

# ALGORITMO E LÓGICA DE PROGRAMAÇÃO

Carlos Veríssimo

```
80
81
82 return array(
83     'code' => $captcha_config['code'],
84     'image_src' => $image_src
85 );
86
87
88 if( !function_exists('hex2rgb') ) {
89     function hex2rgb($hex_str, $return_string = false, $separator = ',') {
90         $hex_str = preg_replace("/[^0-9A-Fa-f]/", '', $hex_str); // Get
91         $rgb_array = array();
92         if( strlen($hex_str) == 6 ) {
93             $color_val = hexdec($hex_str);
94             $rgb_array['r'] = 0xFF & ($color_val >> 0x10);
95             $rgb_array['g'] = 0xFF & ($color_val >> 0x8);
96             $rgb_array['b'] = 0xFF & $color_val;
97         } elseif( strlen($hex_str) == 3 ) {
98             $rgb_array['r'] = hexdec(str_repeat(substr($hex_str, 0, 1), 2));
99             $rgb_array['g'] = hexdec(str_repeat(substr($hex_str, 1, 1), 2));
100             $rgb_array['b'] = hexdec(str_repeat(substr($hex_str, 2, 1), 2));
101         } else {
102             return false;
103         }
104         return $return_string ? implode($separator, $rgb_array) : $rgb_array;
105     }
106 }
```

E-book 4

**FAM**  
ONLINE

# Neste E-Book:

<b>INTRODUÇÃO .....</b>	<b>3</b>
<b>MATRIZES E STRINGS .....</b>	<b>4</b>
Vetores Unidimensionais.....	4
Strings.....	9
Matriz .....	15
Ponteiros.....	23
Recursão.....	26
<b>FUNÇÕES .....</b>	<b>32</b>
Declaração de Funções .....	32
Chamada por valor e por referência.....	34
Função com parâmetro do tipo matriz .....	35
O comando return .....	39
Funções do tipo void.....	40
A função main().....	40
Ordenação e Busca.....	41
Quicksort .....	54
Métodos de pesquisa .....	57
Uso de arquivos de texto .....	59
Arquivos.....	59
Abrindo Arquivos.....	61
Fechando um Arquivo.....	62
<b>CONSIDERAÇÕES FINAIS .....</b>	<b>65</b>
<b>SÍNTESE .....</b>	<b>67</b>

# INTRODUÇÃO

Olá, estudante!

Seja bem-vindo a esta unidade, que completa o ciclo de aprendizado na lógica de programação. Trata-se de uma unidade que abordará tópicos mais avançada da lógica de programação, onde aprenderemos a dominar as técnicas de programação profissionais.

Para este fim, abordaremos uma estrutura de dados muito utilizada na programação: Vetores e Matrizes. Abordaremos também a utilização de ponteiros, a ordenação de dados em vetores, métodos de pesquisa em vetores desordenados e vetores ordenados. Por fim, você aprenderá a manipular arquivos, nos quais você poderá acessar, bem como, gravar dados.

Para cada tópico, temos os conceitos e, logo em seguida, um exemplo codificado (na íntegra) e comentado. Neste aspecto, é muito importante que você refaça cada exemplo, em seu ambiente de desenvolvimento (ou no ambiente on-line explicado anteriormente). Seguindo à codificação, temos sempre uma figura que mostra o resultado do referido código.

Vamos em frente, espero que você tenha um excelente aproveitamento deste maravilhoso mundo da programação de computadores.

Bons Estudos!

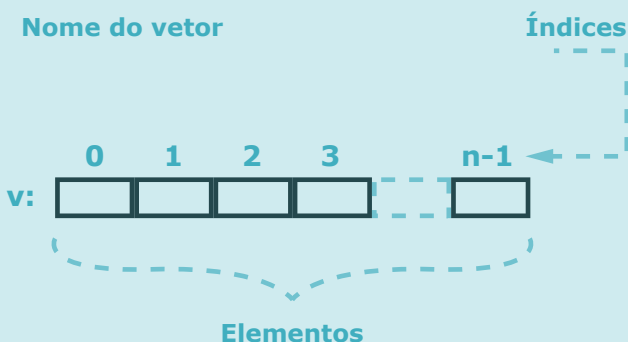
# MATRIZES E STRINGS

Matriz é um dos mais simples e importantes tipos agregados de dados que podemos utilizar em muitas linguagens de programação. Por meio do seu uso, podemos armazenar e manipular grandes quantidades de dados.

## Vetores Unidimensionais

Uma matriz ou vetor é uma coleção de variáveis de um mesmo tipo, que compartilham o mesmo nome e que ocupam posições consecutivas de memória, em que o endereço mais baixo corresponde ao primeiro elemento e o mais alto, ao último elemento. Um elemento específico em um vetor é acessado por meio de um índice.

Se **v** é um vetor com **n** posições, seus elementos são  $v[0]$ ,  $v[1]$ ,  $v[2]$ , ...,  $v[n-1]$ , conforme podemos observar a representação esquemática na Figura 1.



**Figura 1:** Representação esquemática de um vetor. **Fonte:** Elaboração Própria

## FIQUE ATENTO

Em **C** os vetores são sempre indexados a partir de zero e, portanto, o último elemento de um vetor de tamanho **n** ocupa a posição **n-1** do vetor.

Para criar um vetor, basta declarar uma variável com sufixo [**n**], sendo **n** uma constante indicando o número de elementos a serem alocados no vetor.

Em **C** a sintaxe para a declaração de um vetor é:

**tipo** nome\_var [tamanho];

Onde: **tipo** declara o **tipo** de base do vetor, que é o tipo de cada elemento do vetor; **nome\_var** é o nome pelo qual faremos referência ao vetor; **tamanho** define quantos elementos o vetor irá guardar.

Por exemplo: para declarar um vetor de 100 elementos, chamado *amostraTemperatura* e do tipo decimal (*double*), utilizamos o seguinte comando em linguagem **C**:

```
double amostraTemperatura[100];
```

A quantidade de armazenamento necessário para guardar um vetor está diretamente ligado com o seu tamanho e seu tipo. Para um vetor unidimensional, o tamanho total em bytes é calculado pela fórmula:

total em bytes = sizeof(tipo) \* tamanho do vetor

Vetores unidimensionais são, essencialmente, listas de informações do mesmo tipo, que são armazenadas em posições contíguas de memória. Partindo-se do princípio que um programa, ao ser alocado em memória, ocupa um espaço a partir de um determinado endereço de memória, vamos supor que declaremos um vetor chamado **notas** de 5 posições. Vamos supor que o início deste vetor esteja no endereço 1000 de memória, observe na Figura 2 como este vetor ocupara o espaço.

Elemento	→	nota[0]	nota[1]	nota[2]	nota[3]	nota[4]
Endereço	→	1000	1001	1002	1003	1004

**Figura 2:** Representação na memória do vetor de 5 elementos. **Fonte:** Elaboração Própria.

## Exemplos de manipulação de vetores unidimensionais

Vetores são automaticamente zerados pelo compilador. Mas, se for desejado, podemos inicializá-los explicitamente no momento em que os declaramos. Nesse caso, os valores iniciais devem ser fornecidos entre chaves e separados por vírgulas (Linha 4). O código abaixo ilustra esta situação:

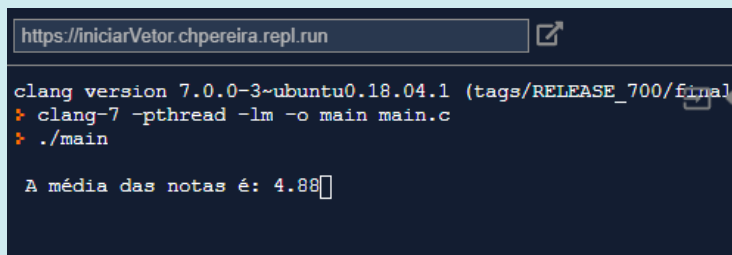
```
1  #include <stdio.h>
2  int main(void)
3  {
4      float notas[4] = {4.5, 5.0, 4.0, 6.0};
5      float soma = 0;
6      for (int indice; indice < 4; indice++)
```

```

7          // Laço de repetição para acessar as 4 posições do
          vetor
8      {
9          soma = soma + notas[indice];
10     }
11     printf("\n A média das notas é: %.2f", (soma / 4));
12     return 0;
13 }

```

Observe na Figura 3 o resultado do código. Note que a média foi obtida a partir dos dados que constam no **vetor notas**.



```

https://iniciarVetor.chpereira.repl.run
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
clang-7 -pthread -lm -o main main.c
./main

A média das notas é: 4.88

```

**Figura 3:** Resultado da execução do código que exemplificou a inicialização de vetor. **Fonte:** Elaboração Própria.

Agora vamos exemplificar como incluir valores no vetor de **forma dinâmica**. Partindo do exemplo anterior, o programa abaixo irá solicitar 4 notas; armazená-las em um vetor e obter a média aritmética das notas, mostrando na tela.

Note que foram utilizados dois laços de repetição **for**: O laço da **linha 8** obtém as notas a partir da digitação, pelo teclado, as coloca no vetor "**notas**". O laço da **linha 14** acessa cada posição do vetor "**notas**" do vetor notas para fazer o cálculo da média.

```

1  #include <stdio.h>
2  int main(void)
3  {
4      float notas[4]; //vetor para receber as notas digitadas
5      float soma = 0;
6      printf("\n Exemplo de inicialização dinâmica de vetor");
7      printf("\n *-----*");
8      for (int indice1=0;indice1 < 4; indice1++)
9          //laço repetição para solicitar 4 notas
10         {
11             printf("\ninforme a nota %d = ",indice1+1);
12             scanf("%f",&notas[indice1]); //Le do teclado a opcao
13         }
14         for (int indice2=0;indice2<4;indice2++)
15             // Laço de repetição para acessar as 4 posições do
            vetor – Acumula na variável soma cada nota do vetor
16         {
17             soma = soma + notas[indice2];
18         }
19         printf("\n A média das notas é: %.2f", (soma / 4 ));
20         return 0;
21     }

```

Observe na Figura 4 o resultado do código que inclui valores de forma dinâmica no vetor.



