

# Ponteiros

Prof. Gabriel Barbosa da Fonseca  
Email: [gbfonseca@sga.pucminas.br](mailto:gbfonseca@sga.pucminas.br)

# Introdução

Para dominar a linguagem C, é essencial dominar ponteiro. Algumas razões para o uso de ponteiros:

- Manipular elementos de **vetores** e **matrizes**;
- Receber argumentos em funções que necessitem modificar o argumento original (**parâmetro por referência**);
- Criar estruturas de dados complexas, como **listas encadeadas** e **árvores binárias**, em que um item deve conter referências a outro;
- **Alocar e desalocar** memória do sistema;
- Passar para uma função o **endereço** de outra.

# Introdução

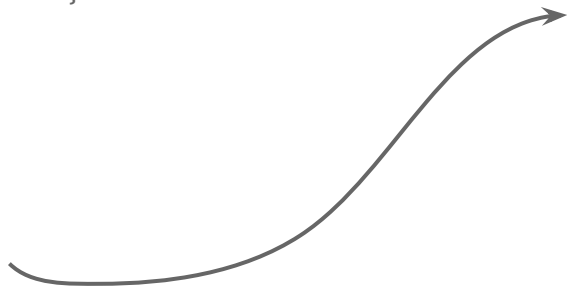
Declaração de uma variável:

- Tipo,
- Nome,
- ENDEREÇO DE MEMÓRIA!

Ex:

```
int x = 0;
```

```
float y;
```



Memória RAM		
Endereço	Variável	Valor
7ffe5367e000	x	0
7ffe5367e001		
7ffe5367e002	y	“lixo”
7ffe5367e003		
7ffe5367e004		
7ffe5367e005		

# Introdução

- Cada endereço representa o espaço de 1 byte (8 bits).
- O endereço da variável é o endereço inicial

Ex:

```
int x = 0;
```

```
float y;
```

Memória RAM		
Endereço	Variável	Valor
7ffe5367e000	x	0
7ffe5367e001		
7ffe5367e002	y	“lixo”
7ffe5367e003		
7ffe5367e004		
7ffe5367e005		

# Introdução

- Para acessarmos o endereço de uma variável, utilizamos o operador **ADDRESSOF** (endereço de), representado pelo símbolo **&**

```
...
```

```
int x = 0;
```

```
&x; //retorna o valor do endereço de x: 7ffe5367e000
```

```
...
```

# Introdução

```
#include <stdio.h>

int main(){
    int i, j, k;
    printf("%p\n%p\n%p\n", &i, &j, &k);
    return 0;
}
```

## Resultado:

```
0028FF1C
0028FF18
0028FF14
```

# Introdução

- Um ponteiro, diferentemente de uma variável comum, **contém um endereço de uma variável** que contém um valor específico
- Um ponteiro **referência um valor indiretamente**;
- Ponteiros **devem ser definidos antes de sua utilização, como qualquer outra variável**

Exemplo:

```
int count;
```

```
int *countPtr;
```

# Declaração – sintaxe

Cada ponteiro precisa ser declarado com o \* prefixado ao nome.

Exemplo: `int *xPtr, *yPtr, x;` //xPtr e yPtr são ponteiros, e x é uma variável comum

Inclua as letras **Ptr** nos nomes de variáveis de ponteiros para deixar claro que essas variáveis são ponteiros.

**Inicialize os ponteiros para evitar resultados inesperados.**



# Declaração – sintaxe

- Quando declaramos um ponteiro, ele é inicializado com o valor **NULL**
- NULL indica que o ponteiro ainda não aponta para nada
- Além do null, podemos utilizar o valor **0** para indicar um ponteiro “vazio”

```
int count;
```

```
int *countPtr = NULL;
```

# Atribuição

Exemplo:

```
int count = 5;
```

```
int *countPtr = NULL;
```

```
countPtr = &count;
```

Memória RAM		
Endereço	Variável	Valor
7ffe5367e000	count	5
7ffe5367e001		

O que temos na variável countPtr???

# Atribuição

Exemplo:

```
int count = 5;
```

```
int *countPtr = NULL;
```

```
countPtr = &count;
```

Memória RAM		
Endereço	Variável	Valor
7ffe5367e000	count	5
7ffe5367e001		

O que temos na variável countPtr???

