Ponteiros em C

Aleksander Pires Calixto Negrão

Carlos Emanuel Magalhães Silva

José Carlos Silva Santos

José Davi Alexandre dos Santos

José Vinicius Cavalcante Soares

Equipe

O que veremos - Ponteiros

1. Definições e fundamentos
2. Alocação dinâmica e liberação de memória
3. Aritmética de ponteiros
4. Ponteiros como parâmetros
5. Ponteiro para função
6. Armadilhas e boas práticas

Fundamentos

Endereço de variável

Definição de **ponteiro**

Desreferenciamento explícito

Fundamentos

Endereço de variável

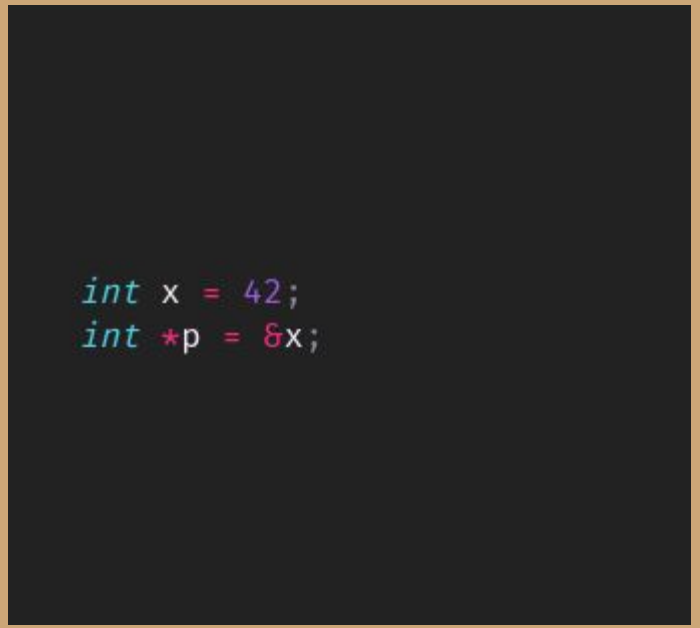
O **operador &** retorna o endereço de memória da variável

```c
int x = 42;
printf("%p\n", &x); // 0x7fffffffd1ac
```

Fundamentos

Ponteiro

Um **ponteiro** é uma variável que armazena um **endereço de memória** como valor.



Fundamentos

Ponteiro

Um **ponteiro** é uma variável que armazena um **endereço de memória** como valor.

- Fortemente tipados

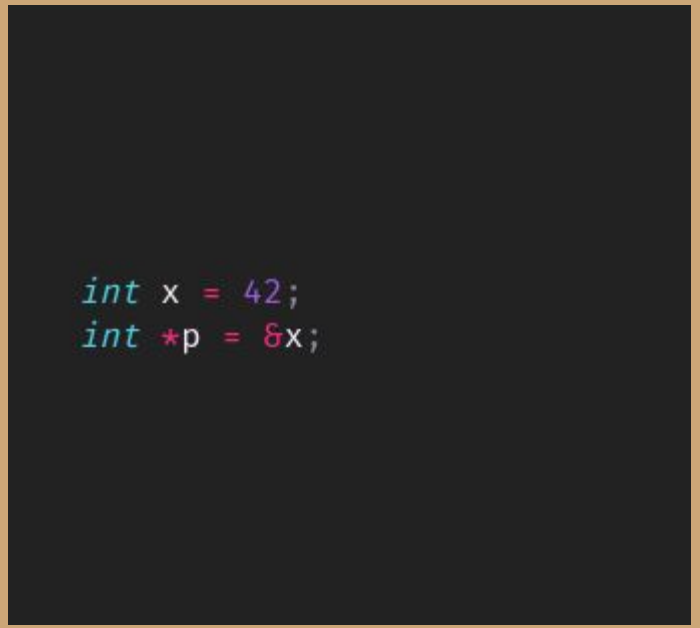
```
int x = 42;
int *p = &x;
```

Fundamentos

Ponteiro

Um **ponteiro** é uma variável que armazena um **endereço de memória** como valor.

