

Estrutura de Dados - I

Endereços e ponteiros

Prof. MSc. Rafael Staiger Bressan
rafael.bressan@unicesumar.edu.br



Conteúdo Programático

- **Endereços e ponteiros**
 - Conceitos;
 - Endereços e ponteiros;



Endereços e ponteiros

- Os conceitos de endereço e ponteiro são fundamentais em qualquer linguagem de programação, embora fiquem ocultos em algumas linguagens. Em C, esses conceitos são explícitos.
- O conceito de ponteiro não é fácil; é preciso fazer algum esforço para dominá-lo.



Endereços

- A memória RAM de qualquer computador é uma sequência de bytes.
- Cada byte armazena um de 256 possíveis valores.
- Os bytes são numerados sequencialmente.
 - O número de um byte é o seu *endereço* (= *address*).



Endereços

- Cada objeto na memória do computador ocupa um certo número de bytes consecutivos.
- Um ***char*** ocupa 1 byte.
- Um ***int*** ocupa 4 bytes em alguns computadores e 8 em outros (o número exato é dado pela expressão sizeof (int)).



Endereços

- Um ***double*** ocupa usualmente 8 bytes (o número exato é dado pela expressão `sizeof (double)`).



Endereços

- Cada objeto na memória tem um **endereço**.
- Na maioria dos computadores, o endereço de um objeto é o endereço do seu primeiro **byte**. Por exemplo, depois das declarações

```
char c;  
int i;  
struct {  
    int x, y;  
} ponto;  
int v[4];
```



Endereços

- Os endereços das variáveis poderiam ser:

<code>c</code>	<code>89421</code>
<code>i</code>	<code>89422</code>
<code>ponto</code>	<code>89426</code>
<code>v[0]</code>	<code>89434</code>
<code>v[1]</code>	<code>89438</code>
<code>v[2]</code>	<code>89442</code>



Endereços

- (os endereços são fictícios).
- O endereço de uma variável é dado pelo operador `&`. (Não confunda esse uso de `&` com o operador lógico ***and***, que em C se escreve `&&`.) Se `i` é uma variável então `&i` é o seu endereço.
- No exemplo acima, `&i` vale **89422** e `&v[3]` vale **89446**.



Endereços

- EXEMPLO: O segundo argumento da função de biblioteca [scanf](#) é o endereço da posição na memória onde devem ser depositados os objetos lidos do dispositivo padrão de entrada:

```
int i;  
  
scanf ("%d", &i);
```



Ponteiros

- Um **ponteiro** (= apontador = pointer) é um tipo especial de variável que armazena endereços.
- Um ponteiro pode ter o valor especial NULL que não é endereço de lugar algum.
- A constante NULL está definida no arquivo-interface stdlib e seu valor é 0 na maioria dos computadores.



Ponteiros

- Se um ponteiro **p** armazena o endereço de uma variável **i**, podemos dizer **p aponta para i** ou **p é o endereço de i**. (Em termos um pouco mais abstratos, diz-se que **p** é uma referência à variável **i**.)
- Se um ponteiro **p** tem valor diferente de **NULL** então ***p** é o valor do objeto apontado por p. (Não confunda esse uso de ***** com o operador de multiplicação!)

Ponteiros

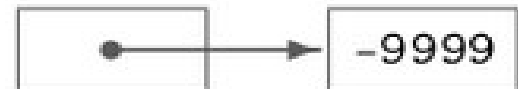
- Por exemplo, se **i** é uma variável e **p** é **&i** então dizer ***p** é o mesmo que dizer **i**.

89422

90001

-9999

89422



p



Ponteiros

- Há vários tipos de ponteiros:
 - ponteiros para caracteres, ponteiros para inteiros, ponteiros para ponteiros para inteiros, ponteiros para registros etc.
- O computador precisa saber de que tipo de ponteiro você está falando.



Ponteiros

- Para declarar um ponteiro p para um inteiro, diga **int *p**;
- Para declarar um ponteiro p para um registro cel, diga **struct cel *p**;
- Um ponteiro r para um ponteiro que apontará um inteiro é declarado assim: **int **r**;

Exemplos

- Suponha que a, b e c são variáveis inteiras. Eis um jeito bobo de fazer $c = a+b$:

```
int *p;  
int *q;  
p = &a;  
q = &b;  
c = *p + *q;
```

```
/* p é um ponteiro para um inteiro */  
  
/* o valor de p é o endereço de a */  
/* q aponta para b */
```




