

# Ponteiros

SCC0202 - Algoritmos e Estruturas de Dados I

Prof. Fernando V. Paulovich

*\*Baseado no material do Prof. Gustavo Batista*

<http://www.icmc.usp.br/~paulovic>  
[paulovic@icmc.usp.br](mailto:paulovic@icmc.usp.br)

Instituto de Ciências Matemáticas e de Computação (ICMC)  
Universidade de São Paulo (USP)

22 de julho de 2013



# Sumário

- 1 Conceitos Introdutórios sobre Ponteiros
- 2 Operações com Ponteiros
- 3 Ponteiros e Matrizes
- 4 Ponteiros e Estruturas (structs)
- 5 Ponteiros e Alocação de Memória
- 6 Ponteiros e Passagem por Referência de Funções

# Sumário

- 1 **Conceitos Introdutórios sobre Ponteiros**
- 2 Operações com Ponteiros
- 3 Ponteiros e Matrizes
- 4 Ponteiros e Estruturas (structs)
- 5 Ponteiros e Alocação de Memória
- 6 Ponteiros e Passagem por Referência de Funções

# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória

# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória
  - Indica onde uma variável está armazenada, não o que está armazenado

# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória
  - Indica onde uma variável está armazenada, não o que está armazenado
  - Proporciona um modo de acesso a uma variável sem referenciá-la

# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória
  - Indica onde uma variável está armazenada, não o que está armazenado
  - Proporciona um modo de acesso a uma variável sem referenciá-la

# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória
  - Indica onde uma variável está armazenada, não o que está armazenado
  - Proporciona um modo de acesso a uma variável sem referenciá-la

## Por que são usados?

- Manipular elementos de uma matriz



# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória
  - Indica onde uma variável está armazenada, não o que está armazenado
  - Proporciona um modo de acesso a uma variável sem referenciá-la

## Por que são usados?

- Manipular elementos de uma matriz
- Receber argumentos em funções que necessitem modificar o argumento original

# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória
  - Indica onde uma variável está armazenada, não o que está armazenado
  - Proporciona um modo de acesso a uma variável sem referenciá-la

## Por que são usados?

- Manipular elementos de uma matriz
- Receber argumentos em funções que necessitem modificar o argumento original
- Criar estruturas de dados complexas

# Introdução

## O que são ponteiros?

- Um ponteiro é um endereço de memória
  - Indica onde uma variável está armazenada, não o que está armazenado
  - Proporciona um modo de acesso a uma variável sem referenciá-la

## Por que são usados?

- Manipular elementos de uma matriz
- Receber argumentos em funções que necessitem modificar o argumento original
- Criar estruturas de dados complexas
- etc.

# Ponteiros Constantes e Ponteiros Variáveis

## Tipos de Ponteiros

- **Ponteiro const** não pode ter seu valor alterado (matrizes)

# Ponteiros Constantes e Ponteiros Variáveis

## Tipos de Ponteiros

- **Ponteiro constante** não pode ter seu valor alterado (matrizes)
- **Ponteiro variável** é um tipo especial de variável que contém o endereço de outra variável

# Ponteiros Constantes e Ponteiros Variáveis

## Tipos de Ponteiros

- **Ponteiro const** não pode ter seu valor alterado (matrizes)
- **Ponteiro variável** é um tipo especial de variável que contém o endereço de outra variável
  - Dizemos que uma variável aponta para outra variável quando a primeira contém o endereço da segunda

# Ponteiros Constantes e Ponteiros Variáveis

## Tipos de Ponteiros

- **Ponteiro constante** não pode ter seu valor alterado (matrizes)
- **Ponteiro variável** é um tipo especial de variável que contém o endereço de outra variável
  - Dizemos que uma variável aponta para outra variável quando a primeira contém o endereço da segunda

# Ponteiros Constantes e Ponteiros Variáveis

## Tipos de Ponteiros

- **Ponteiro constante** não pode ter seu valor alterado (matrizes)
- **Ponteiro variável** é um tipo especial de variável que contém o endereço de outra variável
  - Dizemos que uma variável aponta para outra variável quando a primeira contém o endereço da segunda

