# TÉCNICAS DE PROGRAMAÇÃO

## UNIDADE 3 - ALOCAÇÃO DE MEMÓRIA ESTÁTICA E DINÂMICA

Fernando Cortez Sica

# Introdução

Iniciamos o nosso terceiro encontro sobre Técnicas de Programação. Conversaremos sobre um ponto que ajudará o programa a se tornar mais flexível e, em certos casos, mais eficiente; essa flexibilidade é conseguida pela utilização de ponteiros. Mas, já não usamos ponteiros na passagem de parâmetros por referência? Sim, exato! O primeiro uso e contato com os ponteiros foi na passagem de parâmetros por referência (usando os símbolos "&" e "*"). Mas, aonde mais usamos ponteiros? Podemos encontrar ponteiros nos códigos que manipulam os dispositivos do *hardware* (os chamados *device drivers*), no *kernel* do sistema operacional, nos sistemas de banco de dados e em outras diferentes aplicações! Quando conversamos sobre vetores e falamos que seria necessário definirmos um tamanho já na declaração de variáveis, você pode ter se perguntado: mas, caso a quantidade de itens manipulados aumentasse, teríamos que modificar o programa, alterando-se o tamanho do vetor? Sim, no caso dos vetores, já que eles são estruturas de tamanho fixo, pré-definido. Mas, então, com a utilização de ponteiros, esse problema é eliminado? Sim, veremos uma estratégia denominada alocação de memória dinâmica que servirá, dentre outras coisas, para eliminar esse problema.

No entanto, pode surgir uma outra questão: se os ponteiros nos possibilitam uma manipulação de memória mais poderosa, então é mais difícil de programar? Vamos tentar não usar o termo dificuldade e, sim, falar que o uso de ponteiros requer um pouco mais de atenção devido ao seu alto poder de abstração.

Neste capítulo, falaremos sobre os conceitos e manipulação de ponteiros. Já que mencionamos, anteriormente, que os vetores na passagem de parâmetros das funções, são considerados como ponteiros, então veremos como definir vetores e matrizes usando ponteiros; em seguida, conversaremos sobre uma outra forma de manipular informações, no caso, informações heterogêneas: o uso de estruturas ou registros. Por fim, vamos abordar a representação de tais estruturas como ponteiros.

## 3.1 Ponteiros

Um ponteiro é uma referência a uma posição de memória. A diferença básica entre uma variável do tipo ponteiro e uma não ponteiro consiste no fato de que uma variável que não seja ponteiro armazena um valor propriamente dito. Por sua vez, uma variável do tipo ponteiro, realiza uma indireção, ou seja, faz uma referência indireta a um valor. Mas, na declaração de uma variável, como diferenciar e manipular os dois tipos? Para respondermos essa pergunta, vamos nos amparar com o código abaixo:

```
#include <stdio.h>

int main()
{
  int a=10;          //variavel "normal"                          //linha 1
  int *b;            //variavel do tipo ponteiro para int         //linha 2
  b = &a;                                                         //linha 3
  printf("'a'=%d\t'*b'=%d   (posicao 'a'=%p\tponteiro b=%p\n",
         a,*b,&a,b);                                              //linha 4
  a = 30;                                                         //linha 5
  printf("Valor de 'b' modificado para %d\n",*b);                 //linha 6
  return 0;
}
```

Neste código, temos as explanações para as linhas descritas a seguir. Clique nos itens e confira!

**Linha 1**

Declaração da variável “a” do tipo “int” (sem ser ponteiro), instanciada com o valor 10.

**Linha 2**

Declaração da variável “b” como sendo um ponteiro para um “int”. Para declarar um ponteiro, utiliza-se “*” antes do nome da variável (como denotado por “*b”).

**Linha 3**

Nesta linha temos a indicação que a posição de memória apontada por “b” passa a ser o endereço da variável “a”. O ponteiro “b”, neste caso, não fez uso do “*” o precedendo, pois, estamos a manipular o próprio valor do ponteiro. Para denotarmos a posição de memória de uma variável usamos o símbolo “&” (como denotado por “&a”). Você pode se perguntar: é por esse motivo que usamos esse símbolo “&” na passagem de parâmetros por referência nas funções? Sim, exatamente. Na chamada da função, os parâmetros passados por referência são precedidos por “&” pois passamos para a função, na verdade, o endereço de memória da variável envolvida.

**Linha 4**

A função “printf” imprime os valores da variável “a” e do conteúdo da posição de memória apontado por “b” e, também, a posição de memória na qual foi alocada a variável “a” e a posição do ponteiro “b”. Na execução, poderemos notar que esses dois últimos valores são iguais pois referenciam a mesma posição de memória.

**Linha 5**

O valor da variável “a” é modificada. Como o ponteiro “b” aponta para “a”, qualquer modificação de “a” altera prontamente, o valor do conteúdo de “b”. Sendo assim, a função “printf” da linha 6 imprimirá o valor atribuído a “a”; no caso, imprimirá o valor 30.

O fato da modificação da variável “**a**” impactar o valor do conteúdo apontado pela variável “**b**”, após a execução da linha 3, pode ser explicado por intermédio da figura a seguir:

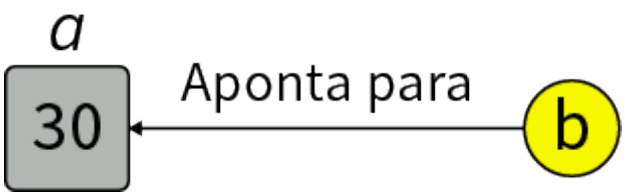


Figura 1 - Estados das variáveis após a execução da linha 5 do código acima. O ponteiro “b” aponta para a mesma posição de memória da variável “a”.
Fonte: Elaborada pelo autor, 2019.

A figura acima ilustra que após a execução da linha contendo a instanciação “b = &**a**”, o ponteiro “**b**” aponta para a mesma posição de memória da variável “**a**”. Sendo assim, qualquer alteração de “**a**” altera o conteúdo do ponteiro “**b**” (“***b**”).

