

# Programação de Computadores I

DUODÉCIMA PARTE

CAPÍTULO V

SUBPROGRAMAS  
& PONTEIROS

PARTE II – PONTEIROS

# Programação de Computadores I

prof. Marco Villaça

# FUNÇÕES E PONTEIROS EM C

- As funções vistas até agora podiam retornar apenas um valor.
- Utilizando ponteiros, é possível superar esta limitação.
- Um tipo ponteiro é aquele em que as variáveis guardam uma faixa de valores que consistem em endereços de memória.

ponteiro = 0x0A102045



ENDEREÇO	CONTEÚDO
...	...
0x0A102047	12
0x0A102046	34
0x0A102045	15

# PONTEIROS

---

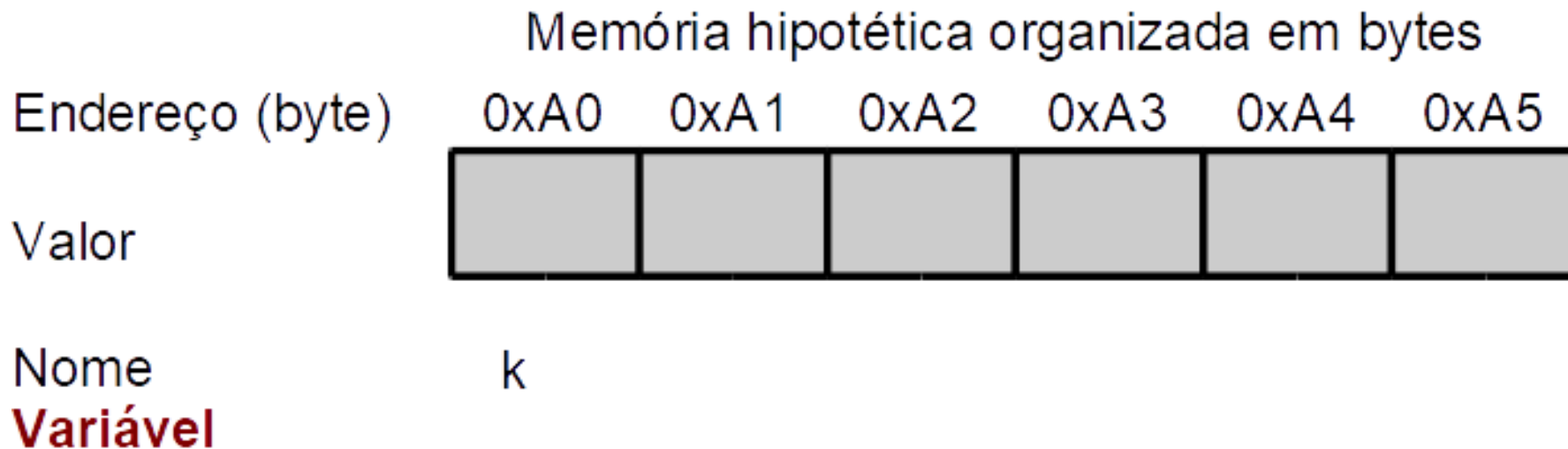
- Conceito importantíssimo em C:
  - Extremamente flexíveis, mas devem ser usados com muito cuidado;
  - Podem apontar para qualquer variável, independentemente de onde ela estiver alocada
  - Usado na alocação dinâmica de dados **e para emular mais de um retorno em funções em C.**

# PONTEIROS

## RECAPITULANDO VARIÁVEIS

---

- Variável é um espaço em memória com um nome específico e com valor que pode mudar
  - ✓ Tamanho do espaço depende do tipo da variável



# PONTEIROS

## RECAPITULANDO VARIÁVEIS

---

- Quando se declara em C

```
char k;
```

- 1 byte (8 bits) de memória é reservado (para guardar um valor inteiro)
- Uma tabela de símbolos mapeia o endereço reservado para o identificador k

# PONTEIROS

## RECAPITULANDO VARIÁVEIS

---

- Quando, no programa, define-se

$k = 2;$

- O valor 2 é colocado na porção de memória reservada para k

Endereço (byte)	0xA0	0xA1	0xA2	0xA3	0xA4	0xA5
Valor	00000010					
Nome Variável	k					

