



# LINGUAGEM DE PROGRAMAÇÃO

AULA 2



Prof. Sandro de Araujo



## CONVERSA INICIAL

Esta aula tem como base os livros *Fundamentos da Programação de Computadores: Algoritmos, Pascal, C/C++* e *Treinamento em Linguagem C*. Em caso de dúvidas ou para aprofundamento, consulte-os em nossa Biblioteca Virtual Pearson.

A aula apresenta a seguinte estrutura de conteúdo:

1. Endereços de memória;
2. Ponteiros;
3. Ponteiros – tamanho e endereçamento;
4. Ponteiros e vetores;
5. Passagem de parâmetros por referência.

O objetivo desta aula é conhecer o conceito de ponteiro e sua aplicação em algoritmos computacionais; entender como um dado é acessado na memória; e sua relação com vetores e funções.

## TEMA 1 – ENDEREÇOS DE MEMÓRIA

A memória de um computador é dividida em *bytes*, numerados de zero até o limite de memória da máquina. Esses números são chamados endereços de *bytes*, usados como referências, pelo computador, para localizar as variáveis (Mizrahi, 2008).

Toda variável tem uma localização na memória, e o endereço de identificação desta variável é o primeiro *byte* ocupado por ela, conforme explicado anteriormente.

Se o programa contém a informação somente do endereço do primeiro *byte* da variável, como ele sabe quais endereços deve ler? Bom, para obter esta resposta, o programa deve saber que toda variável está armazenada em *bytes* sequenciais e, identificando o tamanho desta variável pelo seu tipo, infere até onde a leitura dos endereços deve ir.

Por exemplo, uma variável do tipo *int* em C com tamanho 4 *bytes*, ou seja, sabendo o endereço inicial, sabe que, a partir, dele temos mais três endereços sequenciais que correspondem à variável desejada.

Quando o programa é carregado na memória, ocupa certa quantidade de *bytes*, e toda variável e função desse programa terão seu espaço e endereço



particular. Para conhecer o endereço em que uma variável está alocada, usa-se o operador de endereços **&**. Observe a Figura 1, que mostra um algoritmo que vai imprimir os endereços de três variáveis usando a função **printf()** nas linhas 9, 10 e 11.

Figura 1 – Algoritmo que imprime o endereço de três variáveis

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main() {
5
6      int x, y, z;
7
8      /* %p para ponteiros */
9      printf("O endereço de x: %p \n", &x);
10     printf("O endereço de y: %p \n", &y);
11     printf("O endereço de z: %p \n\n", &z);
12
13     system("pause");
14     return 0;
15 }
```

A Figura 2 mostra a saída do algoritmo acima após a sua execução.

Figura 2 – Saída do algoritmo

```
O endereço de x: 000000000062FE4C
O endereço de y: 000000000062FE48
O endereço de z: 000000000062FE44

