

# Ponteiros

Prof.: Leonardo Tórtoro Pereira

[leonardop@usp.br](mailto:leonardop@usp.br)

# Ponteiros [1]

- A memória de um computador para um programa em C é como uma sucessão de células de memória, cada uma de 1 byte
- Quando quer-se guardar os dados de algo representado por mais de um byte (ex: um inteiro) ocupa-se células com endereços adjacentes
- Portanto, a célula com endereço 42 vem depois da célula 41 e antes da 43. Assim como 100 unidades antes da 142

# Ponteiros [1]

- Quando uma variável é declarada, a memória necessária para guardar seu valor é designada a um local específico na memória
  - ◆ O seu endereço
  - ◆ Isso é definido pelo ambiente em que o código é executado
    - Geralmente, o SO
  - ◆ Mas isso não nos impede de obter esse valor :)

# Ponteiros [1]

→ Como vimos em aulas anteriores, isso é possível através do símbolo `&`, conhecido como operador “endereço de”

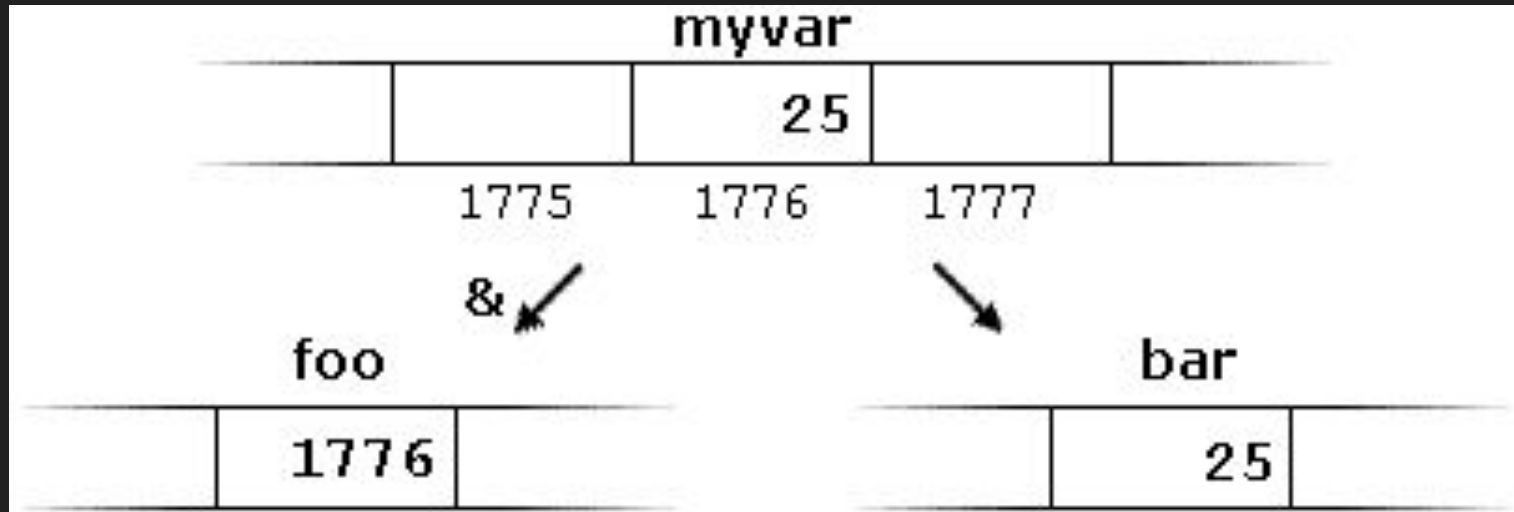
◆ `endereco = &variavel;`

→ Considerando o código abaixo:

```
myvar = 25;
```

```
foo = &myvar; //foo é um ponteiro!
```