# PONTEIROS

## RECAPITULANDO VARIÁVEIS

---

- Observe que ao elemento  $k$  estão associadas duas informações
  - ✓ O próprio inteiro que está armazenado (**2**)
  - ✓ O “valor” da localização de memória (**0xA0**), ou seja, o endereço de  $k$
- Há situações em que o que se deseja armazenar é um endereço
- Ponteiro é uma variável que armazena um endereço de memória, ou seja, aponta para um endereço



# PONTEIROS

- Um ponteiro é uma variável que contém um endereço de memória
  - ✓ Esse endereço é normalmente a posição de uma outra variável na memória
- Se uma variável contém o endereço de uma outra, então a primeira variável aponta para a segunda
  - ✓ Por isso o nome ponteiro:

ponteiro = 0x0A102045



ENDEREÇO	CONTEÚDO	NOME
...	...	...
0x0A102047	12	M
0x0A102046	34	L
0x0A102045	15	K

# PONTEIROS

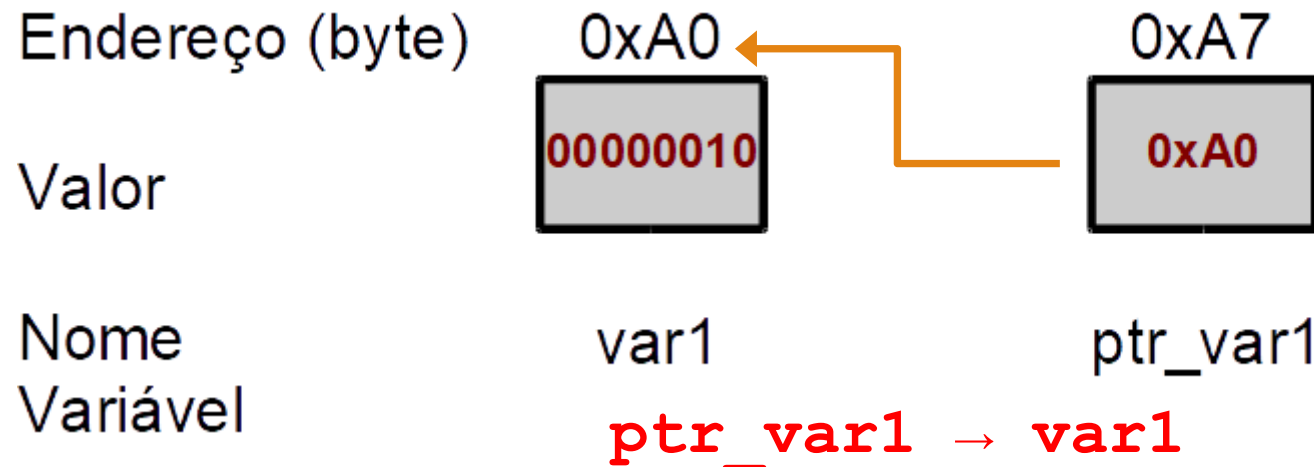
- Na figura, as caixas representam espaços em memória
  - var1 é o nome de um desses espaços
  - “2” é o valor que está armazenado nesse espaço
  - “0xA0” é o endereço desse espaço na memória

Endereço (byte)	0xA0	0xA7
Valor	00000010	0xA0
Nome Variável	var1	ptr_var1

# PONTEIROS

também são chamados de  
variáveis de endereço

- ptr\_var1 é uma outra variável
- ptr\_var1 armazena o valor (endereço) 0xA0
- Neste exemplo, é o endereço da variável var1
- Diz-se que ptr\_var1 aponta para var1



# PONTEIROS

## DECLARAÇÃO

---

- Colocar um asterisco na frente do nome da variável

```
int var1;    // declara uma variável do tipo int
```

```
int *ptr_num; //declara um ponteiro do tipo int
```