## Observação

- Um ponteiro constante é um endereço



# Ponteiros Constantes e Ponteiros Variáveis

## Tipos de Ponteiros

- **Ponteiro constante** não pode ter seu valor alterado (matrizes)
- **Ponteiro variável** é um tipo especial de variável que contém o endereço de outra variável
  - Dizemos que uma variável aponta para outra variável quando a primeira contém o endereço da segunda


## Observação

- Um ponteiro constante é um endereço
- Um ponteiro variável é um lugar para guardar endereços

# Endereços de Memória e Variáveis

- Um endereço é a referência que o computador usa para localizar variáveis
- Toda variável ocupa uma certa localização na memória, e seu **endereço** é o **primeiro byte** ocupado por ela

Endereço	Conteúdo
0xFF7	
0xFF6	A
0xFF5	
0xFF4	B
0xFF3	
0xFF2	
0xFF1	
0xFF0	



# Endereços de Memória e Variáveis

- Um endereço é a referência que o computador usa para localizar variáveis
- Toda variável ocupa uma certa localização na memória, e seu **endereço** é o **primeiro byte** ocupado por ela

Endereço	Conteúdo
0xFF7	
0xFF6	A
0xFF5	
0xFF4	B
0xFF3	
0xFF2	
0xFF1	
0xFF0	

- Inteiro (int) ocupa 2 bytes na memória
- Número de ponto flutuante (float) ocupa 4 bytes na memória

# Operador de Endereços &

- Para conhecer o endereço ocupado por uma variável usamos o operador de endereços (&)

```
1  int main() {  
2      int i, j, k;  
3      printf("Endereço de i -> %u \n", &i);  
4      printf("Endereço de j -> %u \n", &j);  
5      printf("Endereço de k -> %u \n", &k);  
6      return 0;  
7  }
```

## SAÍDA

Endereço de i -> 0xFF4

Endereço de j -> 0xFF2

Endereço de k -> 0xFF0

# Operador de Endereços &

- O operador de endereços (&) só pode ser usado com nomes de variáveis.

```
1 &(i+1) //ERRADO  
2 &5 //ERRADO
```

# Declarando Variável Ponteiro

- Endereços são armazenados de forma semelhante a outras variáveis

# Declarando Variável Ponteiro

- Endereços são armazenados de forma semelhante a outras variáveis
- É necessário saber qual o tamanho que ocupa a variável apontada por um ponteiro

# Declarando Variável Ponteiro

- Endereços são armazenados de forma semelhante a outras variáveis
- É necessário saber qual o tamanho que ocupa a variável apontada por um ponteiro
  - O tipo da variável apontada (portanto seu tamanho) é fornecido na declaração do ponteiro



# Declarando Variável Ponteiro

- Endereços são armazenados de forma semelhante a outras variáveis
- É necessário saber qual o tamanho que ocupa a variável apontada por um ponteiro
  - O tipo da variável apontada (portanto seu tamanho) é fornecido na declaração do ponteiro

# Declarando Variável Ponteiro

- Endereços são armazenados de forma semelhante a outras variáveis
- É necessário saber qual o tamanho que ocupa a variável apontada por um ponteiro
  - O tipo da variável apontada (portanto seu tamanho) é fornecido na declaração do ponteiro

```
1 int *px, *py;
```

# Declarando Variável Ponteiro

- Endereços são armazenados de forma semelhante a outras variáveis
- É necessário saber qual o tamanho que ocupa a variável apontada por um ponteiro
  - O tipo da variável apontada (portanto seu tamanho) é fornecido na declaração do ponteiro

```
1 int *px, *py;
```

- px e py são ponteiros que armazenam endereços de variáveis do tipo **int**

# Operador Indireto (\*)

- O operador indireto (\*) tem como operando um endereço e provê acesso ao conteúdo da variável localizada em tal endereço

# Operador Indireto (\*)

- O operador indireto (\*) tem como operando um endereço e provê acesso ao conteúdo da variável localizada em tal endereço
  - Provê acesso ao valor da variável apontada