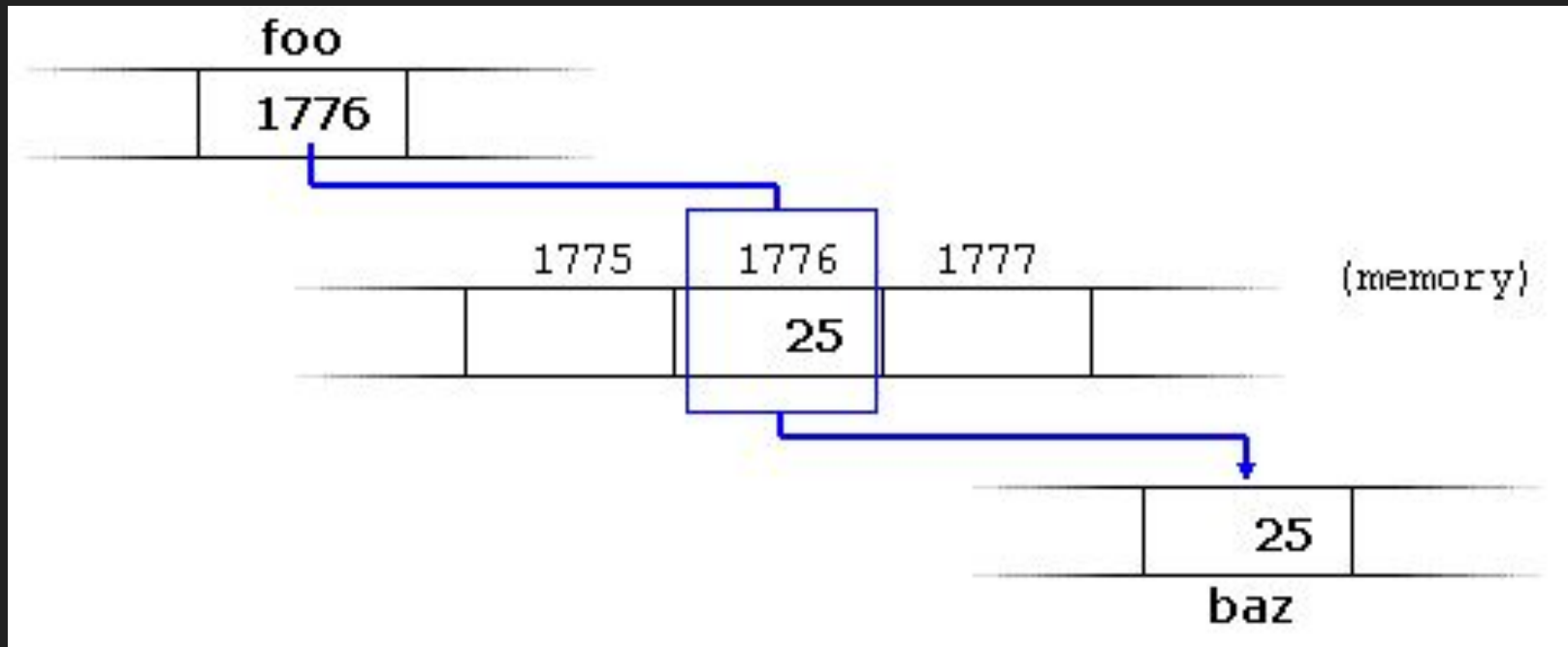
```
bar = myvar;
```



Fonte: <http://www.cplusplus.com/doc/tutorial/pointers/>

# Ponteiros [1]

- Ponteiros são variáveis que “apontam” para a variável das quais eles guardam o endereço
- É possível acessar diretamente a variável pras quais os ponteiros apontam
  - ◆ Usa-se o operador de “desreferência” \*
    - Lê-se: “valor apontado por”
- No exemplo anterior:  
`baz = *foo;`



Fonte: <http://www.cplusplus.com/doc/tutorial/pointers/>

# Ponteiros [1]

- Como um ponteiro pode apontar diretamente para o valor que ele aponta para, ele tem diferentes propriedades dependendo do tipo do valor que ele aponta para
  - ◆ int, char, float, etc.
- Independente disso, eles ocupam o mesmo tamanho na memória!
  - ◆ Esse valor depende da plataforma e de quantos bytes é preciso para endereçar qualquer célula de memória