- Cuidado com declaração de mais de um ponteiro na mesma linha)

```
int  p,  q,  r;           // três variáveis comuns
```

```
int *p,  q,  r; // cuidado! só p será um ponteiro!
```

```
int *p, *q, *r;          // agora temos três ponteiros
```

# PONTEIROS

## OPERADORES

---

Operador	Significado
*	Dereference (dado um ponteiro, obtém o elemento referenciado)
&	Address_of (dado um elemento, aponta para o mesmo)

# PONTEIROS

## OPERADORES

---

- O operador **&** pode ser imaginado como “**o endereço de**”, assim como o comando abaixo significa “p recebe o endereço de count”:

- `p = &count ;`

- O operador **\*** é o complemento de **&**. É um operador unário que devolve o valor da variável localizada no endereço que o segue

- `q = *p;`

- O operador **\*** pode ser imaginado como “**valor no endereço apontado por**”, assim o comando acima significa “**q recebe o valor no endereço apontado por p.**”

# PONTEIROS

## OPERADORES

---

- Memorizar

Se

```
int *p, v;
```

```
p = &v;
```

Significa que:

**$p \rightarrow v$  ou  $*p \equiv v$**

# PONTEIROS

## INICIALIZAÇÃO

---

- Ponteiros devem ser inicializados antes de serem usados, ou seja, têm que apontar para um endereço específico antes do uso

Fazer o seguinte levará a uma falha de segmentação

```
int *p; /*ponteiro não inicializado*/
```

```
*p = 9;
```

```
/*o endereço físico para guardar o número 9 pode  
não ser válido ou permitido*/
```



# PONTEIROS

## RESUMO ATÉ AQUI

---

- Até aqui, vimos o que são ponteiros e como operá-los
- Vamos ver, agora, uma aplicação importante em C
  - ✓ Funções com mais de um retorno através do uso de ponteiros

