# Capítulo 3

# Ponteiros e Alocação Dinâmica

Este capítulo descreve sobre um dos mais interessantes recursos da linguagem C: Ponteiros. Além disso, apresenta os mecanismos necessários para manipular dinamicamente a memória e por consequência ponteiros e vetores.

## 3.1 Ponteiros

Os **ponteiros** são variáveis especiais que contém **ENDEREÇOS DE MEMÓRIA**. Este endereço de memória por sua vez contém um valor. Isto é, este valor é uma variável.

- Ponteiro: Variável que faz **referência indireta** ao valor; diz-se que o ponteiro *aponta para outra variável*;
- Variável: Variável que faz **referência direta** ao valor.

A figura 3.1 ilustra o conceito de ponteiros.

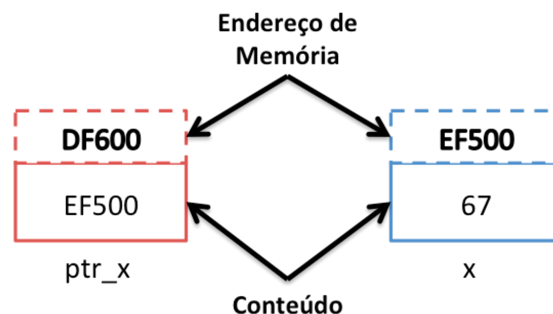


Figura 3.1: Representação gráfica de um ponteiro

É possível notar na figura que existem duas variáveis: ptr_x e x. A variável **ptr_x** possui em seu conteúdo o endereço de memória da variável **x**. A variável **x** no entanto possui um tipo conhecido de dado, isto é, um valor inteiro.

### 3.1.1 Declaração de Ponteiros

Os ponteiros como quaisquer outras variáveis precisam ser declarados antes de serem utilizados. Declara-se uma variável do tipo ponteiro deste modo:

```
1 tipo *nome_do_ponteiro;
```

Exemplo:

```
1 int *Ptr, *ptr_x, *ptr;
```

Onde:

- **tipo**: indica o tipo da variável para qual o ponteiro irá apontar;
- *: operador utilizado para indicar que a variável declarada é do tipo ponteiro.

### 3.1.2 Operadores de Ponteiros

Existem dois operadores especiais para manipular ponteiros.

- **(Asterisco) ***: Operador de **referência indireta** ou operador de **desreferenciamento**. Retorna o valor da variável para qual o ponteiro aponta.
- **(E-comercial) &**: Operador de ponteiro: É um operador unário que retorna o endereço de memória do seu operando (variável).

Exemplo:

```
1 int y = 5;
2 int *ptr_y = NULL; /*Declara o ponteiro para inteiro e
    inicializa-o com NULL*/
3 ptr_y = &y; /*Atribui o endereço da variável 'y' ao
    ponteiro ptr_y. Diz-se que ptr_y aponta para 'y'*/
```

A figura 3.2 ilustra o comportamento das variáveis em relação à sua ocupação na memória.

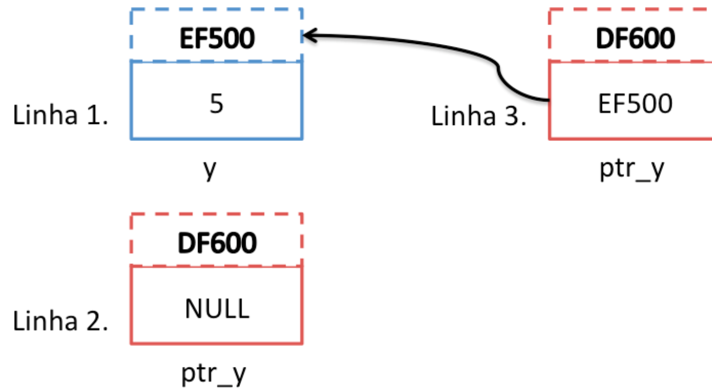


Figura 3.2: Atribuição de valores às variáveis 'y' e 'ptr_y'.

