Fundamentos de Programação

Aula 06 - Ponteiros

Arnaldo Barreto Vila Nova

Sumário

- Endereço de Memória
- Conceito de Ponteiros
- Declaração e Operações
- Alocação de Memória
- Exercícios

Endereço de Memória

- A memória RAM de um computador é separada em uma sequência de bytes
- Cada byte da memória é numerado de forma sequencial (byte 0, byte 1, byte 2, … , byte 3456088…)
- Esta numeração é chamada de **endereço de memória**
- Ou seja, um PC com 8GB de RAM vai ter 8.589.934.592 endereços de memória
- Tudo que é carregado na memória ocupa uma certa quantidade de endereços

Endereço de Memória

- O valor de uma variável ocupa bytes consecutivos na memória
- Ex.:
  - `char letra` → ocupa 1 byte
  - `int num` → ocupa 4 bytes
  - `float num2` → ocupa 4 bytes
  - `double num3` → ocupa 8 bytes

Endereço de Memória

- Você pode usar a função **sizeof** para saber o quanto de memória um tipo ou uma variável utiliza
- Ex.:
  - `print("int ocupa %d bytes", sizeof(int));`
  - `print("var ocupa %d bytes", sizeof(var));`

Endereço de Memória

- Em C, o endereço de memória pode ser acessado pelo **operador de endereço** ou **operador de ponteiro** (**&**)
- Ex.:

```
int x;
scanf(“%d”, &x);
printf(“O valor de x: %d”, x);
printf(“O endereço de x: %p”, &x);
printf(“O endereço em decimal: %d”, &x);
```

Endereço de Memória

- Em C, o endereço de memória pode ser acessado pelo **operador de endereço** ou **operador de ponteiro** (**&**)
- Ex.:

```
int x;
scanf("%d", &x);
printf("O valor de x: %d", x);
printf("O endereço de x: %p", &x);
printf("O endereço em decimal: %d", &x);
```

&x representa o endereço de memória da variável x

Endereço de Memória

- Em C, o endereço de memória pode ser acessado pelo **operador de endereço** ou **operador de ponteiro** (**&**)
- Ex.:

```
int x;
scanf("%d", &x);
printf("O valor de x: %d", x);
printf("O endereço de x: %p", &x);
printf("O endereço em decimal: %d", &x);
```

%p é um formato especial para ponteiros, normalmente exibido em hexadecimal

Ponteiros

- A linguagem C permite armazenar e manipular endereços de memória através dos chamados **ponteiros**
- Um ponteiro é uma variável especial que armazena um endereço de memória
- Dizemos que uma variável comum é uma referência direta a um valor, e um ponteiro é uma referência indireta a esse valor
- Usar ponteiros é essencial para gerenciar o uso de memória dos seus programas

Declaração

- Declarar um ponteiro em C é semelhante a uma variável comum, mas com um asterisco (*), chamado de **operador de referência indireta** ou **operador de desreferenciamento**, antes do nome.
- Ex.:
  - `int *p;`
  - `float *pont_x, *pont_y;`
  - `char *frase;`

Declaração

- O tipo identifica o tipo do valor para o qual ele aponta
- Podemos vincular um ponteiro a uma variável pela operação de atribuição passando um endereço de memória para ele
- Ex.:

```
int x;
int *p_x = &x;
float num;
float *p_num = &num;
```

Declaração

- Cuidado ao declarar vários ponteiros na mesma linha!
- Ex.:

```
int x, y, z; //  3 variáveis comuns
int *p, q, r; //  1 ponteiro e 2 variáveis comuns
int *p, *q, *r; //  3 ponteiros
```

Declaração

- Cuidado ao declarar vários ponteiros na mesma linha!
- Ex.:

```
int x, y, z; //  3 variáveis comuns
int *p, q, r; //  1 ponteiro e 2 variáveis comuns
int *p, *q, *r; //  3 ponteiros
```

- Outras opções...

```
int *p = &x, *q = &y, *r = &z;
int x, *p = &x;
```

Acessando o valor apontado

- Usando ponteiros podemos acessar ou manipular valores diretamente pelo endereço de memória
- Ex.:

```
int x = 10;
int *p = &x;
printf("o valor de x: %d\n", x);
printf("o endereço de x %p\n", &x);
printf("o valor de p: %p\n", p);
printf("o valor apontado por p: %d\n", *p);
```

Acessando o valor apontado

- Não defina um valor diretamente para um ponteiro
- Você não sabe se você tem permissão de acesso a este endereço de memória, nem sabe qual o tipo de informação ele está guardando
- Ex.:

```
int *p = 254;
printf("o valor apontado por p: %d\n", *p);
```

provavelmente este código vai dar um erro de execução por você não ter acesso a este endereço de memória ou por este endereço não estar armazenando um inteiro

Operações

- Atribuição
- Ex.:

```
int x=10, y=5;
int *p1 = &x, *p2 = &y;

p1 = p2;
```

Operações

- Atribuição
- Ex.:

```
int x=10, y=5;
int *p1 = &x, *p2 = &y;

p1 = p2; → p1 passa a apontar para y
```

Operações

- Atribuição
- Ex.:

```
int x=10, y=5;
int *p1 = &x, *p2 = &y;

*p1 = *p2;
```

Operações

- Atribuição
- Ex.:

```
int x=10, y=5;
int *p1 = &x, *p2 = &y;
```

`*p1 = *p2;` → o valor apontado por p1 passa a ser o valor apontado por p2;

Alocação de memória

- Podemos alocar um espaço de memória livre usando a função **malloc**, da biblioteca <stdlib.h>
- Para isso passamos a quantidade de bytes para alocar
- Esta função irá procurar esta quantidade de espaço sequencial livre para uso e reservar para a execução do seu programa
- Ex.:

```
int *p1 = malloc(4), *p2 = malloc(sizeof(int));
*p1 = 2560;
*p2 = 1240;
printf("%d", *p1 - *p2);
```

Alocação de memória

- É uma boa prática de programação liberar a memória alocada anteriormente depois que ela não for mais necessária com a função **free**, também da <stdlib.h>
- Ex.:

```
int *p1 = malloc(4), *p2 = malloc(sizeof(int));
*p1 = 2560;
*p2 = 1240;
printf("%d", *p1 - *p2);
free(p1);
free(p2);
```

Alocação de memória

- Alocar e liberar memória durante o código e reutilizar ponteiros visando usar o menor espaço possível é o que chamamos de **Alocação Dinâmica de Memória**
- O uso de memória é uma das métricas usadas para se definir se um código é mais eficiente do que outro ou não

Exercício

- O que será impresso na tela?

```
int x = 10;
int *p_x = &x;
int **p_p_x = &p_x;

printf("%d\n", x);
printf("%d\n", *p_x);
printf("%d\n", **p_p_x);
```

É possível ter um ponteiro de ponteiro, também chamado de ponteiro duplo.

Exercício

- Indique os erros no código abaixo

```
int *p, i;
for (i = 0; i < 10; i++)
{
    p = malloc(8);
    scanf(“%d”, p);
    printf(“valor digitado %d”, *p);
}
```