<https://inicializaVetor2.chpereira.repl.run>

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
❯ clang-7 -pthread -lm -o main main.c
❯ ./main

Exemplo de inicialização dinâmica de vetor
*-----*
informe a nota 1 = 4.5

informe a nota 2 = 5.0

informe a nota 3 = 4.0

informe a nota 4 = 6.0

A média das notas é: 4.88
```

**Figura 4:** Resultado da execução do código que exemplificou a inicialização dinâmica de vetor. **Fonte:** Elaboração Própria.

## FIQUE ATENTO

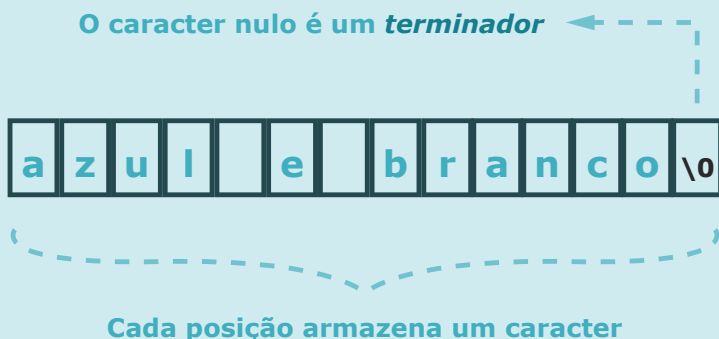
Note que nos dois exemplos acima, ao manipular o vetor “**notas**”, os programas inicializaram seus respectivos **índices** com valor **0** (zero).

Lembre-se: a primeira nota (nos dois exemplos) está na **posição 0** (zero) do vetor “**notas**”.

## Strings

O uso mais comum de vetores unidimensionais é como string de caracteres. A string é talvez uma das mais importantes formas de armazenamento de dados na maioria das linguagens de programação. Em **C**, entretanto, ela não é um tipo de dado básico, pois uma string é uma série de caracteres terminada com um caractere nulo, representado por ‘\0’.

Na forma de uma constante, a **string** aparece como uma série de caracteres delimitada por aspas; como por exemplo, “azul e branco”. **Internamente**, essa **string** é armazenada conforme ilustrado na Figura 5:



**Figura 5:** Armazenamento de uma string em memória.  
**Fonte:** Elaboração Própria.

Devido à necessidade do ‘\0’, os vetores que armazenam **strings** devem ter sempre uma posição a mais do que o número de caracteres a serem armazenados.

## SAIBA MAIS

O caracter nulo ‘\0’ é o primeiro da tabela ASCII e tem código igual a **zero**.

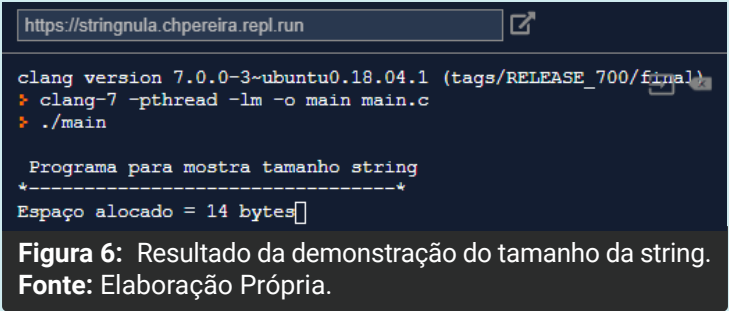
Cuidado para não confundi-lo com o caracter ‘0’, que tem código ASCII 48.

No código abaixo, podemos comprovar a quantidade de bytes alocados para a **string** “azul e branco”. Se contarmos a quantidade de caracteres, temos **13 caracteres** (Os espaços também são caracteres).

Porém, o compilar gera automaticamente o tamanho de 15, pois, o caractere \0 foi colocado automaticamente ao final.

```
1  #include <stdio.h>
2  int main(void) {
3      printf("\n Programa para mostra tamanho string");
4      printf("\n *-----*");
5      printf("\n Espaço alocado = %ld bytes", sizeof("azul e branco"));
6  }
```

Observe na Figura 6 o resultado obtido no programa, note que o tamanho obtido foi 14, e não 13 (que é a quantidade de caracteres digitadas na **string**)



**Figura 6:** Resultado da demonstração do tamanho da string.  
**Fonte:** Elaboração Própria.

## Manipulação de strings

A linguagem C disponibiliza funções de manipulação de strings, conforme listada na Tabela 1:

Função	Objetivo	Exemplo	Comentário
strcpy	Copiar strings	strcpy(s1, s2)	Copia s2 em s1

Função	Objetivo	Exemplo	Comentário
strcat	Concatenar strings	strcat(s1, s2)	Concatena s2 ao final de s1
strlen()	Descobrir o tamanho de uma string	strlen(s1)	Retorna o tamanho da string s1
strcmp()	Comparar strings	strcmp(s1, s2, tamanho)	Retorna 0 se s1 e s2 forem iguais;  Menor que 0 se s1 < s2 e Maior que 0 se s1 > s2

**Tabela 1:** Funções para manipulação de string. **Fonte:** Elaboração Própria.

Para utilização dessas funções, faz-se necessário a inclusão da biblioteca *string*, utilizando-se o comando `#include <string.h>`.

Temos, abaixo, um código que exemplifica a utilização da função que faz a cópia de *strings*(**strcpy**) e retorna o tamanho da string(**strlen**):

```

1  #include <stdio.h>
2  #include <string.h> //necessário para strcpy
3  int main (void)
4  {
5      char nome[15];
6      strcpy(nome, "Fulano de Tal");
7      // strcpy(string_destino, string_origem);
8      //note que a string de destino é nome
9      //a string de origem é "Fulano de Tal"
10     printf("\n Exemplo de cópia de string");
11     printf("\n *-----");
12     printf("\n Nome = %s", nome);

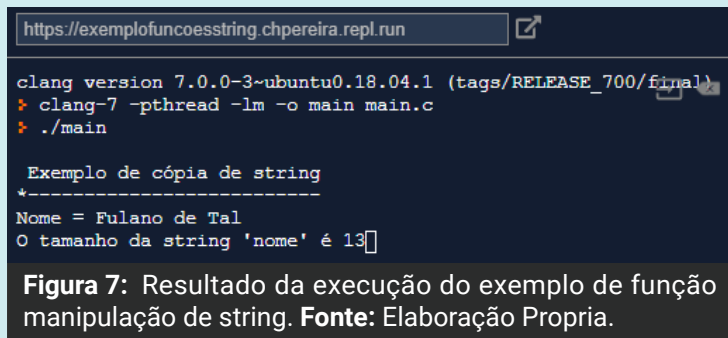
```

```

14         printf("\n O tamanho da string 'nome' é %li",strlen(nome));
15         return 0;
16     }

```

Observe na Figura 7 o resultado desta execução:



**Figura 7:** Resultado da execução do exemplo de função manipulação de string. **Fonte:** Elaboração Propria.

## ■ Comparação entre duas strings

Como a *string* não é um tipo de dados básico da linguagem **C**, operações simples, como atribuição e comparação, não podem ser feitas diretamente com os operadores disponíveis. A comparação entre duas *strings* pode ser feita de duas formas: **1-)** Percorrendo os vetores e comparar seus caracteres correspondentes, um a um ou **2-)** utilizar a função **strcmp()**.

O código abaixo faz a comparação entre duas *strings* **str1** e **str2**. Para isso, utilizamos a função **strcmp()** – observe a linha 10. Note que a estrutura do comando: **strcmp(s1, s2, tamanho)**, onde o campo “*tamanho*” é a quantidade caracteres comparados. Em nosso caso, foram comparados os 8 caracteres.

```

1     #include <stdio.h>

```

```

2  #include <string.h>
3  int main ()
4  {
5      char *str1 = "banana2";
6      char *str2 = "banana1";
7      int ret;
8      printf("\n Manipulação de String : Comparação entre 2
strings");
9      printf("\n
*-----*");
10     ret = strncmp(str1, str2, 8); //Compara as 8 posições das
duas strings
11     if(ret > 0)
12     {
13         printf("\n str1 é maior");
14     }
15     else if(ret < 0)
16     {
17         printf("\nstr2 é maior");
18     }
19     else
20     {
21         printf("\nAs duas palavras são iguais");
22     }
23     return(0);
24 }

```

Observe o resultado do código analisado na Figura 8. A variável **str1** é maior que a variável **str2**, pois a função comparou caractere por caractere e detectou diferença na oitava posição. Importante destacar que, se a comparação fosse somente das 7 posições, o resultado sairia pela igualdade (linha 21 do programa)

<https://stringcompara.chpereira.repl.run>

```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang-7 -pthread -lm -o main main.c
> ./main
```

Manipulação de String : Comparação entre 2 strings

\*-----\*

str1 é maior

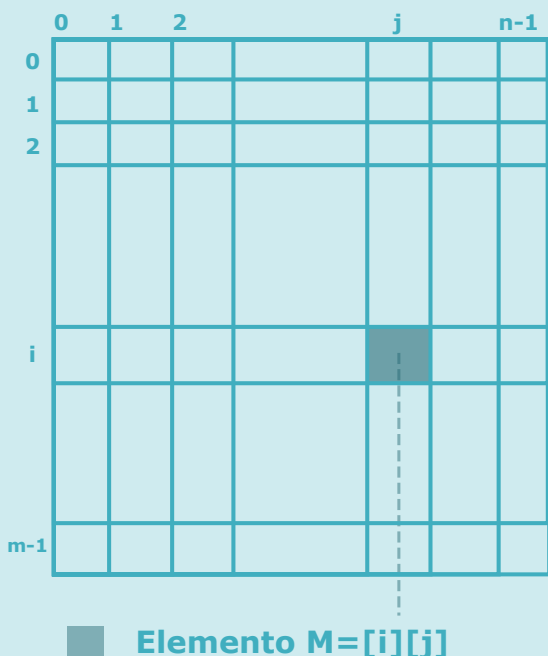
**Figura 8:** Execução do programa que compara duas strings.

**Fonte:** Elaboração Própria.

## Matriz

Uma matriz é uma coleção **homogênea** bidimensional, cujos elementos são distribuídos em linhas e colunas. Para ter acesso aos elementos da matriz, devemos considerar que se **M** é uma matriz **m×n**, então, podemos indexar suas linhas de 0 a **m-1** e suas colunas de 0 a **n-1**. Portanto para acessar um elemento em particular de **M**, escrevemos **M[i][j]**, onde **i** é o número da linha e **j** é o número da coluna que o elemento ocupa.

Podemos ilustrar o acesso a um elemento na Figura 9 na qual notamos que, para acessar o **elemento em destaque** (cor escura), basta obter as coordenadas compostas pela posição da linha **i** com a posição da linha **j**, onde obtemos **M(i,j)**, ou na notação **M[i][j]**. Exemplo: Supondo a situação em que **i=5** e **j=6**. Então para acessar este elemento em destaque, basta indexar **M=[5][6]**.



**Figura 9:** Representação de uma matriz bidimensional.  
**Fonte:** Elaboração Própria.

## FIQUE ATENTO

Do ponto de vista técnico, a linguagem não suporta diretamente matrizes e, para criar uma matriz, devemos declarar um vetor cujos elementos são vetores, ou seja, vetores de vetores unidimensionais.

Em **C** a sintaxe para a declaração de uma matriz bidimensional é:

**tipo** nome\_var [tamanho de i] [tamanho de j];



Onde: **tipo** declara o **tipo** de base da matriz, que é o tipo de cada elemento do vetor; **nome\_var** é o nome pelo qual faremos referência à matriz; **tamanho de i** define quantos elementos o **vetor i** irá guardar e **tamanho de j** define quantos elementos o **vetor j** irá guardar.

## ■ Inicialização de Matriz

Vamos imaginar a seguinte situação:

Você deverá produzir um programa para mostrar as 4 maiores temperaturas do último trimestre, cujo cenário é exemplificado na Tabela 2.

	temperatura1	temperatura2	temperatura3	temperatura4
mês 1	25.4	27.09	28.90	29.50
mês 2	20.05	20.90	21.8	22.90
mês 3	17.70	18.8	19.00	20.10

**Tabela 2:** Maiores Temperaturas registradas nos últimos três meses **Fonte:** Elaboração Própria.

## Solução:

Vamos declarar a matriz que deverá CONTER a tabela com as seguintes características: Uma matriz com 3 linhas e 4 colunas, chamada *amostraTemperaturaTrimestral*.

Abaixo temos a codificação do programa. Pedimos uma **especial atenção** às **linhas 5, 6, 7 e 8**, pois são exatamente as linhas onde se localizam a especificação da matriz dimensional desejada, cujo nome para referência é **amostraTemperaturaTrimestral**,

e que foi declarada para ter **3 colunas** [3] e **4 linhas** [4]. Observe também que, no mesmo comando, inicializamos a matriz com os dados desejadas (tabela de temperaturas). Para tal, abrimos chave { (*linha 6*), para conter o conjunto de vetores com as 4 temperaturas de cada mês. Na *linha 8* fechamos chave }, onde termina a especificação da matriz.

Para acessar os dados da matriz, utilizamos dois índices, **índice\_i** e **índice\_j**, que apontam respectivamente para **linha** e **coluna** da matriz.

Para **percorrer as linhas** da matriz, declaramos o laço de repetição **for** na **linha 11**. Note que, na declaração deste laço, limitamos **índice\_i** a 3, pois, conforme especificação, correspondem aos 3 últimos meses.

Para **percorrer as colunas** da matriz, declaramos o laço de repetição **for** na **linha 15**. Note que na declaração deste laço, limitamos **índice\_i** a 4, pois, conforme especificação, correspondem aos 3 últimos meses. Observe também que tivemos a preocupação de identificar, de forma separada, cada mês de referência (*linha 14*); esta informação é obtida a partir do próprio índice(**índice\_i**). Também identificamos a sequência de temperatura a partir do índice de coluna (**índice\_j**).