# Ponteiros [1]

```
int main() {  
    int firstvalue, secondvalue;  
    int * mypointer;  
    mypointer = &firstvalue;  
    *mypointer = 10;  
    mypointer = &secondvalue;  
    *mypointer = 20;  
    printf("firstvalue is %d\n", firstvalue);  
    printf("secondvalue is %d\n", secondvalue);  
    return 0;  
}
```

# Ponteiros [1]

- É importante notar que o \* usado para declarar *mypointer* como um ponteiro não é o operador de “desreferência”
  - ◆ É parte do especificador de tipo da linguagem C
  - ◆ Eles apenas usam o mesmo símbolo!

# Ponteiros [1]

```
int main(){
    int firstvalue = 5, secondvalue = 15;
    int * p1, * p2;
    p1 = &firstvalue; // p1 = address of firstvalue
    p2 = &secondvalue; // p2 = address of secondvalue
    *p1 = 10;          // value pointed to by p1 = 10
    *p2 = *p1;          // value pointed to by p2 = value pointed to by p1
    p1 = p2;           // p1 = p2 (value of pointer is copied)
    *p1 = 20;          // value pointed to by p1 = 20

    printf("firstvalue is %d\n", firstvalue);
    printf("secondvalue is %d\n", secondvalue);
    return 0;
}
```

# Ponteiro vs Vetor

# Ponteiros [1]

- Os conceitos de vetor e ponteiro são relacionados
  - ◆ Vetores funcionam basicamente como ponteiros para seus elementos iniciais
  - ◆ E também podem ser implicitamente convertidos para um ponteiro do tipo adequado
  - ◆ Suportam os mesmos conjuntos de operações

`a[5] = 0;                    // a [offset of 5] = 0`

`*(a+5) = 0;                // pointed to by (a+5) = 0`

# Ponteiros [1]

```
int main ()
{
    int numbers[5];
    int * p;
    p = numbers;  *p = 10;
    p++;  *p = 20;
    p = &numbers[2];  *p = 30;
    p = numbers + 3;  *p = 40;
    p = numbers;  *(p+4) = 50;
    for (int n=0; n<5; n++)
        printf("%d, ", numbers[n]);
    return 0;
}
```

## Ponteiros [1, 2]

- Mas um ponteiro pode receber outro endereço
  - ◆ Um vetor, não!
- O *sizeof()* de um vetor retorna o tanto de memória usado por todos os elementos do vetor
  - ◆ O do ponteiro retorna apenas quanta memória foi usada pela variável do ponteiro em si