- Fortemente tipados
- Não armazena o *valor* da variável



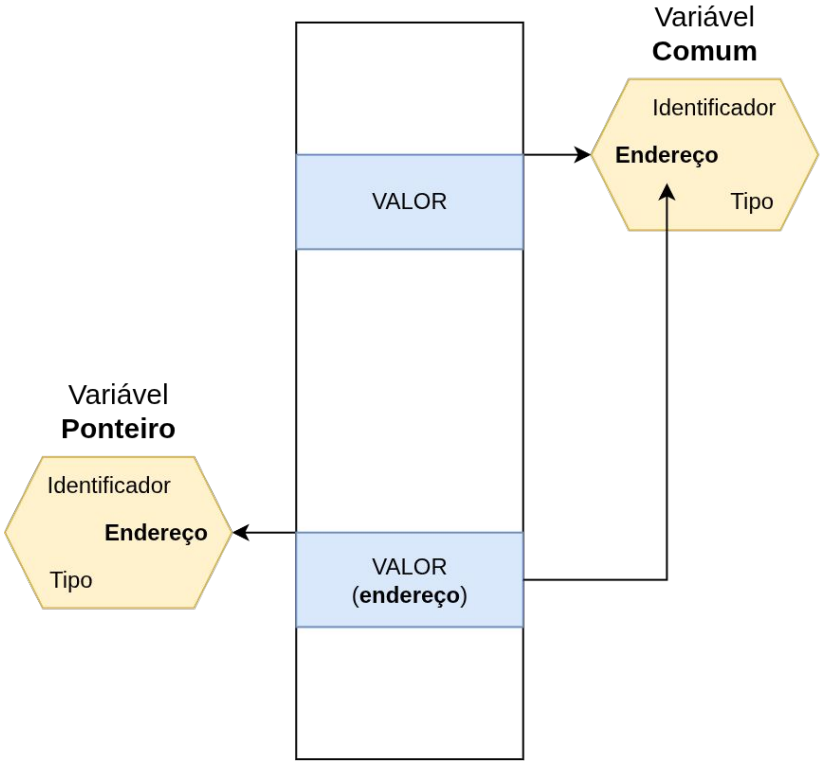
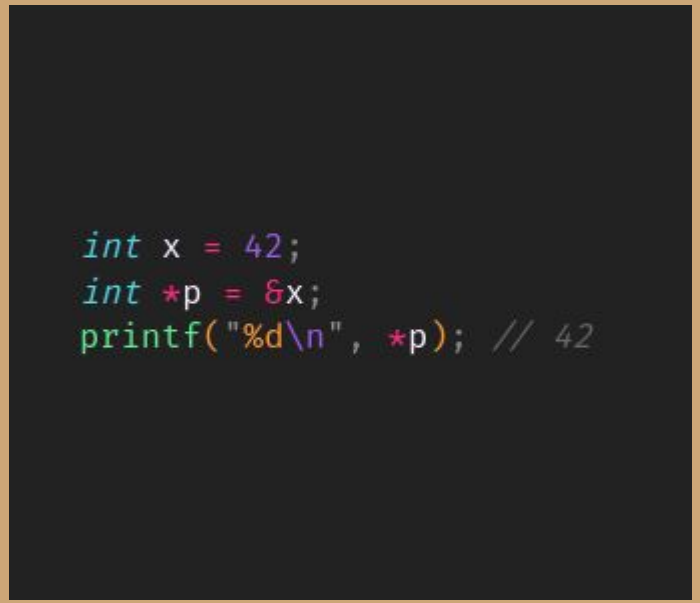