```



1  #include <stdio.h>
2  int main(void)
3  {
4      //vetor para receber as notas digitadas
5      float amostraTemperaturaTrimestral[3][4] =
6          {{25.4, 27.09, 28.90, 29.50},
7           {21.8, 20.05, 22.90, 20.90},
8           {18.8, 20.10, 17.70, 19.00}}
9      printf("\n Exemplo de acesso a Matriz Bidimensional");
10     printf("\n *-----*");
11     for (int indice_i=0;indice_i < 3; indice_i++)
12     {
13         //laço repetição para controlar as 3 LINHAS da
matriz
14         printf("\n *-----Temperaturas do Mes : %d
-----*",indice_i+1);
15         for (int indice_j=0;indice_j < 4; indice_j++)
16         {
17             //laço repetição para controlar as 4 COLUNAS
da matriz
18             printf("\n Temperatura %d = %.2f", indice_j+1,
amostraTemperaturaTrimestral[indice_i][indice_j]);
19         }
20     }
21     return 0;
22 }

```

## FIQUE ATENTO

Na declaração dos dois **laços de repetição**, foram iniciados seus respectivos índices com **valor 0** (zero), pois a posição “0” do vetor corresponde à primeira posição. Exemplo amostraTemperaturaTrimestral [0][0] corresponde à primeira temperatura do primeiro mês (No nosso caso é o valor 25.4)

Observe o resultado da execução do programa na Figura 10:

```
https://matrizTemperatura.chpereira.repl.run   
  
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)   
❖ clang-7 -pthread -lm -o main main.c  
❖ ./main  
  
Exemplo de acesso a Matriz Bidimensional  
*-----*  
*-----Temperaturas do Mes : 1 -----*  
Temperatura 1 = 25.40  
Temperatura 2 = 27.09  
Temperatura 3 = 28.90  
Temperatura 4 = 29.50  
*-----Temperaturas do Mes : 2 -----*  
Temperatura 1 = 21.80  
Temperatura 2 = 20.05  
Temperatura 3 = 22.90  
Temperatura 4 = 20.90  
*-----Temperaturas do Mes : 3 -----*  
Temperatura 1 = 18.80  
Temperatura 2 = 20.10  
Temperatura 3 = 17.70  
Temperatura 4 = 19.00□
```

**Figura 10:** Resultado da execução do programa. **Fonte:** Elaboração Própria.

## ■ Inicialização Dinamicamente de Matriz

Para inicializar de **forma dinâmica** uma matriz, é necessário que utilizemos dois laços de repetição: O **laço externo** controla as **linhas**, enquanto o **laço interno** controla as **colunas**. Demonstramos esta operação, tendo como base nosso último programa desenvolvido, com a seguinte adaptação: As temperaturas agora são digitadas (informadas pelo teclado). Observe o código abaixo e seus comentários logo em seguida:

```
1  #include <stdio.h>
2  //variáveis GLOBAIS
3  //vetor para receber as temperaturas digitadas
4  float amostraTemperaturaTrimestra[3][4];
5  void entradaTemperaturas()
6  {
7      for (int indice1=0;indice1<3;indice1++)
8      {
9          for(int indice2=0;indice2<4;indice2++)
10         {
11             printf("\n Informe a temperatura #%d para o
MES #%d: ",indice2+1,indice1+1);
12             scanf("%f", &amostraTemperaturaTrimestra[in
dice1][indice2]);
13         }
14     }
15 }
16 int main(void)
17 {
18     entradaTemperaturas();
19     printf("\n Exemplo de acesso a Matriz Bidimensional");
```

```

20         printf("\n *-----*");
21         for (int indice_i=0;indice_i < 3; indice_i++)
22         {
23             printf("\n *-----Temperaturas do Mes : %d -----*",
                indice_i+1);
24             //laço repetição para solicitar 4 notas
25             for (int indice_j=0;indice_j < 4; indice_j++)
26             {
27                 printf("\n Temperatura %d = %.2f", indice_j+1,
                amostraTemperaturaTrimestra[indice_i][indice_j]);
28             }
29         }
30         return 0;
31     }

```

## Comentários:

O programa possui uma função **entradaTemperaturas()** que solicita a entrada das quatro temperaturas. Note que esta função é o primeiro comando (**linha 18**) da função **main()**.

Na função de entrada de temperatura temos dois laços de repetição:

- O comando de repetição da **linha 7** – Controla as linhas da matriz (de 1 a 3 meses)
- O comando de repetição da **linha 9** – Controla as colunas da matriz (de 1 a 4 temperaturas)

Depois que termina o ciclo de entrada de dados, o programa volta ao seu ciclo, na **linha 19**. Em seguida, na **linha 21**, temos o comando para controle do laço de repetição dos meses que contêm as temperaturas.

Na **linha 25**, temos o controle do laço de repetição da impressão das quatro temperaturas.

## Ponteiros

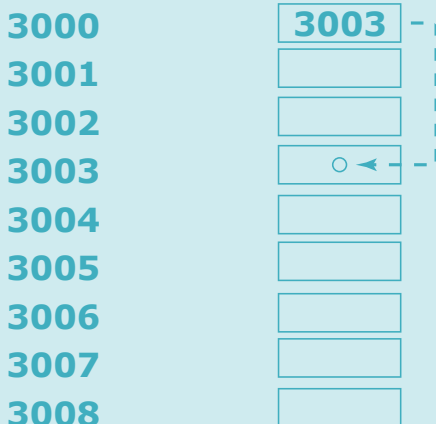
O correto entendimento do uso de ponteiros é muito crítico para que tenhamos sucesso em programação na linguagem **C**, pois a utilização de ponteiros aumenta a eficiência de certas rotinas, usam-se ponteiros em alocação de dinâmica de memória e finalmente, com o uso de ponteiros as funções podem modificar seus argumentos.

### ■ O Que São Ponteiros?

Um ponteiro é uma variável que contém uma referência a um endereço de memória. Este endereço é normalmente a posição de uma outra variável na memória, conforme podemos observar na Figura 11, na qual podemos notar a variável, que está no endereço de memória 3000, apontando para a variável que está no endereço 3003.

## Endereço na Memória

## Variável na Memória



**Figura 11:** Representação do uso de ponteiros – Uma variável aponta para o endereço de outra. **Fonte:** Elaboração Própria.

## Variáveis de Ponteiros

Para uma variável conter um ponteiro, esta deve ser declarada como tal, ou seja, tipo, o caráter \* (asterisco) e o nome da variável:

*tipo \* nome;*

Onde: *tipo* é qualquer tipo válido em **C** e *nome* é o nome da variável ponteiro. Exemplo: *char \*p;*

## O Operadores de Ponteiros

O **C** possui dois operadores especiais de ponteiros, representados pelos caracteres \* e &. O operador & é um operador unário que devolve o endereço na memória de seu operando. Por exemplo: *somaParcial*



= **&soma**; coloca na variável *somaParcial* o endereço da memória que contém a variável *soma*. A variável *somaParcial* **não recebe** o valor que contém a variável *soma*. Devemos entender esta operação da seguinte forma: *somaParcial* **recebe** o endereço de *soma*.

Por exemplo: Supondo que a variável *soma* utiliza a posição de memória 3003 para armazenar seu valor, por exemplo 4500. Logo, após a atribuição feita em nosso exemplo, *somaParcial* terá o valor 3003 (endereço de memória), e não 4500 (valor da variável).

## FIQUE ATENTO

Se o **tipo** de um ponteiro **p** é diferente do tipo de uma variável **v**, então, o endereço de **v** não deve ser atribuído a **p**, ou seja, **p** e **&v** **não são compatíveis de atribuição**.

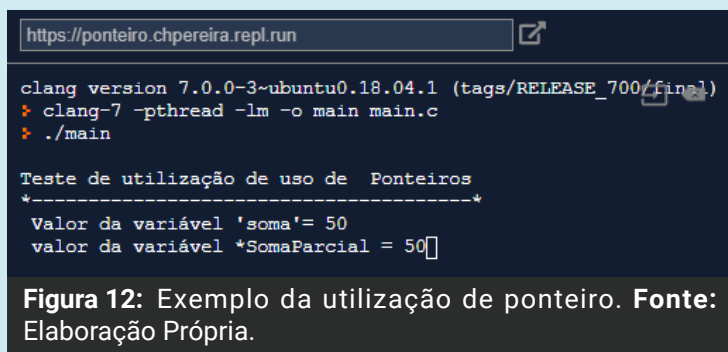
Temos, no código abaixo, um exemplo da utilização de ponteiros. Note que na linha 5 fazemos a declaração da variável ponteiro **\*somaParcial**. Importante destacar que esta variável não contém o valor 50 (soma de 20+30). Ela contém o endereço de memória da variável **soma** (Esta sim contém o valor 50). Portanto, o comando *printf* da variável **\*somaParcial** (linha 9) o resultado que está no endereço apontado pela variável **soma**.