## Ponteiros [2]

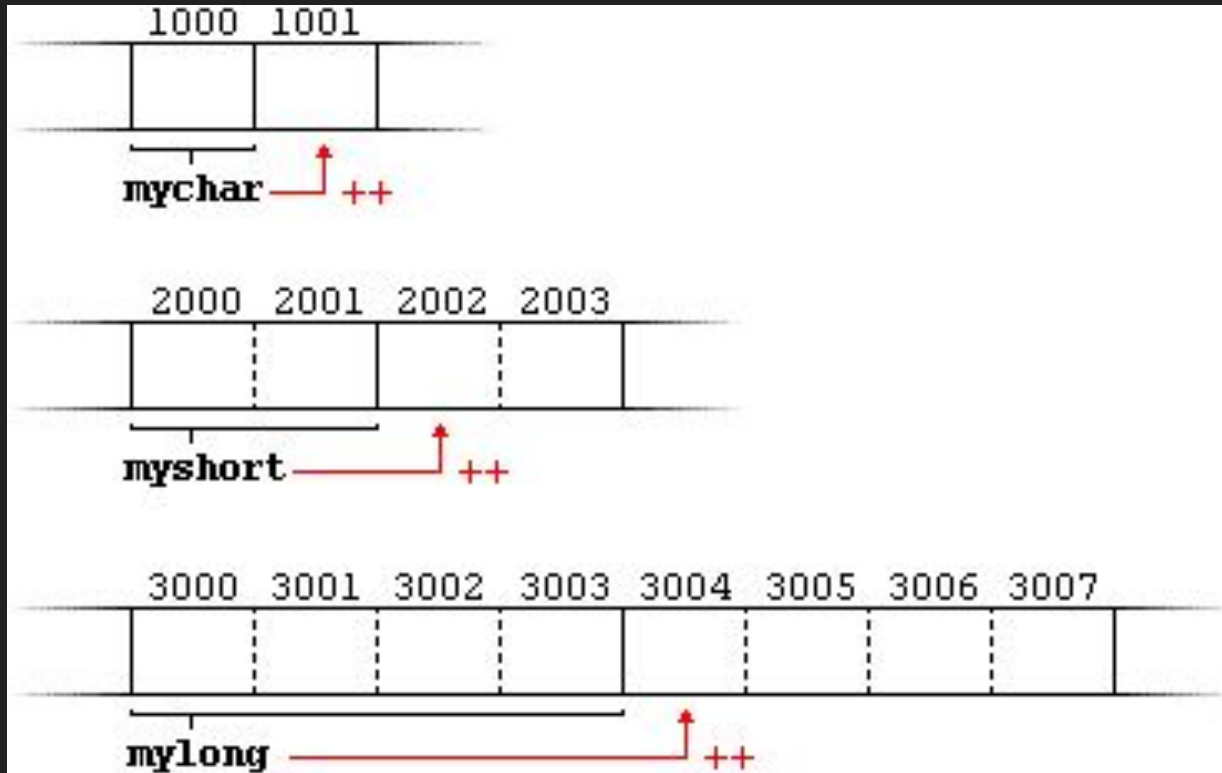
- *&array* equivale a *&array[0]* e retorna o endereço do primeiro elemento do vetor
  - ◆ *&pointer* retorna o endereço do ponteiro
- Inicializar um vetor de caracteres como *char array[] = "abc"* adiciona o *'\0'* ao final
  - ◆ Num ponteiro *char \*pointer = "abc"*, apenas atribui ao ponteiro o valor do endereço de "abc"
- É possível fazer aritmética em ponteiros (ex: soma)



# Aritmética de Ponteiros

# Ponteiros [1]

- É possível usar soma e subtração em ponteiros
  - ◆ Variam de comportamento de acordo com o tamanho do tipo de dado que o ponteiro aponta para
- Quando adicionamos/subtraímos uma unidade de um ponteiro, o que acontece é que ele aponta para o próximo/anterior elemento do mesmo tipo do ponteiro
  - ◆ Portanto, soma-se/subtrai-se o tamanho em bytes do tipo, e não uma unidade



Fonte: <http://www.cplusplus.com/doc/tutorial/pointers/>

# Ponteiros [1]

- Isso vale para qualquer valor de incremento/decremento!
- Cuidado com o uso de `*` e `++` em ponteiros:

`*p++` // same as `*(p++)`: increment pointer, and dereference unincremented address

`*++p` // same as `*(++p)`: increment pointer, and dereference incremented address

`++*p` // same as `++(*p)`: dereference pointer, and increment the value it points to

`(*p)++` // dereference pointer, and post-increment the value it points to

Ponteiro *void*

# Ponteiros [1]

- Existe um tipo especial de ponteiro em c, o tipo *void*
  - ◆ Representa a ausência de um tipo
  - ◆ Apontam para um valor que não tem tipo
    - Não tem um comprimento determinado
    - Não tem propriedades de desreferenciamento
- Portanto, são muito flexíveis
  - ◆ Podem apontar para qualquer tipo de dado
  - ◆ Mas **precisam** ser transformados nesse tipo de dado

# Ponteiros [1]

```
void increase (void* data, int psize) {  
    if ( psize == sizeof(char) ) {  
        char* pchar;  
        pchar=(char*)data;  
        ++(*pchar);  
    }  
    else if (psize == sizeof(int) ) {  
        int* pint;  
        pint=(int*)data;  
        ++(*pint);  
    }  
}
```