Diagrama de ponteiro e variável na memória

Fundamentos

Desreferenciamento

Desreferenciamento é a operação de *acessar o valor* do endereço do ponteiro



Alocação dinâmica e liberação de memória

Memória estática, heap e stack

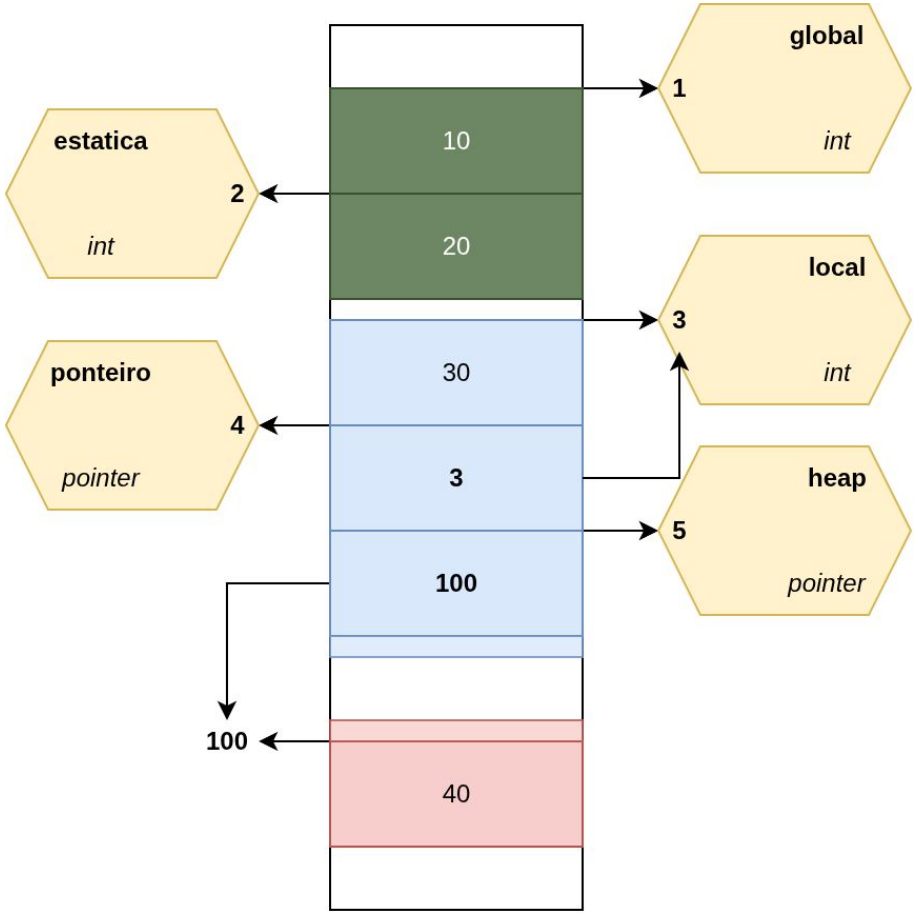
malloc()

free()

Memória estática, Heap e Stack

- **Memória estática**: variáveis estáticas e globais
- **Memória stack**: variáveis locais, parâmetros e retorno de funções
- **Memória heap**: variáveis de alocação dinâmica

```c
#include <stdlib.h>

int global = 10;
static int estatica = 20;

int main(void) {
    int local = 30;
    int *ponteiro = &local;

    int *heap = malloc(sizeof(int));
    *heap = 40;

    free(heap);
    return 0;
}
```



Memória estática, Heap e Stack

Alocação dinâmica

***malloc*()**

- reservar espaço na memória heap
- tamanho arbitrário
- em tempo de execução

```c
int *ptr = malloc(sizeof(*ptr));
free(ptr);
```

Alocação dinâmica

***malloc*()**

- retorna um ponteiro para o início do bloco reservado
- ponteiro do tipo genérico **void***
- atribuição com conversão implícita ao tipo do ponteiro (coerção)

Alocação dinâmica

***malloc*()**

- necessidade de liberar memória explicitamente (*free*)
- alocação não inicializa a memória
- retorna *NULL* caso a heap esteja cheia

```c
int *ptr = malloc(sizeof(int));
if (ptr == NULL) {
    return 1;
}
free(ptr);
```

Aritmética de ponteiros

Relação entre ponteiros e arrays

Aritmética de ponteiros e arrays

Aritmética de ponteiros e strings

Relação entre ponteiros e **arrays**

Decaimento

Um array expresso como ponteiro decai para o endereço do seu primeiro elemento

```c
int a[] = {1, 2, 3};
int *p = a;
int *q = &(a[0]);
```

Relação entre ponteiros e arrays

Incremento a ponteiro de array

- **p++** avança ***sizeof***(***p**) bytes na memória
- equivale ao **próximo elemento** do array

```
int a[] = {1, 2, 3};
int *p = a;
p++;
```