```

1  #include <stdio.h>
2  int main(void)
3  {
4      int soma = 20 + 30;
5      int *somaParcial = &soma;
6      printf("\nTeste de utilização de uso de Ponteiros");
7      printf("\n*-----*");
8      printf("\n Valor da variável 'soma' = %i", soma);
9      printf("\n valor da variável *SomaParcial = %i",
10     *somaParcial);
11     return 0;
12 }

```

Temos, na Figura 12, o resultado deste código:



```

https://ponteiro.chpereira.repl.run
clang version 7.0.0-3-ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang-7 -pthread -lm -o main main.c
> ./main

Teste de utilização de uso de Ponteiros
*-----*
Valor da variável 'soma' = 50
valor da variável *SomaParcial = 50

```

**Figura 12:** Exemplo da utilização de ponteiro. **Fonte:** Elaboração Própria.

## Recursão

Até o momento, tratamos de programas estruturados com funções que chamam funções de forma hierárquica, ou seja, uma função que é chamada, ao terminar, devolve o controle para a função que a chamou. Para muitos problemas, é útil ter as funções a chamar umas às outras. Uma **função recursiva** é

uma função que chama a si mesma. A recursividade é um princípio que nos permite obter a solução de um problema a partir da solução de uma instância menor de si mesmo.

Toda recursividade é composta por um **caso base** e pelas **chamadas recursivas**, onde

- **Caso base:** é o caso mais simples. É usada uma condição em que se resolve o problema com facilidade;
- **Chamadas Recursivas:** procuram simplificar o problema de tal forma que convergem para o caso base.

Um exemplo prático é um cálculo matemático popular: O fatorial de um número **n** inteiro não negativo, escrito **n!** (e pronunciado com “**n fatorial**”), é o produto

$$n.(n-1).(n-2)...1$$

Com 1! Igual a 1 e 0! Definido como 1. Por exemplo, 5! É o produto

$$5 . 4 . 3 . 2 . 1 = 125$$

Lembrando que o fatorial de um número pode ser resolvido **iterativamente** (não recursivamente) pelo laço de repetição **for**, conforme demonstrado no quadro abaixo:

```
numero =5;
fatorial = 1;
for (int contador = numero;contador>=1;contador--)
    fatorial *= contador;
```

Porém, nosso foco é demonstrar uma solução utilizando a recursividade. Portanto, chegaremos a uma definição recursiva da função fatorial observando o seguinte relacionamento:

$$n! = n \cdot (n-1)!$$

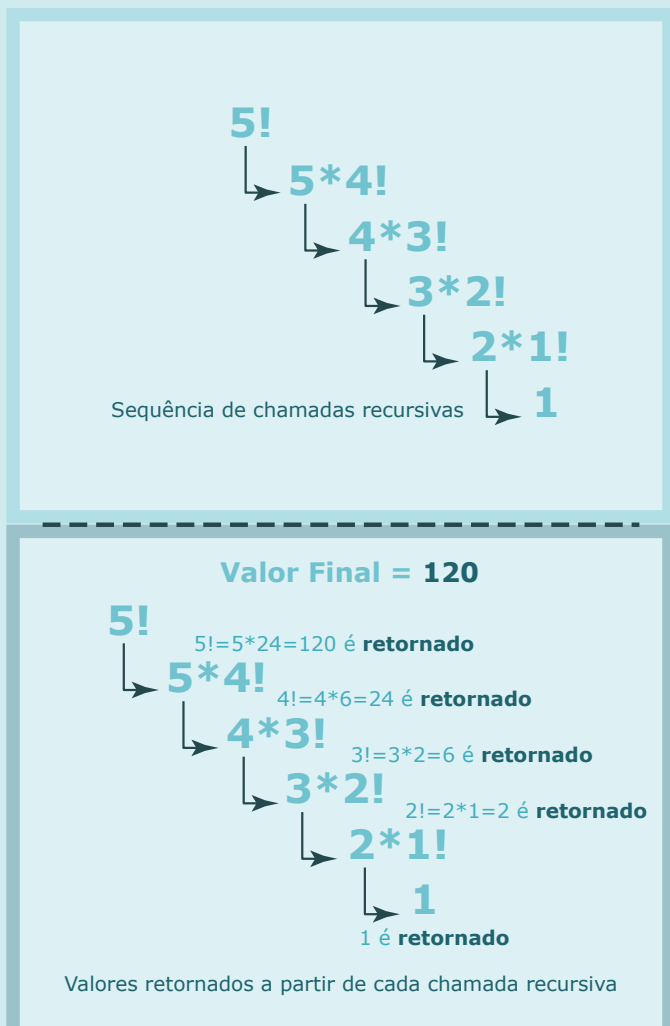
Por exemplo:  $5!$  É equivale à relação Como mostrado abaixo:

$$5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1$$

$$5! = 5 \cdot (4 \cdot 3 \cdot 2 \cdot 1)$$

$$5! = 5(4!)$$

Observe, na Figura 13, a avaliação de  $5!$  em que a sucessão de chamadas recursivas prossegue até  $1!$ , que é avaliado como 1, o que termina a recursão.



**Figura 13:** Avaliação recursiva de 5! **Fonte:** Elaboração Própria.


Agora, acompanhe o código **C** que implementa lógica de recursividade:

```

1 //Cálculo de fatorial com função recursiva
2 #include <stdio.h>
3 //Função recursiva que calcula o fatorial
4 //de um numero inteiro n
5 double fatorial(int entrada)
6 {
7     double vfatorial;
8     printf("\n Sequencia de Chamadas recursivas = %d",entrada);
9     if ( entrada <= 1 )
10         //Caso base: fatorial de n <= 1 retorna 1
11         return (1);
12     else
13     {
14         //Chamada recursiva
15         vfatorial = entrada * fatorial(entrada - 1);
16         printf("\nValores Retornados = %2.f",vfatorial );
17         return (vfatorial);
18     }
19 }
21 int main(void)
22 {
23     int numero;
24     double f;
25     printf("Digite o numero que deseja calcular o fatorial: ");
26     scanf("%d",&numero);
27     printf("\n Programa Cálculo de fatorial");
28     printf("\n *-----*");
29     //chamada da função fatorial
30     f = fatorial(numero);
31     printf("\nFatorial de %d = %.0lf",numero,f);
32     return 0;
33 }

```

Observe, na Figura 14, o resultado do programa, em que a saída reflete exatamente o conceito estudado, ou seja, a sequência de chamadas recursivas e os valores retornados a partir de cada chamada recursiva.

```
https://recursividade.chpereira.repl.run   
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)  
❯ clang-7 -pthread -lm -o main main.c  
❯ ./main  
Digite o numero que deseja calcular o fatorial: 5  
  
Programa Cálculo de fatorial  
*-----*  
Sequencia de Chamadas recursivas = 5  
Sequencia de Chamadas recursivas = 4  
Sequencia de Chamadas recursivas = 3  
Sequencia de Chamadas recursivas = 2  
Sequencia de Chamadas recursivas = 1  
Valores Retornados = 2  
Valores Retornados = 6  
Valores Retornados = 24  
Valores Retornados = 120  
Fatorial de 5 = 120
```

**Figura 14:** Resultado da execução do programa de recursividade. **Fonte:** Elaboração Própria.

## FIQUE ATENTO

Para ser útil, uma função recursiva deve ter um ponto de parada, ou seja, deve ser capaz de interromper as chamadas recursivas e executar em tempo finito.

Para saber mais sobre vetores, acesse o **Podcast** “Por dentro dos vetores”.

**Acesse o Podcast 1 em Módulos**

# FUNÇÕES

Funções (também chamada de sub-rotinas) são blocos de códigos escritos na linguagem **C**. As funções são a alma da programação em **C**, pois é onde ocorrem todas as atividades do programa. Podemos afirmar que um programa em **C** é uma sequência de execução de funções.

## Declaração de Funções

A forma geral da declaração de uma função é:

```
especificador_de_tipo nome_da_função(lista de parâmetros)  
{  
    corpo da função  
}
```

O **especificador\_de\_tipo** especifica o tipo de valor que o comando **return** da função devolve, sendo que deve ser qualquer tipo de dado válido em **C**. Se nenhum tipo for especificado, então o compilador assume que a função irá devolver um valor inteiro.

O **nome\_da\_função** deverá ser uma identificação válida para a linguagem **C**. A chamada da função se dará por este nome.

A lista de parâmetros é uma lista de variáveis, precedidas de seus respectivos **tipos de dados**. A função pode não ter parâmetros, neste caso a lista é



declarada vazia com os símbolos () – abre e fecha parênteses.

Na Tabela 3 temos três exemplos de declaração de função. Observe a particularidade de cada declaração:

Função	Breve descrição da função (Hipoteticamente)	retorno	Lista de Parâmetros	Particularidades
void mostraDados();	Mostrar dados específicos em tela	não	não	1 - Lista de parâmetros de entrada vazia 2 - função não devolve nada ao final 3 - Emite somente comando return;
int mostraSomaValores (int valor1, int valor2);	Faz a soma de dois valores inteiros informados	O resultado da soma	valor1 valor2	1 - Necessita de dois valores de entrada 2 - Função devolve valor tipo int 3 - Ao final emite comando return valorx;
void atualizaSaldo (float valorSaldo);	Atualiza o Saldo no BD	não	valorSaldo	1 - Lista de parâmetros de entrada: 1 valor tipo float 2 - função não devolve nada ao final 3 - Emite somente comando return;

**Tabela 3:** Exemplos de utilização de função. **Fonte:** Elaboração Própria.

## Chamada por valor e por referência

O **C** aceita a passagem de argumentos entre funções de duas maneiras: **Chamada por valor** e **chamada por referência**.


A chamada por valor, a mais utilizada pelo **C**, copia um valor do argumento no parâmetro formal da sub-rotina. Já a chamada por referência, o endereço de um argumento é copiado no parâmetro.

Exemplificamos a passagem de parâmetros no código abaixo:

```
1  #include <stdio.h>
2  int sqr(int entrada)
2  {
3      entrada = entrada*entrada;
4      return(entrada);
5  }
6  int main(void)
6  {
7      int t=4;
8      printf("\n *exemplo de função por REFERÊNCIA*");
9      printf("\n *-----*");
10     printf("\n %d %d", sqr(t), t);
11     return 0;
12 }
```

Podemos observar que, neste código, o valor do argumento para a função **sqr()**, 4, é **copiado** no parâmetro **entrada**. Quando a atribuição **entrada = entrada \* entrada** ocorre, apenas a variável local **entrada** é

modificada. A variável `t`, usada para chamar a função, ainda tem o valor 4. Observe, na Figura 15, o resultado da saída:

```
https://funcaoPorReferencia.chpereira.repl.run 
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
✧ clang-7 -pthread -lm -o main main.c
✧ ./main

*exemplo de função por REFERÊNCIA*
*-----*
16 4
```

**Figura 15:** Resultado da execução do programa exemplo de função por referência. **Fonte:** Elaboração Própria.

## Função com parâmetro do tipo matriz

Quando uma matriz é usada como argumento para uma função, apenas o endereço da matriz é passado, e não uma cópia da matriz inteira. A declaração de parâmetros deve ser de um tipo de ponteiro compatível.

### FIQUE ATENTO

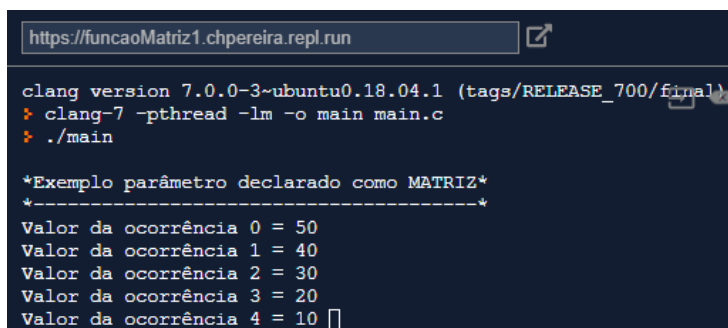
Um nome de matriz sem qualquer índice é um ponteiro para o primeiro elemento da matriz.

Existem três formas de se declarar um parâmetro que receberá um ponteiro para a matriz:

## Primeira Forma – Declarado como uma matriz

```
1  #include <stdio.h>
2  void imprimeValores(int valores[5])
3  {
4      for(int controle=0;controle<5;controle++)
5      {
6          printf("\n Valor da ocorrência %d = %d",controle,valores[controle]);
7      }
8  }
9  int main(void)
10 {
11     printf("\n *Exemplo parâmetro declarado como
    MATRIZ*");
12     printf("\n *-----*");
13     int valoresOriginais [5] = {50,40,30,20,10};
14     imprimeValores(valoresOriginais);
15     return 0;
16 }
```

Na Figura 16, temos a execução do exemplo de função, cujo parâmetro foi declarado como uma matriz:



```
https://funcaoMatriz1.chpereira.repl.run
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
❏ clang-7 -pthread -lm -o main main.c
❏ ./main

*Exemplo parâmetro declarado como MATRIZ*
*-----*
Valor da ocorrência 0 = 50
Valor da ocorrência 1 = 40
Valor da ocorrência 2 = 30
Valor da ocorrência 3 = 20
Valor da ocorrência 4 = 10 □
```

**Figura 16:** Execução de função - parâmetro declarado como uma matriz. **Fonte:** Elaboração Própria.

## Segunda Forma – Especificado como matriz sem dimensão

```
1  #include <stdio.h>
2  void imprimeValores(int valores[]) //MATRI SEM DIMENSÃO
3  {
4      for(int controle=0;controle<5;controle++)
5      {
6          printf("\nValor da ocorrência %d = %d",controle,valores[controle]);
7      }
8  }
9  int main(void)
10 {
11     printf("\n *Exemplo parâmetro declarado como MATRIZ sem dimensão*");
12     printf("\n *-----*");
13     int valoresOriginais [5] = {501,401,301,201,101};
14     imprimeValores(valoresOriginais);
15     return 0;
16 }
```

Na Figura 17, temos a execução do exemplo de função, cujo parâmetro foi declarado como uma matriz sem dimensão:

<https://funcaoMatriz1.chpereira.repl.run>



```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
clang-7 -pthread -lm -o main main.c
./main

*Exemplo parâmetro declarado como MATRIZ sem dimensão*
*-----*
Valor da ocorrência 0 = 501
Valor da ocorrência 1 = 401
Valor da ocorrência 2 = 301
Valor da ocorrência 3 = 201
Valor da ocorrência 4 = 101
```

**Figura 17:** Execução de função - parâmetro declarado como uma matriz sem dimensão. **Fonte:** Elaboração Própria.

## Terceira Forma – Declarado como um ponteiro

```
1  #include <stdio.h>
2  void imprimeValores(int *valores) //MATRIZ como ponteiro
3  {
4      for(int controle=0;controle<5;controle++)
5      {
6          printf("\nValor da ocorrência %d = %d",controle,valores[controle]);
7      }
8  }
9  int main(void)
10 {
11     printf("\n *Exemplo parâmetro declarado como PONTEIRO*");
12     printf("\n *-----*");
13     int valoresOriginais [5] = {511,411,311,211,111};
14     imprimeValores(valoresOriginais);
15     return 0;
16 }
```

Na Figura 18, temos a execução do exemplo de função, cujo parâmetro foi declarado como PONTEIRO:

```
https://funcaoMatriz1.chpereira.repl.run

clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
❯ clang-7 -pthread -lm -o main main.c
❯ ./main

*Exemplo parâmetro declarado como PONTEIRO*
*-----*
Valor da ocorrência 0 = 511
Valor da ocorrência 1 = 411
Valor da ocorrência 2 = 311
Valor da ocorrência 3 = 211
Valor da ocorrência 4 = 111
```

**Figura 18:** Execução de função - parâmetro declarado como PONTEIRO. **Fonte:** Elaboração Própria.

## O comando return

O comando **return** possui duas importantes utilizações dentro de uma função:

1. Provoca uma saída imediata da função (termina a função);
2. Devolve um valor

Importante destacar que uma função pode conter mais de um comando **return**. Mas, ao executar qualquer deles, a execução desta função termina.

Todas as funções, exceto as do tipo **void**, devolve um valor e este valor é especificado explicitamente pelo comando **return**.

## SAIBA MAIS

Geralmente, os compiladores **C** devolvem 0 (zero) quando nenhum valor de retorno é explicitamente especificado.

## Funções do tipo void

Um dos usos do void é declarar, de forma explícita, funções que não devolvem valores. Isso é importante, pois dá a liberdade ao programador de codificar uma função que atua internamente, sem a necessidade de devolver um valor. Por exemplo: Imagine que você desenvolveu uma função somente para imprimir um aviso em tela. A função emite o aviso e não precisa devolver ao chamador algum dado manipulado em seu interior. Por exemplo:

```
2 void imprimeValores(int *valores)
3 {
4     for(int controle=0;controle<5;controle++)
5     {
6         printf("\nValor da ocorrência %d = %d",controle,valores[controle]);
7     }
8 }
```

## A função main()

A função **main()** é a função que **inicia um programa em linguagem C**. Ela devolve um inteiro para o processo chamador: geralmente é o sistema operacio-



nal. Isso é equivalente à chamada da função **exit()** com o mesmo valor.

## Ordenação e Busca

A ordenação é um processo de arranjar um conjunto de informações semelhantes numa ordem crescente ou decrescente. Isso ocorre, em especial, em uma lista ordenada  $i$  de  $n$  elementos. Nesta seção, trataremos de **ordenação de dados em vetores**, mas o mesmo princípio é aplicado em outros tipos de área de memória.

A despeito de muitos compiladores em **C**, já trazerem embutido função específica para ordenação, trata-se do **qsort()**, é muito importante que você aprenda o paradigma da ordenação, pois pode existir alguma estrutura que os algoritmos de classificação, então, consigam fazê-lo. Existem três tipos de ordenação:

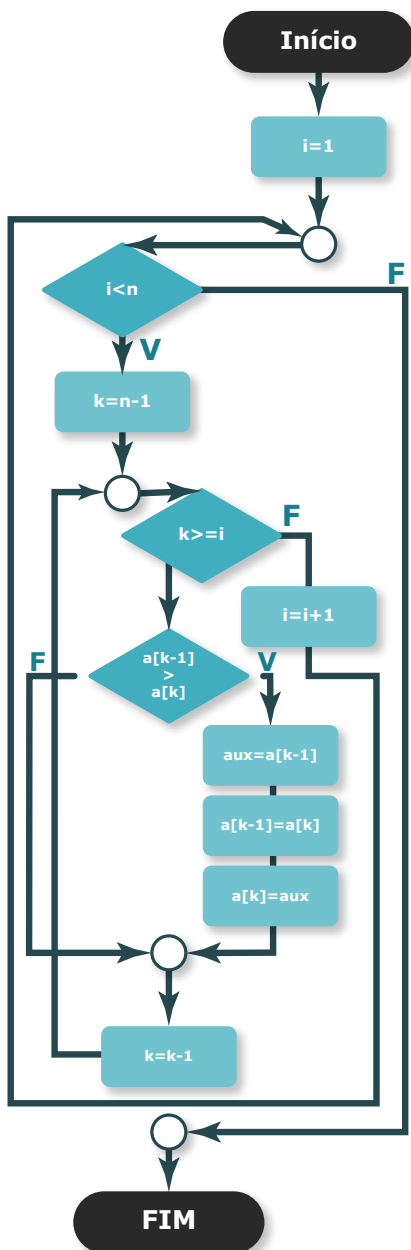
1. Por troca
2. Por seleção
3. Por inserção

### ■ Ordenação por Troca

A ordenação por troca, cuja técnica é conhecida por “Ordenação Bolha”, e envolve repetidas comparações e, se necessário, a troca de dois elementos adjacentes. Este nome de “bolha” se dá porque os elementos se comportam como bolhas de sabão, em que cada uma procura o seu próprio nível em um tanque. A Figura 19 mostra o esquema desta técnica em um diagrama de blocos (utilizando o paradigma de laço

de repetição **for**). Observe, na Figura 19, que existem dois laços de repetição, e admitamos que **n** é o **tamanho do vetor "a"** que deverá ser classificado. O primeiro laço (mais externo) está sendo controlado pela **variável controle i**. Já o segundo laço de repetição está sendo controlado pela **variável de controle k**. Note que **i**, irá de 0 (primeira posição do vetor) até **n** (tamanho do vetor). Já o controle **k**, inicia em **n - 1**, e que **k** representa as posições que serão comparadas com **i**.

O laço mais interno é o ponto que compara a posição **[k-1]** com cada posição que está à sua frente **[k]**, e, se esta for maior (posição **[k-1]** for maior posição **[k]**), então deverá ser feita a troca, pois o valor maior (posição **[k-1]**) deve trocar de lugar com a posição **[k]**. Neste caso, precisamos de uma variável **auxiliar** para salvar a posição **[k-1]**, e colocar o valor da posição **[k]** em seu lugar. Depois, coloca-se o valor da variável **auxiliar**, na posição **[k]**.



**Figura 19:** Diagrama de blocos da ordenação por TROCA (Ordenação BOLHA). **Fonte:** Elaboração Própria.

Abaixo, temos o código da implementação algoritmo de ordenação por troca:

```
1  #include <stdio.h>
2  int main(void)
3  {
4      int vetor[5] = {76,63,46,14,02}; //Vetor sem classificação
5      int limite=5;
6      printf("\n *-----*");
7      printf("\n *Vetor original (Antes da classificação*)");
8      printf("\n *-----*");
9      for (int w=0;w<limite;w++)
10     {
11         printf("\n vetor[%d] = %d",w,vetor[w]);
12     }
13     // Aqui começa a classificação por TROCA
14     for (int i=1;i<limite;i++)
15     {
16         for (int k=limite-1;k>=i;k--)
17         {
18             if (vetor[k-1]>vetor[k])
19             {
20                 int aux=vetor[k-1]; //Neste Ponto efetua-se
a TROCA
21                 vetor[k-1]=vetor[k]; //Neste Ponto efetua-se
a TROCA
22                 vetor[k] = aux; //Neste Ponto efetua-se a
TROCA
23             }
24         }
25     } //Aqui termina a classificação por TROCA
26     printf("\n !!!");
27     printf("\n *-----*");
```

```

28      Vetor Classificado por TROCA*");
29      printf("\n *-----*");
30      for (int z=0;z<limite;z++)
31      {
32          printf("\n vetor[%d] = %d",z,vetor[z]);
33      }
34      return 0;
35  }

```

Na Figura 20, temos o resultado do programa codificado:

```

https://sortbolha2.chpereira.repl.run
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang-7 -pthread -lm -o main main.c
> ./main

*-----*
* Vetor original (Antes da classificação)*
*-----*
vetor[0] = 76
vetor[1] = 63
vetor[2] = 46
vetor[3] = 14
vetor[4] = 2
!!!
*-----*
* Vetor Classificado por TROCA*
*-----*
vetor[0] = 2
vetor[1] = 14
vetor[2] = 46
vetor[3] = 63
vetor[4] = 76

```

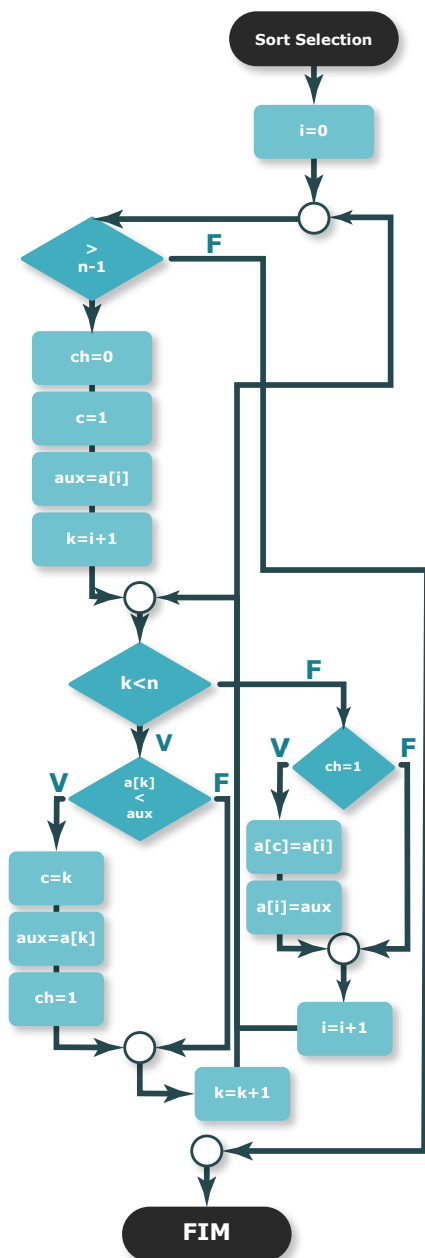
**Figura 20:** Resultado do programa de ordenação por troca.  
**Fonte:** Elaboração Própria.

## Ordenação por Seleção

A ordenação por seleção aplica a técnica de seleção do elemento de menor valor e troca-o pelo primeiro elemento. Então, para os elementos restantes, é en-

contrado o elemento de menor chave, trocando pelo segundo elemento, e assim sucessivamente. A troca continua até os dois últimos elementos.

Temos, na Figura 21, o diagrama que representa este algoritmo:



**Figura 21:** Diagrama de blocos da ordenação por Seleção.  
**Fonte:** Elaboração Própria.

Abaixo, temos o código em **C** que implementa este algoritmo:

```
1  #include <stdio.h>
2
3  int main(void)
4  {
5      int vetor[5] = {76,63,46,14,02};
6      int limite=5;
7      int exchange=0;
8      printf("\n *-----*");
9      printf("\n Vetor original (Antes da classificação*");
10     printf("\n *-----*");
11     for (int w=0;w<limite;w++)
12     {
13         printf("\n vetor[%d] = %d",w,vetor[w]);
14     }
15     for(int i=0;i<limite-1;i++) //Aqui começa a classificação
16     {
17         exchange = 0;
18         int c=i;
19         int aux = vetor[i];
20         for(int j=i+1;j<limite;j++)
21         {
22             if(vetor[j]<aux)
23             {
24                 c=j;
25                 aux=vetor[j];
26                 exchange=1;
27             }
28         }
29         if (exchange)
```



```

30         {
31             vetor[c]=vetor[i];
32             vetor[i]=aux;
33         }
34     }
35     printf("\n!!!"); //printf("\n!!!"); printf("\n!!!");
36     printf("\n*-----*");
37     printf("\n* Vetor Classificado por SELEÇÃO*");
38     printf("\n*-----*");
39     for (int z=0;z<limite;z++)
40     {
41         printf("\n vetor[%d] = %d",z,vetor[z]);
42     }
43     return 0;
44 }

```

Observe, na Figura 22, o resultado da execução do programa de ordenação por seleção.

<https://orderSelecao.chpereira.repl.run>



```
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
❏ clang-7 -pthread -lm -o main main.c
❏ ./main
```

```
*-----*
* Vetor original (Antes da classificação)*
*-----*
vetor[0] = 76
vetor[1] = 63
vetor[2] = 46
vetor[3] = 14
vetor[4] = 2
!!!
*-----*
* Vetor Classificado por SELEÇÃO*
*-----*
vetor[0] = 2
vetor[1] = 14
vetor[2] = 46
vetor[3] = 63
vetor[4] = 76
```

**Figura 22:** Resultado do programa de ordenação por seleção. **Fonte:** Elaboração Própria.

Como na ordenação bolha, o laço mais externo é executado vezes e o laço interno vezes. Como resultado, a ordenação por seleção requer comparações, fato que torna este tipo de ordenação muito lenta para um número elevado de itens.

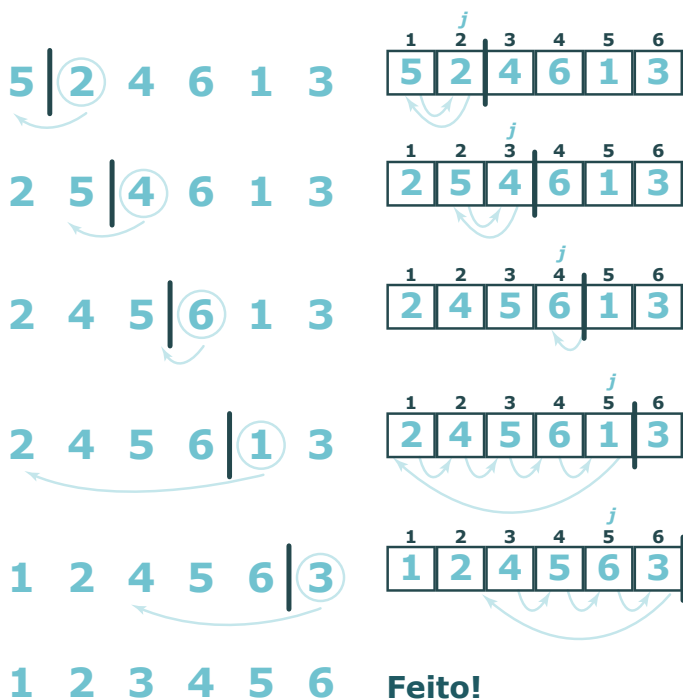
Para saber mais sobre modelos de classificação de dados, acesse o Podcast “Classificando dados na memória”.

**Acesse o Podcast 2 em Módulos**

## ■ Ordenação por Inserção

Inicialmente, esta ordena os dois primeiros membros do vetor. Em seguida, o algoritmo insere o tercei-

ro membro de sua posição ordenada com relação aos dois primeiros membros. Então, insere o quarto elemento na lista dos três elementos. O processo continua até que todos os elementos tenham sido ordenados. Este modelo de ordenação é demonstrado na Figura 23.



**Figura 23:** Esquema da ordenação por inserção. Fonte: [miftyisbored](#)

Analogamente, este método de ordenação equivale à forma como as pessoas ordenam cartas de baralho. Inicialmente, com a mão esquerda vazia e as cartas viradas com a face para baixo na mesa. Em seguida, removeremos uma carta de cada vez da mesa,

inserindo-a na posição correta na mão esquerda. Para encontrar a posição correta de uma carta, vamos compará-la a cada uma das cartas que já estão na mão, da direita para a esquerda.

Abaixo temos o código **C** que implementa o algoritmo da ordenação por inserção:

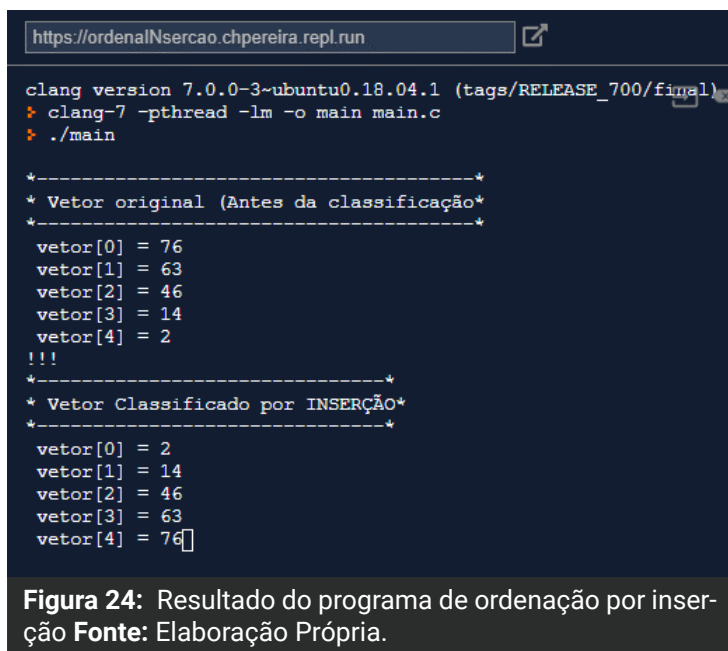
```
1  #include <stdio.h>
2  int main(void)
3  {
4      int vetor[5] = {76,63,46,14,02};
5      int limite=5;
6      printf("\n *-----*");
7      printf("\n *Vetor original (Antes da classificação*)");
8      printf("\n *-----*");
9      int j=0;
10     int i=0;
11     for (int w=0;w<limite;w++)
12     {
13         printf("\n vetor[%d] = %d",w,vetor[w]);
14     }
15     //Aqui começa a ordenação por INSERÇÃO
16     for(i=1;i<limite;i++)
17     {
18         int aux = vetor[i];
19         for(j=i-1;j>=0 && aux<vetor[j];j--)
20         {
21             vetor[j+1]=vetor[j];
22         }
23         vetor[j+1]=aux;
24     } //fim da ordenação por inserção
```