O endereço de count: 7ffe5367e000

# Manipulação

- Como o ponteiro contém um endereço, podemos também atribuir um valor à variável guardada nesse endereço, ou seja, à variável apontada pelo ponteiro.
- Para isso, usamos o operador \* (asterisco), que basicamente significa "o valor apontado por".

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

## Saída:

```
5  5  0028FF18
```

# Exercício

```
#include <stdio.h>
int main()
{
    int x = 4, y = 7, *px, *py;
    printf("\n &x=%x e x=%d",&x,x);
    printf("\n &y=%x e y=%d",&y,y);
    printf("\n");
    px=&x;
    py=&y;
    printf("\nPX=%x e *PX=%d",px,*px);
    printf("\nPY=%x e *PY=%d",py,*py);
    printf("\n");
    return 0;
}
```

# Exercício

```
#include <stdio.h>
int main()
{
    int x = 4, y = 7, *px, *py;
    printf("\n &x=%x e x=%d",&x,x);
    printf("\n &y=%x e y=%d",&y,y);
    printf("\n");
    px=&x;
    py=&y;
    printf("\nPX=%x e *PX=%d",px,*px);
    printf("\nPY=%x e *PY=%d",py,*py);
    printf("\n");
    return 0;
}
```

Resultado:

```
&x=2811c0c8 e x=4
&y=2811c0c4 e y=7

PX=2811c0c8 e *PX=4
PY=2811c0c4 e *PY=7
```

# Manipulação

Suponhamos dois ponteiros inicializados p1 e p2.

Podemos fazer dois tipos de atribuição entre eles:

1. Esse primeiro exemplo fará com que p1 aponte para o mesmo lugar que p2. Ou seja, usar p1 será equivalente a usar p2 após essa atribuição:

**p1 = p2;**

2. Nesse segundo caso, igualamos os valores apontados pelos dois ponteiros: alteramos o valor apontado por p1 para o valor apontado por p2:

**\*p1 = \*p2;**

# Exercício

Determine o valor especificado em cada item abaixo considerando que foi executado as seguintes instruções (assuma que o endereço de x é 1000 e de y é 1004):

- (a) x
- (b) y
- (c) &x
- (d) &y
- (e) p1
- (f) p2
- (g) \*p1 + \*p2
- (h) \*(&x)
- (i) &(\*p2)

```
int x = 10, y=20;  
int* p1;  
int* p2;  
p1 = &x;  
p2 = &y;  
(*p1)++;
```



# Exercício

Determine o valor especificado em cada item abaixo considerando que foi executado as seguintes instruções (assuma que o endereço de x é 1000 e de y é 1004):

- (a)  $x \rightarrow 11$
- (b)  $y \rightarrow 20$
- (c)  $\&x \rightarrow 1000$
- (d)  $\&y \rightarrow 1004$
- (e)  $p1 \rightarrow 1000$
- (f)  $p2 \rightarrow 1004$
- (g)  $*p1 + *p2 \rightarrow 31$
- (h)  $\&(*x) \rightarrow 11$
- (i)  $\&(*p2) \rightarrow 1004$

```
int x = 10, y=20;  
int* p1;  
int* p2;  
p1 = &x;  
p2 = &y;  
(*p1)++;
```

# Passagem de parâmetros em Funções

Passagem de parâmetros por valor significa que a função trabalhará com **cópias dos valores** passados no momento de sua chamada. Para entender melhor esse processo, observe o programa a seguir.