# Operador Indireto (\*)

- O operador indireto (\*) tem como operando um endereço e provê acesso ao conteúdo da variável localizada em tal endereço
  - Provê acesso ao valor da variável apontada

# Operador Indireto (\*)

- O operador indireto (\*) tem como operando um endereço e provê acesso ao conteúdo da variável localizada em tal endereço
  - Provê acesso ao valor da variável apontada
- Ponteiros são sempre inicializados com valor **0** ou **NULL**

# Operador Indireto (\*)

- O operador indireto (\*) tem como operando um endereço e provê acesso ao conteúdo da variável localizada em tal endereço
  - Provê acesso ao valor da variável apontada
- Ponteiros são sempre inicializados com valor **0** ou **NULL**
- C garante que **NULL** não é um endereço válido



# Operador Indireto (\*)

- O operador indireto (\*) tem como operando um endereço e provê acesso ao conteúdo da variável localizada em tal endereço
  - Provê acesso ao valor da variável apontada
- Ponteiros são sempre inicializados com valor **0** ou **NULL**
- C garante que **NULL** não é um endereço válido
  - Antes de usar um ponteiro devemos atribuir algum endereço válido à ele

# Exemplo de Ponteiros

```
1 int main() {  
2     int x, y;  
3     int *px, *py;  
4  
5     x = 10;  
6     y = 20;  
7  
8     printf("Valor de x: %d \n", x);  
9     printf("Valor de y: %d \n", y);  
10  
11     px = &x;  
12     py = &y;  
13  
14     printf("Conteúdo apontado por px: %d \n", *px);  
15     printf("Conteúdo apontado por py: %d \n", *py);  
16  
17     *px = 3;  
18     *py = 5;  
19  
20     printf("Conteúdo apontado por px: %d \n", *px);  
21     printf("Conteúdo apontado por py: %d \n", *py);  
22  
23     printf("Valor de x: %d \n", x);  
24     printf("Valor de y: %d \n", y);  
25  
26     return 0;  
27 }
```

# Sumário

- 1 Conceitos Introdutórios sobre Ponteiros
- 2 Operações com Ponteiros**
- 3 Ponteiros e Matrizes
- 4 Ponteiros e Estruturas (structs)
- 5 Ponteiros e Alocação de Memória
- 6 Ponteiros e Passagem por Referência de Funções

# Operações com Ponteiros

- A linguagem C oferece cinco operações básicas que podem ser executadas com ponteiros

# Operações com Ponteiros

- A linguagem C oferece cinco operações básicas que podem ser executadas com ponteiros
  - Soma

# Operações com Ponteiros

- A linguagem C oferece cinco operações básicas que podem ser executadas com ponteiros
  - Soma
  - Subtração

# Operações com Ponteiros

- A linguagem C oferece cinco operações básicas que podem ser executadas com ponteiros
  - Soma
  - Subtração
  - Operações lógicas (<,>,==,!=)

# Operações com Ponteiros

- A linguagem C oferece cinco operações básicas que podem ser executadas com ponteiros
  - Soma
  - Subtração
  - Operações lógicas (<,>,==,!=)
  - Operador de dereferência ou indireto (\*)



# Operações com Ponteiros

- A linguagem C oferece cinco operações básicas que podem ser executadas com ponteiros
  - Soma
  - Subtração
  - Operações lógicas (<,>,,==,! =)
  - Operador de dereferência ou indireto (\*)
  - Operador de endereço (&)

# Operações com Ponteiros

- A linguagem C oferece cinco operações básicas que podem ser executadas com ponteiros
  - Soma
  - Subtração
  - Operações lógicas (<, >, ==, !=)
  - Operador de dereferência ou indireto (\*)
  - Operador de endereço (&)
- O seguinte programa ilustra tais operações

