

Estruturas de Dados – Aula 01

Prof. Dr. Eduardo Takeo Ueda
eduardo.tueda@sp.senac.br

Plano de Ensino da Disciplina

O plano de ensino permanecerá disponível no Blackboard, para consulta, durante todo o período letivo da disciplina!

Processo de Avaliação da Disciplina

Instrumento de avaliação	Período previsto para aplicação	Devolução
Exercícios (individual)	Semanal	1 semana depois
1ª Avaliação individual	8ª semana	1 semana depois
2ª Avaliação individual	17ª semana	1 semana depois

Composição da Nota Final da Disciplina

$$\text{Nota Final (NF)} = 0.1(E1) + 0.1(E2) + 0.8(NP)$$

onde:

E1 = Exercícios (individuais) em laboratório (Blackboard)

E2 = Exercícios (individuais) extraclasse (Blackboard) - ADOs

NP = Média aritmética das notas A1 e A2

A1: 1ª Avaliação individual

A2: 2ª Avaliação individual

Será aprovado na disciplina o aluno que obtiver Nota Final (NF) maior ou igual a 6 (seis inteiros).

Revisão de endereços e ponteiros

- Os conceitos de **endereço** e **ponteiro** são fundamentais em qualquer linguagem de programação.
- Em linguagem C, esses conceitos são **explícitos**; em algumas outras linguagens (como Java e Python) eles são **implícitos** (representados pelo conceito mais abstrato de **referência**).

Endereços

- A memória (RAM) de qualquer computador é uma sequência de bytes; os bytes são numerados sequencialmente e o número de um byte é o seu **endereço** (*address*).
- Cada variável de um programa ocupa um certo número de bytes consecutivos na memória do computador; o número exato de bytes de uma variável é dado pelo operador **sizeof**.

Endereços

- Cada variável (em particular, cada *array* e cada *struct*/registro) na memória tem um endereço; na maioria dos computadores, o endereço de uma variável é o endereço do seu primeiro byte.
- O endereço de uma variável é dado pelo operador **&**; assim, se **i** é uma variável então **&i** é seu endereço.
- É muito comum imprimir endereços em notação **hexadecimal**, usando formato **%p**, que exige o *cast* **(void *)**.

Endereços

```
1  #include <stdio.h>
2
3  int main(void) {
4
5      char c = 'A';
6      int i = 42;
7
8      printf("Valor de c (caractere): %c\n", c);
9      printf("Valor de i (inteiro): %d\n", i);
10
11     printf("Tamanho de c: %lu byte(s)\n", sizeof(c));
12     printf("Tamanho de i: %lu byte(s)\n", sizeof(i));
13
14     printf("Endereço (em hexadecimal) de c: %p\n", (void *) &c);
15     printf("Endereço (em hexadecimal) de i: %p\n", (void *) &i);
16
17     printf("Endereço (em decimal) de c: %ld\n", (long int) &c);
18     printf("Endereço (em decimal) de i: %ld\n", (long int) &i);
19
20     return 0;
21 }
```


Endereços

```
Valor de c (caractere): A
Valor de i (inteiro): 42
Tamanho de c: 1 byte(s)
Tamanho de i: 4 byte(s)
Endereço (em hexadecimal) de c: 0x7ffe1926bd13
Endereço (em hexadecimal) de i: 0x7ffe1926bd14
Endereço (em decimal) de c: 140729320389907
Endereço (em decimal) de i: 140729320389908
```

Ponteiros

- Um **ponteiro** (*pointer*) é um tipo especial de variável que armazena um endereço; um ponteiro pode ter o valor **NULL**, que é um endereço “inválido”.
- A macro **NULL** está definida na interface **stdlib.h** e seu valor é **0 (zero)** na maioria dos computadores.
- Se um ponteiro **p** armazena o endereço de uma variável **i**, podemos dizer “**p aponta para i**” ou “**p é o endereço de i**”.

Ponteiros

- Se um ponteiro **p** tem valor diferente de **NULL**, então ***p** é o **valor** da variável apontada por **p**; por exemplo, se **i** é uma variável e **p** é igual a **&i** então temos que “***p**” é o mesmo que “**i**”.