Pressione qualquer tecla para continuar. . .
```

Quando se define uma variável como ponteiro, dizemos que o endereço de uma variável simples está guardado em um ponteiro que pode ser utilizado como parâmetro para uma função. Para que isso ocorra, basta colocar o



operador “\*” antes da variável e o operador “&”<sup>1</sup> na chamada do parâmetro (Mizrahi, 2008).

O resultado do operador &, "endereço de", sempre será o endereço de memória do elemento em questão, normalmente é o local onde uma variável está alocada na memória. Isto é, esse operador gera um ponteiro.

A função scanf() espera que o usuário digite algum dado de entrada e o operador ‘&’, acompanhado da variável, serve para especificar o lugar certo onde esse dado vai ficar posicionado na memória. Portanto, o uso do operador de endereço para essa função se faz necessário (Ascencio, 2012).

O operador oposto é o \* (asterisco), que pega o valor apontado pelo endereço.

No exemplo do algoritmo da Figura 1, foram declaradas três variáveis inteiras. Obtivemos como saída a impressão dos seus endereços em hexadecimal com o uso do operador &. O endereço alocado depende de vários fatores, dentre eles, o tamanho da palavra<sup>2</sup>, se há ou não outros programas usando a memória, entre outros.

Por essas razões, podemos encontrar endereços diferentes na passagem de parâmetros e execução do algoritmo (Mizrahi, 2008) para cada nova execução de um problema. Faça o teste você mesmo. Implemente o exemplo anterior no seu compilador e mande-o executar diversas vezes. Cada nova execução gerará valores diferentes de endereços alocados.

Mizrahi (2008) descreve memória como uma unidade organizada logicamente em palavras. Uma palavra é uma unidade lógica de informação constituída por um número de *bits* de único endereço, conseqüentemente, um conjunto de palavras armazenadas na memória é um programa, e pode ser dividido em duas categorias:

- Instruções – operações (programa propriamente dito) realizadas pela máquina;
- Dados – variáveis, ou valores, processadas nessas operações.

Cada palavra é identificada por meio de um endereço de memória sem ambiguidade. Observe a Tabela 1.

---

<sup>1</sup> O operador unário & retorna o endereço na memória de seu operando (Mizrahi, 2008).

<sup>2</sup> Unidade de informação para cada tipo de computador (Mizrahi, 2008).



Tabela 1 – Exemplo de endereços de palavras

Ordem na memória	Endereço na memória	Palavras
0	000	Palavra 0
1	001	Palavra 1
2	010	Palavra 2
3	011	Palavra 3
4	100	Palavra 4
5	101	Palavra 5
6	110	Palavra 6
7	111	Palavra 7

A capacidade, ou tamanho, de uma memória vai depender do número de palavras que ela pode suportar. A posição de uma palavra dentro da memória é tida como o seu endereço. A primeira palavra da memória tem o endereço 000, a próxima, 001, e assim por diante (Mizrahi, 2008).

## TEMA 2 – PONTEIROS

O ponteiro é uma ferramenta poderosa oferecida em linguagens de programação e considerada, pela maioria dos programadores, um dos tópicos mais difíceis (Mizrahi, 2008; Ascencio, 2012).

Apontadores, ou ponteiros, são variáveis que armazenam o endereço de outras variáveis na memória. Ou seja, em vez de termos um valor numérico ou caracteres, por exemplo, armazenado na variável, temos um endereço. Dizemos que **um ponteiro “aponta” para uma variável na memória quando este contém o endereço daquela variável.**

O uso de ponteiros é muito útil quando um dado deve ser acessado na memória em diferentes partes de um programa. Assim, podem existir vários ponteiros espalhados, indicando a localidade da variável que contém o dado desejado. Caso este dado seja atualizado, todas as partes que apontam para a variável serão atualizados simultaneamente (Ascencio, 2012).

De acordo com Mizrahi (2008), estas são algumas razões para o uso de ponteiros:

1. Fornecem maneiras com as quais as funções podem realmente modificar os argumentos que recebem (passagem de parâmetros por referência);
2. Criar estruturas de dados complexas, como listas encadeadas e árvores binárias, em que um item deve conter referências a outro;
3. Alocar e desalocar memória dinamicamente do sistema;
4. Passar para uma função o endereço de outra função.



A sintaxe de declaração de um ponteiro é:

**tipo \*nome\_ponteiro;**

Em que temos:

- **tipo** – refere-se ao tipo de dado da variável armazenada que é apontada pelo endereço do ponteiro;
- **\*nome\_ponteiro** – o nome da variável ponteiro;
- O uso do asterisco \* serve para determinar que a variável usada será um ponteiro.

Um ponteiro, como qualquer variável, deve ser tipificado, que é a identificação do tipo da variável para a qual ele aponta. Para declarar um ponteiro, especifica-se o tipo da variável para a qual ele aponta com o nome precedido por asterisco. Exemplo:

```
int ponteiro;      // declaração de uma variável comum do tipo inteiro
int *ponteiro;     // declaração de um ponteiro para um inteiro
```

É importante prestar bastante atenção na hora de declarar vários ponteiros em uma linha, visto que o asterisco deve vir antes de cada nome de variável. Exemplos:

```
int x, y, z;       // Essa instrução declara três variáveis comuns.
int *x, y, z;      // Essa instrução declara somente x como ponteiro.
int *x, *y, *z;    // Essa instrução declara três ponteiros.
```

Um ponteiro é uma variável que armazena um endereço de memória, a localização de outra variável. Dizemos que uma variável aponta para outra quando a primeira contém o endereço da segunda. Observe os exemplos mostrados nas figuras 3 e 4.

Figura 3 – Exemplo de ponteiros

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main ()
5  {
6      int *y, x;
7
8
9      printf( "Digite um numero: ");
10     scanf("%d", &x);
11
12     y = &x; // y recebe o endereço de x
13
14     // Imprime o dado que y aponta
15     printf("\n Voce digitou o numero: %d \n", y);
16
17     system ("pause");
18     return 0;
19 }
20
```



A Figura 4 mostra a saída do algoritmo acima após a sua execução.