Exemplos

- Suponha que precisamos de uma função que troque os valores de duas variáveis inteiras, digamos i e j. É claro que a função

```
void troca (int i, int j) /* errado! */  
{  
    int temp;  
    temp = i; i = j; j = temp;  
}
```



Exemplos

- Não produz o efeito desejado, pois recebe apenas os valores das variáveis e não as variáveis propriamente ditas.
- A função recebe cópias das variáveis e troca os valores dessas cópias, enquanto as variáveis originais permanecem inalteradas.
- Para obter o efeito desejado, é preciso passar à função os endereços das variáveis:



Exemplos

```
void troca (int *p, int *q) {  
    int temp;  
    temp = *p;  
    *p = *q;  
    *q = temp;  
}
```

- Para aplicar a função às variáveis *i* e *j* basta dizer ***troca (&i, &j);***



Exemplos

- ou ainda

```
int *p, *q;
```

```
p = &i;
```

```
q = &j;
```

```
troca (p, q);
```

Exemplos

```
1  #include <stdio.h>
2
3  void swap(int i, int j)
4  {
5      int temp;
6      temp = i;
7      i = j;
8      j = temp;
9  }
10
11 int main()
12 {
13     int a, b;
14     a = 5;
15     b = 10;
16     printf ("%d %d\n", a, b);
17     swap (a, b);
18     printf ("%d %d\n", a, b);
19     return 0;
20 }
```

Exemplos

```
1  #include <stdio.h>
2
3  void swap (int *i, int *j)
4  {
5      int temp;
6      temp = *i;
7      *i = *j;
8      *j = temp;
9  }
10
11 int main ()
12 {
13     int a, b;
14     a = 5;
15     b = 10;
16     printf ("\n\nEles valem %d, %d\n", a, b);
17     swap (&a, &b);
18     printf ("\n\nEles agora valem %d, %d\n", a, b);
19     return 0;
20 }
```

Exemplos

```
1  #include<stdio.h>
2  int cubo_valor( int );
3  int cubo_referencia( int * );
4
5  int main(){
6      int number = 5;
7      printf("\nO valor original eh: %d", number );
8      number = cubo_valor( number );
9      printf("\nO novo valor de number eh: %d", number);
10     printf("\n-----");
11     number = 5;
12     printf("\nO valor original eh: %d", number );
13     cubo_referencia( &number );
14     printf("\nO novo valor de number eh: %d", number);
15     return 0;
16 }
17 int cubo_valor( int a){
18     return a * a * a;
19 }
20 int cubo_referencia( int *aPtr ){
21     *aPtr = *aPtr * *aPtr * *aPtr;
22 }
```

Exemplos

```
1  #include <stdio.h>
2  int main ()
3  {
4      int i;
5      int vetorTeste[3] = {4, 7, 1};
6      int *ptr = vetorTeste;
7      printf("%p\n", vetorTeste);
8      printf("%p\n", ptr);
9      printf("%p\n", &ptr);
10     for (i = 0; i < 3; i++)
11     {
12         printf("O endereço do índice %d do vetor é %p\n", i, &ptr[i]);
13         printf("O valor do índice %d do vetor é %d\n", i, ptr[i]);
14     }
15     return 0;
16 }
```




Vetores e endereços

- Os elementos de qualquer vetor (= array) têm endereços consecutivos na memória do computador.
- (Na verdade, os endereços não são consecutivos, pois cada elemento do vetor pode ocupar vários bytes. Mas o compilador C acerta os detalhes internos de modo a criar a ilusão de que a diferença entre os endereços de elementos consecutivos vale 1.)

Vetores e endereços

- Na verdade, vetores "*são ponteiros*" — um uso particular dos ponteiros.

```
1  #include <stdio.h>
2  int main ()
3  {
4      int i;
5      int vetorTeste[3] = {4, 7, 1};
6      int *ptr = vetorTeste;
7      printf("%p\n", vetorTeste);
8      printf("%p\n", ptr);
9      printf("%p\n", &ptr);
10     for (i = 0; i < 3; i++)
11     {
12         printf("O endereço do índice %d do vetor é %p\n", i, &ptr[i]);
13         printf("O valor do índice %d do vetor é %d\n", i, ptr[i]);
14     }
15     return 0;
16 }
```



Vetores e endereços

- Começamos declarando um vetor com três elementos; depois, criamos um ponteiro para esse vetor. Mas repare que **não colocamos o operador de endereço** em vetorTeste; fazemos isso porque um vetor já representa um endereço, como você pode verificar pelo resultado da primeira chamada a printf().