Relação entre ponteiros e arrays

Equivalência de expressões

O acesso de índice equivale ao desreferenciamento da sua soma ao ponteiro

```c
int a[] = {1, 2, 3};
int *p = a;

if (a[1] == *(p+1)) {
    // true
}
```

Aritmética de ponteiros e arrays

Subtração de ponteiros

Retorna o número de elementos entre eles

```c
int a[] = {1, 2, 3};
int diff = &a[2] - &a[0];
//  diff = 2
```

Aritmética de ponteiros e strings

Subtração de ponteiros

- Strings são arrays de char terminados com `**\0**`

```c
char s[] = "hello";
for (char *c = s; *c != '\0'; c++) {
    *c = *c - 32; // transforma em maiúscula
}
```

Ponteiros como parâmetros

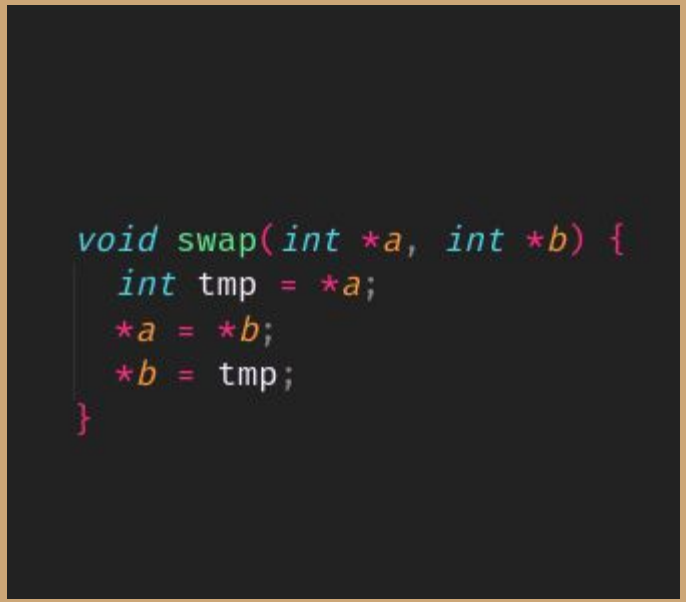
Passando por referência

Passando ponteiros de ponteiros

Ponteiros parâmetros

Passagem por referência

Permite que uma função modifique o valor de uma **variável do chamador**.



```c
void swap(int *a, int *b) {
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

Ponteiros parâmetros

Ponteiro para ponteiro

Permite que a função **altere o ponteiro do chamador**, útil para alocação dinâmica

```c
void make_array(int **out, size_t n) {
  *out = malloc(n * sizeof(int));
}
```

Ponteiro para função

Exemplo prático

Assinatura

Ponteiro para função

Exemplo prático

- armazenar o endereço de uma função e chamá-la indiretamente.
- útil para callbacks e predicados
- **qsort**

```c
int cmp_ints(const void *a, const void *b) {
    int ia = *(const int*)a;
    int ib = *(const int*)b;
    return ia - ib;
}

int main(void) {
    int v[] = {42, 5, 17, 3, 99, 42};
    qsort(v, 6, sizeof(int), cmp_ints);
}
```

Ponteiro para função

Assinatura

tipo_retorno (*ptr)(*args*)

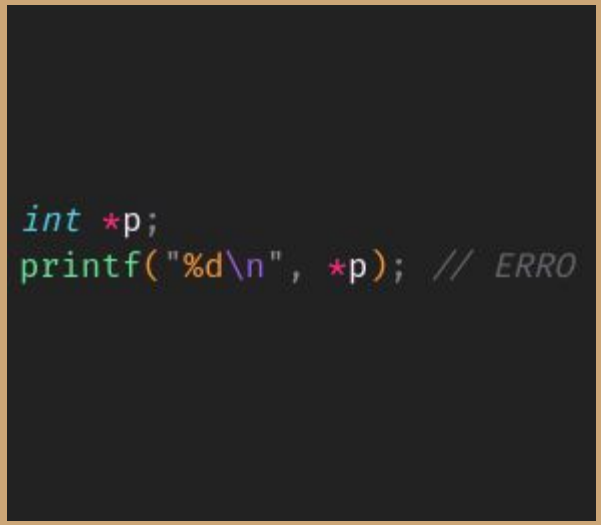
```c
int soma(int a, int b) {
    return a + b;
}