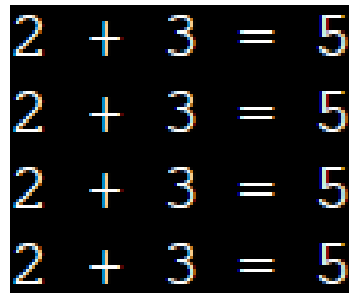
Declaração de ponteiros

- Existem vários tipos de ponteiros: ponteiros para caracteres, ponteiros para inteiros, ponteiros para structs/registros, etc; o computador precisa saber de que tipo de ponteiro estamos falando.
- Declaração de um ponteiro para um inteiro:
`int *p;`
- Declaração de um ponteiro para um struct/registro:
`struct registro *p;`

Declaração de ponteiros

```
1  #include <stdio.h>
2
3  int main(void) {
4      int a = 2, b = 3, c;
5
6      int *p; // p é um ponteiro para um inteiro
7      int *q; // q é um ponteiro para um inteiro
8
9      p = &a; // o valor de p é o endereço de a
10     q = &b; // q aponta para b
11
12     c = *p + *q; // c = a + b
13
14     printf("%d + %d = %d\n", a, b, a + b);
15     printf("%d + %d = %d\n", a, b, c);
16     printf("%d + %d = %d\n", *p, *q, c);
17     printf("%d + %d = %d\n", *p, *q, *p + *q);
18
19     return 0;
20 }
```

Declaração de ponteiros

A 4x4 grid of the text "2 + 3 = 5". Each character in the grid has a glitch or double-exposure effect, appearing slightly offset or semi-transparent, creating a digital or corrupted visual style. The background of the grid is black, and the text is white with a yellowish-orange glow.

Ponteiros e funções

- Suponha que precisamos de uma função para trocar os valores de duas variáveis inteiras, digamos *i* e *j*.
- Neste caso, uma função que recebe “cópias” das variáveis e troca os valores dessas cópias não resolve, pois as variáveis originais permanecem inalteradas.
- Para obtermos o efeito desejado, é preciso passar para a função os **endereços** das variáveis.

Passagem de parâmetros por valor

```
1  #include <stdio.h>
2
3  void troca (int i, int j) {
4      int tmp;
5
6      tmp = i;
7      i = j;
8      j = tmp;
9  }
10
11 int main(void) {
12
13     int a = 3, b = 7;
14
15     printf("Antes da troca : a = %d, b = %d\n", a, b);
16
17     troca(a, b);
18
19     printf("Depois da troca: a = %d, b = %d\n", a, b);
20
21     return 0;
22 }
```


Passagem de parâmetros por valor

```
Antes da troca : a = 3, b = 7  
Depois da troca: a = 3, b = 7
```

Passagens de parâmetros por referência

```
1  #include <stdio.h>
2
3  void troca (int *i, int *j) {
4      int tmp;
5
6      tmp = *i;
7      *i = *j;
8      *j = tmp;
9  }
10
11 int main(void) {
12
13     int a = 3, b = 7;
14
15     printf("Antes da troca : a = %d, b = %d\n", a, b);
16
17     troca(&a, &b);
18
19     printf("Depois da troca: a = %d, b = %d\n", a, b);
20
21     return 0;
22 }
```

Passagens de parâmetros por referência

```
Antes da troca : a = 3, b = 7  
Depois da troca: a = 7, b = 3
```

Passagens de parâmetros por referência

```
1  #include <stdio.h>
2
3  void troca (int *i, int *j) {
4      int tmp;
5
6      tmp = *i;
7      *i = *j;
8      *j = tmp;
9  }
10
11 int main(void) {
12
13     int a = 3, b = 7;
14     int *p = &a, *q = &b;
15
16     printf("Antes da troca : a = %d, b = %d\n", *p, *q);
17
18     troca(p, q);
19
20     printf("Depois da troca: a = %d, b = %d\n", a, b);
21
22     return 0;
23 }
```

Passagens de parâmetros por referência

```
Antes da troca : a = 3, b = 7  
Depois da troca: a = 7, b = 3
```

Ponteiros e vetores

- Os elementos de qualquer vetor/array são alocados em bytes consecutivos na memória do computador.
- Por exemplo, depois dos comandos a seguir o ponteiro **v** aponta para o primeiro elemento de um **vetor/array** de 10 inteiros.

```
int *v;  
v = malloc (10 * sizeof (int));
```