Como mencionado em Deitel (2011), todo ponteiro, para ser efetivamente manipulado, deve ser instanciado. Porém, podemos, em algumas ocasiões, definir um ponteiro apontando para um valor nulo. Neste caso, para criarmos um ponteiro nulo poderemos proceder de alguma das formas a seguir:

```
int * p = NULL, *q = 0;
```

Na linha de código acima, temos a instanciação de dois ponteiros ("p" e "q") com o valor nulo. A definição do "NULL" encontra-se no arquivo *header* "<stddef.h>".
Falamos até aqui, de um ponteiro apontando para uma variável não ponteiro. Mas, um ponteiro também pode apontar para um ponteiro? Veremos isso a seguir.

## 3.2 Ponteiros para ponteiros

Um ponteiro para ponteiro significa que a variável do tipo ponteiro não faz referência a uma posição de memória e, sim, faz referência à um outro ponteiro. Esse último, por sua vez, aponta uma posição de memória que contém, efetivamente, o valor armazenado. Para falarmos de ponteiros para ponteiros, vamos imaginar uma utilidade: suponha que uma matriz seja implementada com ponteiros (como veremos adiante); e que, em um certo momento, necessitemos realizar uma operação de ordenação, sabemos que a ordenação da matriz necessita da movimentação de dados. Uma última suposição: cada célula da matriz armazena uma informação de tamanho relativamente grande. Com esse cenário, podemos concluir que demandaremos para realizar a operação de ordenação, uma grande movimentação de informações. Para atenuar isso, podemos construir uma matriz formada por ponteiros para ponteiros. Neste caso, cada ponteiro aponta para uma posição da matriz, ou seja, um ponteiro aponta para cada ponteiro que caracteriza a matriz. Sendo assim, ordenar uma matriz, significa instanciar ponteiros e não movimentar grandes volumes de informação. (MIZRAHI, 2008).
Mas, como podemos definir ponteiros para ponteiros? Vamos analisar o código a seguir:

```
#include <stdio.h>

int main()
{
  int a = 10, *b, **c;                                                    //1
  b = &a;                                                                 //2
  c = &b;                                                                 //3
  printf("a=%d  *b=%d  **c=%d\n",a,*b,**c);                               //4
  printf("pos 'a'=%p  pos 'b'=%p  pos '**c'=%p  pos '*c'=%p  pos 'c'=%p\n",
          &a,b,**c,*c,c);                                                 //5
  return 0;
}
```

Clique nas abas abaixo e conheça a descrição de cada linha do código acima.

- **Linha 1**

  A variável " **b**" é um ponteiro para " **int**" (uso do símbolo " *" precedendo o nome da variável). Por sua vez, o uso de dois sinais " *" denota, em " **c**" um ponteiro para ponteiro.

- **Linha 2**

  Instanciação do ponteiro " **b**" recebendo o endereço da variável " **a**".

- **Linha 3**

  instanciação do ponteiro para ponteiro " **c**" recebendo o endereço do ponteiro " **b**".

- **Linha 4 e 5**

  Impressão dos conteúdos e dos endereços das variáveis e ponteiros. Nota-se que, para imprimir o valor final (no caso 10), devemos indicar, no caso do ponteiro para ponteiro " **c**" que a impressão se refere ao conteúdo do conteúdo de " **c**", ou seja, valor do ponteiro que aponta para o ponteiro " **b**" que, por sua vez, aponta para a variável " **a**" – esse é o motivo da utilização de "**" precedendo o nome da variável.

A figura a seguir ilustra melhor esse encadeamento de indireções.

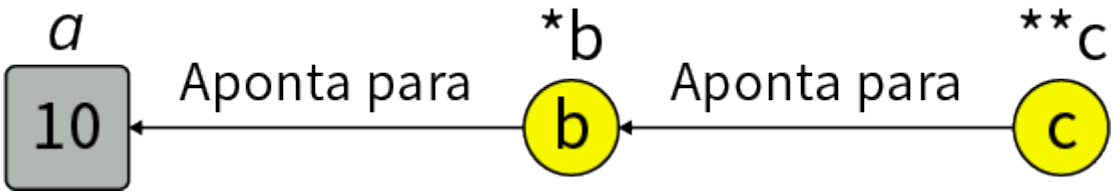


Figura 2 - Relação dos apontamentos feitos por um ponteiro ("*b") e por um ponteiro para ponteiro ("**c").
Fonte: Elaborada pelo autor, 2019.

Em C, podemos aumentar a indireção criando, por exemplo, ponteiro para ponteiro para ponteiro. Neste caso, a criação da variável seria feita da seguinte forma, por exemplo: "**int ***p**". A quantidade de símbolos "*" indica a quantidade de indireções desejadas.

## VOCÊ SABIA?

Em algumas ocasiões, tais como sistemas embarcados, funções para o tratamento de interrupções e implementação de "*threads*", temos que usar ponteiros para funções. Desta forma, a chamada de funções torna-se dinâmica, facilmente configurável. Para saber um pouco mais sobre o assunto, clique neste link <https://www.embarcados.com.br/ponteiro-em-c-funcoes/>. (GARCIA, 2015).

Quando mencionamos uma das aplicações de ponteiros para ponteiros, mencionamos o caso de matrizes definidas com ponteiros. Para chegar lá, vamos compreender antes, como definimos vetores com ponteiros.