```

25     printf("\n!!!"); //printf("\n!!!"); printf("\n!!!");
26     printf("\n *-----*");
27     printf("\n *Vetor Classificado por INSERÇÃO*");
28     printf("\n *-----*");
29     for (int z=0;z<limite;z++)
30     {
31         printf("\n vetor[%d] = %d",z,vetor[z]);
32     }
33     return 0;
34 }

```

Conforme observamos na Figura 24, o resultado da execução do programa de ordenação por inserção:



```

https://ordenalNsercao.chpereira.repl.run
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang-7 -pthread -lm -o main main.c
> ./main

*-----*
* Vetor original (Antes da classificação)
*-----*
vetor[0] = 76
vetor[1] = 63
vetor[2] = 46
vetor[3] = 14
vetor[4] = 2
!!!
*-----*
* Vetor Classificado por INSERÇÃO
*-----*
vetor[0] = 2
vetor[1] = 14
vetor[2] = 46
vetor[3] = 63
vetor[4] = 76

```

**Figura 24:** Resultado do programa de ordenação por inserção **Fonte:** Elaboração Própria.

Ao contrário da ordenação bolha e da ordenação por inserção, o número de comparações que ocorrem

durante a ordenação por inserção depende de como a lista está inicialmente ordenada. Se a lista estiver em ordem, o número de comparações será  $n - 1$ . Se estiver fora de ordem, o número de comparações será  $1/2(n^2 + n)$ .

## Quicksort

A *Quicksort* é superior a todas as outras ordenações que abordamos e é considerada o melhor algoritmo de ordenação de propósito geral. É baseada no método de ordenação por trocas e na ideia de **partições**.

O procedimento é selecionar um valor, chamado de **comparando** e, então, fazer a partição do vetor em duas seções, com todos os elementos maiores ou iguais ao valor da partição de um lado e os menores do outro. Este processo é repetido para cada seção restante, até que o vetor esteja ordenado.

Abaixo, temos o código **C** que implementa o algoritmo da **Quicksort**:

```
1  #include <stdio.h>
2  int vetor[5] = {76,63,46,14,02};
3  int limite=5;
4  void qs(int *vetor, int left, int right)
5  { //início da função "qs"
6      int i, j;
7      int x, y;
8      i = left; j=right;
9      x = vetor[(left+right)/2];
10     do
```

```

11      {
12          while (vetor[i]<x && i<right) i++;
13          while (x<vetor[j] && j>left) j--;
14          if (i<=j)
15              {
16                  y = vetor[i];
17                  vetor[i]=vetor[j];
18                  vetor[j]=y;
19                  i++;j--;
20              }
21      } while (i<=j);
22      if (left<j) qs(vetor,left,j);
23      if (i<right) qs(vetor,i,right);
24  } //FIM da função "qs"
25  //-----
26  void quick(int *vetor, int count)
27  { //Início da função "quick"
28      printf("\n *-----*");
29      printf("\n *Vetor original (Antes da classificação*)");
30      printf("\n *-----*");
31      for (int z=0;z<=limite;z++)
32          {
33              printf("\n vetor[%d] = %d",z,vetor[z]);
34          }
35      qs(vetor, 0, count-1); //chama a função "qs"
36  } //FIM da função "quick"
37  //-----
38  //início do programa (main)
39  //-----
40  int main(void)
41  {

```

```

42      quick(vetor, 5); //Chama a função que inicia a ordenação
    (quick)
43      printf("\n!!!"); //printf("\n!!!"); printf("\n!!!");
44      printf("\n*-----*");
45      printf("\n* Vetor Classificado QUICKSORT*");
46      printf("\n*-----*");
47      for (int z=0;z<limite;z++)
48      {
49          printf("\n vetor[%d] = %d",z,vetor[z]);
50      }
51      return 0;
52  }

```

Na Figura 25, temos o resultado do programa de classificação **Quicksort**:

```

https://ordenacaoquicksort.chpereira.repl.run
clang version 7.0.0-3~ubuntu0.18.04.1 (tags/RELEASE_700/final)
✚ clang-7 -pthread -lm -o main main.c
✚ ./main

*-----*
* Vetor original (Antes da classificação*
*-----*
vetor[0] = 76
vetor[1] = 63
vetor[2] = 46
vetor[3] = 14
vetor[4] = 2
vetor[5] = 5
!!!
*-----*
* Vetor Classificado QUICKSORT*
*-----*
vetor[0] = 2
vetor[1] = 14
vetor[2] = 46
vetor[3] = 63
vetor[4] = 76

```

**Figura 25:** Resultado do programa de ordenação Quicksort.  
**Fonte:** Elaboração Própria.



Nessa versão, a função **quick()** executa a chamada à função de ordenação principal **qs()**. Isso permite manter a interface comum com vetor e limite. Isso dá um número médio de comparações de .

## Métodos de pesquisa

Encontrar informações em um vetor desordenado requer uma pesquisa, começando no primeiro elemento do vetor. Essa busca termina quando o elemento for encontrado ou chegar o final do vetor. Isso ocorre quando temos dados desordenados. Porém, quando temos um vetor já ordenado (você pode utilizar uma das técnicas que já estudamos para ordenação), temos um método de busca chamado **pesquisa binária**.

A pesquisa binária utiliza o método “**dividir para conquistar**”. Por esse método, primeiro encontramos o elemento central. Se este elemento for maior que a chave procurada, ele testa o elemento central da primeira metade; caso contrário, ele testa o elemento central da segunda metade. Esse procedimento é repetido até que o elemento seja encontrado ou que **não haja mais elementos** a testar (neste último caso, conclui-se que o elemento não foi encontrado).

Abaixo, temos o código **C** que implementa o algoritmo de pesquisa binária:

```
1  #include <stdio.h>
2  #include <stdbool.h>
3  int vetor[5] = {02, 14, 46, 63, 76};
4  int limite=5;
```

```

5  bool buscaBinaria(int *vetor, int limitex,int chave)
6  {
7      int low, high, mid;
8      low=0; high=limitex-1;
9      while(low<=high)
10     { //Início do laço de repetição
11         mid = (low+high)/2;
12         if (chave<vetor[mid]) high = mid-1;
13         else if(chave>vetor[mid]) low=mid+1;
14         else return true;
15     } //fim do laço de repetição
16     return false;
17 }
18 //-----
19 //início do programa (main)
20 //-----
21 int main(void)
22 {
23     int elemento;
24     printf("\nInforme o número ");
25     scanf("%d",&elemento);
26     if(buscaBinaria(vetor, limite,elemento))
27     {
28         printf("\nO elemento %d foi encontrado no
29         vetor",elemento);
30     } else{
31         printf("\nO elemento %d NÃO foi encontrado no
32         vetor",elemento);
33     }
34     return 0;
35 }

```

Na Figura 26, temos dois resultados para execução do programa de pesquisa binária: **Caso 1** – Encontrou o argumento de pesquisa; e **Caso 2**–Não encontrou o argumento de pesquisa.

Caso #1

```
https://pesquisabinaria.chiperli.a rept.run
clang version 7.0.0-3-ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang-7 -pthread -lm -o main main.c
> ./main
Informe o número 14
O elemento 14 foi encontrado no vetor
```

O elemento de pesquisa (chave) **foi** encontrado

Caso #2