O próximo exemplo mostra em um único código várias formas de como pode-se obter os valores e seus respectivos endereços de memória utilizando ponteiros.

```
1  #include <stdio.h>
2
3  int main(){
4
5          int a, *ptr_a = NULL; /*'a' e um inteiro e 'ptr_a
               ' e um ponteiro para inteiro*/
6          a = 7;
7          ptr_a = &a; /*'ptr_a' recebe o endereco de 'a'*/
8
9          printf("\n\tEndereco de 'a': %p\n\tEndereco de '
              ptr_a': %p", &a, &ptr_a);
10         printf("\n\tValor de 'a': %i\n\tValor de 'ptr_a':
               %p", a, ptr_a);
11         printf("\n\tValor de '*ptr_a': %i", *ptr_a);
12
13 return 0;
14 }
```

### 3.1.3 Ponteiros e Vetores

Até o presente momento, vetores e ponteiros são vistos em nossos estudos como elementos distintos. Este conceito, entretanto, deixa de existir com o estudo desta seção. Nesta etapa iremos conhecer e compreender a relação que existe entre um **vetor** e um **ponteiro**.

Em C, ponteiros e vetores são vistos pela linguagem como uma única estrutura de dados. Um ponteiro pode ser utilizado como um vetor e vice-versa.

Na realidade, um vetor é um **ponteiro** constante para o primeiro elemento, ou seja, o menor endereço de memória (lembre-se que um vetor é um conjunto contíguo de endereços de memória). O vetor é acessado pelo seu nome e por um índice indicado entre colchetes ([ ]).

Desta forma, pode-se declarar um ponteiro para manipular um vetor, veja o exemplo:

```
1  #include <stdio.h>
2
3  int main(){
4
5          int vetor[5], i, *ptr_v = NULL;
6
7          ptr_v = vetor; /*O ponteiro recebe o endereço de
              memória do primeiro elemento, vetor == &vetor
              [0]*/
8
9          for(i = 0; i < 5; i++)
10                 ptr_v[i] = 7 + i;
11
12         for(i = 0; i < 5; i++)
13                 printf("\n\tVetor[%d]: %d / ptr_v[%d]: %d
                      ", i, vetor[i], i, ptr_v[i]);
14
15 printf("\n\n");
16 return 0;
17 }
```

**Dicas:**

- Após declarar um ponteiro inicialize-o com **NULL** para evitar problemas em tempo de execução. Ex.: ptr = NULL;

## 3.2 Alocação Dinâmica

A linguagem C permite, com a utilização de ponteiros, alocar memória dinamicamente conforme a necessidade do programador. Este recurso é bastante útil quando se deseja armazenar um conteúdo do qual não se tem o conhecimento prévio de seu tamanho (em *bytes*). Isto otimiza o uso da memória principal do computador.

Existem 4 funções associadas a alocação dinâmica de memória, são elas: *malloc(), calloc(), realloc() e free()*. As funções são acessíveis com a utilização da biblioteca padrão #**include** <**stdlib.h**>. Os protótipos e as descrições das funções são apresentados a seguir:

### 3.2.1 malloc()

- **malloc()**: *void *malloc(size_t* `n_Bytes`*);* A função **malloc()** aloca um espaço contíguo na memória de tamanho: `n_Bytes` * **size_t** e retorna um ponteiro do tipo **void ***, onde:

  `n_Bytes` indica o número de *Bytes* correspondente a um determinado tipo. Isto é, se `n_Bytes` for **sizeof(int)**, ele assumirá o valor de 4 *Bytes*. Em outras palavras, o parâmetro `n_Bytes` indica o tamanho (obtido com o operador **sizeof()**) em *Bytes* do tipo de dado que se deseja armazenar (char, int, float, double, etc.).

  **size_t** é definida na biblioteca <stdlib.h> como sendo um **typedef unsigned int size_t**. Isto é, um inteiro sem sinal. Este parâmetro é utilizado como fator de multiplicação do tipo de dado (`n_Bytes`). Note que este valor é sempre um inteiro positivo uma vez que somente é permitido alocar um número positivo de posições de memória.

  **Exemplo:** Alocar um vetor de 5 posições:

```
1 int *ptr;  //declara um ponteiro do tipo inteiro
2 ptr = (int *) malloc(5 * sizeof(int)); /*ptr
     recebe o endereço de memória do primeiro
     elemento do vetor de inteiros*/
```

No comando da linha 2. o compilador irá obter o tamanho do tipo de dado inteiro (**sizeof(int)**) e multiplicá-lo por 5. Assim, o ponteiro **ptr** irá apontar para um vetor de 5 posições do tipo inteiro.

