

Disciplina: Programação Estruturada Turmas: A1 e A2 – Noturno

Prof. Dr. Jesús P. Mena-Chalco
Assistente Docente: Rafael J. P. Damaceno



Ponteiro

Neste tutorial serão trabalhados os itens descritos na lista abaixo, sobre ponteiros, adaptada do livro “Linguagem C” de Luís Damas (10ª edição, 2007).

1. Um ponteiro é uma **variável** que **contém** o **endereço** de outra variável
2. Sua declaração é feita usando o **tipo de variável** para a qual se quer apontar, **seguido do operador asterisco (*)**
3. Ponteiros devem ser iniciados ou com o endereço de uma variável ou com NULL
4. **&** é o operador utilizado para se obter o **endereço** de uma variável
5. ***** é o operador utilizado para se obter o **conteúdo** de uma variável apontada por um ponteiro
6. Ponteiros possuem uma aritmética própria, que permite realizar operações de incremento, decremento, diferença e comparação
7. O nome de um vetor **v** corresponde ao **endereço do primeiro elemento** de **v**

Cada variável ocupa um determinado número de **bytes** consecutivos na memória. Por exemplo, uma variável do tipo **int** ocupa 4 **bytes** e uma do tipo **double** ocupa 8 **bytes**. Na maioria dos computadores, o endereço de uma variável é o endereço do seu primeiro **byte**. A seguir há quatro exemplos que exploram os conceitos envolvidos nos itens 1 a 5.

Exemplo 1. Declaração de variáveis e impressão de seus endereços.

```
1  #include <stdio.h>
2  int main() {
3      int x1 = 2006;           // inteiro x1 recebe 2006
4      int *p1 = &x1;          // ponteiro p1 recebe endereço de x1
5      int x2 = 2017;          // inteiro x2 recebe 2017
6      int *p2 = &x2;          // ponteiro p2 recebe endereço de x2
7      char c = 'u';           // caractere c recebe 'u'
8      printf("%p\n", &c);      // imprime um endereço (algo como 0x7ffffe0c697f)
9      printf("%p\n", &x1);      // imprime um endereço (algo como 0x7ffffe0c6980)
10     printf("%p\n", &x2);      // imprime um endereço (algo como 0x7ffffe0c6984)
11     printf("%p\n", p1);
12     printf("%p\n", p2);
13     printf("%d\n", *p1 + *p2)
14     return 0;
15 }
```

Lembre-se de que na função **printf()**, **%p** é usado para imprimir endereços. Observe que o endereço de **x1** é acessado por meio de **p1** (um ponteiro) e também por **&x1** (linhas 11 e 9, respectivamente). Outro ponto importante é que na declaração de ponteiros (linhas 4 e 6), usa-se ***** seguido do nome da variável. Caso não fosse atribuído um endereço, nesses dois casos, o recomendado seria atribuir o valor **NULL**, como segue: **int *p1 = NULL**. Por outro lado, o uso de ***** fora de atribuições é feito para acessar o conteúdo para o qual um determinado ponteiro aponta (linha 13). Perceber os diferentes papéis que o ***** pode exercer em um código é fundamental para compreender ponteiros. Pratique esses e outros exemplos!

Pergunta: O que as linhas 11, 12 e 13 do Exemplo 1 imprimem?

O trecho de código `float *p3 = 100.0` declara um ponteiro `p3` para `float` e atribui-lhe o valor `100.0`. **Porque esse trecho de código está incorreto?** As linhas 3 e 4 do Exemplo 2 a seguir apresenta uma das formas corretas de fazer essa atribuição.

Exemplo 2. Mais declaração de variáveis e impressão de seus endereços.

```
1 #include <stdio.h>
2 int main() {
3     float *p3 = NULL;
4     float x3 = 100.0;
5     p3 = &x3;
6     printf("%f\n", *p3);           // imprime o conteúdo apontado por p3
7     *p3 = 10.5;
8     printf("%f\n", *p3);           // imprime o conteúdo apontado por p3
9     printf("%f\n", x3);            // imprime o valor de x3
10    printf("%p\n", &p3);           // imprime o endereço de p3
11    printf("%p\n", p3);            // imprime o endereço p3
12    printf("%p\n", &x3);           // imprime o endereço de x3
13    return 0;
14 }
```

As linhas 10, 11 e 12 do Exemplo 2 imprimem endereços. Observe a diferença sutil entre as linhas 10 e 11. A primeira imprime o endereço do ponteiro `p3` (pois ponteiros também são variáveis, e por isso, também possuem endereço) e a segunda imprime o endereço que `p3` armazena (o conteúdo de `p3` é o endereço de `x3`, ou seja, `&x3`). Rode esses exemplos para ver na prática!

Pergunta: (a) Os endereços impressos nas linhas 10, 11 e 12 são iguais? (b) Qual é o valor final de `x3`?