## 3.3 Alocação de vetores como ponteiros

Quando mencionamos o fato de que na passagem de parâmetros de funções, os vetores se comportam como passagem por referência, queremos dizer que, na verdade, vetores são manipulados como ponteiros. De fato, o compilador cria um espaço de memória de tamanho *N*K*, onde *N* representa a dimensão do vetor e *K* o espaço gasto para armazenar um item de um determinado tipo de dado. Então, podemos definir um vetor como ponteiros da seguinte forma:

```
int *vetor;
```

Mas, como inserir e ler os valores contidos em um vetor declarado desta forma? Para que possamos entrar neste assunto, veremos o exemplo de codificação a seguir:

```
#include <stdio.h>

int main()
{
  int vet[10]={0,1,2,3,4,5,6,7,8,9};                          //Linha 1
  int *pvet;                                                  //Linha 2
    pvet = vet;                                               //Linha 3
    *(pvet+3) = 13;                                           //Linha 4
    vet[5] = 15;                                              //Linha 5
    for(int i=0; i<10; i++)                                   //Linha 6
      printf("vet[%d] = %d\t*(pvet+%d)=%d\n",i,vet[i],i,*(pvet+i));   //Linha 7
    return 0;
}
```

Para conhecer a descrição do código apresentado, clique nas abas abaixo.

| | |
|---|---|
| **Linha 1** | Criação e instanciação do vetor "**vet**" de valores inteiros de 10 posições. |
| **Linha 2 e 3** | Criação de uma variável do tipo ponteiro para inteiro "p**vet**" (linha 2) e a sua instanciação (na linha 3), recebendo o endereço ocupado pelo vetor "**vet**". Nota-se que, nesta linha, não aparece o símbolo "**&**" precedendo a variável "**vet**" pois ela, sendo um vetor, já é considerada como um ponteiro. Sendo assim, a partir deste momento, a variável "**pvet**" aponta para o início do vetor "**vet**". Essa instanciação poderia ser feita, também, da seguinte forma: "**pvet = &vet[0]**" – o ponteiro "**pvet**" recebe o endereço da posição 0 do vetor "**vet**". |
| **Linha 4** | Instanciação de uma posição do vetor. O código "***(pvet + 3) = 13**" pode ser traduzido para a seguinte forma: o conteúdo do vetor cuja posição é o início do vetor deslocada de 3 itens recebe o valor 13. |
| **Linha 5** | O mesmo processo de acessar a posição 5 a partir do início do vetor para que seja instanciada com o valor 15. Essa mesma interpretação deve ser feita na linha 7, para proceder a impressão do conteúdo do vetor. |

O processo de acessar uma posição *N* a partir do início do vetor é ilustrada na figura a seguir:

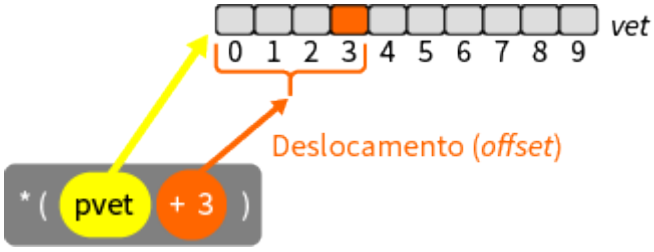


Figura 3 - Acessando a posição 3 de um vetor através do uso de ponteiro. O valor 3 representa o deslocamento (offset) a partir do início do vetor.
Fonte: Elaborada pelo autor, 2019.

Como ilustrado na figura acima, o índice utilizado para o acesso à uma posição específica (no caso, utilizado como exemplo, o valor 3), representa o deslocamento a partir da posição inicial do vetor. Mas, quantos *bytes* representa esse deslocamento? A quantidade de *bytes* saltada por unidade de índice varia em função do tamanho do item representado. Para saber o tamanho de um tipo, pode-se utilizar a função "**sizeof(<tipo>)**" da seguinte forma:

```
printf("Tamanho de um int = %d\n",sizeof(int));
```

Na linha acima, a função "**sizeof**" retornará um valor inteiro que representa a quantidade de *bytes* utilizada por um "**int**". Voltando ao deslocamento, você pode se perguntar: se eu posso adicionar um valor à um ponteiro eu obtenho uma posição específica de memória, então, eu posso incrementar um ponteiro? Sim, é possível realizar incrementações com ponteiros. Por exemplo, caso façamos, "**p++**" (sendo "**p**" um ponteiro), significa que "**p**" apontará para a próxima posição deslocados *N* bytes (sendo que *N* denota o tamanho ocupado pelo tipo manipulado). Para testar esse comportamento, tomaremos como exemplo, o trecho a seguir:

```
for(int i=0; i<10; i++)
  printf("vet[%d] = %d\t*(pvet+%d)=%d\n",i,vet[i],i,*(pvet+i));
```

Vamos, então, substituir o "***(pvet+i)**" por "***(pvet++)**":

```
for(int i=0; i<10; i++)
  printf("vet[%d] = %d\t*(pvet+%d)=%d\n",i,vet[i],i,*(pvet++));
```

Teremos o mesmo resultado. Porém, ao invés de incrementarmos o índice "**i**", estamos a mudar o próprio ponteiro, fazendo-o apontar para a posição subsequente de memória a cada iteração do laço de repetição. Neste caso, para que voltemos à posição inicial do vetor, precisaremos decrementar o ponteiro ou ter salvo, em outra variável, a referência de memória relativa à posição 0. Até o momento, usamos ponteiros apenas para referenciar vetores criados estaticamente, ou seja, referenciar vetores criados através do código: "**<tipo> nome[dimensão]** ". Seria possível criar um vetor dinamicamente? O que seria alocação dinâmica de memória? Alocação dinâmica

consiste em reservar espaços de memória sob demanda, ela é útil para, por exemplo, quando não se conhece previamente a quantidade de informações que serão manipuladas, ou ainda, quando é necessário criar estruturas temporárias de armazenamento na memória principal.

## VOCÊ QUER VER?

A alocação dinâmica de memória é realizada em uma região denominada "*heap*". O computador, para gerenciar os programas usa, além do "*heap*", a pilha ("*stack*"). O vídeo postado pelo professor da USP de São Carlos, Rodrigo Mello, mostra a diferença entre as duas porções de memória. Você pode assisti-lo neste link <https://www.youtube.com/watch?v=i_0IBJkXn2M>, e aumentar seus conhecimentos sobre o assunto.

O processo de criação de estruturas dinâmicas (vetores ou registros) é feito por intermédio da função "**malloc**" ou "c**alloc**", para a liberação da memória previamente alocada pelo "**malloc**", utiliza-se a função "**free**" ("**malloc**" e "**free**" estão definidas no arquivo header "**stdlib.h**").