# Operações com Ponteiros

```
1 int main() {
2     int x = 5, y = 6;
3     int *px, *py;
4
5     px = &x;
6     py = &y;
7
8     if ( px < py ) printf("py - px = %u \n", py-px);
9     else printf("px - py = %u \n", px-py);
10
11     printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);
12     printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);
13
14     px++;
15     printf("px = %u, *px = %d, &px = %u\n", px, *px, &px);
16
17     py = px + 3;
18     printf("py = %u, *py = %d, &py = %u\n", py, *py, &py);
19     printf("py - px = %u\n", py - px);
20
21     return 0;
22 }
```

# Operações com Ponteiros

## SAÍDA

```
py - px = 1
```

```
px = 65488, *px = 5, &px = 65492
```

```
py = 65490, *py = 6, &py = 65494
```

```
px = 65490, *px = 6, &px = 65492
```

```
py = 65496, *py = -28, &py = 65494
```

```
py - px = 3
```

# Precedência

- Como a expressão `*lista++` é interpretada pelo compilador?

```
1 *lista++ == *(lista++), ou  
2 *lista++ == (*lista)++
```

# Precedência

- Como a expressão `*lista++` é interpretada pelo compilador?

```
1 *lista++ == *(lista++), ou  
2 *lista++ == (*lista)++
```

- Quando operadores têm a mesma precedência, como é o caso do operador indireto e do operador de incremento, eles são resolvidos por associação: da direita para esquerda ou da esquerda para a direita
- Como operadores unários são resolvidos da direita para a esquerda, temos que a expressão é interpretada como:

# Precedência

- Como a expressão `*lista++` é interpretada pelo compilador?

```
1 *lista++ == *(lista++), ou  
2 *lista++ == (*lista)++
```

- Quando operadores têm a mesma precedência, como é o caso do operador indireto e do operador de incremento, eles são resolvidos por associação: da direita para esquerda ou da esquerda para a direita
- Como operadores unários são resolvidos da direita para a esquerda, temos que a expressão é interpretada como:

```
1 *lista++ == *(lista++)
```

# Sumário

- 1 Conceitos Introdutórios sobre Ponteiros
- 2 Operações com Ponteiros
- 3 Ponteiros e Matrizes**
- 4 Ponteiros e Estruturas (structs)
- 5 Ponteiros e Alocação de Memória
- 6 Ponteiros e Passagem por Referência de Funções



# Ponteiros e Matrizes

- Existe um estreito relacionamento entre Matrizes e Ponteiros em C de forma que qualquer operação que possa ser feita com índice de matrizes pode ser feita com ponteiros

# Ponteiros e Matrizes

- Existe um estreito relacionamento entre Matrizes e Ponteiros em C de forma que qualquer operação que possa ser feita com índice de matrizes pode ser feita com ponteiros
- O nome de uma matriz é um endereço, ou seja, um ponteiro

# Ponteiros e Matrizes

- Existe um estreito relacionamento entre Matrizes e Ponteiros em C de forma que qualquer operação que possa ser feita com índice de matrizes pode ser feita com ponteiros
- O nome de uma matriz é um endereço, ou seja, um ponteiro
- Na verdade o nome da matriz é um ponteiro constante

# Ponteiros e Matrizes

- Versão usando notação de matriz

```
1 int main() {  
2     int nums[] = {92,81,70,69,58};  
3     for(int d = 0; d < 5; d++) printf("%d\n", nums[d]);  
4     return 0;  
5 }
```

- Versão usando notação de ponteiros

```
1 int main() {  
2     int nums[] = {92,81,70,69,58};  
3     for(int d = 0; d < 5; d++) printf("%d\n", *(nums+d));  
4     return 0;  
5 }
```

# Ponteiros e Matrizes

- De forma geral, temos

```
1 *(nome_matriz + índice) = nome_matriz[índice]
```

# Ponteiros e Matrizes

- O endereço de um elemento de uma matriz pode ser referenciado de duas formas
  - em notação ponteiro ( $\text{nums}+d$ )
  - em notação de matriz ( $\&\text{nums}[d]$ )

# Ponteiros e Matrizes

- O endereço de um elemento de uma matriz pode ser referenciado de duas formas
  - em notação ponteiro ( $\text{nums} + d$ )
  - em notação de matriz ( $\&\text{nums}[d]$ )
- Considerando uma matriz de inteiros, no endereço 3000

