

Endereços de Memória e Ponteiros em Go

Um guia prático para entender como Go trabalha com memória, referências e mutabilidade de dados

Integrantes:

Gustavo Maldanis Zanini

Rafael Carvalho de Souza

Samuel Dias Medeiros

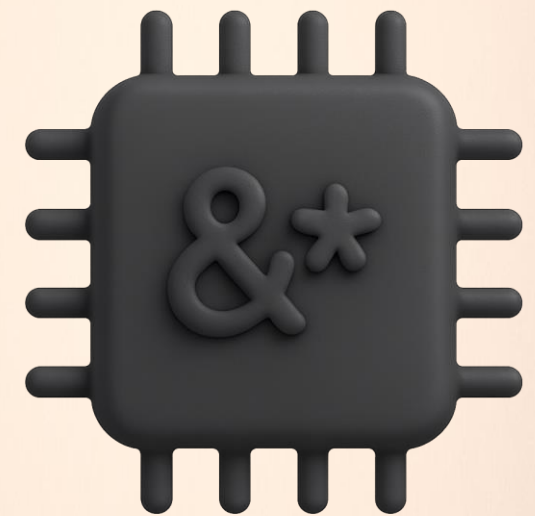
Mahgid Perez Thomé

Caio Lopes Vieira

João Gustavo Cardoso Freitas

Alberto Alexandre Suave

Robert Willian Vicente





O que é um Endereço de Memória?

O Conceito

Cada variável em um computador armazena seus dados em uma localização física na memória RAM. O **endereço de memória** funciona como o "CEP" dessa localização - um identificador único que marca onde os dados estão guardados.

Em Go, quando declaramos `var x int = 10`, o número 10 é armazenado em algum lugar específico da memória. Podemos descobrir e usar esse endereço.

O Ponteiro

Um **ponteiro** é uma variável especial que não armazena um valor direto como 10 ou "oi", mas sim o endereço de memória de outra variável.

Ele literalmente "aponta" para onde o valor real está guardado, permitindo acesso indireto aos dados.



Os Operadores Mágicos: & e *

Em Go, usamos dois operadores principais para trabalhar com ponteiros. Dominar estes símbolos é essencial:

Operador & (Address Of)

Nome: Endereço de

Função: Obtém o endereço de memória de uma variável

```
p := &x
```

Lê-se: "p recebe o endereço de x"

Operador * (Dereference)

Nome: Desreferência

Função: Obtém o valor armazenado no endereço apontado pelo ponteiro

```
fmt.Println(*p)
```

Lê-se: "imprime o valor apontado por p"

Exemplo Básico: Obtendo Endereço e Valor

Vamos ver na prática como declarar e usar ponteiros em Go:

```
package main

import "fmt"

func main() {
    // 1. Variável Comum
    num := 42
    fmt.Printf("Valor de 'num': %d\n", num)
    // Usamos '%p' para formatar o endereço de memória
    fmt.Printf("Endereço de 'num' (&num): %p\n", &num)

    // 2. Variável Ponteiro
    // Declaramos 'p' como um ponteiro para um 'int' (*int)
    var p *int

    // Atribuímos o endereço de 'num' ao ponteiro 'p'
    p = &num

    fmt.Printf("\nValor do ponteiro 'p' (Endereço): %p\n", p)

    // 3. Desreferenciando: Acessando o valor no endereço
    fmt.Printf("Valor apontado por 'p' (*p): %d\n", *p)
}
```



Resultado esperado: Valor de 'num': 42 | Endereço de 'num': 0xc000014080 | Valor do ponteiro 'p': 0xc000014080 | Valor apontado por 'p': 42

Nota: O endereço será diferente a cada execução do programa

Modificando Valores com Ponteiros

A principal utilidade dos ponteiros é permitir a **mutabilidade** - a capacidade de alterar o valor original através de seu endereço de memória.

```
package main

import "fmt"

func main() {
    x := 10
    p := &x // p aponta para o endereço de x
    fmt.Printf("Antes: x = %d\n", x) // x = 10

    // Usamos o operador * para alterar o valor no endereço
    // que p aponta
    *p = 20

    fmt.Printf("Depois: x = %d\n", x) // x = 20 (Alterado!)

    // x também mudou, pois p alterou o valor
    // diretamente na memória!
}
```

Observe que modificamos `*p`, mas a variável `x` foi alterada. Isso acontece porque ambas apontam para o mesmo local na memória.