## VOCÊ O CONHECE?

Uma das primeiras ideias para a alocação dinâmica de memória surgiu com os manuscritos de Von Neumann, estudado e publicado em 1970, por Donald Knuth, através do artigo "*Von Neumann´s first Computer Program*" (O primeiro programa de computador do Von Neumann). Para saber um pouco mais sobre a importância de Knuth para o mundo dos algoritmos, acesse <https://internacional.estadao.com.br/noticias/nytiw,conheca-o-cientista-considerado-o-guia-espiritual-dos-algoritmos,70002654893>.

Mas, como utilizar as funções "malloc" e "free"? Para responder a esta pergunta, vamos analisar antes as suas sintaxes para depois, darmos uma olhada na sua forma de uso.

**void * malloc(quantidade_de_bytes_a_serem_alocados);**
**void *calloc(quantidade_de_itens, tamanho_de_um_item);**
**void *realloc(ponteiro_a_ser_realocado, quantidade_de_bytes);**
**free(ponteiro_a_ser_desalocado);**

A diferença básica entre as funções "**malloc()**" e "c**alloc()**" consiste no fato de que a "**calloc**", além de alocar espaço da memória, o preenche com o valor 0 (zero). Por sua vez, a função "**realloc()**" tem a função de realocar um espaço de memória. A função retorna um novo ponteiro, cujo tamanho é especificado como parâmetro.

Então como utilizar, efetivamente, as funções "**malloc**", "**calloc**", "**realloc**" e "**free**"? Para isso, vejamos o código a seguir:

```c
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int *vetor;
  char *str;
  int NumCelulas = 5, TamanhoStr = 10;

  //alocando espaco para vetor e para a string
  vetor = (int *) malloc(NumCelulas * sizeof(int));
  str = (char *) calloc(TamanhoStr,sizeof(char));

  //realocando espaco para o vetor - aumentando o vetor em 5 posicoes
  vetor = realloc(vetor,(NumCelulas+5)*sizeof(int));

  //liberando os espacos previamente alocados
  free(vetor);
  free(str);
  return 0;
}
```

No trecho de código acima, podemos encontrar a utilização de alocação dinâmica para a construção de um vetor de inteiros (com a dimensão de "**NumCelulas**") e uma *string* de tamanho "**TamanhoStr**". Nota-se que foram utilizadas as funções "**malloc()**" e "**calloc()**" para a alocação dinâmica inicial de memória. Para realizar o redimensionamento do vetor, após a sua criação, utilizou-se a função "**realloc()**", tomando-se por base o próprio ponteiro "**vetor**" para a sua nova instanciação.

## 3.4 Alocação de matrizes como ponteiros

Assim como os vetores, as matrizes também podem ser manipuladas através da utilização de ponteiros. O processo de criação e manipulação é análogo aos vetores, porém, adicionaremos uma dimensão a mais: vamos lembrar que para a criação de um vetor, são alocados *N*K bytes* na memória, onde *N* representa a dimensão do vetor e *K* o espaço gasto para armazenar um item de um determinado tipo de dado. Por sua vez, para a criação de uma matriz, são alocados *N*M*K bytes*, onde *N* e *M* representam a dimensão da matriz e *K* denota o tamanho de cada item (em função do tipo de dado a ser armazenado).
Desta forma, para acessar um item da matriz, procedemos na forma:

```c
. . .
int *matriz;
. . .
*(pmatriz+i+j) = x;
. . .
```

Para entendermos melhor, vamos analisar o código a seguir:

```
#include <stdio.h>

#define NLINHAS  4                                                    //Linha 1
#define NCOLUNAS 4                                                    //Linha 2

int main()
{
  int matriz[NLINHAS][NCOLUNAS]={{0,1,2,3},{4,5,6,7},                 //Linha 3
                                 {8,9,10,11},{12,13,14,15}},
      *pmat,                                                          //Linha 4
      linha=2,coluna=1;                                               //Linha 5
  pmat = matriz[0];  //ou pmat=&(matriz[0][0]);                       //Linha 6
  *(pmat+linha*NCOLUNAS+coluna) = 13;                                 //Linha 7
  matriz[1][3] = 15;
  for(linha=0; linha<NLINHAS; linha++)
    for(coluna=0; coluna<NCOLUNAS; coluna++)
       printf("M[%d][%d] = %d\t*(pmat+(%d %d))=%d\n",                 //Linha 8
              linha,coluna,matriz[linha][coluna],
              linha,coluna,*(pmat+linha*NCOLUNAS+coluna));
  return 0;
}
```

No código acima, temos as seguintes funcionalidades das linhas marcadas como comentários, clique nos itens e confira.

**Linhas 1 e 2**

Definições de constantes de compilação para facilitar a parametrização da matriz; dessa forma, torna-se mais prático quando necessário alterar as dimensões da matriz pois não se necessita mudar o código no tangente às dimensões da matriz.

**Linha 3**

Criação da matriz do tipo “int” com dimensões NLINHAS x NCOLUNAS e a sua respectiva instanciação.

**Linha 4**

Criação do ponteiro para inteiro que ficará a cargo de referenciar a posição de memória alocada para armazenar a matriz. Além disso, foram instanciadas variáveis que serão os índices manipulados pelos laços de repetição.

**Linha 6**

Instanciação do ponteiro para a matriz denominada como “matriz”. A referência, neste caso, é feita através da indicação do início da linha 0 ou passando o endereço do elemento “matriz[0][0]”.

**Linha 7**

Instanciação do elemento “matriz[2][1]” atribuindo-lhe o valor 13. Nota-se que foi realizado o cálculo “linha*NCOLUNAS+coluna” pelo fato de que uma matriz é transformada em uma estrutura linear para que seja armazenada na memória principal do dispositivo computacional. Essa linearização é realizada por linha, ou seja, o primeiro elemento da segunda linha encontra-se após todos os elementos da primeira e assim sucessivamente. Sendo assim, a cada linha, deve-se saltar NCOLUNAS itens.

**Linha 8**

Impressão do conteúdo das células da matriz. Nota-se, também, a mesma operação "linha*NCOLUNAS+coluna" para o acesso ao conteúdo.