# Ponteiros e Matrizes

- O endereço de um elemento de uma matriz pode ser referenciado de duas formas
  - em notação ponteiro (`nums+d`)
  - em notação de matriz (`&nums[d]`)
- Considerando uma matriz de inteiros, no endereço 3000

```
1 int nums[] = {92,81,70,69,58}
2
3 &nums[2] == (nums+2) == 3004
4 nums[2] == *(nums+2) == 70
```



# Ponteiros e Matrizes

- Considere o seguinte código

```
1 int vetor[10];  
2  
3 for(int i=0; i < 10; i++) {  
4     *(vetor+i) = i;  
5 }
```

- Posso fazer?

```
1 *(vetor++) = i;
```

# Ponteiros e Matrizes

- Considere o seguinte código

```
1 int vetor[10];  
2  
3 for(int i=0; i < 10; i++) {  
4     *(vetor+i) = i;  
5 }
```

- Posso fazer?

```
1 *(vetor++) = i;
```

# Ponteiros e Matrizes

- Considere o seguinte código

```
1  int vetor[10];  
2  
3  for(int i=0; i < 10; i++) {  
4      *(vetor+i) = i;  
5  }
```

- Posso fazer?

```
1  *(vetor++) = i;
```

- Erro pois **vetor** é um ponteiro constate, portanto não pode ser alterado

# Ponteiros e Matrizes

## Observação

- Não se pode alterar o valor de um ponteiro constante, somente de um ponteiro variável

# Ponteiros para matrizes usando funções

- Na função `adconst()` a declaração `int *ptr` é equivalente a `int ptr[]`

```
1  int const TAM = 5;
2
3  void adconst(int *ptr, int num, int con ) {
4      for (int k = 0; k < num; k++) *(ptr) = *(ptr++) + con;
5  }
6
7  int main() {
8      int matriz[TAM] = {3,5,7,9,11};
9      int c = 10;
10
11     adconst(matriz, TAM, c);
12
13     for(int j = 0; j < TAM; j++) printf("%d", *(matriz+j));
14
15     return 0;
16 }
```

# Sumário

- 1 Conceitos Introdutórios sobre Ponteiros
- 2 Operações com Ponteiros
- 3 Ponteiros e Matrizes
- 4 Ponteiros e Estruturas (structs)**
- 5 Ponteiros e Alocação de Memória
- 6 Ponteiros e Passagem por Referência de Funções

# Ponteiros e Estruturas (structs)

- Como em qualquer outro tipo, ponteiros para estruturas podem ser definidos

```
1 typedef struct ITEM {  
2     int chave;  
3     int valor;  
4 };  
5  
6 int main() {  
7     struct ITEM item; //define uma variável  
8     struct ITEM *pitem; //define um ponteiro  
9  
10    pitem = &item; //ponteiro aponta para variável  
11  
12    (*pitem).chave = 1;  
13    (*pitem).valor = 59;  
14  
15    return 0;  
16 }
```

# Ponteiros e Estruturas (structs)

- Uma notação do tipo **(\*pitem).chave** é confusa, de forma que a linguagem C define um operador adicional (**->**) para acessar membros de estruturas através de ponteiros



# Ponteiros e Estruturas (**structs**)

- Uma notação do tipo **(\*pitem).chave** é confusa, de forma que a linguagem C define um operador adicional (**->**) para acessar membros de estruturas através de ponteiros
- O operador **->** substitui o operador **.** no caso da utilização de um ponteiro para uma estrutura (**struct**)

# Ponteiros e Estruturas (**structs**)

- Uma notação do tipo **(\*pitem).chave** é confusa, de forma que a linguagem C define um operador adicional (**->**) para acessar membros de estruturas através de ponteiros
- O operador **->** substitui o operador **.** no caso da utilização de um ponteiro para uma estrutura (**struct**)

