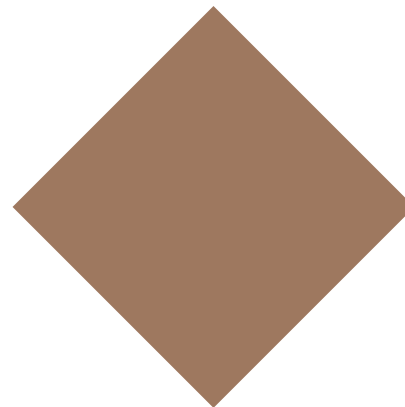
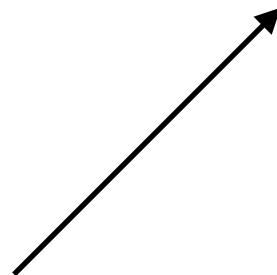
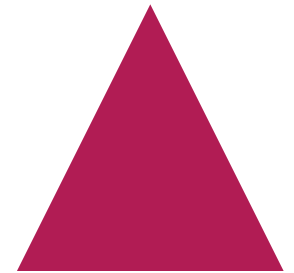
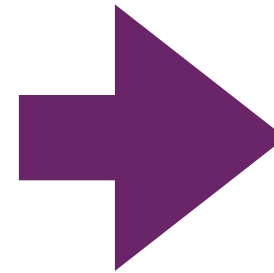
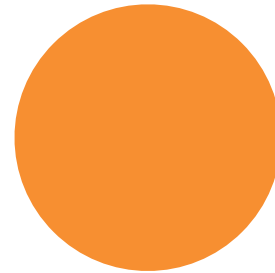
*Sometimes you need a blank template.*