Figura 4 – Saída do algoritmo

```
Digite um numero: 6582
Voce digitou o numero: 6582
Pressione qualquer tecla para continuar. . .
```

Figura 5 – Exemplo de ponteiros

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      int x = 4 , y = 7 ; //varaveis do tipo inteiro x e y
7      int *px, *py;      //ponteiros do tipo inteiro px e py
8
9      //imprime os enderecos e os dados das variaveis x e y
10     printf ("Endereco (&x) = %p --- Dado (x) = %d \n" , &x, x);
11     printf ("Endereco (&y) = %p --- Dado (y) = %d \n\n" , &y, y);
12
13     px = &x; //px recebe o endereço de x
14     py = &y; //py recebe o endereço de y
15
16     /*imprime os enderecos apontados e os dados
17     das variaveis referenciadas */
18     printf ("Endereco (px) = %p --- Dado (*px) = %d \n" , px, *px);
19     printf ("Endereco (py) = %p --- Dado (*py) = %d \n\n" , py, *py);
20
21     system ("pause");
22     return 0 ;
23 }
24
25
```

A Figura 6 mostra a saída do algoritmo acima após a sua execução.

Figura 6 – Saída do algoritmo

```
Endereco (&x) = 000000000062FE3C --- Dado (x) = 4
Endereco (&y) = 000000000062FE38 --- Dado (y) = 7

Endereco (px) = 000000000062FE3C --- Dado (*px) = 4
Endereco (py) = 000000000062FE38 --- Dado (*py) = 7

Pressione qualquer tecla para continuar. . .
```



Figura 7 – Exemplo de ponteiros

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int  main()
5  {
6      int x, y;
7      int *px = &x; // *px recebe o endereço de x
8
9      x = 14;        //x recebe o numero 14
10     y = *px;       //y = recebe o dado de x pelo endereço de *px
11     px = &y;       // px recebe o endereço de y
12     x = 16;        //alterado o valor de x
13
14     printf ( "x = %d\n" , x);
15     printf ( "Endereco de x.....= %p\n", &x); //endereço de y
16
17     printf ( "y = %d\n" , y);
18     printf ( "Endereco de y -> px = %p\n", px); //endereço de y em px
19     printf ( "Endereco de y.....= %p\n\n", &y); //endereço de y
20
21     system ("pause");
22     return 0 ;
23 }
```

A Figura 8 mostra a saída do algoritmo acima após a sua execução.

Figura 8 – Saída do algoritmo

```
x = 16
Endereco de x.....= 000000000062FE44
y = 14
Endereco de y -> px = 000000000062FE40
Endereco de y.....= 000000000062FE40

Pressione qualquer tecla para continuar. . . _
```

### TEMA 3 – PONTEIROS: TAMANHO E ENDEREÇAMENTO

Um ponteiro também é uma variável e também ocupa espaço na memória. Normalmente, o tamanho de um ponteiro independe do tipo de dados da variável da qual está apontando e ocupa o espaço de um inteiro (Mizrahi, 2008).

Para obter o tamanho de um tipo de variável na linguagem de programação C, utiliza-se a função *sizeof*. A Figura 9 mostra um algoritmo que imprime o tamanho das variáveis mais usadas na escrita de um programa em linguagem de programação C (Mizrahi, 2008; Ascencio, 2012).