Passagem por Valor: A Cópia

Regra de Ouro em Go

Argumentos de funções são passados **por valor** - ou seja, são copiados. Este é o comportamento padrão da linguagem.

Comportamento: A função recebe uma cópia do dado. Alterar a cópia não afeta a variável original.

```
func dobraValor(v int)
    v = v * 2 // Altera apenas a CÓPIA local de 'v'
{
func main() {
    a := 5
    dobraValor(a)
    fmt.Println("Valor de 'a' após a função:", a)
    // Output: 5 (Não mudou!)
}
```

A variável `a` permanece com valor 5 porque a função trabalhou apenas com uma cópia dela.

Passagem por Ponteiro: A Referência

Comportamento: A função recebe o **endereço** (o ponteiro) da variável, não uma cópia do valor.

Vantagem: Permite que a função altere a variável original fora de seu escopo, tornando a modificação permanente.

```
func dobraPonteiro(p *int) {  
    // Desreferencia para alterar o valor no endereço  
    *p = *p * 2  
}  
  
func main() {  
    a := 5  
    // Passamos o ENDEREÇO de 'a' (o ponteiro)  
    dobraPonteiro(&a)  
    fmt.Println("Valor de 'a' após a função:", a)  
    // Output: 10 (Mudou!)  
}
```



Por que isso funciona?

A função recebeu o endereço de memória onde **a** está armazenado, então ao modificar o conteúdo desse endereço, a variável original é alterada.



Receivers de Método: Valor vs Ponteiro

Em Go, ao definir um método para uma struct, devemos decidir se o **receiver** será um valor ou um ponteiro. Esta escolha impacta diretamente o comportamento.

Receiver por Valor

Sintaxe: `func (p Pessoa) ...`

Quando usar:

- Apenas para leitura de dados
- Quando a struct é pequena
- Quando não precisa alterar o estado

Receiver por Ponteiro

Sintaxe: `func (p *Pessoa) ...`

Quando usar:

- Se o método alterar o estado da struct
- Se a struct for muito grande (evita cópias)
- Para garantir mutabilidade



Exemplo Prático: Receivers em Ação

```
package main

import "fmt"

type Pessoa struct {
    Nome string
    Idade int
}

// Método com Receiver por VALOR (Recebe uma cópia)

func (p Pessoa) FazAniversarioValor() {
    p.Idade++ // Altera SÓ a cópia
    fmt.Println("Na função (Valor):", p.Idade)
}

// Método com Receiver por PONTEIRO (Recebe o endereço)

func (p *Pessoa) FazAniversarioPonteiro() {
    p.Idade++ // Altera a struct ORIGINAL
    fmt.Println("Na função (Ponteiro):", p.Idade)
}

func main() {
    p1 := Pessoa{Nome: "Alice", Idade: 30}

    // 1. Chamada por Valor: Não Altera a original
    p1.FazAniversarioValor()
    fmt.Println("Original (Valor):", p1.Idade) // 30 (Não mudou)

    // 2. Chamada por Ponteiro: Altera a original
    p1.FazAniversarioPonteiro()
    fmt.Println("Original (Ponteiro):", p1.Idade) // 31 (Mudou!)
}
```

Resumo e Melhores Práticas



O que são Ponteiros

Permitem acessar e modificar o valor de uma variável através de seu endereço de memória



Mutabilidade

Use quando uma função ou método precisa alterar o estado de uma variável (como em `json.Unmarshal`)



Eficiência

Evite cópias desnecessárias de grandes structs e arrays ao passá-los para funções



Cuidado Importante: Evite desreferenciar um ponteiro `nil`! Isso causa um panic em tempo de execução e encerra o programa.

Use ponteiros conscientemente. O Garbage Collector de Go gerencia automaticamente a limpeza de memória, mas cabe ao programador decidir quando a mutabilidade e a eficiência são necessárias.