Observe na linha 2. a instrução (**int** *) antes do operador '='. Esta instrução indica para o compilador que ele deve EXPLICITAMENTE converter o ponteiro de retorno sem tipo (**void** *) para um ponteiro do tipo inteiro (**int** *).

Para efeito de ilustração, o comando da linha 2. pode ser comparado a seguinte declaração de vetor:

```
1 int ptr[5];  //declara um vetor do tipo inteiro
     com 5 elementos
```

### 3.2.2 calloc()

- **calloc()**: *void *calloc(size_t num, site_t size);* A função **calloc()** aloca `num` elementos com dimensão `size`. Isto é, ela funciona da mesma forma que a função **malloc** exceto por uma diferença: após alocar os espaços de memória ela zera todos os elementos.

  Os parâmetros da função também são organizados de outra maneira, onde:

  `num` inteiro positivo que indica a quantidade de elementos.

  `size` equivalente ao `n_Bytes` da função malloc(). Isto é, o número de *Bytes* correspondente a um determinado tipo. Se `size` for **sizeof(int)**, ele assumirá o valor de 4 *Bytes*. Em outras palavras, o parâmetro `size` indica o tamanho (obtido com o operador **sizeof()**) em *Bytes* do tipo de dado que se deseja armazenar (char, int, float, double, etc.).

  **Exemplo:** Alocar um vetor de 5 posições:

```
int *ptr;  //declara um ponteiro do tipo inteiro
ptr = (int *) calloc(5, sizeof(int)); /*ptr
    recebe o endereço de memória do primeiro
    elemento do vetor de inteiros*/
```

No comando da linha 2. o compilador irá obter o tamanho do tipo de dado inteiro (**sizeof(int)**) e multiplicá-lo por 5. Note que na função **calloc()** a multiplicação ocorre de maneira implícita.

Para efeito de ilustração, o comando da linha 2. pode ser comparado a seguinte declaração de vetor:

```
int ptr[5];  //declara um vetor do tipo inteiro
    com 5 elementos
```

### 3.2.3 realloc()

- **realloc()**: *void *realloc(void *ptr, size_t new_size);* A função **realloc()** permite alterar a quantidade de posições de memórias previamente alocadas com as funções **malloc()** ou **calloc()**.

  Esta função possui um parâmetro adicional (void *ptr) que corresponde ao ponteiro associado ao espaço de memória do qual se deseja expandir. Se o ponteiro (ptr) for fornecido o conteúdo do atual espaço de memória é mantido. Se **NULL** for fornecido o conteúdo é perdido.

  O parâmetro size_t **new_size** indica o novo tamanho a ser alocado. Este parâmetro é equivalente aos parâmetros (size_t `n_Bytes`) da função **malloc()**.

  Algumas considerações sobre o **realloc()**:

  - Se o novo tamanho de memória puder ser reservado, a memória é aloca e é retornado o ponteiro com o endereço;
  - Se o espaço não puder ser aumentado contiguamente, um novo espaço de memória com o tamanho total é alocado e novo endereço é retornado para o ponteiro.
  - Se a memória não puder ser realocada ou um novo bloco criado então **NULL** é retornado.

**Exemplo:** Realocar um vetor de 5 posições:

```
1 int *ptr;  //declara um ponteiro do tipo inteiro
2 ptr = (int *) malloc(5 * sizeof(int)); /*ptr
     recebe o endereço de memória do primeiro
     elemento do vetor de inteiros*/
3 ptr = (int *) realloc(ptr, 10 * sizeof(int)); /*
     ptr recebe o endereço de memória do novo vetor
      agora com 10 elementos*/
```

### 3.2.4 free()

- **free()**: *void free(void *ptr);* A função **free()** libera o espaço de memória alocado com as **malloc(), calloc() ou realloc()**.

  A função deve ser invocada para cada ponteiro que faz referência a espaços de memória alocados dinamicamente durante a execução do programa.

  Dica: A utilização desta função é

  **EXTREMAMENTE RECOMENDADA**

  Tenha o bom hábito de utilizar ao final de cada programa. Assim, você libera a memória para o sistema operacional de modo explícito, não necessitando que as rotinas de alocação de memória do S.O. façam isso pelo seu programa.

