



PROGRAMAÇÃO DE SOFTWARE BÁSICO

## Aula 07

# PONTEIROS NA LINGUAGEM C

Prof. Juliano Dertzbacher

# REVISÃO

- char = 1 byte
- int = 4 bytes
- float = 4 bytes
- double = 8 bytes
  
- 1 byte = 8 bits
- 1 kb = 1024 bytes
- 1 mb = 1024 kb
- 1 gb = 1024 mb
- 8 gb = 8.589.934.592 bytes ( $8 * 1024 * 1024 * 1024$ )

1	
2	
3	79
4	
5	
6	
7	
8	
9	P
...	
8.589.934.592	

# PONTEIROS

- Viabilizam o acesso a variáveis, sem referenciá-las diretamente, utilizando o endereço destas na memória.
- São usados em situações em que a passagem de valores é difícil ou indesejável.
- Fornece uma alternativa para que as funções modifiquem os parâmetros que recebem.
- Permitem a criação de estruturas de dados complexas como listas encadeadas e árvores binárias.
- Ponteiros compilam mais rapidamente tornando o código mais eficiente.

# PONTEIROS CONSTANTES E PONTEIROS VARIÁVEIS

- Ao passarmos vetores e matrizes como parâmetros para uma função não estamos passando o conteúdo destas, mas sim o endereço e, desta forma, já estamos utilizando ponteiros constantes.
- Um ponteiro variável é uma variável que aponta para outra variável, ou seja, possui um endereço de memória com a localização de uma outra variável na memória.
- Em resumo, um ponteiro constante é um endereço e um ponteiro variável é um lugar para guardar endereços.

# DECLARAÇÃO DA VARIÁVEL DO TIPO PONTEIRO

```
int main (void) {  
    int *px, *py;  
    ...  
    return 0;  
}
```

- A declaração de ponteiros tem um sentido diferente de uma variável simples. No exemplo acima, *px* e *py* são ponteiros que contêm endereços de variáveis do tipo *int*.
- No caso dos ponteiros o compilador reserva 4 (plataformas 32 bits) ou 8 (plataformas 64 bits) bytes de memória onde os endereços serão armazenados.

# EXEMPLO DE USO DO TIPO PONTEIRO

```
#include <stdio.h>

int main (void) {
    int a, b;
    a = 8;
    b = 2;
    int *p_i = NULL; //declaração de ponteiro

    p_i = &a; //inicialização de ponteiro
    *p_i = 1000; //acesso através de ponteiro, atribuindo um novo valor para "a"
    p_i = &b; //ponteiro passa a apontar para outra variável
    *p_i = 1234; //acesso através de ponteiro, atribuindo um novo valor para "b"
    printf ("a = %d\n", a);
    printf ("b = %d\n", b);
    printf ("sizeof (p_i) = %ld\n", sizeof(p_i)); //tamanho em bytes
    printf ("p_i = %p\n", p_i); //endereço de memória atual
    return 0;
}
```

# PASSANDO ENDEREÇOS PARA A FUNÇÃO

```
#include <stdio.h>
```

```
void change (int *p_i) { /*p_i recebe o endereço do parâmetro  
    *p_i = 2; //com este é possível alterar definitivamente e não na cópia  
}
```

```
int main (void) {  
    int a = 1;  
    printf ("a (antes da chamada) = %d\n", a);  
    change (&a); //a variável 'a' é efetivamente alterada dentro da função  
    printf ("a (depois da chamada) = %d\n", a);  
    return 0;  
}
```

# PASSANDO ENDEREÇOS PARA A FUNÇÃO

```
#include <stdio.h>
```

```
//exemplo de troca de variáveis referenciadas através de seus ponteiros
```

```
void swap (int *p_x, int *p_y) {  
    int temp = *p_x;  
    *p_x = *p_y;  
    *p_y = temp;  
}
```

```
int main (void) {  
    int a = 1;  
    int b = 2;  
    printf ("antes:  a=%d b=%d\n", a, b);  
    swap (&a, &b);  
    printf ("depois: a=%d b=%d\n", a, b);  
  
    return 0;  
}
```



# OPERAÇÕES COM PONTEIROS

- Atribuição: um endereço pode ser atribuído a um ponteiro com o operador **&** junto a uma variável simples.
- Buscar o endereço do ponteiro: o operador **&** retorna a posição de memória onde o ponteiro está localizado.
  - O nome do ponteiro retorna o endereço para o qual ele aponta;
  - O operador **&** junto ao nome do ponteiro retorna o endereço do ponteiro e;
  - O operador **\*** junto ao nome do ponteiro retorna o conteúdo da variável apontada.