Figura 9 – Tamanho de variáveis mais usadas em C

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      printf("Tamanho de char: %i Byte\n", sizeof(char));
7      printf("Tamanho de int: %i Bytes\n", sizeof(int));
8      printf("Tamanho de float: %i Bytes\n", sizeof(float));
9      printf("Tamanho de double: %i Bytes\n", sizeof(double));
10
11     return 0;
12 }
```

A Figura 10 mostra a saída do algoritmo acima após a sua execução.

Figura 10 – Saída do algoritmo

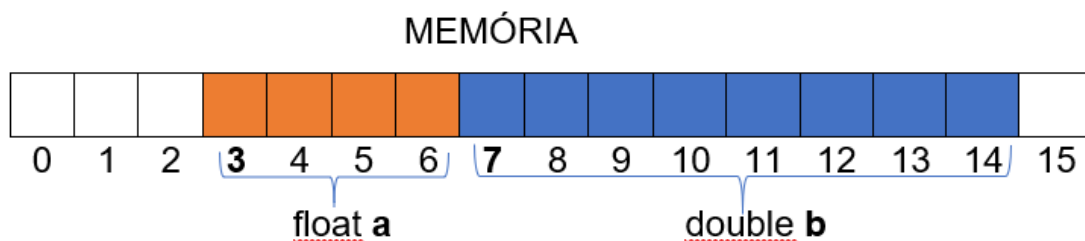
```
Tamanho de char: 1 Byte
Tamanho de int: 4 Bytes
Tamanho de float: 4 Bytes
Tamanho de double: 8 Bytes

-----
Process exited after 0.3406 seconds with return value 0
Pressione qualquer tecla para continuar. . .
```

A saída gerada, na Figura 9, após a execução da função *sizeof*, tem como retorno o tamanho dos tipos de variáveis em *bytes*.

Como exemplo, vamos considerar a declaração de duas destas variáveis listadas no algoritmo, em uma memória endereçada *byte a byte*, uma variável do tipo *float* que ocupará 4 *bytes* e outra variável do tipo *double*, que ocupará 8 *bytes*. A Figura 11 ilustra o endereço base dessas duas variáveis.

Figura 11 – Exemplo de duas variáveis na memória



No cenário da Figura 11, considera-se como endereço o menor valor na região ocupada. Sendo assim, a variável **a** terá como endereço o numeral **3**, e a variável **b**, o numeral **7**.



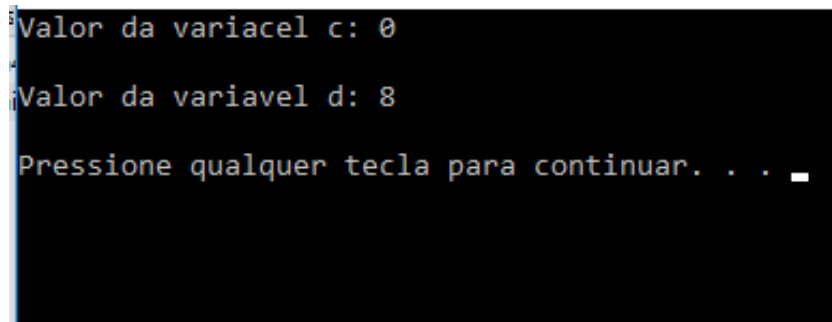
Trabalha-se com ponteiros quando existe a necessidade de ter os valores das variáveis alterados diretamente na memória mostra um. A Figura 12 mostra um algoritmo que evidencia esse conceito.

Figura 12 – Manipulação de dados com ponteiros

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6  int *x, *y, c = 5, d = 3;
7
8  x = &c;    // x aponta para c
9  y = &d;    // y aponta para d
10 *y = 8;    // alterado o valor existente na variavel d
11 *x = *y;   // copia o valor de d (opontado pory) para c (apontado por x
12 *x = 1;    // altera o valor da variavel c
13 y = x;     // y aponta para o mesmo lugar que x, ou seja, para c
14 *y = 0;    // altera o valor de c
15
16 printf("Valor da variacel c: %d\n\nValor da variavel d: %d\n\n", c, d);
17
18 system("pause");
19 return 0;
20 }
```

A Figura 13 mostra a saída do algoritmo acima após a sua execução.

Figura 13 – Saída do algoritmo



```
Valor da variacel c: 0
Valor da variavel d: 8
Pressione qualquer tecla para continuar. . . _
```

No momento em que declarou as variáveis **c** e **d**, também se fixaram os valores 5 e 3, respectivamente. Após a execução das instruções, conforme mostrado na Figura 6, seus valores foram alterados para 0 e 8 com o uso de ponteiros.