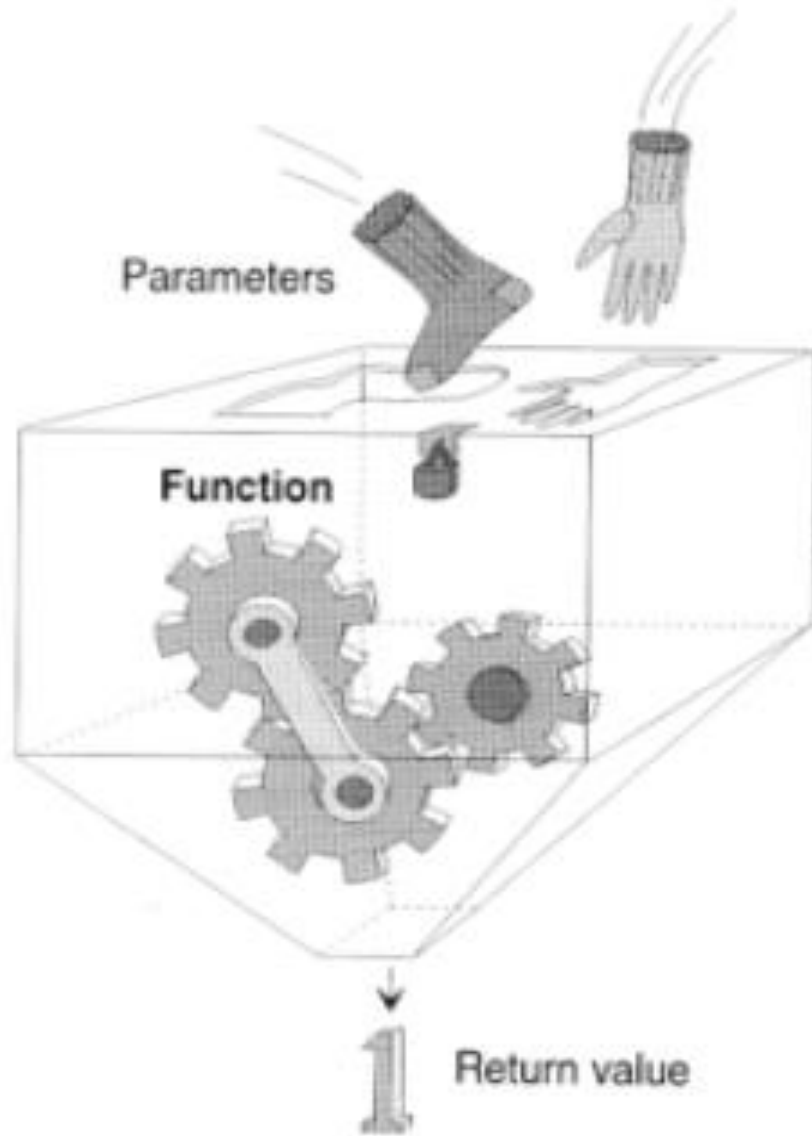
# PONTEIROS

## COMO ARGUMENTOS DE FUNÇÕES EM C

---

- Em C, parâmetros são passados para uma função através de uma chamada por valor
- Valor do argumento é copiado para dentro da função

```
c = celsius(f);           /* Chamada da função */
printf("Celsius = %.2f\n", c);
return 0; }               ARGUMENTO: VALOR PASSADO PARA A FUNÇÃO
/* Definição da função */
float celsius(float fahr) {
    float c;
    c = (fahr - 32.0) * 5.0/9.0;
    return c; }           RETORNO: VALOR DEVOLVIDO PELA FUNÇÃO
```



# PONTEIROS COMO ARGUMENTOS DE FUNÇÕES EM C

NA CHAMADA POR  
VALOR O CAMINHO SÓ  
TEM UMA DIREÇÃO

# PONTEIROS

## COMO ARGUMENTOS DE FUNÇÕES EM C

---

- Mas usando ponteiros, é possível alterar o próprio parâmetro de entrada
- Assim, função poderá emular a existência de mais de um retorno
- Passagem por referência
- Vamos ver um exemplo prático

# PROGRAMA PARA TROCAR O VALOR DE DUAS VARIÁVEIS

---

```
#include <stdio.h>
int main() {
    int a = 5, b = 10, temp;
    printf ("%d %d\n", a, b);
    temp = a;
    a = b;
    b = temp;
    printf ("%d %d\n", a, b);
    return 0;
}
```

# PROGRAMA PARA TROCAR O VALOR DE DUAS VARIÁVEIS

---

- E se essa operação tiver que ser repetida várias vezes? Como colocá-la em uma função?
- Verifique se o código seguinte faria isso.

# PROGRAMA PARA TROCAR O VALOR DE DUAS VARIÁVEIS

---

```
#include <stdio.h>
void swap(int , int);
int main()
{
    int a, b;
    a = 5;
    b = 10;
    printf ("%d %d\n", a, b);
    swap (a, b);
    printf ("%d %d\n", a, b);
    return 0;
}
```

```
void swap(int i, int j)
{
    int temp;
    temp = i;
    i = j;
    j = temp;
}
```

# PROGRAMA PARA TROCAR O VALOR DE DUAS VARIÁVEIS COMO FAZER COM PONTEIROS

---

```
#include <stdio.h>
void swap(int * , int* );
int main()
{
    int a, b;
    a = 5;
    b = 10;
    printf ("%d %d\n", a, b);
    swap (&a, &b);
    printf ("%d %d\n", a, b);
    return 0;
}
```

```
void swap(int *i, int *j)
{
    int temp;
    temp = *i;
    *i = *j;
    *j = temp;
}
```



# EXEMPLO 1

Fazer **uma** função que receba o raio de um círculo e devolva o comprimento da circunferência e a área.

Não use variáveis globais.

Use ponteiros

int main() pergunta o raio para o usuário,  
chama a função e depois imprime o resultado

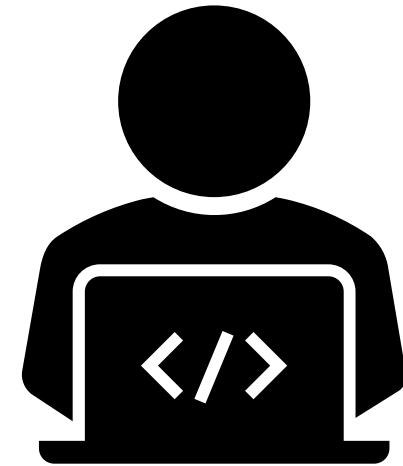


Fazer **uma** função que receba dois resistores e devolva o valor de sua associação em série e o valor de sua associação em paralelo

Não use variáveis globais.