Exemplo 3. Declaração de variáveis e impressão do tamanho (em bytes) que ocupam na memória.

```
1 #include <stdio.h>
2 int main() {
3     printf("-----\n");
4     printf("tipo          tamanho (bytes)\n");
5     printf("-----\n");
6     printf("char          %zu\n", sizeof(char));
7     printf("short         %zu\n", sizeof(short));
8     printf("int           %zu\n", sizeof(int));
9     printf("long          %zu\n", sizeof(long));
10    printf("-----\n");
11    short *p1 = NULL;
12    printf("* para short %zu\n", sizeof(p1));
13    printf("* para short %zu\n", sizeof(*p1));
14    float **p2 = &p1; // ponteiro para ponteiro para float
15    printf("** para float %zu\n", sizeof(p2));
16    return 0;
17 }
```

Complemente o Exemplo 3 para verificar o tamanho (em bytes) ocupado pelos seguintes tipos: (a) `long long`, (b) `float`, (c) `double` e (d) `long double`. Verifique também o tamanho (em bytes) ocupado por um ponteiro para `long double` e um ponteiro para ponteiro para `long double`. Observe a diferença sutil entre as linhas 12 e 13. Na linha 12 é impresso o tamanho do espaço ocupado por um ponteiro. Já na linha 13 é impresso o tamanho do espaço que é apontado por um ponteiro, portanto, depende do tipo apontado por `p1`. De forma alternativa, seria o mesmo que escrever `sizeof(short)`, pois `p1` aponta para `short`.

Pergunta: (a) Um ponteiro para `short` e um ponteiro para `long double` ocupam diferentes tamanhos (em bytes)? (b) E um ponteiro para ponteiro, que tamanho ocupa comparado a esses dois?

A seguir serão abordados os itens 6 e 7 da lista e um pouco sobre alocação dinâmica de memória. Nas situações em que não se sabe previamente a quantidade de memória a ser utilizada, por exemplo quando não se conhece quantos elementos um vetor conterá, pode-se utilizar a função `malloc`. Essa função aloca uma quantidade de bytes consecutivos na memória e retorna o endereço desse bloco, ou seja, um ponteiro. Para isso, precisa-se passar como parâmetro a quantidade de bytes que queremos alocar, o que é feito com auxílio da função `sizeof()` vista no Exemplo 3.

Exemplo 4. Alocação dinâmica de memória e aritmética de ponteiros.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int n;
5     printf("n = ");
6     scanf("%d", &n);
7     int *v = malloc(n * sizeof(int));
8     for(int i = 0; i < n; i++)
9         v[i] = i;
10    printf("n[0]: %d\n", *(v));
11    printf("n[0]: %d\n", v[0]);
12    printf("n[n-1]: %d\n", *(v + n - 1));
13    printf("n[n-1]: %d\n", v[n - 1]);
14    free(v);
15    v = NULL;
16    return 0;
17 }
```

Note que no Exemplo 4 foi necessário carregar a biblioteca `stdlib` (linha 2), que contém a função `malloc`. Na linha 7 `malloc` aloca-se n vezes o tamanho do tipo `int` e retorna o endereço desse bloco. A variável `v` é um ponteiro para `int` que recebe esse endereço, que é na prática, o endereço do primeiro elemento do vetor de `int`. Na versão estática, para $n = 100$, seria o mesmo que fazer `int v[100]`, sendo o primeiro elemento representado por `v[0]`. Cabe lembrar que usando a notação de ponteiros, `*(v)` aponta para o primeiro elemento do bloco alocado. Por exemplo, para acessar o conteúdo da quinta posição pode-se fazer `*(v + 5)`. Para acessar o conteúdo da última posição, pode-se fazer `*(v + n - 1)`. Seriam o mesmo que fazer `v[5]` ou `v[n - 1]`.

No Exemplo 5 é alocada memória o suficiente para armazenar 10 elementos do tipo `int` em um vetor (linhas 4 a 8), e também 20 elementos do tipo `int` em uma matriz com 4 linhas e 5 colunas (linhas 9 a 12). Note que uma matriz é um vetor de vetor e sua alocação na memória é feita de forma sequencial, como se fosse um único vetor com o total de elementos. Uma matriz composta por 4 linhas e 5 colunas é, na memória, um único vetor composto por 20 elementos.

Exemplo 5. Aritmética de ponteiros, vetores e matrizes.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 int main() {
4     int *v = malloc(10 * sizeof(int)); // início de alocação de vetor
5     for (int i = 0; i < 10; i++){
6         *(v + i) = i * 5;
7         printf("v[%d] : %d (%p)\n", i, *(v + i), v + i);
8     }
9     int **m; // início de alocação de matriz
10    m = malloc (4 * sizeof (int *)); // o endereço para o bloco alocado
11    for (int i = 0; i < 4; i++)
12        *(m + i) = malloc (5 * sizeof (int));
13    for (int i = 0; i < 4; i++){ // percorrendo matriz
14        for (int j = 0; j < 5; j++){
15            m[i][j] = (i * 5) + j; // atribui um valor qualquer a m[i][j]
16            printf("m[%d][%d] = %d (%p)\n", i, j, m[i][j], &m[i][j]);
17        }
18    }
19    return 0;
20 }
```

A linha 10 aloca um bloco com que ocupa **4 vezes o tamanho de um ponteiro para int**. Observe os diferentes usos do operador `*`: (i) pode indicar a declaração de ponteiro, (ii) o acesso ao conteúdo apontado por um ponteiro, e (iii) uma simples multiplicação. Na mesma linha 10 esse operador aparece duas vezes. **Que papéis ele exerce?**

Pergunta: (a) Na alocação de um vetor com 10 elementos do tipo `int`, os endereços dos 10 elementos são sequenciais? (b) E no caso de matrizes?

Teste interativo usando <https://cdecl.org>

Para cada uma das seguintes declarações veja exatamente o que está sendo definido.

- `int *A`
- `int **A`
- `int **A[17]`
- `float *p3`
- `float &p3`
- `void (*pf)(int)`
- `double (*pf)(int)`

Tutorial interativo

- Aritmética de ponteiros (http://www.learn-c.org/en/Pointer_Arithmetics)
- Ponteiros a função (http://www.learn-c.org/en/Function_Pointers)