## TEMA 4 – PONTEIROS E VETORES

Vetores unidimensionais, ou *arrays*, consistem em um conjunto de dados de mesmo tipo armazenados em posições sequenciais na memória, caracterizando uma estrutura de dados homogênea. O nome do vetor é um



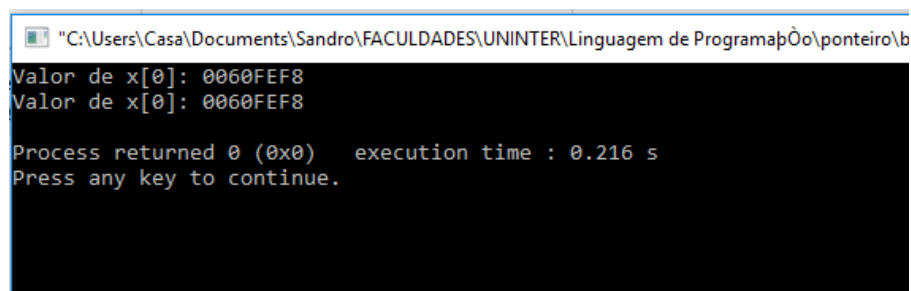
ponteiro que aponta para o primeiro elemento do vetor. Observe o algoritmo da Figura 14.

Figura 14 – Ponteiros e vetores

```
1  #include<stdio>
2  #include<stdlib>
3
4  int main()
5  {
6      int x[ ] = {2, 16, 15, 3, 10};
7      int *pont;
8
9      pont = x; //atribui o endereço do vetor
10
11     printf ("Valor de x[0]: %p\n", x);
12     printf ("Valor de x[0]: %p\n", pont);
13
14     return 0;
15 }
```

A Figura 15 mostra a saída do algoritmo acima após a sua execução.

Figura 15 – Saída do algoritmo



```
"C:\Users\Casa\Documents\Sandro\FACULDADES\UNINTER\Linguagem de Programação\ponteiro\b"
Valor de x[0]: 0060FEF8
Valor de x[0]: 0060FEF8

Process returned 0 (0x0)   execution time : 0.216 s
Press any key to continue.
```

As instruções acima são usadas para criar um ponteiro que vai apontar para o primeiro elemento do vetor **x[ ]**. A expressão “**pont = x;**” faz com que o ponteiro “**pont**” atribua o endereço do primeiro elemento do vetor **x[ ]**. A Figura 16 ilustra a atribuição do endereço ao ponteiro “**pont**”.

Figura 16 – Representação de um ponteiro com o endereço de um vetor





Para obter o endereço do primeiro elemento, basta escrever:

```
1. int x[ ] = {2, 16, 15, 3, 10};  
2. int *pont;  
3.  
4. pont = x.
```

ou

```
1. int x[ ] = {2, 16, 15, 3, 10};  
2. int *pont;  
3.  
4. pont = &x[0].
```

As instruções anteriores produzem resultados equivalentes. Logo, as instruções abaixo serão usadas de forma análoga para obter o endereço do quinto elemento:

```
1. int x[] = {2, 16, 15, 3, 10};  
2. int *pont;  
3.  
4. pont = &x[4].
```

Para obter o endereço de outro índice, é necessário utilizar o operador '&'. Observe o código acima e ilustrado na Figura 17.

Figura 17 – Ponteiro com o endereço do quinto elemento do vetor

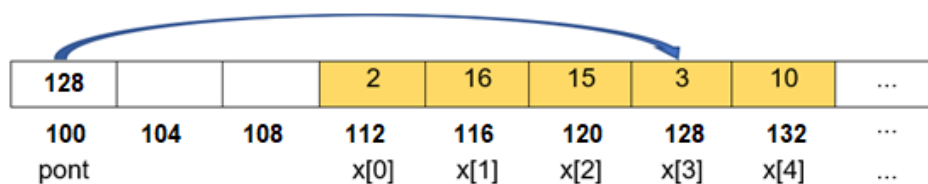




Figura 18 – Exemplo de ponteiro e vetor

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      /* O primeiro endereço de um vetor é o endereço do primeiro elemento
7       deste vetor */
8      int V[5] = {32 , 56 , 78 , 32 , 44};
9      int i;
10
11     // IMPRIME OS ELEMENTOS DO VETOR DE FORMA CONVENCIONAL
12     printf("IMPRIMINDO O VETOR DIRETAMENTE\n");
13     for (i = 0; i < 5; i++)
14         printf("Elemento do VETOR sem ponteiros [%d] = %d \n", i, V[i]);
15
16     printf ("\n\n");
17
18     // IMPRIME OS ELEMENTOS DO VETOR USANDO A NOTACÃO DE PONTEIROS
19     printf("\nIMPRIMINDO O VETOR COM A NOTACAO *(V + i)\n");
20     for (i = 0; i < 5; i++)
21         printf ("Elemento do VETOR com ponteiros [%d] = %d \n", i, *(V + i));
22
23     printf ("\n\n");
24
25     system("pause");
26     return 0;
27 }
28
```

A Figura 19 mostra a saída do algoritmo acima após a sua execução.

Figura 19 – Saída do algoritmo