# Ponteiros [1]

```
int main ()
{
    char a = 'x';
    int b = 1602;
    increase (&a, sizeof(a));
    increase (&b, sizeof(b));
    printf("%c, %d\n", a , b);
    return 0;
}
```



Ponteiro *null*

# Ponteiros [1]

- Ponteiros deveriam apontar para um endereço válido
- Mas erros podem acontecer, como o uso de ponteiros não inicializados ou que apontem para elementos de um vetor não existentes
  - ◆ E desde que o valor não seja acessado, não causa um erro de compilação em C
  - ◆ Causa um acesso a um valor aleatório ou um erro em tempo de execução

# Ponteiros [1]

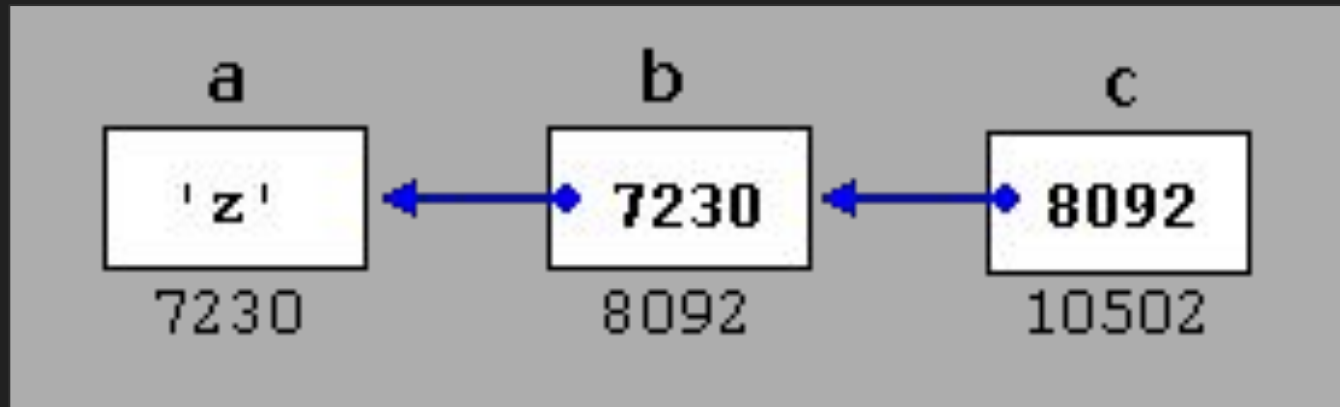
- Mas, às vezes, é preciso que um ponteiro aponte para NENHUM LUGAR
- Existe um valor especial para isso, o valor *null* de ponteiro
  - ◆ Corresponde ao valor inteiro 0
  - ◆ Ou à palavra chave *nullptr*
  - ◆ Ou à constante *NULL*
- `int *p = 0, *q = nullptr, *r = NULL;`
- Todos apontam para o nada.