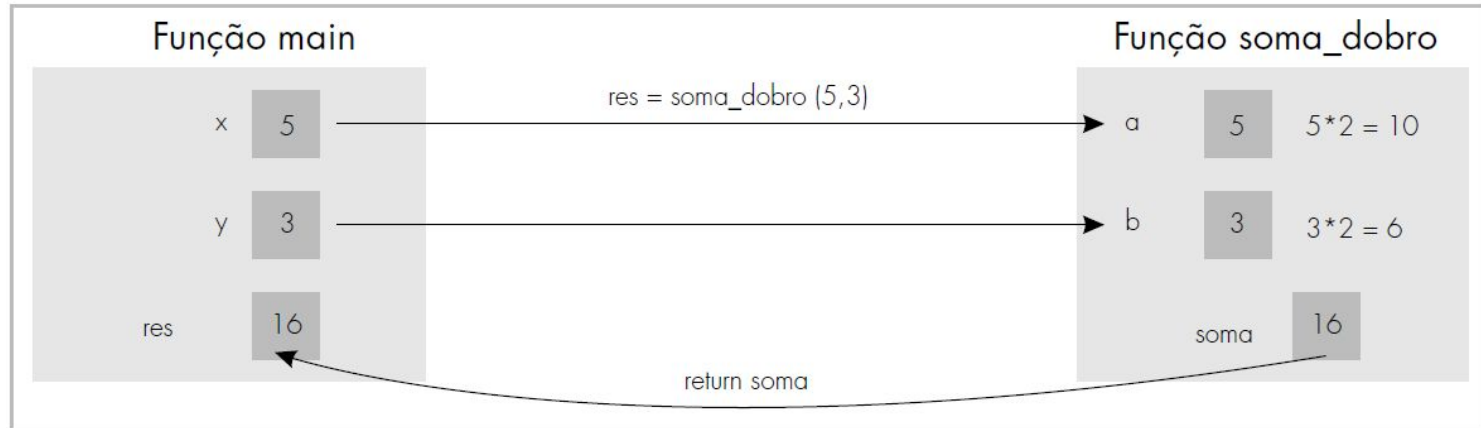
```
1. #include <stdio.h>
2. int soma_dobro(int a, int b);
3. int main()
4. {
5.     int x, y, res;
6.     x = 5;
7.     y = 3;
8.     res = soma_dobro(x,y);
9.     printf("\nA soma do dobro dos números %d e %d = %d",x,y,res);
10.    return 0; }
11. int soma_dobro(int a, int b){
12.     int soma;
13.     a = 2 * a;
14.     b = 2 * b;
15.     soma = a + b;
16.     return soma; }
```

# Passagem por VALOR

Passagem de parâmetros por valor significa que a função trabalhará com **cópias dos valores** passados no momento de sua chamada. Para entender melhor esse processo, observe o programa a seguir.

```
1. #include <stdio.h>
2. int soma_dobro(int a, int b);
3. int main()
4. {
5.     int x, y, res;
6.     x = 5;
7.     y = 3;
8.     res = soma_dobro(x,y);
```

```
9.     printf("\nA soma do dobro dos
números %d e %d = %d",x,y,res);
10.    return 0; }
11. int soma_dobro(int a, int b){
12.     int soma;
13.     a = 2 * a;
14.     b = 2 * b;
15.     soma = a + b;
16.     return soma; }
```