int main(void) {

  int (*funcao)(int, int) = soma;
  int res = funcao(3, 4);
  //  res = 7

  return 0;
}
```

Armadilhas e boas práticas

Armadilhas

Ponteiro selvagem

Problema: Ponteiro não inicializado



Armadilhas

Ponteiro selvagem

Problema: Ponteiro não inicializado

Solução: Inicializar como **NULL**

Armadilhas

Ponteiro pendurado (dangling)

Problema: ponteiro apontando para memória liberada

Armadilhas

Ponteiro pendurado (dangling)

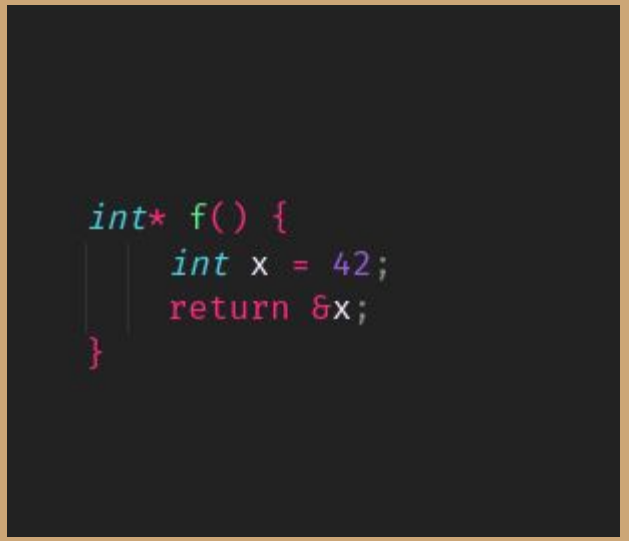
Problema: ponteiro apontando para memória liberada

Solução: atribuir a **NULL**

Armadilhas

Ponteiro pendurado à variável local

Problema: Ponteiro apontando para variável local (stack) fora do escopo

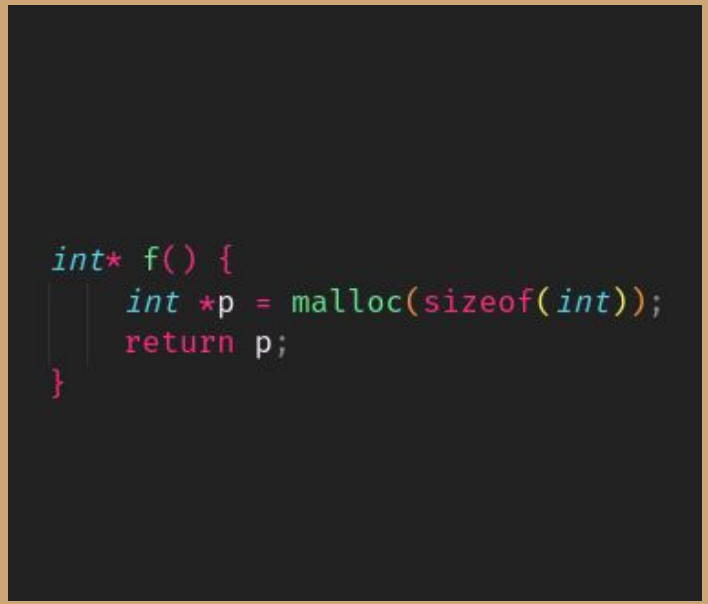


Armadilhas

Ponteiro pendurado à variável local

Problema: Ponteiro apontando para variável local (stack) fora do escopo

Solução: alocar na heap



Armadilhas

Double free

Chamar `**free**()` duas vezes para o mesmo ponteiro

Armadilhas

Vazamento (leak) de memória

Memória alocada dinamicamente que perde todos os seus ponteiros sem ser liberada por `free`.

Armadilhas

Vazamento (leak) de memória

Memória alocada dinamicamente que perde todos os seus ponteiros sem ser liberada por `free`.

Solução: chamar `free` antes de nova alocação

Armadilhas

malloc NULL

malloc() retorna **NULL** quando a heap está cheia, o que deve ser checado