```
IMPRIMINDO O VETOR DIRETAMENTE
Elemento do VETOR sem ponteiros [0] = 32
Elemento do VETOR sem ponteiros [1] = 56
Elemento do VETOR sem ponteiros [2] = 78
Elemento do VETOR sem ponteiros [3] = 32
Elemento do VETOR sem ponteiros [4] = 44

IMPRIMINDO O VETOR COM A NOTACAO *(V + i)
Elemento do VETOR com ponteiros [0] = 32
Elemento do VETOR com ponteiros [1] = 56
Elemento do VETOR com ponteiros [2] = 78
Elemento do VETOR com ponteiros [3] = 32
Elemento do VETOR com ponteiros [4] = 44

Pressione qualquer tecla para continuar. . . _
```



Figura 20 – Exemplo de ponteiro e vetor

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main (){
5
6      //Declaração de um vetor com tres posicoes
7      int vetor[3];
8      int *v, i; //Declaração de um ponteiro
9
10     v = vetor; // variavel v recebe o endereco do vetor
11
12     vetor[0] = 248; // Vetor na posicao 0 recebe 248
13     vetor[1] = 954; // Vetor na posicao 0 recebe 954
14     vetor[2] = 587; // Vetor na posicao 0 recebe 587
15
16     printf("IMPRESSAO ACESSANDO O VETOR\n\n");
17     for(i=0; i<3; i++){
18         printf ("vetor[%d] = %d\n\n",i, vetor[i]);
19     }
20
21     printf("IMPRESSAO ACESSANDO O PONTEIRO\n\n");
22     for(i=0; i<3; i++){
23         printf ("vetor[%d] = %d\n\n",i, *(v + i));
24     }
25
26     system ("pause");
27     return 0;
28 }
29
```

A Figura 21 mostra a saída do algoritmo acima após a sua execução.

Figura 21 – Saída do algoritmo

```
IMPRESSAO ACESSANDO O VETOR
vetor[0] = 248
vetor[1] = 954
vetor[2] = 587
IMPRESSAO ACESSANDO O PONTEIRO
vetor[0] = 248
vetor[1] = 954
vetor[2] = 587
Pressione qualquer tecla para continuar. . . _
```



A Figura 23 mostra a saída do algoritmo acima após a sua execução.

Figura 23 – Saída do algoritmo

```
Os valores gravados no vetor foram:

VETOR[0] COM ENDERECO: (000000000062FE30) = 66
VETOR[1] COM ENDERECO: (000000000062FE34) = 55
VETOR[2] COM ENDERECO: (000000000062FE38) = 89

Pressione qualquer tecla para continuar. . . _
```

## 4.1 Vetor de ponteiros

Os ponteiros também podem ser declarados na forma de uma estrutura de dados homogênea (Mizrahi, 2008). Para evidenciar esse conceito, temos o algoritmo na Figura 24, que define um vetor de ponteiros com 4 elementos, e mais quatros vetores de 3 elementos.

Figura 24 – Vetor de ponteiros

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4
5  int main()
6  {
7      int *pont[4];           // vetor de ponteiros do tipo inteiro
8
9      int x[3] = {1, 22, 322}; // primeiro vetor com três elementos
10     int y[3] = {4, 51, 66};  // segundo vetor com três elementos
11     int z[3] = {7, 83, 99};  // terceiro vetor com três elementos
12     int w[3] = {10, 11, 12}; // quarto vetor com três elementos
13
14     pont[0] = x;             // atribui o endereço do x para o ponteiro pont[0]
15     pont[1] = y;             // atribui o endereço do y para o ponteiro pont[1]
16     pont[2] = z;             // atribui o endereço do z para o ponteiro pont[2]
17     pont[3] = w;             // atribui o endereço do w para o ponteiro pont[3]
18
19     printf ("Valor de x[0]: %d\n", *pont[0]);
20     printf ("Valor de y[0]: %d\n", *pont[1]);
21     printf ("Valor de z[0]: %d\n", *pont[2]);
22     printf ("Valor de w[0]: %d\n", *pont[3]);
23
24     return 0;
25 }
```



A Figura 25 mostra a saída do algoritmo da Figura 24 após a sua execução.

Figura 25 – Saída do algoritmo