## 3.3 Exemplos

Alguns exemplos de programas utilizando as funções de alocação dinâmicas são apresentados a seguir.

**Figura 3.3: Exemplo de alocação dinâmica utilizando malloc()**
**Figura 3.4: Saída do programa da figura 3.3.**
**Figura 3.5: Exemplo de alocação dinâmica utilizando malloc() com 'char'**
**Figura 3.6: Saída do programa da figura 3.5.**

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5
6     //declara duas variaveis inteiras
7     int i, tam = 5;
8     //declara um ponteiro para inteiro chamado 'ptr'
9     int *ptr;
10
11    //aloca dinamicamente um "vetor com 5 elementos" apontado por 'ptr'
12    ptr = (int *) malloc(tam * sizeof(int));
13
14    //preenche o vetor apontado por 'ptr' e mostra os valores
15    for(i = 0; i < tam; i++){
16        ptr[i] = i+5;
17        printf("\n\tValor de ptr[%i]: %i", i, ptr[i]);
18    }
19    //libera o espaco de memoria alocado para 'ptr'
20    free(ptr);
21
22 printf("\n\n");
23 return 0;
24 }
```

Figura 3.3: Função malloc()

```
hoss@debian8vm:~/ProgII/aula08_ponteiros_malloc$ ./exemplo1

        Valor de ptr[0]: 5
        Valor de ptr[1]: 6
        Valor de ptr[2]: 7
        Valor de ptr[3]: 8
        Valor de ptr[4]: 9

hoss@debian8vm:~/ProgII/aula08_ponteiros_malloc$
```

Figura 3.4: Saída do programa com a função malloc()

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5
6     //declara duas variaveis inteiras
7     int i, tam = 5;
8     //declara um ponteiro para char chamado 'ptr'
9     char *ptr;
10
11    //aloca dinamicamente um "vetor com 5 elementos" apontado por 'ptr'
12    ptr = (char *) malloc(tam * sizeof(char));
13
14    //preenche o vetor apontado por 'ptr' e mostra os valores
15    for(i = 0; i < tam; i++){
16        ptr[i] = i + 65;
17        printf("\n\tValor de ptr[%i]: %c", i, ptr[i]);
18    }
19    //libera o espaco de memoria alocado para 'ptr'
20    free(ptr);
21
22 printf("\n\n");
23 return 0;
24 }
```

Figura 3.5: Função malloc() com char

```
hoss@debian8vm:~/ProgII/aula08_ponteiros_malloc$ ./exemplo2

        Valor de ptr[0]: A
        Valor de ptr[1]: B
        Valor de ptr[2]: C
        Valor de ptr[3]: D
        Valor de ptr[4]: E

hoss@debian8vm:~/ProgII/aula08_ponteiros_malloc$
```

Figura 3.6: Saída do programa com a função malloc() com char

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 
 4 int main(){
 5 
 6     //declara duas variaveis inteiras
 7     int i, tam = 6;
 8     //declara um ponteiro para char chamado 'nome'
 9     char *nome;
10     //aloca dinamicamente um "vetor com 6 elementos" apontado por 'nome'
11     nome = (char *) malloc(tam * sizeof(char));
12 
13     //atribui o texto "Diego" ao vetor apontado por 'nome' e mostra o conteudo
14     nome = "Diego";
15     printf("\n\t%s", nome);
16 
17     /*realoca o espaco para conter o nome completo:
18     Diego Hoss 10 caracteres mais NULL (11 no total)
19     apagando o conteudo original*/
20     nome = (char *) realloc(NULL, 11 * sizeof(char));
21 
22     //atribui e mostra o nome completo
23     nome = "Diego Hoss";
24     printf("\n\t%s", nome);
25 
26     //libera o espaco de memoria alocado para 'nome'
27     free(nome);
28 
29 printf("\n\n");
30 return 0;
31 }
```

Figura 3.7: Função realloc() com char

```
hoss@debian8vm:~/ProgII/aula08_ponteiros_malloc$ ./exemplo3

        Diego
        Diego Hoss

hoss@debian8vm:~/ProgII/aula08_ponteiros_malloc$ _
```

Figura 3.8: Saída do programa com a função realloc() com char

**Figura 3.7: Exemplo de realocação dinâmica utilizando realloc() com 'char'**
**Figura 3.8: Saída do programa da figura 3.7.**