Use ponteiros

int main() pergunta o valor de R1 e R2 para o usuário, chama a função e depois imprime os resultados Rs e Rp.



## Exercício 1

CAPÍTULO V

SUBPROGRAMAS  
& PONTEIROS

PARTE II – PONTEIROS  
CONTINUAÇÃO

# Programação de Computadores I

prof. Marco Villaça

# PONTEIROS COMO ARGUMENTOS DE FUNÇÕES EM C

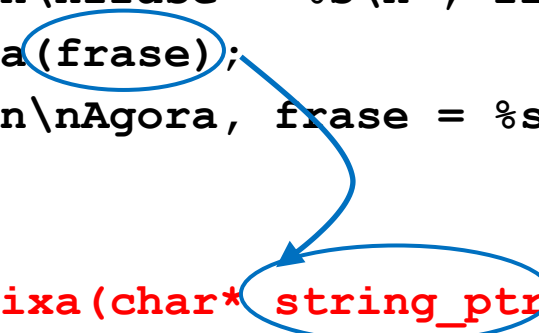
---

- Além de permitir que mais de um valor retorne, a passagem por referência tem outra vantagem.
- Não há cópia dos valores para dentro da função
- Mais rápido e eficiente
- Principalmente para manipulação de vetores e estruturas
- Observe o próximo exemplo:

# PONTEIROS COMO ARGUMENTOS DE FUNÇÕES EM C

```
#include <stdio.h>
#include <ctype.h>
void troca_caixa(char*);
int main () {
    char frase[150]="Esta e uma string grande...";
    printf ("\n\nfrase = %s\n", frase);
    troca_caixa(frase); // frase = &frase[0]
    printf ("\n\nAgora, frase = %s\n", frase);
    return 0;
}

void troca_caixa(char* string_ptr) {
    while(*string_ptr!='\0') {
        *string_ptr = toupper(*string_ptr);
        string_ptr++;
    }
}
```



0x0A102...	'\0'	Frase[149]
...	...	...
0x0A102047	12	frase[2]
0x0A102046	34	frase[1]
0x0A102045	15	frase[0]

string\_ptr



# ARITMÉTICA DE PONTEIROS EM C

---

- A definição do tipo em ponteiros serve para que o compilador saiba quantos bytes copiar para uma posição de memória:

```
int *ptr, var1; //plataforma com int 32  
ptr = &var1;  
*ptr = 2;
```

- Neste exemplo, indica que 32 bits representando o número 2 serão copiados para a área de memória apontada
- Além disso, também serve para fazer operações aritméticas com ponteiros

# ARITMÉTICA DE PONTEIROS EM C

---

```
int *ptr, var1; //plataforma com int 32
```

- Vamos imaginar que o `ptr` do exemplo anterior aponta para o endereço (em bytes) 100.000 (decimal)
- Para qual endereço aponta `(ptr+1)` ?
- Se `ptr` é do tipo `int` (4 bytes), o ponteiro irá apontar para o próximo endereço com um inteiro
- Neste caso hipotético, para o endereço 100.004

# ARITMÉTICA DE PONTEIROS EM C

---

- Os operadores

`++, --`

Exemplo: `ptr++;`

comparação: `>, <, >=, <=, ==, !=`

Exemplo: `*ptr != -1`

- São válidos com ponteiros e operam sobre os endereços



# PONTEIROS E VETORES EM C

---

- Em C, os elementos de um vetor são guardados sequencialmente, a uma “distância” fixa um do outro
- Seja o exemplo

```
char vetor[5] = {5, 10, 15, 20, 25};  
char *vetor_ptr = &vetor[0];
```