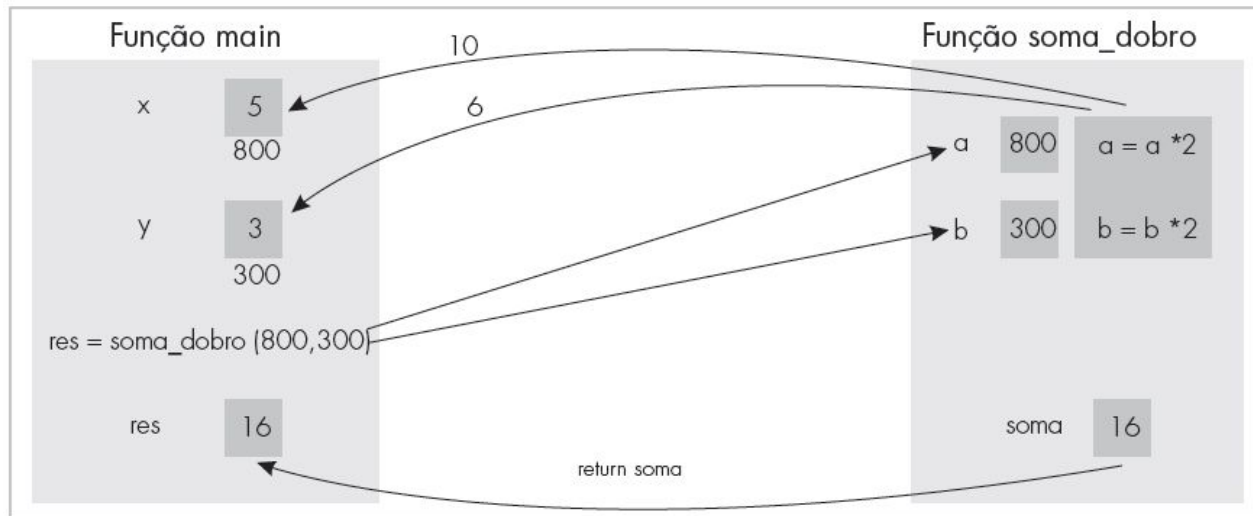
# Passagem por Referência

Passagem de parâmetros por referência significa que os parâmetros passados para uma função correspondem a **endereços de memória ocupados por variáveis**. Dessa maneira, toda vez que for necessário acessar determinado valor, **isso será feito por meio de referência**, ou seja, **apontamento ao seu endereço**.

```
1. #include <stdio.h>
2. int soma_dobro(int *a, int *b);
3. int main()
4. {
5.     int x, y, res;
6.     x = 5;
7.     y = 3;
8.     res = soma_dobro(&x,&y);
9.     printf("\nA soma dos números %d e %d = %d",x,y,res);
10.    return 0;}
11. int soma_dobro(int *a, int *b){
12.     int soma;
13.     *a = 2*(*a);
14.     *b = 2*(*b);
15.     soma = *a + *b;
16.     return soma;}
```

# Passagem por Referência

```
1. #include <stdio.h>
2. int soma_dobro(int *a, int *b);
3. int main()
4. {
5.     int x, y, res;
6.     x = 5;
7.     y = 3;
8.     res = soma_dobro(&x,&y);
9.     printf("\nA soma dos números
%d e %d = %d",x,y,res);
10.    return 0;}
11. int soma_dobro(int *a, int
*b){
12.    int soma;
13.    *a = 2*(*a);
14.    *b = 2*(*b);
15.    soma = *a + *b;
16.    return soma;}
```



# Alocação dinâmica de memória

A área de alocação dinâmica, também chamada heap, consiste em toda memória disponível que não foi usada para outro propósito. Em outras palavras heap, é o resto da memória.


A linguagem C oferece um conjunto de funções que permitem a alocação ou a liberação dinâmica de memória: **malloc()**, **realloc()** e **free()**. As funções estão disponíveis na biblioteca “**stdlib.h**”

# Malloc

A função “**malloc**” ou “alocação em memória” em C é usada para alocar dinamicamente um único bloco de memória com o tamanho especificado. Ela retorna um ponteiro do tipo **void** que pode ser convertido em um ponteiro de qualquer tipo. A posição alocada conterá “lixo”.

Sintaxe:

```
ptr = (cast_type *) malloc (byte_size);
```



```
int *ptr;  
int n = 5;  
  
ptr = (int*)malloc(n * sizeof(int));
```

Serão alocados 20 bytes por cada inteiro ter tamanho de 4 bytes.

Se não houver espaço disponível, malloc retorna **NULL**.

# Realloc

A função de “realocação” em C realloc é usada para alterar dinamicamente uma alocação feita anteriormente com “malloc”.

A realocação de memória mantém a informação já armazenada e os blocos extras alocados serão inicializados com “lixo”. Se não houver espaço, retorna NULL.

```
int * ptr1 = (int*) malloc(5 * sizeof(int));  
  
int * ptr2 = ptr1;  
  
//recebe o primeiro ponteiro e realoca,  
//dobrando o espaço inicialmente alocado  
ptr2 = (int*) realloc(ptr2, 10 * sizeof(int));
```



# Realloc

## CUIDADOS:

Quando realloc é chamado, a localização de memória apontada por **ambos os ponteiros pode ser desalocada** (no caso de o espaço contíguo não estar disponível logo após o bloco de memória). **ptr2** agora apontará para o local recém-deslocado no heap (retornado por realloc), mas **ptr1** ainda está apontando para o local antigo (que agora é desalocado).

```
int * ptr1 = (int*) malloc(5 * sizeof(int));
```

```
int * ptr2 = ptr1;
```

```
//recebe o primeiro ponteiro e realoca,
```

```
//dobrando o espaço inicialmente alocado
```

```
ptr2 = (int*) realloc(ptr2, 10 * sizeof(int));
```

# Free()

A função **free()** em C é usada para desalocar dinamicamente a memória.

**A memória alocada com as funções malloc () e realloc () não é desalocada automaticamente.**

A função free é ajuda a reduzir o desperdício de memória ao liberá-la.

```
int *ptr;  
int n = 5;
```

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