

# Programação Estruturada

## Vetores e matrizes

---

Professores Emílio Francesquini e Carla Negri Lintzmayer

2018.Q3

Centro de Matemática, Computação e Cognição  
Universidade Federal do ABC



# Introdução

---

# Motivação

Suponha que desejamos guardar notas de alunos.

Com o que aprendemos até agora, como armazenaríamos 3 notas?

---

```
1 float nota1, nota2, nota3;
2
3 printf("Nota do aluno 1: ");
4 scanf("%f", &nota1);
5 printf("Nota do aluno 2: ");
6 scanf("%f", &nota2);
7 printf("Nota do aluno 3: ");
8 scanf("%f", &nota3);
```

---

# Motivação

Com o que sabemos, como armazenaríamos 100 notas?

---

```
1 float nota1, nota2, nota3,..., nota100;
2
3 printf("Nota do aluno 1: ");
4 scanf("%f", &nota1);
5 printf("Nota do aluno 2: ");
6 scanf("%f", &nota2);
7 ...
8 printf("Nota do aluno 100: ");
9 scanf("%f", &nota100);
```

---

Apesar de ainda ser viável, criar 100 variáveis distintas não é uma solução elegante para este problema. E se precisássemos armazenar 1.000.000 notas? Ou  $n$  notas?

# Vetores

---

# Definição de vetores

- Um vetor em C é uma coleção de variáveis de um mesmo tipo que são referenciadas por um **identificador único**.
- Características de um vetor:
  - As variáveis ocupam posições contíguas na memória.
  - O acesso se dá por meio de um índice inteiro.
  - O vetor possui um tamanho pré-definido.
  - O acesso do vetor com um índice fora dos limites pode causar comportamento anômalo do programa.

# Declaração de um vetor

Para declarar um vetor usamos a seguinte sintaxe:

---

```
1 tipo identificador[tamanho];
```

---

Exemplos:

---

```
1 /* vetor "notas" equivale a 100 variáveis do tipo  
   ↳ float */  
2 float notas[100];  
3  
4 /* vetor "primos" equivale a 20 variáveis do tipo  
   ↳ int */  
5 int primos[20];
```

---

## Usando um vetor

- Após declarada uma variável do tipo vetor, pode-se acessar uma determinada posição utilizando-se um índice de valor inteiro.
- Sendo  $n$  o tamanho do vetor, os índices válidos para o vetor vão de 0 até  $n - 1$ .
  - A primeira posição de um vetor tem índice 0.
  - A última posição de um vetor tem índice  $n - 1$ .
  - A  $i$ -ésima posição tem índice  $i - 1$ .
  - A sintaxe para acesso de uma determinada posição é:  
**identificador[posicao]**
    - Lê-se **vet[4]** como “vetor **vet** na posição 4”
    - **vet[4]** é o quinto elemento do vetor **vet**



## Usando um vetor

Uma posição específica de um vetor tem o mesmo comportamento que uma variável simples.

---

```
1  int nota[10];  
2  int a;  
3  /* "nota[5]" corresponde a uma variável inteira */  
4  nota[5] = 95;  
5  a = nota[5];
```

---

## Usando um vetor

- Você deve usar apenas valores inteiros como índice para acessar uma posição do vetor.
- O valor pode ser inclusive uma outra variável inteira.

---

```
1 int g, vet[10];  
2 for(g = 0; g < 10; g++)  
3     vet[g] = 5 * g;
```

---

Quais valores estarão armazenados em cada posição do vetor após a execução deste código?

# Vetores e a memória

Suponha o código:

---

```
1  int d;  
2  int vetor[5];  
3  int f;
```

---

Na memória temos:

Nome	d	vetor					f
Índice	-	0	1	2	3	4	-

# Vetores e a memória

Ao executar o comando

```
1 vetor[3] = 10;
```

Temos em memória:

Nome	d	vetor					f
Índice	-	0	1	2	3	4	-
					10		

# Vetores e a memória

E ao executar os comandos a seguir?