Vetores e endereços

- Podemos usar a sintaxe $*(ptr + 1)$ para acessar o inteiro seguinte ao apontado pelo ponteiro *ptr*. Mas, se o ponteiro aponta para o vetor, o próximo inteiro na memória será o próximo elemento do vetor! De fato, em C as duas formas $*(ptr + n)$ e $ptr[n]$ são equivalentes.

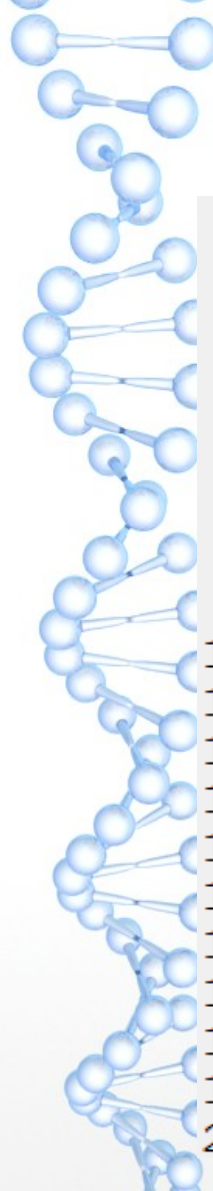


Vetores e endereços

- Não é necessário criar um ponteiro para usar essa sintaxe; como já vimos, o vetor em si já é um ponteiro, de modo que qualquer operação com *ptr* será feita igualmente com *vetorTeste*.
- Todas as formas abaixo de acessar o segundo elemento do vetor são equivalentes:

```
10  vetorTeste[1];  
11  *(vetorTeste + 1);  
12  ptr[1];  
13  *(ptr + 1)
```

Vetores e endereços



```
1  #include <stdio.h>
2  int main()
3  {
4      int numbers[5];
5      int *p;
6      int n;
7      p = numbers;
8      *p = 10;
9      p++;
10     *p = 20;
11     p = &numbers[2];
12     *p = 30;
13     p = numbers + 3;
14     *p = 40;
15     p = numbers;
16     *(p + 4) = 50;
17     for (n = 0; n < 5; n++)
18         cout << numbers[n] << ", ";
19     return 0;
20 }
```

Ele resume as várias formas de acessar elementos de um vetor usando ponteiros.

Vetores e endereços

```
1 #include <stdio.h>
2 int main ()
3 {
4     int i;
5     int vetor[10];
6     for (i = 0; i < 10; i++) {
7         printf ("Digite um valor para a posicao %d do vetor: ", i + 1);
8         scanf ("%d", &vetor[i]); //isso é equivalente a fazer *(x + i)
9     }
10    for (i = 0; i < 10; i++)
11        printf ("%d\n", vetor[i]);
12
13    return (0);
14 }
```

Vetores e endereços

```
1  #include <stdio.h>
2  int main()
3  {
4      int vetorTeste[3] = {4, 7, 1};
5      int *ptr = vetorTeste;
6      int i = 0;
7      while (ptr <= &vetorTeste[2])
8      {
9          printf("O endereço do índice %d do vetor é %p\n", i, ptr);
10         printf("O valor do índice %d do vetor é %d\n", i, *ptr);
11         ptr++;
12         i++;
13     }
14     return 0;
15 }
```


Vetores e endereços

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int x = 1;
5      int *p_x = &x;      // p_x aponta para x
6      int **p_p_x = &p_x; // p_p_x aponta para o ponteiro p_x
7      printf("%d\n", x);   // Valor da variável
8      printf("%d\n", *p_x); // Apontada por p_x
9      printf("%d\n", **p_p_x); // Apontada pelo endereço apontado por p_p_x
10     return 0;
11 }
```

Vetores e endereços