Da mesma forma dos vetores, poderemos alocar as matrizes dinamicamente usando também as funções "**malloc()**" ou "**calloc()**" e liberando o seu espaço com a função "**free()**" conforme as linhas a seguir:

```
int *matriz;
matriz = (int *) calloc(num_linhas*num_colunas , sizeof(int));
. . .
free(matriz);
```

No trecho de código acima temos a criação de forma dinâmica de uma matriz de dimensões "**num_linhas**" por "**num_colunas**" (totalizando "**num_linhas * num_colunas**" células) do tipo "**int**".

Mencionamos, anteriormente, que uma matriz pode ser construída a partir do uso de ponteiros para ponteiros. Mas como seria isso? Para entender, vamos observar o código abaixo:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
  int **matriz, linha=4,coluna=3,i,j;                    //Linha 1
  matriz = (int **)malloc(sizeof(int)*linha);            //Linha 2
  for(i=0; i<linha; i++)
    matriz[i] = (int *)malloc(sizeof(int)*coluna);       //Linha 3
  for(i=0; i< linha; i++)
    for(j=0; j<coluna; j++)
      matriz[i][j]=i*10+j;                               //Linha 4
  for(i=0; i<linha; i++)
    {
      for(j=0; j<coluna; j++)
          printf("%d ",matriz[i][j]);                    //Linha 5
      printf("\n");
    }
  return 0;
}
```

Neste código, temos as linhas numeradas como comentários com algumas funcionalidades; clique nos itens e confira.

**Linha 1**

Declaração das variáveis, destacando-se a variável "matriz" que é do tipo "ponteiro para ponteiro de inteiro".

**Linha 2 e 3**

Na linha 2, ocorre a alocação de espaço de memória para receber um vetor de ponteiros para ponteiros de tamanho "linha". Cada "linha" da matriz, na linha 4 do código, recebe o endereço de um ponteiro que denota um vetor correspondente às colunas.

**Linha 4**
Ocorre a instanciação das células da matriz. Note que podemos utilizar também, colchetes para delimitar os índices da matriz. Foi escolhida a expressão "i*10+j" para que cada célula contenha a referência da coordenada correspondente. Por exemplo, a posição "[3][2]" corresponderá a "3*10+2", ou seja, "32".
**Linha 5**
Contém o código para realizar a impressão na tela das células da matriz.

Essa aplicação de mapeamento de matrizes utilizando-se ponteiro para ponteiro é extremamente útil para a passagem de parâmetro por referência. No caso, uma matriz poderá ser recebida como "**int **matriz**"

## CASO

Um projetista de software recebeu um projeto de um sistema computacional para que fosse implementado, fazia parte deste um problema de otimização para gerenciar recursos energéticos, o chamado *Smart Grid* (Grade Inteligente, na tradução literal). Esses recursos envolvem a parte de geração (por operadores de energia, geração particular nas residências e indústrias) e distribuição de energia. No projeto, ele recebeu um número específico de pontos que deveriam ser alocados em matrizes, o processo de otimização envolve a manipulação de forma massiva, de tais matrizes. Ele teve os seguintes pensamentos: será que a solução dada ao problema poderia ser usada em outras circunstâncias? Será que a quantidade de elementos poderia mudar (para mais ou para menos) em projetos similares futuros? Será que poderiam ser implementadas matrizes esparsas ao invés de matrizes que alocam todos os seus elementos (inclusive os valores nulos) na memória? Informamos aqui, que somente as células não nulas de uma matriz esparsa são alocadas na memória. Então, para resolver esses questionamentos, o projetista implementou o seu sistema usando alocação dinâmica para representar os dados a serem manipulados. Com isso, ele introduziu em seu sistema, o conceito de escalabilidade, que consiste em criar um programa que seja fácil se adequar a outros volumes de dados para o processamento. Essa adequação deve ser realizada com a menor quantidade possível de alterações no código do programa.

Até o momento conversamos apenas sobre a manipulação de estruturas para o armazenamento de informações do mesmo tipo, ou seja, estruturas homogêneas. Mas, seria possível o armazenamento de informações de vários tipos em uma única estrutura? Para isso, existem os registros ou estruturas, que veremos a seguir.

# 3.5 Tipos de dados heterogêneos

Algumas vezes, quando implementamos programas é útil criarmos variáveis que conseguem armazenar vários itens com tipos diferentes de informações. Vamos pensar, que devemos manipular informações de uma pessoa para depois, por exemplo, salvar em um arquivo; ela pode ter atributos tais como: nome, endereço, dia, mês e ano de nascimento. Para o processo de gravação ou recuperação do arquivo, neste caso, manipulamos todas as informações simultaneamente ao invés de efetuarmos a gravação ou recuperação campo por campo. (ASCENCIO, 2012). Mas, como declarar variáveis que admitem informações heterogêneas? É o que veremos agora.

### *3.5.1 Declaração de Variáveis do Tipo Registro*

Como mencionamos a pouco, existe a possibilidade de criarmos estruturas de dados que conseguem armazenar, de forma agrupada, informações heterogêneas. Nomeia-se como dados heterogêneos aqueles constituídos de informações que possuem tipos distintos. Na programação em *C*, esse agrupamento é conseguido através da utilização de estruturas de dados denominadas como "**struct**". Mas, como criar uma "**struct**"? Abordaremos quatro formas para a criação de "**struct**", na figura a seguir:

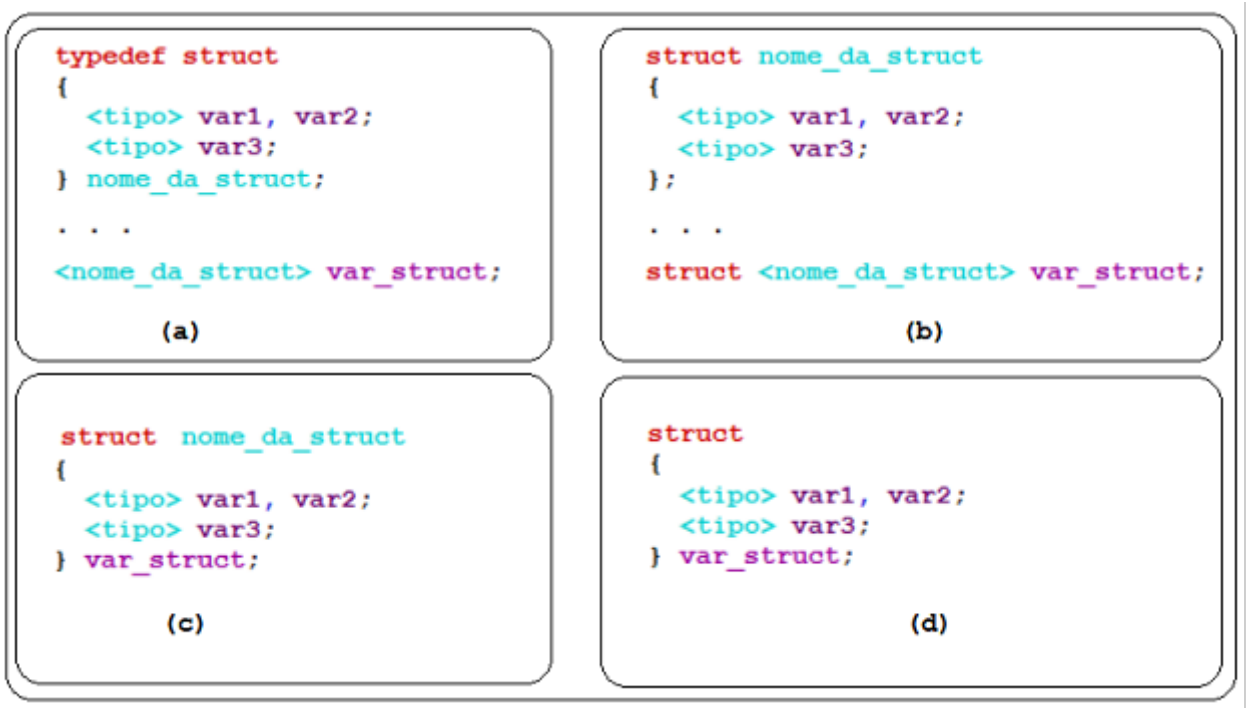


Figura 4 - Formas de declaração e instanciação de registros ("struct"). Nota-se que a "struct" é formada por campos – podendo possuir tipagens distintas.
Fonte: Elaborada pelo autor, 2019.

Temos, na figura acima, em (a), a criação de um novo tipo de dados para representar "**struct**". O termo "**typedef struct**", identifica que estamos criando um novo tipo de dados no formato de estrutura, esse novo tipo foi batizado como "**nome_da_struct**". Ele será usado, posteriormente, para a criação de variáveis na forma "**<tipo> nome_da_variável**", como especificado em "**<nome_da_struct> var_struct**".
Ainda em relação à figura acima, temos em (b), (c) e (d) outras formas de criação de variáveis do tipo "**struct**". Em todas as formas, temos os dois momentos distintos: a criação da "**struct**" e depois, a criação das variáveis que manipularão as informações heterogêneas.
Para melhor ilustrar, vamos analisar o trecho de código abaixo:

```
typedef struct                          //1
{
  char nome[30], ender[50];             //2
  int dia, mês;                         //3
  int ano;                              //4
} PESSOA;                               //5

int main()
{
  PESSOA p;                             //6
  . . .
  return 0;
}
```

No código acima, temos entre as linhas 1 e 5, a criação de um novo tipo de dado do tipo "**struct**" batizado como "**PESSOA**" (linha 5). Os campos que compõem a estrutura integram as linhas 2 a 4. A declaração de uma variável do tipo "**PESSOA**", intitulada "**p**" é realizada na linha 6.
Uma outra forma de se declarar um registro utilizando-se a forma indicada por (b) na figura anterior, seria:

```
struct PESSOA
{
  char nome[30], ender[50];
  int dia, mês, ano;
  int ano;
};

int main()
{
  struct PESSOA p;
  . . .
  return 0;
}
```

Nota-se, no exemplo de código acima, como não foi criado um novo tipo de dado (ausência do "**typedef**"), houve a necessidade da utilização de "**struct**" antes do tipo "**PESSOA**" para a criação da variável "**p**".

# VOCÊ QUER LER?

Nas estruturas, podemos definir os tamanhos dos campos (em bits), desta forma, é possível criar os chamados campos de *bits*; estes são utilizados, sobretudo, na manipulação de sistemas em baixo nível, como por exemplo, para a escrita de *device drivers*, sistemas embarcados e kernel de sistemas operacionais. Para saber mais sobre o assunto, acesse este link < https://docs.microsoft.com/pt-br/cpp/c-language/c-bit-fields?view=vs-2019>.

Até o momento, realizamos a criação de variáveis do tipo "**struct**". Mas como manipulá-las? Como acessar e instanciar os seus campos? Veremos a manipulação de variáveis do tipo "**struct**" a seguir.

## *3.5.2 Acesso aos membros de um registro*

O acesso ao campo se faz de forma igual, tanto para a instanciação quanto para a coleta dos valores de uma variável do tipo "**struct**"; mencionando o nome da variável e o nome do membro da estrutura a ser acessada (esses dois itens separados por um ponto): ***variável.membro***.
Para ilustrar melhor, convido-o para analisar o código a seguir:

```
#include <stdio.h>
typedef struct                      //1
{
  char nome[30], ender[50];         //2
  int dia, mes;                     //3
  int ano;                          //4
} PESSOA;                           //5

int main()
{
  PESSOA p;                         //6
  strcpy(p.nome ,"Abcde");          //7
  scanf("%s",p.ender);              //8
  p.dia = 15;                       //9
  p.mes = 8;                        //10
  scanf("%d",&p.ano);               //11
  . . .
  return 0;
}
```

Como já mencionamos, as linhas 1 a 5 do código acima visam a definição da "**struct**" intitulada como "**PESSOA**", enquanto a linha 6 realiza a criação da variável "**p**" do tipo "**PESSOA**". O acesso aos seus campos podem ser observados nas linhas de 7 a 10. Nota-se o formato "**var.campo**" na manipulação dos campos da estrutura.

## VOCÊ QUER VER?

Uma forma de estruturar e encapsular informações heterogêneas é a base da programação orientada por objetos. Orientação por objeto é a base de muitos sistemas atuais, escritos por exemplo, em C++ e em Java.

Acesse este vídeo <https://www.youtube.com/watch?v=fyvhl_EostE>, para conhecer mais dos conceitos deste paradigma de programação.

Além de facilitar o processo de manipulação de arquivos, a utilização de estruturas deixa o código mais organizado, aumenta o fator de abstração pelos desenvolvedores e, dentre outras vantagens, facilita a implementação quando desejamos passar informações para outros módulos/funções do programa.
Mas, já que mencionamos ponteiros anteriormente, estruturas podem ser mapeadas como ponteiros? A resposta a essa pergunta veremos adiante.

## 3.6 Ponteiros como Registros

Uma questão que poderá ser levantada agora reside no fato de como usar as "**struct**" como ponteiros? Essa é uma informação importante pois poderemos usá-la inicialmente, em dois momentos: um relativo à passagem de parâmetro por referência nas funções e o outro que consiste em alocação dinâmica.
Para o primeiro momento, vamos manipular uma "**struct**" como um ponteiro alocado estaticamente, na forma:

```
typedef struct
{
  char nome[30], ender[50];
  int  dia, mes, ano;
} PESSOA;

int main()
{
  PESSOA Pessoa;
  PESSOA *pPessoa;
  pPessoa = &Pessoa;
  . . .
```

No código acima, podemos visualizar que o ponteiro para o tipo "**PESSOA**", intitulado como "**pPessoa**", recebeu a posição de memória alocada para a variável "**Pessoa**" (**pPessoa = &Pessoa**). Porém, o acesso ao campo tem uma

alteração de sintaxe em relação ao acesso de campos de "**struct**" que não sejam ponteiro. Nesse momento, o acesso se dará na forma: "var->campo" ou "**(*var).campo**". Sendo assim, no trecho de código acima, para o caso desejarmos instanciar o campo "**dia**" do ponteiro "**pPessoa**", teremos que fazer, por exemplo:

**(*pPessoa).dia = 5; //1**

**scanf("%d",&pPessoa->mes); //2**

Nas linhas acima, temos em "1", a instanciação direta do campo "**dia**" do ponteiro "**pPessoa**". Já, na linha 2, o campo "mes" encontra-se passado por referência à função "**scanf**". Nota-se que, nas duas linhas, foram usadas duas formas distintas para o acesso aos membros do registro.

Até o momento, conversamos sobre alocações estáticas, as quais, temos que conhecer previamente a quantidade de informações que serão manipuladas. Veremos a seguir, como podemos realizar alocações dinâmicas e, mais especificamente, como podemos realizar busca, inserção e remoção de registros alocados dinamicamente.

### *3.6.1 Operações sobre ponteiros: busca, inserção e remoção*

Já que falamos sobre ponteiros e registros, que tal praticarmos operações básicas sobre uma lista de registros usando ponteiros? Vamos voltar ao exemplo do registro que denota uma pessoa, e, a partir dele, vamos criar uma lista para armazenar no máximo 10 pessoas. Para facilitar, vamos criar um registro descritor que contém além do ponteiro da lista, um campo que armazena a quantidade de itens armazenados:

```
typedef struct
{
  PESSOA  *pPessoa;
  int QtdPessoas;
}
```

Utilizando a estrutura acima, vamos implementar as funções para realizar o processo de busca, inserção e remoção de registros. Porém, inicialmente, temos que criar uma lista vazia, alocando o espaço para o recebimento das informações relativas a no máximo 10 pessoas:

```
void CriarListaVazia(LISTAPESSOAS *listaP)
{
  listaP->pPessoas = (PESSOA *)malloc(10*sizeof(PESSOA));
  listaP->QtdPessoas=0;
}
```

No código acima, a função "**CriarListaVazia()**" visa a alocação dinâmica do vetor com capacidade para o armazenamento das informações relativas a 10 pessoas e a iniciação, com 0, do campo que indica a quantidade de pessoas armazenadas.

A partir da criação da lista, poderemos, então, realizar as operações básicas. Vamos iniciar pela operação de busca, cuja função é apresentada a seguir:

```
int BuscarRegistro(char *nome,LISTAPESSOAS listaP,PESSOA *pessoa)
{
  int i=0;
  if(listaP.QtdPessoas==0)
    return -1;  //indicativo de lista vazia
  for(; i<listaP.QtdPessoas; i++)
    if(!strcmp(nome,(listaP.pPessoas+i)->nome))
      {
          pessoa = (listaP.pPessoas+i);
          return i;
      }
  return -2; //indicativo de registro nao encontrado
}
```

Como se observa no código acima, o processo de busca é realizado percorrendo-se todo vetor "**pPessoas**" por intermédio da utilização do índice "**i**" somado ao endereço base do ponteiro "**listaP.pPessoas**" de modo a comparar o conteúdo do campo "nome" com o argumento passado como parâmetro `("if(!strcmp(nome,(listaP.pPessoas+i)->nome))")`. Caso seja encontrado, o endereço do item achado é copiado para o ponteiro `"pessoa"` `("pessoa = (listaP.pPessoas+i);")` e retornado o índice no qual encontra-se o elemento encontrado.
Para a inserção na posição "pos", é necessário realizar o deslocamento das informações para a direita a partir da posição na qual ocorrerá a inserção. Esse deslocamento é mostrado através do código a seguir:

```
int InserirRegistro(PESSOA pessoa,LISTAPESSOAS *listaP,int pos)
{
  int i;
  if(listaP->QtdPessoas==10)
    return -1;  //indicativo lista cheia
  if(pos > listaP->QtdPessoas)
    return -2;  //indicativo posicao extrapola limite do vetor
  for(i=listaP->QtdPessoas; i>=pos;i--)
    memcpy((listaP->pPessoas+i),(listaP->pPessoas+i-1),sizeof(PESSOA));
  memcpy((listaP->pPessoas+pos),&pessoa,sizeof(PESSOA));
  (listaP->QtdPessoas)++;
}
```

Na função de inserção acima, nota-se que a cada iteração `"for(i=listaP->QtdPessoas; i>=pos;i--)"` é feita uma cópia da célula "**i - 1**" para a célula "**i**", liberando-se, assim, a célula "**pos**" para receber as informações da pessoa a ser cadastrada `("memcpy((listaP->pPessoas+i),(listaP->pPessoas+i-1),sizeof(PESSOA));")` . Nesta etapa, foi utilizada a função "**memcpy**" (definida no arquivo header "**string.h**") para efetuar a cópia das informações de acordo com o formato: "**memcpy(destino,fonte,quantidade_bytes);**". Além de efetuar o deslocamento das informações dentro do vetor, essa função incrementa o campo que indica a quantidade de itens no vetor `("(listaP->QtdPessoas)++;")`.

Por fim, para realizar a remoção de um item da posição “pos”, pode-se proceder como exemplificado no código a seguir:

```
int RemoverRegistro(LISTAPESSOAS *listaP,int pos)
{
  int i;
  if(listaP->QtdPessoas==0)
    return -1;  //indicativo lista vazia
  if(pos >= listaP->QtdPessoas)
    return -2;  //indicativo posicao extrapola qtd pessoas armazenadas
  for(i=pos; i < listaP->QtdPessoas-1; i++)
    memcpy((listaP->pPessoas+i),(listaP->pPessoas+i+1),sizeof(PESSOA));
  (listaP->QtdPessoas)--;
}
```

Ao contrário da função de inserção, a remoção realiza um deslocamento do conteúdo para a esquerda em cada iteração do laço de repetição. Neste caso, a posição “**i**” recebe o conteúdo da posição “**i + 1**”. Ao final do deslocamento, a quantidade de itens armazenados no vetor é decrementada em uma unidade.

Na prática existem diversas outras estratégias para a manipulação de estruturas usando a alocação dinâmica, dentre as quais podemos citar: listas, filas, pilhas e árvores. (PUGA, 2016). Alocações dinâmicas são extremamente úteis para a criação de códigos escaláveis. Escalabilidade de um programa refere-se à sua facilidade para se adequar a outras dimensões de massa de dados que será manipulada. Além das listas encadeadas, podemos usar a alocação dinâmica em outras estruturações, tais como: listas duplamente encadeadas e árvores.

# Síntese

Chegamos ao fim de nossa terceira conversa sobre técnicas de programação; nesta, pudemos ampliar o universo da programação através da conceituação e experimentação da utilização de ponteiros. Conseguimos verificar o uso de ponteiros nos processos de alocação estática e alocação dinâmica. Com os pontos abordados, você já conseguirá implementar programas mais complexos e torná-los mais eficientes, estruturados e escaláveis através da utilização dos ponteiros. Através dos conteúdos apresentados nesta terceira unidade, esperamos que você continue o processo de incremento em relação à criação de programas computacionais de modo a torná-los ainda mais eficientes, organizados e, agora, escaláveis.

Nesta unidade, você teve a oportunidade de:

- ter contato com conceitos inerentes aos ponteiros;
- analisar as vantagens para a manipulação de ponteiros;
- modelar vetores e matrizes através de ponteiros;
- conhecer e empregar estruturas de dados heterogêneas;
- analisar e utilizar alocações dinâmicas de memória.

# Bibliografia


ASCENCIO, A. F. G. **Fundamentos de Programação de Computadores: Algoritmos, PASCAL, C/C++ (Padrão ANSI) e Java**. 3. ed. São Paulo: Pearson Education do Brasil, 2012. Disponível em: <https://laureatebrasil.blackboard.com/>. Acesso em: 11/07/2019.

DEITEL, P. J.; DEITEL, H. **C: Como Programar**. 6. ed. São Paulo: Pearson Prentice Hall, 2011. Disponível em: < https://laureatebrasil.blackboard.com/>. Acesso em: 11/07/2019.

FERREIRA, D. Como funciona memória: de C a JavaScript (memory management). 2017. 22 min. Disponível em: < https://www.youtube.com/watch?v=82yMCuCY2-k>. Acesso em 11/07/2019.

GARCIA, F. D. **Ponteiro em C**: Funções. 2015. Disponível em: <https://www.embarcados.com.br/ponteiro-em-c-funcoes/>. Acesso em: 03/07/2019.

LACERDA, L.C.; RAMOS, J.M.B. Instituto Federal de Educação, Ciência e Tecnologia. Programação orientada a objetos. Aula 01. Rondônia, 2015. 1h15min. Disponível em: < https://www.youtube.com/watch?v=l-CaVliXaA0>. Acesso em: 11/07/2019.

MELLO, R. **Introdução ao Funcionamento da Memória Stack e Heap**. 11:45min. 2015. Disponível em: < https://www.youtube.com/watch?v=i_0IBJkXn2M>. Acesso em: 14/07/2019.

MICROSOFT. **Campos de Bits C**. 2016. Disponível em: <https://docs.microsoft.com/pt-br/cpp/c-language/c-bit-fields?view=vs-2019>. Acesso em: 11/07/2019.

MIZRAHI, V. V. **Treinamento em Linguagem C**. 2. ed. São Paulo: Pearson Prentice Hall, 2008. Disponível em: < https://laureatebrasil.blackboard.com/>. Acesso em: 11/07/2019.

PUGA, S.; RISSETTI, G. **Lógica de Programação e Estruturas de Dados – com Aplicações em Java**. 3. ed. São Paulo: Pearson Education do Brasil, 2016. Disponível em: <https://laureatebrasil.blackboard.com/>. Acesso em: 11/07/2019.

ROBERTS, S. Conheça o cientista considerado o 'guia espiritual' dos algorítimos. **The New York Times**. Nova York, 25, dez 2018. International Weekly Estadão. Disponível em: < https://internacional.estadao.com.br/noticias/nytiw,conheca-o-cientista-considerado-o-guia-espiritual-dos-algoritmos,70002654893>. Acesso em 11/07/2019.