# OPERAÇÕES COM PONTEIROS

- Comparações: testes relacionais com  $\geq$ ,  $\leq$ ,  $>$  e  $<$  são aceitos entre ponteiros somente quando os dois operandos são ponteiros e são do mesmo tipo.

# PONTEIROS E VETORES

```
#include <stdio.h>

//exemplo da equivalência entre vetores e ponteiros quando passados como parâmetros

void func1 (int vi[], int n) {
    printf ("sizeof vi=%d vi[3]=%d\n", n, vi[3]);
}

void func2 (int *p_i, int n) {
    printf ("sizeof p_i=%d p_i[3]=%d\n", n, p_i[3]);
}

int main (void) {
    int x[] = {0, 1, 2, 3, 4, 5, 6, 7, 8};
    int t = sizeof (x) / sizeof (int);
    func1 (x, t);
    func2 (x, t);
    return 0;
}
```

# PONTEIROS E STRINGS

```
#include <stdio.h>

//exemplo de strings implementadas como vetores de caracteres e como ponteiros caracteres,
//porém, somente junto à sua declaração os vetores mantêm informação sobre seu tamanho

int main(void) {
    char s1[] = "blablablabla";
    char *s2 = "blablablabla";

    printf ("s1=%s sizeof=%ld\n", s1, sizeof(s1)); //tamanho real do vetor
    printf ("s2=%s sizeof=%ld\n", s2, sizeof(s2)); //tamanho de um ponteiro

    return 0;
}
```

# EXERCÍCIOS

01. Faça um programa que imprima o tamanho de diversas variáveis do tipo ponteiro (ponteiro para caracter, ponteiro para inteiro e ponteiro para double), utilizando o operador `sizeof()`.
02. Crie um programa que declare e inicialize três variáveis dos tipos `char`, `double` e `int`. Então crie três ponteiros, um para cada tipo. A seguir, inicialize os ponteiros para que apontem para as respectivas variáveis. Finalmente, imprima na tela o valor apontado por cada ponteiro.
03. Crie uma função `spaces()` que recebe um vetor de caracteres como uma string. Essa função deve contar o número de ocorrências de caracteres de espaço (usando a função adequada do `include ctype`) e retornar um valor inteiro com a contagem feita. Teste esta função a partir da função `main()`.
04. Copie todo o programa anterior, mas altere apenas o tipo do parâmetro da função `spaces()`, que deve ser agora um ponteiro para caracter, ou seja, `char *p_c`. É necessário modificar mais alguma coisa nesta função para que ela funcione como anteriormente? Nota: não altere a função `main()`.
05. Faça um programa que percorra o seguinte vetor de strings, e que imprima a string mais longa:  
`char *vs[] = {"jfd", "kj", "usjkhcs", "nbxh", "yt", "muoi", "x", "rexhd"};`
06. Escreva uma função chamada `limits()`, que recebe quatro parâmetros: um vetor de inteiros, o tamanho deste vetor, um ponteiro `p_min` para inteiro e um ponteiro `p_max` para inteiro. Essa função retorna `void`, e deve encontrar o menor e o maior elemento do vetor passado, e armazená-los em `p_min` e `p_max`, respectivamente. Teste esta função chamando ela a partir de `main()`.
07. Escreva uma função chamada `round_double_pointer()`, que recebe um ponteiro para `double` e altera o valor apontado. Essa função deve arredondar um valor fracionário para o inteiro mais próximo. Por exemplo, 2.3 é arredondado para 2.0, 3.7 é arredondado para 4.0, e 6.5 é arredondado para 7.0. Teste a função chamando esta a partir de `main()`, para os valores citados. Dica: utilize a função `floor()` da biblioteca matemática.