```
https://pesquisabinaria.chiperli.a rept.run
clang version 7.0.0-3-ubuntu0.18.04.1 (tags/RELEASE_700/final)
> clang-7 -pthread -lm -o main main.c
> ./main
Informe o número 22
O elemento 22 NÃO foi encontrado no vetor
```

O elemento de pesquisa (chave) **não foi** encontrado

**Figura 26:** Resultado dos dois casos de execução do programa de busca binária **Fonte:** Elaboração Própria.

## Uso de arquivos de texto

A linguagem **C** não possui nenhum comando específico de entrada e saída(**E/S**). Todas as operações de **E/S** ocorrem mediante chamadas das funções da biblioteca **C** padrão. Isso proporciona que dados possam ser transferidos na sua representação binária interna ou em formato de texto legível por humanos.

## Arquivos

Em **C**, um arquivo pode ser qualquer dispositivo, por exemplo, arquivo em disco, impressora, ou um terminal. Um *stream* de dados é associado com arquivo específico a partir de sua **abertura**. Uma vez que o arquivo tenha sido aberto, as informações podem ser trocadas entre este e o seu programa. Para desasso-

ciar este arquivo a um *stream* de dados específico, usa-se uma operação de **fechamento**.

Cada *stream* associado a um arquivo tem uma estrutura de controle de arquivo do tipo **FILE**. Esta estrutura é definida no cabeçalho **STDIO.H**.

O sistema de arquivos em **C** é composto por diversas funções, conforme demonstrado na Tabela 4.

Função	Objetivo
<i>fopen()</i>	Abre um arquivo
<i>fclose()</i>	Fecha um arquivo
<i>putc()</i>	Escreve um caractere em um arquivo
<i>getc()</i>	Lê um caractere de um arquivo
<i>fsseek()</i>	Posiciona o arquivo em um byte específico
<i>fprintf()</i>	Imprime no arquivo conjunto de caracteres
<i>fscanf()</i>	Transfere caractere do arquivo para o programa
<i>feof()</i>	Indicar fim de arquivo atingido (volta <b>verdadeiro</b> )
<i>ferror()</i>	Indica erro na operação do arquivo (volta <b>verdadeiro</b> )
<i>rewind()</i>	Recoloca o indicador de posição de arquivo no início do arquivo
<i>remove()</i>	Apaga um arquivo
<i>fflush()</i>	Descarrega um arquivo

**Tabela 4:** Funções C para manipulação de arquivo **Fonte:** Elaboração Própria.

## SAIBA MAIS

Todos os arquivos abertos em um programa são fechados automaticamente quando este termina normalmente, com **main()** retornando ao sistema operacional código de retorno 0 (zero), ou com a chamada da função **exit()**.

Os arquivos não são fechados quando um programa quebras(**crash**) ou quando chama a função **abort()**.

## Abrindo Arquivos

A função **fopen()** abre uma *stream* para uso e associa a um arquivo. Ela retorna um ponteiro de arquivo associado a este arquivo. Esta função tem a seguinte estrutura:

**FILE \*fopen**(const char\* *nomearq*, const char\* *modo*);

Onde:

- *nomearq* = É um ponteiro para uma cadeia de caracteres que forma um nome válido de arquivo e pode incluir especificação de caminho (*path*)
- *modo* = Indica como este arquivo será aberto (conforme possibilidade indicada na Tabela 5)

Modo	Significado
<b>R</b>	Abre um arquivo texto para leitura
<b>W</b>	Cria um arquivo texto para escrita
<b>A</b>	Anexa a um arquivo texto
<b>Rb</b>	Abre um arquivo binário para leitura
<b>wb</b>	Cria um arquivo binário para escrita
<b>Ab</b>	Anexa a um arquivo binário
<b>r+</b>	Abre um arquivo texto para leitura/escrita
<b>w+</b>	Cria um arquivo texto para a leitura /escrita
<b>a+</b>	Anexa ou cria um arquivo texto para leitura/escrita
<b>r+b</b>	Abre um arquivo binário para leitura/escrita
<b>w+b</b>	Cria um arquivo binário para leitura/escrita
<b>a+b</b>	Anexa a um arquivo binário para leitura / escrita

**Tabela 5:** Opções de modos de abertura de arquivo. **Fonte:** Elaboração Própria.

## Fechando um Arquivo

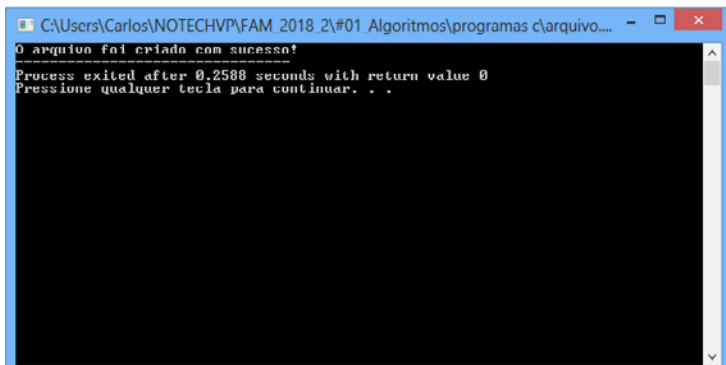
A função ***fclose()*** fecha uma *stream* que foi aberta por meio de uma chamada ***fopen()***. Esta função ainda escreve dados que ainda estejam no ***buffer*** de disco no arquivo, e somente a partir daí fecha o arquivo efetivamente. A sintaxe deste comando é: ***fclose(nome\_do\_arquivo)***. Um valor de retorno zeros significa que o arquivo foi fechado normalmente. Qualquer outro valor, significa erro.

Temos, no código abaixo, um exemplo de manipulação de arquivo, em que criamos um arquivo de nome “arquivo.txt”. Abrimos o arquivo com a posição de

gravação “w” e gravamos no diretório “desktop”, conforme caminho que consta na linha 8 do programa. Gravamos neste arquivo quatro linhas, cujo conteúdo é uma **string** (texto). Note que, ao final de cada **string**, utilizamos o caractere que indica salto de linha “\n”. Por fim, na linha 14, fechamos o arquivo com o comando **fclose()**.

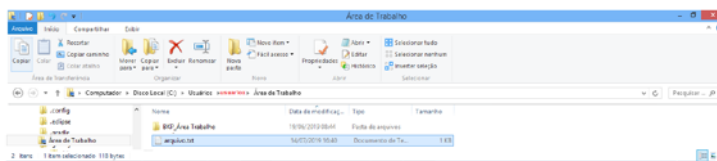
```
1  #include <stdio.h>
2  #include <stdlib.h>
3  int main(void)
4  {
5      // criando a variável ponteiro para o arquivo
6      FILE *file;
7      //abrindo o arquivo
8      file = fopen("C:\\Users\\usuariox\\Desktop\\arquivo.txt",
9                  "w");
10     fprintf(file,"Esta é a linha 1 do arquivo\n");
11     fprintf(file,"Esta é a linha 2 do arquivo\n");
12     fprintf(file,"Esta é a linha ... do arquivo\n");
13     fprintf(file,"Esta é a linha n do arquivo\n");
14     // fechando arquivo
15     fclose(file);
16     //mensagem para o usuário
17     printf("O arquivo foi criado com sucesso!");
18     return(0);
19 }
```

Na Figura 27, temos o resultado da console com a execução do programa:



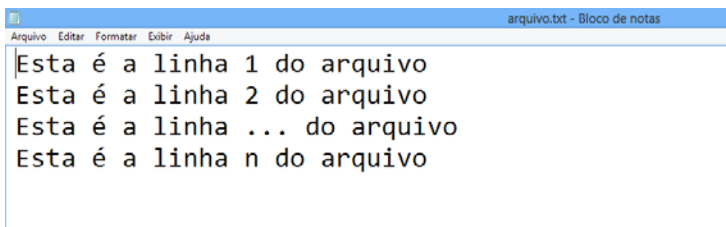
**Figura 27:** Resultado da execução – Console. **Fonte:** Elaboração Própria.

Na Figura 28, temos o local onde o arquivo foi gerado.



**Figura 28:** Resultado da execução - Local de gravação do arquivo gerado **Fonte:** Elaboração Própria.

Na Figura 29, temos o conteúdo gerado no arquivo “arquivo.txt”.



**Figura 29:** Resultado da execução - Conteúdo do arquivo gerado **Fonte:** Elaboração Própria.



# CONSIDERAÇÕES FINAIS

Caro estudante, nesta unidade, estudamos juntos tópicos avançados na programação de computadores, em que você foi elevado a um nível de programação. Aprendemos a tratar vetores unidimensionais, bem como vetores bidimensionais (matrizes), que é um ponto muito importante em qualquer linguagem de programação, pois permite ao programador solucionar os problemas computacionais com melhor aproveitamento das técnicas de programação. E em muitas necessidades de negócio, esta técnica é aplicada.

Ao desenvolver as soluções para tratamento de vetores, aplicamos os paradigmas estudados em unidades anteriores, qual seja, a utilização de lógica de laços de repetição. E neste aspecto, praticamos fortemente soluções com uso de laços de repetição *for* e *while* e *do-while*. Portanto, este tópico sobre vetores e matrizes nos fortaleceu muito nos amadurecimentos de conceitos e práticas da programação de computadores.

Outro tópico estudado nesta unidade foi a utilização de ponteiros, que, na linguagem C, é de fundamental importância, pois o domínio da utilização da técnica do uso de ponteiros aumenta a eficiência da performance do programa, bem como a correta utilização em alocação dinâmica de memória.

Aprendemos a tratar os paradigmas de ordenação de dados em memória (vetores), em que mostramos os modelos de ordenação mais utilizados, como a ordenação por troca (ordenação bolha), a ordenação por seleção e a ordenação por inserção. Aprendemos também a encontrar informações em um vetor, seja ele ordenado ou desordenado, respectivamente com os métodos de busca sequencial e busca binária.

Por fim, você aprendeu a manipular arquivos na linguagem **C**, pois este é um tópico muito importante no desenvolvimento de todos os programadores de qualquer linguagem de programação.

Parabéns por ter completado um ciclo muito importante no aprendizado de um profissional de tecnologia da informação. Pois, mesmo que você não busque se desenvolver na programação de computadores, o conhecimento da lógica de programação se faz muito importante para o seu desenvolvimento em outras áreas da tecnologia da informação. Desejo muito sucesso em sua trajetória com o uso dos conceitos da lógica de programação!



## TÓPICOS AVANÇADOS DE PROGRAMAÇÃO

Nesta unidade abordamos tópicos mais avançados da lógica de programação, em que aprenderemos a dominar as técnicas de programação profissionais. Abordaremos a estrutura de dados de Vetores e Matrizes. Abordamos também a utilização de ponteiros, a ordenação de dados em vetores, métodos pesquisa em vetores desordenados e vetores ordenados. Por fim aprendemos a manipular arquivos, acessar e gravar dados.

Para este fim, estruturamos este material da seguinte forma:

### Matrizes e Strings

 Vetores Unidimensionais Strings Vetores Bidimensionais (Matriz) Inicialização de Matriz

### Ponteiros

 Conceito de ponteiros O Operadores de Ponteiros

### Recursão

 Paradigmas de recursividade Implementando recursividade na linguagem

### Funções

 Declaração de Funções Chamada por valor e por referência Função com parâmetro do tipo matriz O comando **return** Funções do tipo **void**

### Ordenação e Busca

 Ordenação por Troca Ordenação por Seleção **Quicksort** Métodos de pesquisa binária

### Uso de arquivos de texto

 Fundamentos de arquivo Abrindo Arquivos Fechando um Arquivo Manipulação de arquivo

# Referências Bibliográficas & Consultadas

---

ALVES, W. P. **Linguagem e Lógica de Programação**. São Paulo: Érica, 2015.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R.L.; STEIN, C. **Algoritmos: Teoria e prática**. Rio de Janeiro: Elsevier, 2002.

EDELWEIS, N.; LIVI, M. A. C.; LOURENÇO, A. E. **Algoritmos e programação com exemplos em Pascal e C**. São Paulo: Bookman, 2014.

FORBELLONE, A. L. V; EBERSPACHER, H. F. **Lógica de Programação: a construção de algoritmos e estruturas de dados**. São Paulo: Prentice Hall, 2015.

MANZANO, J. A. N. G; MATOS, E; LOURENÇO, A. E. **Algoritmos: Técnicas de Programação**. São Paulo: Érica, 2015.

MANZANO, J. A. N. G; OLIVEIRA, J. F. **Algoritmos: Lógica para desenvolvimento de programação de computadores**. São Paulo: Saraiva, 2016.

PERKOVIC, L. **Introdução à computação usando Python**. Rio de Janeiro: LTC-Livros Técnicos e Científicos, 2016.

ZIVIANI, N. **Projeto de Algoritmos:** Com implementações em Pascal e C. São Paulo: Cengage Learning, 2011.

**FaTM**  
**ONLINE**