```
"C:\Users\Casa\Documents\Sandro\FACULDADES\UNINTER\Linguagem de Programação\ponteiro\  
Valor de x[0]: 1  
Valor de y[0]: 4  
Valor de z[0]: 7  
Valor de w[0]: 10  
  
Process returned 0 (0x0)   execution time : 0.223 s  
Press any key to continue.  
_
```

Para acessar os elementos de `pont[0]`, `pont[1]`, `pont[2]` e `pont[3]`, basta manipular os ponteiros utilizando o operador `*` e indicar o índice desejado. Conforme mostrado abaixo:

- `*pont[0]` – é o valor 1, o conteúdo do endereço 116, ou seja, `x[0]` e o mesmo valor pode ser obtido com a instrução
- `*pont[1]` – é o valor 4, o conteúdo do endereço 128, ou seja, `y[0]`;
- `*pont[2]` – é o valor 7, o conteúdo do endereço 140, ou seja, `z[0]`;
- `*pont[3]` – é o valor 10, o conteúdo do endereço 152, ou seja, `w[0]`.

Esse exemplo é ilustrado na Figura 26.

Figura 26 – Vetor de ponteiros do tipo inteiro

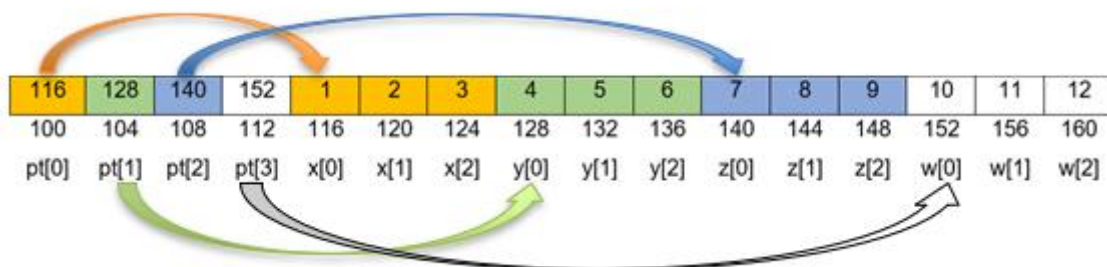






Figura 27 – Exemplo de vetor de ponteiros

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      /* Vetor de 5 ponteiros para números inteiros */
7      int *vetor[5];
8      int num1 = 16, num2 = 22, num3 = 31, num4 = 58, num5 = 63;
9
10     vetor[0] = &num1; /* vetor[0] aponta para num1 */
11     vetor[1] = &num2; /* vetor[1] aponta para num2 */
12     vetor[2] = &num3; /* vetor[2] aponta para num3 */
13     vetor[3] = &num4; /* vetor[3] aponta para num4 */
14     vetor[4] = &num5; /* vetor[4] aponta para num5 */
15
16     /* Imprime "num1: 16, num2: 22"... */
17     printf(" DADOS IMPRESSOS COM VETOR DE PONTEIROS\n");
18     printf(" num1: %i\n num2: %i\n num3: %i\n num4: %i\n num5: %i\n\n",
19         *vetor[0], *vetor[1], *vetor[2], *vetor[3], *vetor[4]);
20
21     system("pause");
22     return 0;
23 }
24
25
```

A Figura 28 mostra a saída do algoritmo da Figura 27 após a sua execução.

Figura 28 – Saída do algoritmo

```
DADOS IMPRESSOS COM VETOR DE PONTEIROS
num1: 16
num2: 22
num3: 31
num4: 58
num5: 63

Pressione qualquer tecla para continuar. . .
```

## TEMA 5 – PASSAGEM DE PARÂMETROS POR REFERÊNCIA

Uma das vantagens obtidas com ponteiros é a possibilidade de alterar o valor de variáveis que estão lugares diferentes do programa. O asterisco é utilizado para indicar que as variáveis são ponteiros e guardam o endereço de outras variáveis simples na memória.