```
1  #include <stdio.h>
2  void atribuiValores(int[], int);
3  void mostraValores(int[], int);
4  int main()
5  {
6      int vetorTeste[3]; // crio um vetor sem atribuir valores
7      atribuiValores(vetorTeste, 3);
8      mostraValores(vetorTeste, 3);
9      return 0;
10 }
11 void atribuiValores(int valores[], int num)
12 {
13     for (int i = 0; i < num; i++)
14     {
15         printf("Insira valor #%d: ", i + 1);
16         scanf("%d", &valores[i]);
17     }
18 }
19 void mostraValores(int valores[], int num)
20 {
21     for (int i = 0; i < num; i++)
22     {
23         printf("Valor #%d: %d\n", i + 1, valores[i]);
24     }
25 }
```



Vetores e endereços

- Repare que passamos dois parâmetros para as funções:
- O "nome" do vetor, que representa o seu endereço na memória. (Temos 3 maneiras para passar o endereço do vetor: diretamente pelo seu "nome", via um ponteiro ou pelo endereço do primeiro elemento.)



Vetores e endereços

- Uma constante, que representa o número de elementos do vetor. Isso é importante pois o C não guarda informações sobre o tamanho dos vetores; você não deve tentar alterar ou acessar valores que não pertencem ao vetor.



Vetores e endereços

- É claro que devemos passar o endereço do vetor (por "referência"), pois os seus valores são alterados pela função `atribuiValores`. De nada adiantaria passar o vetor por valor, pois o valor só seria alterado localmente na função (como já vimos no caso de troca do valor de duas variáveis).



Vetores e endereços

- Por causa dessa equivalência entre vetores e ponteiros, podemos fazer uma pequena alteração no protótipo (tanto na declaração quanto na definição) das funções `atribuiValores` e `mostraValores`, sem precisar alterar o código interno dessas funções ou a chamada a elas dentro da função `main` ? trocando



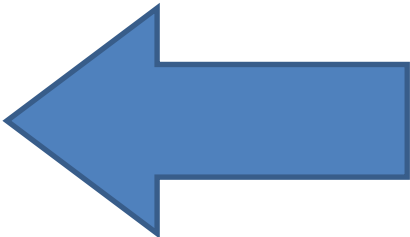
Vetores e endereços

```
void atribuiValores(int[], int);  
void  mostraValores(int[], int);
```

por

```
void atribuiValores(int*, int);  
void  mostraValores(int*, int);
```

Vetores e endereços



```
1  #include <stdio.h>
2  void atribuiValores(int*, int);
3  void mostraValores(int*, int);
4  int main()
5  {
6      int vetorTeste[3]; // crio um vetor sem atribuir valores
7      atribuiValores(vetorTeste, 3);
8      mostraValores(vetorTeste, 3);
9      return 0;
10 }
11 void atribuiValores(int valores[], int num)
12 {
13     for (int i = 0; i < num; i++)
14     {
15         printf("Insira valor #%d: ", i + 1);
16         scanf("%d", &valores[i]);
17     }
18 }
19 void mostraValores(int valores[], int num)
20 {
21     for (int i = 0; i < num; i++)
22     {
23         printf("Valor #%d: %d\n", i + 1, valores[i]);
24     }
25 }
```




Exercícios

- Por que o código abaixo está errado?
- ```
void troca (int *i, int *j) {
 int *temp;
 *temp = *i;
 *i = *j;
 *j = *temp;
}
```



# Exercícios

- Um ponteiro pode ser usado para dizer a uma função onde ela deve depositar o resultado de seus cálculos. Escreva uma função **hm** que converta minutos em horas-e-minutos. A função recebe um inteiro **mnts** e os endereços de duas variáveis inteiras, digamos h e m, e atribui valores a essas variáveis de modo que m seja menor que 60 e que  $60 * h + m$  seja igual a **mnts**. Escreva também uma função main que use a função hm.



# Exercícios

- Escreva uma função **mm** que receba um vetor inteiro  $v[0..n-1]$  e os endereços de duas variáveis inteiras, digamos **min** e **max**, e deposite nessas variáveis o valor de um elemento mínimo e o valor de um elemento máximo do vetor. Escreva também uma função **main** que use a função **mm**.



# Exercícios

- Suponha que os elementos do vetor  $v$  são do tipo `int` e cada `int` ocupa 8 bytes no seu computador. Se o endereço de  $v[0]$  é 55000, qual o valor da expressão  $v + 3$ ?



# Exercícios

O que a seguinte função faz?

```
void imprime (char *v, int n) {
 char *c;
 for (c = v; c < v + n; v++)
 printf ("%c", *c);
}
```



# Exercícios

- O que há de errado com o seguinte trecho de código?

```
char *a, *b;
```

```
a = "abacate";
```

```
b = "uva";
```

```
if (a < b)
```

```
 printf ("%s vem antes de %s no dicionário", a, b);
```

```
else
```

```
 printf ("%s vem depois de %s no dicionário", b, a);
```



# Exercícios

[http://www.ufjf.br/jairo\\_souza/files/2012/11/Lista-de-exercicios-Ponteiros.pdf](http://www.ufjf.br/jairo_souza/files/2012/11/Lista-de-exercicios-Ponteiros.pdf)

[https://www.ic.unicamp.br/~ra144681/courses/mc202/slides/files/Lista01-a\\_pointers.pdf](https://www.ic.unicamp.br/~ra144681/courses/mc202/slides/files/Lista01-a_pointers.pdf)