# Ponteiros e Estruturas (structs)

- Uma notação do tipo **(\*pitem).chave** é confusa, de forma que a linguagem C define um operador adicional (**->**) para acessar membros de estruturas através de ponteiros
- O operador **->** substitui o operador **.** no caso da utilização de um ponteiro para uma estrutura (**struct**)

```
1 pitem->chave = 1; //ao invés de (*pitem).chave = 1;  
2 pitem->valor = 59; //ao invés de (*pitem).valor = 59;
```

# Sumário

- 1 Conceitos Introdutórios sobre Ponteiros
- 2 Operações com Ponteiros
- 3 Ponteiros e Matrizes
- 4 Ponteiros e Estruturas (structs)
- 5 Ponteiros e Alocação de Memória**
- 6 Ponteiros e Passagem por Referência de Funções

# Alocação de Memória

- É possível alocar memória para a qual um ponteiro aponta, ao invés de fazê-lo apontar para uma variável

# Alocação de Memória

- É possível alocar memória para a qual um ponteiro aponta, ao invés de fazê-lo apontar para uma variável
- Para tal, uma chamada ao procedimento predefinido **malloc()** deve ser feita

# Alocação de Memória

- É possível alocar memória para a qual um ponteiro aponta, ao invés de fazê-lo apontar para uma variável
- Para tal, uma chamada ao procedimento predefinido **malloc()** deve ser feita
- Uma vez alocada memória, não esquecer de liberar com **free()**

# Alocação de Memória

- É possível alocar memória para a qual um ponteiro aponta, ao invés de fazê-lo apontar para uma variável
- Para tal, uma chamada ao procedimento predefinido **malloc()** deve ser feita
- Uma vez alocada memória, não esquecer de liberar com **free()**
- Para usar **malloc()** e **free()** não esqueça de incluir **#include <stdlib.h>**



# Alocação de Memória

```
1 typedef struct ITEM {  
2     int chave;  
3     int valor;  
4 };  
5  
6 //aloca memória para um ITEM  
7 struct ITEM *n = (struct ITEM *)malloc(sizeof(struct ITEM))  
8  
9 //libera memória alocada  
10 free(n);
```

# Alocação de Memória

```
1 typedef struct ITEM {  
2     int chave;  
3     int valor;  
4 };  
5  
6 //aloca memória para um ITEM  
7 struct ITEM *n = (struct ITEM *)malloc(sizeof(struct ITEM))  
8  
9 //libera memória alocada  
10 free(n);
```

- Uma chamada a **malloc()** faz duas tarefas
  - Cria uma variável dinâmica do tipo que **n** aponta, em uma área de memória chamada *heap*
  - Faz com que **n** aponte para essa variável dinâmica recém criada

# Sumário

- 1 Conceitos Introdutórios sobre Ponteiros
- 2 Operações com Ponteiros
- 3 Ponteiros e Matrizes
- 4 Ponteiros e Estruturas (structs)
- 5 Ponteiros e Alocação de Memória
- 6 Ponteiros e Passagem por Referência de Funções

# Passagem de Parâmetros

- Argumentos em C são passados para funções usando **Chamada por Valor**
  - É feita uma cópia dos argumentos passados para serem usados dentro da função
- Isso pode causar duas restrições
  - Memória e tempo de processamento extra são necessários para realizar essa cópia
  - Alterações aos argumentos são feitas localmente, não são visíveis fora da função

# Passagem de Parâmetros

- Ponteiros podem ser usados para que seja possível acessar os argumentos originais passados
  - Possibilita “retornar” valores nos argumentos das funções
  - Evita cópia de argumentos muito grandes (p.ex. *structs*)
- Esse tipo de chamada é conhecido como **Chamada por Referência**
  - Na verdade ainda são feito cópias dos ponteiros, mas essas indicam os endereços dos argumentos originais

# Passagem de Parâmetros

```
1 void altera(int *px, int *py) {  
2     *px = *px + 3;  
3     *py = *py + 5;  
4 }  
5  
6 int main() {  
7     int x, y;  
8  
9     x = 10;  
10    y = 20;  
11  
12    printf("X = %d, Y = %d", x, y);  
13  
14    altera(&x, &y);  
15  
16    printf("X = %d, Y = %d", x, y);  
17  
18    return 0;  
19 }
```