Portanto, o conteúdo destas variáveis simples também pode ser modificado diretamente na memória quando passados seus endereços por meio



dos ponteiros para uma função, ou seja, as alterações dos dados sofridas dentro da função também serão sentidas fora dela (Mizrahi, 2008).

O código na Figura 29 mostra a implementação da passagem de parâmetros por referência.

Figura 29 – Passagem de parâmetros por referência

```
1  #include <stdio.h>
2  #include <stdlib.h>
3
4  void soma_mais_1(int *num);
5
6  int main()
7  {
8      int a = 8;
9
10     // Impressão de "a" antes da função.
11     printf("Antes da funcao: a = %d\n", a);
12     // A função recebe o endereço de "a".
13     soma_mais_1(&a);
14     // Impressão de "a" depois da função.
15     printf("Depois da funcao: a = %d\n", a);
16
17     system("pause");
18     return 0;
19 }
20 //Pega o endereço do Parâmetro "a".
21 void soma_mais_1(int *num){
22     *num = *num + 1;
23     printf("Dentro da funcao: a = %d\n", *num);
24 }
25
```

A Figura 30 mostra a saída do algoritmo acima após a sua execução.

Figura 30 – Saída do algoritmo

```
Antes da funcao: a = 8
Dentro da funcao: a = 9
Depois da funcao: a = 9

Pressione qualquer tecla para continuar. . . _
```

No algoritmo da Figura 29, temos os seguintes passos:

1. Na linha 8, declara-se a variável 'a' com o valor 8;
2. Na linha 11, a função printf() imprime o valor da variável 'a' antes do seu endereço ser passado para a função;



3. Na linha 13, a instrução `soma_mais_1(&a)` recebe o endereço da variável 'a';
4. Na linha 22, a função `soma_mais_1(*num)` altera diretamente o dado na memória.
5. Na linha 15, a função `printf()` imprime o valor da variável 'a' depois que a função `soma_mais_1(*num)` foi executada.

Esses efeitos não ocorrem quando os parâmetros são passados por valor (sem o uso do asterisco '\*' e o operador '&'), em que uma cópia do dado é passada como parâmetro para a função e a variável origem não sofre qualquer alteração.

Figura 31 – Exemplo de passagem de parâmetros por referência

```
1  #include <stdio.h>
2  #include<stdlib.h>
3
4  void alterar(int *n) {
5      *n = 2548;
6  }
7
8  int main() {
9      int x = 321;
10     int *p_endereco;
11
12     /* endereço recebe o endereço da variável x */
13     p_endereco = &x;
14
15     /* mostra conteúdo da variável x */
16     printf("Valor de x:   %d\n", x);
17     /* mostra o endereço de x */
18
19     printf("Endereço de x: %p\n\n", p_endereco);
20
21     /* passa o endereço de x como referência, para alteração */
22     alterar(&x);
23
24     /* mostra o novo valor de x */
25     printf("Valor de x alterado pela funcao: %d\n", x);
26
27     /* mostra o endereço que p_endereco aponta */
28     printf("Endereço de x no ponteiro: %p\n", p_endereco);
29
30     /* note que o endereço de x não foi alterado */
31     printf("Endereço da variável x:   %p\n\n", &x);
32
33     system("pause");
34     return 0;
35 }
```

A Figura 32 mostra a saída do algoritmo acima após a sua execução.



Figura 32 – Saída do algoritmo

```
Valor de x: 321
Endereco de x: 000000000062FE44

Valor de x alterado pela funcao: 2548
Endereco de x no ponteiro: 000000000062FE44
Endereco da variavel x: 000000000062FE44

Pressione qualquer tecla para continuar. . . _
```

## FINALIZANDO

Nesta aula, aprendemos os principais conceitos e temas das abordagens sobre ponteiro e sua aplicação em algoritmos computacionais; vimos como um dado é acessado na memória; bem como sua relação com vetores e funções.



---

## REFERÊNCIAS

ASCENCIO, A. F. G. **Fundamentos da programação de computadores:** Algoritmos, Pascal, C/C++ (padrão ANSI) JAVA. 3. ed. São Paulo: Pearson, 2012.

MIZRAHI, V. V. **Treinamento em linguagem C.** 2. ed. São Paulo: Pearson, 2008.