**Apresentação: Endereços de Memória e Ponteiros em Go (Golang)**

**1. Introdução**

* **Título:** Endereços de Memória e Ponteiros em Go (Golang)

**(Slide 2: O que é um Endereço de Memória?)**

* **Conceito:** Cada variável em um computador armazena seus dados em uma localização física na memória RAM. O **Endereço de Memória** é o "CEP" dessa localização.
* **Em Go:** Quando declaramos var x int = 10, o número 10 está armazenado em algum lugar. Podemos descobrir esse endereço.
* **Ponteiro:** Uma variável especial que não armazena um valor (como 10 ou "oi"), mas sim o **endereço de memória** de outra variável. Ele "aponta" para onde o valor real está.

**2. Sintaxe Básica (Os Operadores Mágicos)**

**: Operadores & e \*)**

Em Go, usamos dois operadores principais para trabalhar com ponteiros:

|  |  |  |  |
| --- | --- | --- | --- |
| **Operador** | **Nome** | **Função** | **Exemplo** |
| **&** | **Address Of** (Endereço de) | Obtém o endereço de memória de uma variável. | p := &x |
| **\*** | **Dereference** (Desreferência) | Obtém o valor armazenado no endereço apontado pelo ponteiro. | fmt.Println(\*p) |

**(Slide 4: Exemplo Básico de Ponteiro)**

**Código de Exemplo 1: Obtendo Endereço e Valor**

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 para demonstração)**

Valor de 'num': 42   
Endereço de 'num' (&num): 0xc000014080 (O endereço será diferente a cada execução)   
   
Valor do ponteiro 'p' (Endereço): 0xc000014080   
Valor apontado por 'p' (\*p): 42 

**3. Modificando Valores com Ponteiros**

**(Slide 5: Mutabilidade e Alteração)**

A principal utilidade dos ponteiros é permitir a **mutabilidade** (alteração) do valor original através de seu endereço.

**Código de Exemplo 2: Alterando o Valor Original**

Go

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!   
} 

**4. Passagem de Parâmetros: Valor vs. Ponteiro**

**(Slide 6: Passagem por Valor)**

* **Regra de Ouro em Go:** Argumentos de funções são passados por **Valor** (copiados).
* **Comportamento:** A função recebe uma **cópia** do dado. Alterar a cópia não afeta a variável original.

**Código de Exemplo 3.1: Passagem por Valor (A Cópia)**

Go

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!)   
} 

**(Slide 7: Passagem por Ponteiro)**

* **Comportamento:** A função recebe o **Endereço** (o ponteiro) da variável.
* **Vantagem:** Permite que a função altere a variável original fora de seu escopo.

**Código de Exemplo 3.2: Passagem por Ponteiro (A Referência)**

Go

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!)   
} 

**5. Aplicações Práticas: Métodos em Structs**

**(Slide 8: Receivers de Método)**

Em Go, ao definir um método para uma struct, devemos decidir se o *receiver* será um **Valor** ou um **Ponteiro**.

|  |  |  |
| --- | --- | --- |
| **Receiver** | **Sintaxe** | **Quando Usar** |
| **Valor** | func (p Pessoa) ... | Apenas para leitura. Se a struct for pequena. |
| **Ponteiro** | func (p \*Pessoa) ... | Se o método for **alterar** o estado da struct. Se a struct for **muito grande** (para evitar cópias). |

**(Slide 9: Exemplo de Struct com Receivers)**

**Código de Exemplo 4: Receiver por Valor vs. Ponteiro**

Go

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!)   
} 

**6. Conclusão**

**(Slide 10: Resumo e Melhores Práticas)**

* **Ponteiros em Go:** Permitem acessar e modificar o valor de uma variável em seu endereço de memória.
* **Uso Primário:**
* **Mutabilidade:** Quando uma função ou método precisa alterar o estado de uma variável (como em json.Unmarshal).
* **Eficiência:** Evitar a cópia de grandes structs e arrays ao passar para funções.
* **Lembrete:** Use ponteiros conscientemente. O *Garbage Collector* de Go gerencia a limpeza, mas o programador decide quando a mutabilidade (e a eficiência) é necessária.
* **Cuidado:** Evite desreferenciar um ponteiro nil! Isso causa um *panic* em tempo de execução.