---

```
1 vetor[3] = 10;  
2 vetor[5] = 5;  
3 vetor[-1] = 1;
```

---

Teremos em memória:

Nome	d	vetor					f
Índice	-	0	1	2	3	4	-
	1				10		5

Seu programa estará errado pois você está alterando inadvertidamente valores de outras variáveis.

Ele será encerrado (**Segmentation Fault**) ou poderá continuar executando, mas ocorrerão erros difíceis de serem rastreados.

## Questões importantes sobre vetores

- O tamanho do vetor é pré-definido (durante a execução do programa não pode ser alterado).
- O uso de índices fora dos limites pode causar comportamento anômalo do programa.

## Como armazenar até 100 notas?

---

```
1 float nota[100];
2 int n, i;
3
4 printf("Número de alunos: ");
5 scanf("%d", &n);
6
7 for (i = 0; i < n; i++) {
8     printf("Digite a nota do aluno %d: ", i);
9     scanf("%f", &nota[i]);
10 }
```

---

O programa acima está correto?

# Como armazenar até 100 notas?

Você deve testar se  $n > 100$  para evitar erros!!

---

```
1  float nota[100];
2  int n, i;
3
4  printf("Número de alunos: ");
5  scanf("%d", &n);
6
7  if (n > 100) {
8      n = 100;
9      printf("Numero maximo de alunos alterado para
10 ↪ 100\n");
11 }
12 for (i = 0; i < n; i++) {
13     printf("Digite a nota do aluno %d: ", i);
14     scanf("%f", &nota[i]);
15 }
```



## Exemplo: produto interno de dois vetores

### Problema

Ler dois vetores de dimensão 5 e computar o produto interno (produto escalar) destes.

Quais tipos de variáveis usar?

## Exemplo: produto interno de dois vetores

Abaixo temos o código para ler dois vetores de dimensão 5.

```
1  int main() {
2      double vetor1[5], vetor2[5], resultado;
3      int i;
4
5      for (i = 0; i < 5; i++) {
6          printf("Entre com valor da posição %d para vetor 1: ",
7              ↵ i);
8          scanf("%lf", &vetor1[i]);
9      }
10     printf("\n\n");
11     for (i = 0; i < 5; i++) {
12         printf("Entre com valor da posição %d para vetor 2:",
13             ↵ i);
14         scanf("%lf", &vetor2[i]);
15     }
16
17     /* calculando o produto interno */
18     ...
19 }
```

## Exemplo: produto interno de dois vetores

Abaixo temos a parte do código para computar o produto interno dos vetores.

```
1  int main() {  
2      double vetor1[5], vetor2[5], resultado;  
3      int i;  
4  
5      ...  
6  
7      /* calculando o produto interno */  
8      resultado = 0.0;  
9      for (i = 0; i < 5; i++) {  
10         resultado = resultado + (vetor1[i] * vetor2[i]);  
11     }  
12     printf("\n\n0 produto interno é: %lf\n", resultado);  
13     return 0;  
14 }
```

## Exemplo: produto interno de dois vetores – código completo

```
1  int main() {
2      double vetor1[5], vetor2[5], resultado;
3      int i;
4
5      for (i = 0; i < 5; i++) {
6          printf("Entre com valor da posição %d para vetor 1: ",
7              ↪ i);
8          scanf("%lf", &vetor1[i]);
9      }
10     printf("\n\n");
11     for (i = 0; i < 5; i++) {
12         printf("Entre com valor da posição %d para vetor 2: ",
13             ↪ i);
14         scanf("%lf", &vetor2[i]);
15     }
16     /* calculando o produto interno */
17     resultado = 0.0;
18     for (i = 0; i < 5; i++)
19         resultado = resultado + (vetor1[i] * vetor2[i]);
20     printf("\n\nO produto interno é: %lf\n", resultado);
21     return 0;
22 }
```

## Exemplo: elementos iguais

- Ler dois vetores com 5 inteiros cada.
- Checar quais elementos do segundo vetor são iguais a algum elemento do primeiro vetor.
- Se não houver elementos em comum, o programa deve informar isso.