Ponteiros e vetores

- O endereço do segundo elemento do vetor é $v+1$, o endereço do terceiro é $v+2$, e assim por diante; de tal forma que se i é uma variável do tipo `int` então $v+i$ é o endereço do $(i+1)$ -ésimo elemento do vetor/array.
- As expressões $v+i$ e $\&v[i]$ têm exatamente o mesmo valor e, portanto as atribuições $*(v+i) = 42$ e $v[i] = 42$ tem o mesmo efeito.

Ponteiros e vetores

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(void) {
5
6      int *v; // v é um ponteiro para um inteiro
7      v = malloc (10 * sizeof (int)); // aloca 10 inteiros
8      // as duas linhas acima são equivalentes a int v[10];
9
10     int i;
11
12     for (i = 0; i < 3; i++) {
13         printf("Digite o valor de v[%d]: ", i);
14         scanf ("%d", v + i); // v + i <=> &v[i]
15     }
16
17     for (i = 0; i < 3; i++)
18         printf("v[%d]: %d\n", i, *(v+i)); // *(v+i) <=> v[i]
19
20     return 0;
21 }
```


Ponteiros e vetores

```
Digite o valor de v[0]: 1
Digite o valor de v[1]: 2
Digite o valor de v[2]: 3
v[0]: 1
v[1]: 2
v[2]: 3
```

Ponteiros para ponteiros

- Ponteiros para ponteiros têm várias vantagens em programação, principalmente quando trabalhamos com **alocação dinâmica**.
- Por exemplo, permite criar matrizes de tamanho variável dinamicamente. Como uma matriz é essencialmente um *array* de *arrays*, cada linha pode ser um ponteiro para um *array*, acessado por um ponteiro para ponteiros.

Ponteiros para ponteiros

```
1  #include <stdio.h>
2
3  int main(void) {
4      int i = 42;    // variável inteira
5      int *p = &i;   // ponteiro que aponta para 'i'
6      int **q = &p;  // ponteiro para ponteiro, que aponta para 'p'
7
8      // Imprimindo os valores e endereços
9      printf("Valor de i: %d\n", i);
10     printf("Valor de i via p: %d\n", *p);
11     printf("Valor de i via q: %d\n", **q);
12
13     printf("\nEndereço de i: %p\n", (void *)&i);
14     printf("Endereço de p: %p\n", (void *)&p);
15     printf("Endereço armazenado em q: %p\n", (void *)q);
16
17     return 0;
18 }
```

Ponteiros para ponteiros

```
Valor de i: 42  
Valor de i via p: 42  
Valor de i via q: 42  
  
Endereço de i: 0x7ffddf629d74  
Endereço de p: 0x7ffddf629d78  
Endereço armazenado em q: 0x7ffddf629d78
```

Ponteiros para ponteiros

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5      int linhas = 2, colunas = 3;
6
7      // declaração do ponteiro para ponteiro
8      int **matriz;
9
10     // alocação de memória para as linhas da matriz
11     matriz = (int **) malloc(linhas * sizeof(int *));
12
13     // alocação de memória para cada coluna, em cada linha
14     for (int i = 0; i < linhas; i++)
15         matriz[i] = (int *) malloc(colunas * sizeof(int));
16
```

Ponteiros para ponteiros

```
17     printf("Preenchendo a matriz:\n");
18     for (int i = 0; i < linhas; i++)
19     {
20         for (int j = 0; j < colunas; j++) {
21             matriz[i][j] = i + j; // soma dos índices
22             printf("Matriz[%d][%d] = %d\n", i, j, matriz[i][j]);
23         }
24
25         // Liberando a memória alocada
26         for (int i = 0; i < colunas; i++)
27             free(matriz[i]);
28     }
29     free(matriz);
30     return 0;
31 }
```

Ponteiros para ponteiros

```
Preenchendo a matriz:  
Matriz[0][0] = 0  
Matriz[0][1] = 1  
Matriz[0][2] = 2  
Matriz[1][0] = 1  
Matriz[1][1] = 2  
Matriz[1][2] = 3
```

Fim!