Ponteiros de ponteiros

# Ponteiros [1]

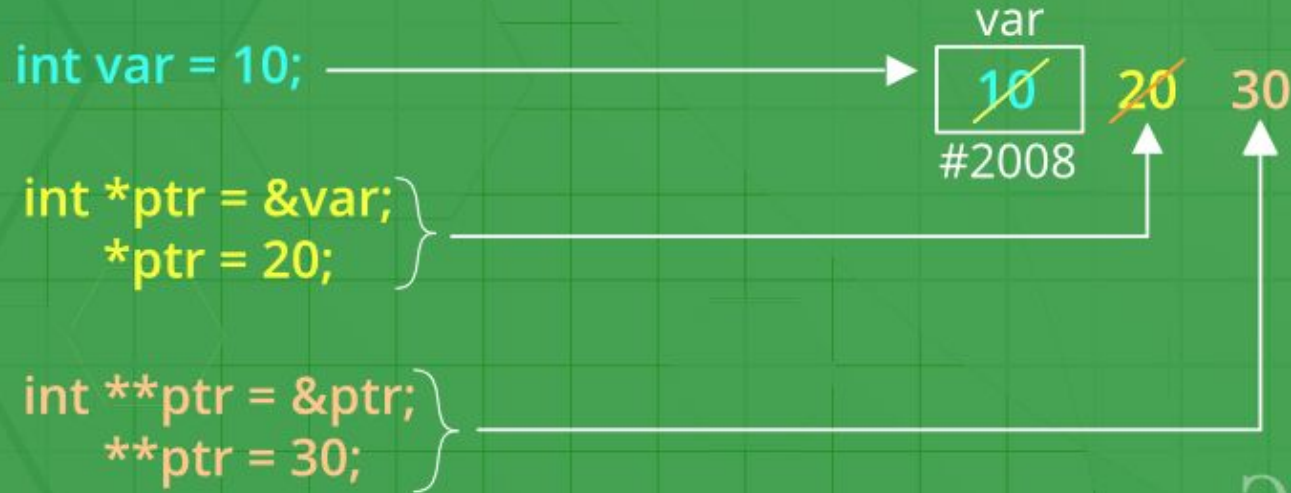
- Ponteiros podem apontar para outros ponteiros
  - ◆ E assim sucessivamente...
  - ◆ Cada novo nível requer um \* a mais na declaração

```
char a;  
char * b;  
char ** c;  
a = 'z';  
b = &a;  
c = &b;
```



Fonte: <http://www.cplusplus.com/doc/tutorial/pointers/>

# How pointer works in C



## Ponteiros [1, 3]

- No exemplo anterior temos os seguintes tipos e valores
  - ◆ *c* é um *char\*\** com valor 8092
  - ◆ *\*c* é um *char\** com valor 7230
  - ◆ *\*\*c* é um *char* com valor 'z'
- No geral, eles são equivalentes a vetores multidimensionais
- Considere a declaração seguinte:
  - ◆ `int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };`



## Ponteiros [3]

Anotação de ponteiro	Anotação de vetor	Valor
<code>**nums</code>	<code>nums[0][0]</code>	16
<code>*(*nums+1)</code>	<code>nums[0][1]</code>	18
<code>*(*nums+2)</code>	<code>nums[0][2]</code>	20
<code>*(*(nums+1)</code>	<code>nums[1][0]</code>	25
<code>*(*(nums+1)+1)</code>	<code>nums[1][1]</code>	26
<code>*(*(nums+1)+2)</code>	<code>nums[1][2]</code>	27

## Ponteiros [3]

```
int main () {  
    int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };  
    int **numsp;  
    numsp = (int**) malloc(sizeof(int*)*2);  
    for(int i = 0; i < 2; ++i)  
        numsp[i] = (int*)malloc(sizeof(int)*3);  
    printf("\nVetor:\n");  
    for(int i = 0; i < 2; ++i) {  
        for(int j = 0; j < 3; ++j) {  
            printf("%d - ", nums[i][j]);  
            *(*(numsp+i)+j) = nums[i][j];  
        }  
        printf("\n");  
    }  
}
```

## Ponteiros [3]

```
printf("\nPonteiro:\n");
for(int i = 0; i < 2; ++i) {
    for(int j = 0; j < 3; ++j) {
        printf("%d - ", (*(numsp+i)+j));
    }
    printf("\n");
}

for(int i = 0; i < 2; ++i)
    free(numsp[i]);
free(numsp);
}
```

# Referências

1. <http://www.cplusplus.com/doc/tutorial/pointers/>
2. <https://www.geeksforgeeks.org/pointer-vs-array-in-c/>
3. <https://www.geeksforgeeks.org/pointers-in-c-and-c-set-1-introduction-arithmetic-and-array/>
4. <https://www.geeksforgeeks.org/dynamic-memory-allocation-in-c-using-malloc-calloc-free-and-realloc/>