---

# SHAPES

---

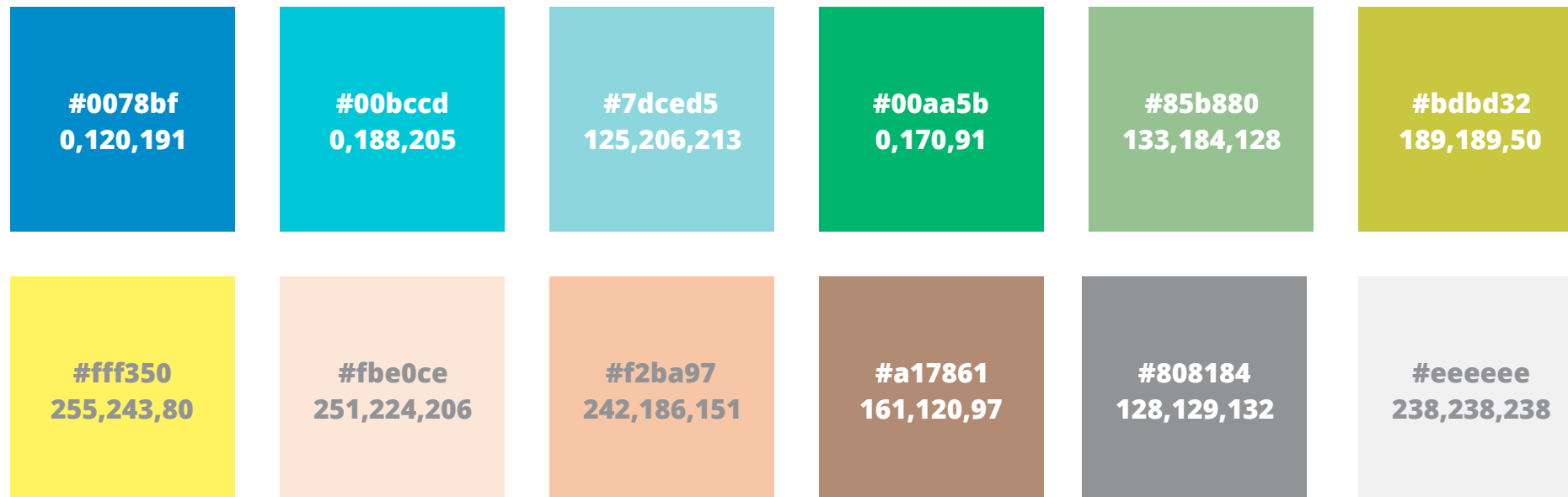


---

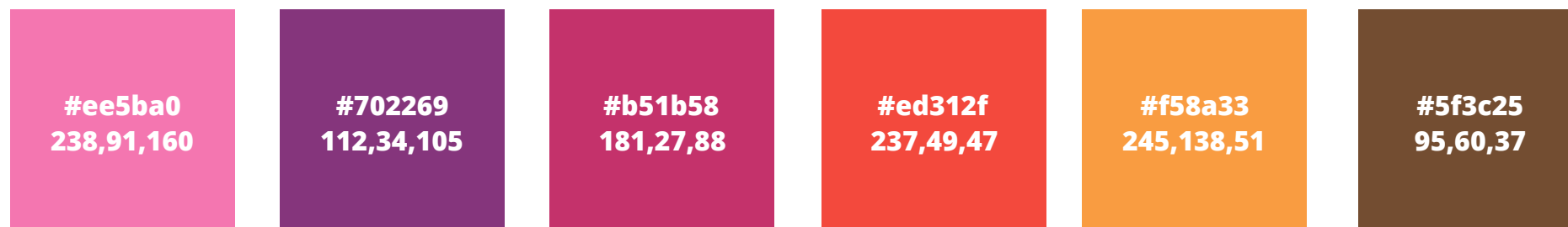
# COLOR-PALETTE

---

## *Safe on Projectors*



## *Risky on Projectors*



# BOXES, SHAPES AND TABLES

---

*By default, a floating text box looks like this*

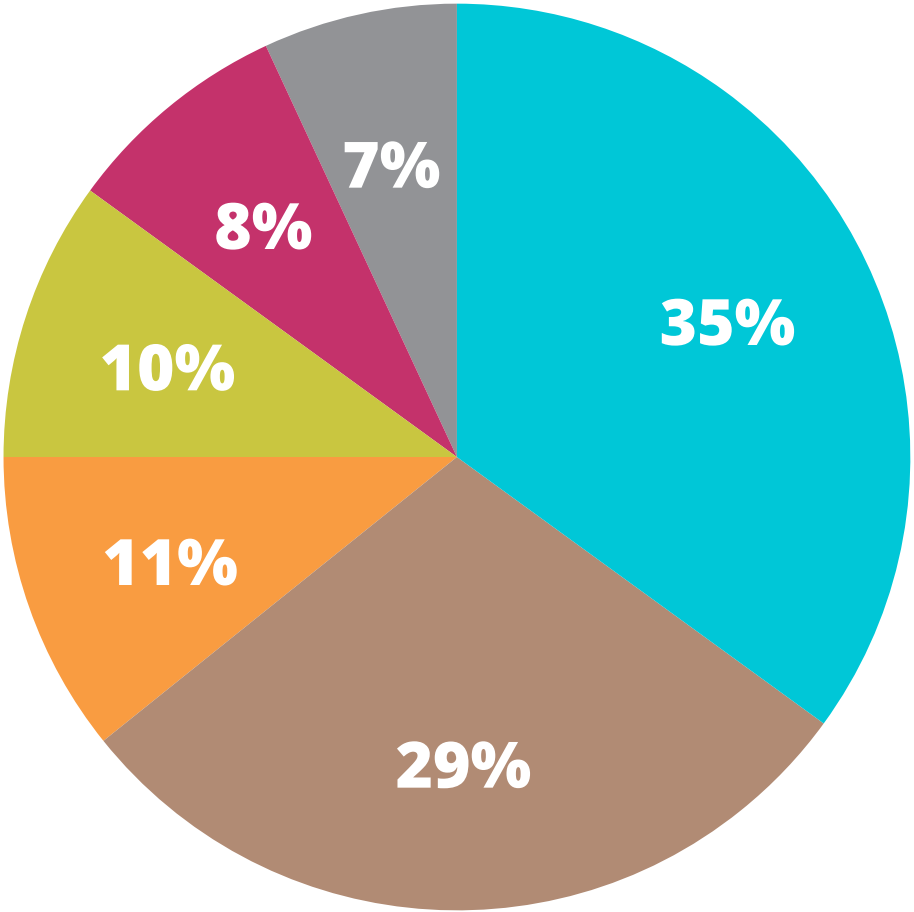
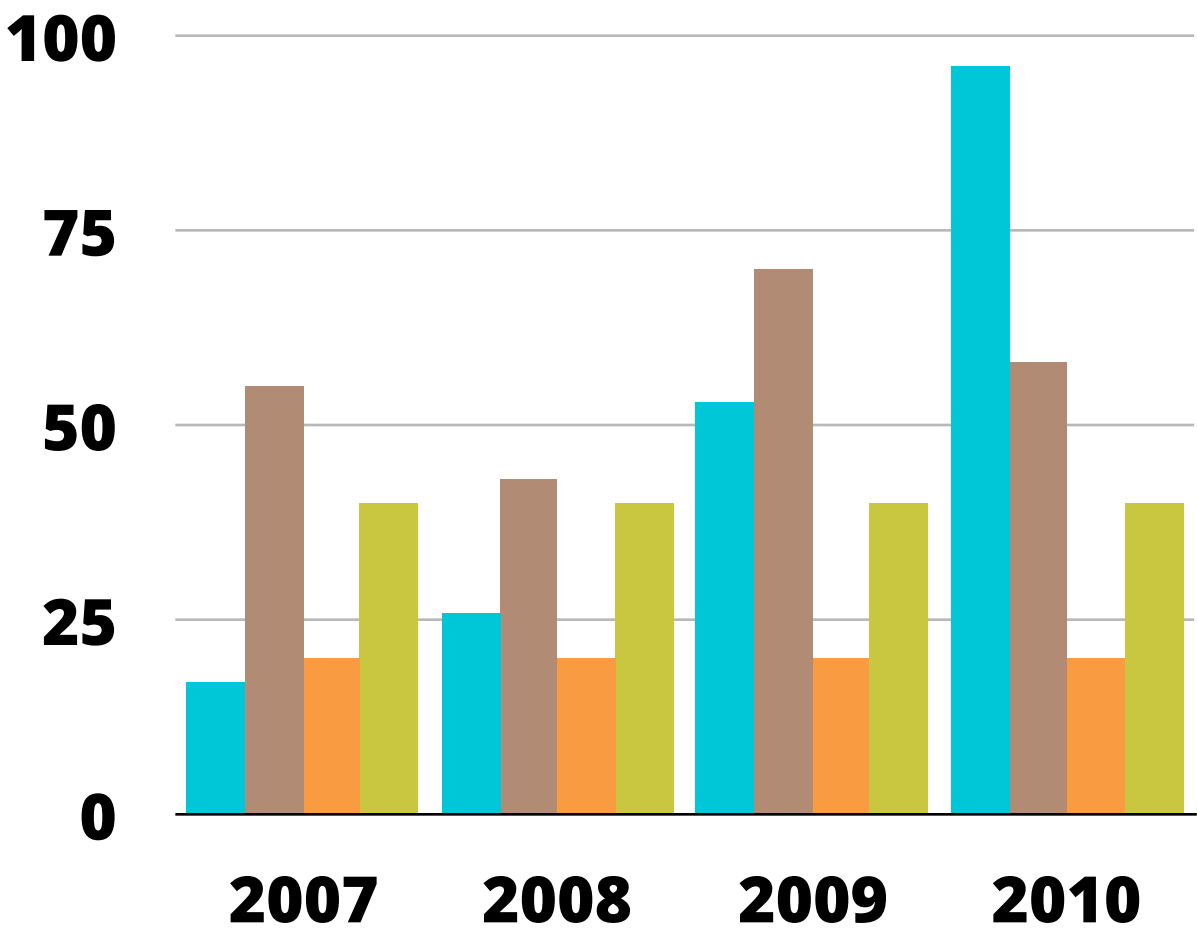
Shapes are tinted gray, with lots of padding.

Works for any shape

	Header rows ...	are black.
Header columns ...	If you insert a table ...	it should automatically ...
are gray.	pick up this style.	Enjoy.

# CHARTS

---



# PARTING THOUGHTS (USING THE CHECKLIST TEMPLATE)

---

First, get the proper font: <http://opensans.com/>

Start with a clean template.

When in doubt, copy and paste from something that already works.

Be brave:

- Learn to use master pages properly.
- Edit! Brevity, not design, makes presentations great.

# DON'T MISS MASTER SLIDES IN OTHER COLORS

---

- Black
- Blue
- Khaki
- Magenta
- Orange
- Green
- Pink

# THANK YOU

*For questions or suggestions:*

*Go to Brand Hub*

*<https://my.thoughtworks.com/groups/brand>*

**ThoughtWorks®**