# PONTEIROS E VETORES EM C

Endereço	Conteúdo	Variável
...	...	...
0x5004	25	vetor[4]
0x5003	20	vetor[3]
0x5002	15	vetor[2]
0x5001	10	vetor[1]
0x5000	5	vetor[0]



```
char vetor[5] = {5, 10, 15, 20, 25};  
char *vetor_ptr = &vetor[0];
```

- Considerando a organização hipotética de memória ao lado, o que é obtido com:

```
✓ printf("%d", *vetor_ptr);           5  
✓ printf("%d", *(vetor_ptr+1));      10  
✓ printf("%d", (*vetor_ptr)+1);      6
```

# PONTEIROS E VETORES EM C

---

- Seja o trecho de programa em C abaixo:

```
int vetor[3] = {4, 7, 1}; *ptr;
```

```
ptr = vetor; // Em C é igual a ptr = &vetor[0]
```

- C trata ponteiros e vetores da mesma forma. Assim, são equivalentes:

```
vetor[i]; // "acesso" padrão vetor
```

```
ptr[i];
```

```
*(vetor + i); // "acesso" padrão ponteiro
```

```
*(ptr + i);
```

# PONTEIROS E VETORES EM C

---

```
#include <stdio.h>

int main()
{
    int vetor[ ] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3};
    int *ptr;
    ptr = vetor;
    printf("Vetor[2] = %d.\n",vetor[2]); // modo vetor
    printf("Vetor[2] = %d.\n",ptr[2]);    // modo vetor
    printf("Vetor[2] = %d.\n",*(ptr+2)); // modo ponteiro
    printf("Vetor[2] = %d.",*(vetor+2)); // modo ponteiro
    return (0);
}
```

# EXEMPLO

Fazer um função que diga quantos elementos há em um vetor de inteiros positivos declarado no programa principal antes do número 0.

Use para teste o seguinte vetor:

```
int vetor[ ] = {4, 5, 8, 9, 8, 1, 0, 1, 9, 3,-1};
```

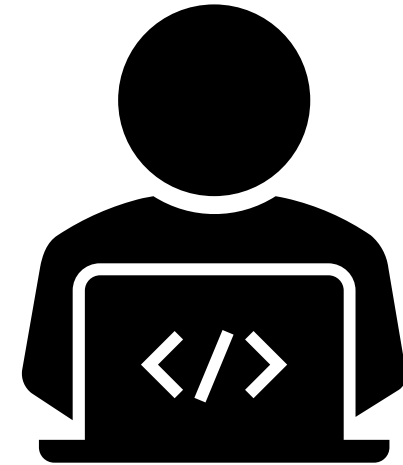


Dado um vetor de reais  $y$  informado pelo teclado pelo usuário dentro de uma faixa de -100 a +100, pede-se:

a) Criar uma função para calcular a média aritmética dos valores.

b) Calcule o desvio padrão em torno da média. Essa função deve chamar a função que calcula a média (item a):

$$s_y = \sqrt{\frac{(\sum (y_i - \bar{y})^2)}{n-1}}$$

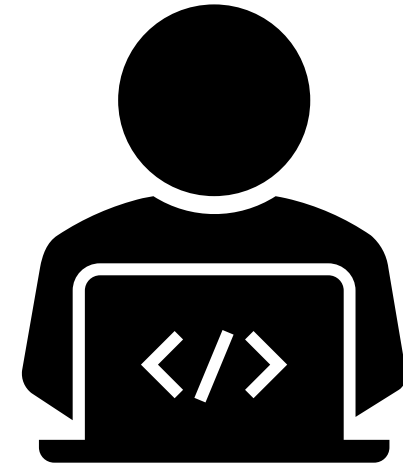


## Exercício 1

Elaborar uma função e um programa em C para testá-la que retire todos os elementos os de um vetor de inteiros de 2 bytes dentro de um intervalo especificado.

O último elemento do vetor é - 32767.

O intervalo deve ser passado para a função junto com o endereço do vetor.



## Exercício 2

# Bibliografia e crédito das figuras

---



OUALLINE, S. Practical C Programming. 3a ed. O'Reilly, 1997.



SEBESTA, R. Conceitos de Linguagens de Programação. 5a ed.  
Porto Alegre: Bookman, 2003.