## Exemplo: elementos iguais

Abaixo está o código que faz a leitura de dois vetores.

---

```
1  int main() {
2      int vetor1[5], vetor2[5];
3      int i, j, umEmComum;
4
5      for (i = 0; i < 5; i++) {
6          printf("Entre com valor da posição %d do vetor
7              ↪ 1: ", i);
8          scanf("%d", &vetor1[i]);
9      }
10     printf("\n\n");
11     for (i = 0; i < 5; i++) {
12         printf("Entre com valor da posição %d do vetor
13             ↪ 2: ", i);
14         scanf("%d", &vetor2[i]);
15     }
16     ...
17 }
```

## Exemplo: elementos iguais

- Para cada elemento do `vetor1` testamos todos os outros elementos do `vetor2` para saber se são iguais.
- Usamos uma variável indicadora para decidir, ao final dos laços encaixados, se os vetores possuem ou não um elemento em comum.

## Exemplo: elementos iguais

```
1  int main() {
2      int vetor1[5], vetor2[5];
3      int i, j, umEmComum;
4      ...
5      umEmComum = 0; /* Assumimos que não haja elementos
   ↪ comuns */
6      for (i = 0; i < 5; i++) {
7          for (j = 0; j < 5; j++) {
8              if (vetor1[i] == vetor2[j]) {
9                  umEmComum = 1; /* Aachamos um elemento
   ↪ comum */
10                 printf("vetor1[%d] == vetor2[%d]\n", i,
   ↪ j);
11             }
12         }
13     }
14     if (!umEmComum)
15         printf("Nenhum elemento em comum!\n");
```



## Informações extras: inicialização de um vetor

- Em algumas situações é necessário declarar e já atribuir um conjunto de valores constantes para um vetor.
- Em C, isto é feito atribuindo-se uma lista de elementos para o vetor na sua criação da seguinte forma:

---

```
1  tipo identificador[] = {elementos separados por  
    ↪ vírgula};
```

---

- Exemplos:

---

```
1  double vet1[] = {2.3, 3.4, 4.5, 5.6};  
2  int vet2[] = {5, 4, 3, 10, -1, 0};
```

---

- Note que automaticamente é criado um vetor com tamanho igual ao número de dados da inicialização.

## Informações extras: inicialização de um vetor

---

```
1  #include <stdio.h>
2
3  int main() {
4      double vet1[] = {2.3, 3.4, 4.5, 5.6};
5      int vet2[] = {5, 4, 3, 10, -1, 0};
6      int i;
7
8      for (i = 0; i < 4; i++)
9          printf("%lf\n", vet1[i]);
10
11     for (i = 0; i < 6; i++)
12         printf("%d\n", vet2[i]);
13
14     return 0;
15 }
```

---

# Strings

---

# Strings em C

- A linguagem C não possui o tipo *string* explicitamente, mas podemos considerar um vetor de caracteres como uma string.
- Em C, uma string é sempre terminada pelo caractere `'\0'`.

## Definição

Uma string em C corresponde a um vetor de caracteres terminado pelo caractere especial `'\0'`.

- Sempre declare uma string com um caractere a mais do que precisa, já que também será preciso armazenar o `'\0'`.
  - Se por exemplo, estivermos trabalhando com strings de 10 caracteres, declare uma variável com tamanho 11:

---

```
1  char st[11];
```

---

# Strings em C

**Lembre-se:** o caractere ‘\0’ identifica o final da string.

No programa abaixo gostaríamos que fosse impresso “ola”.

---

```
1  int main() {
2      char st[80];
3
4      st[0] = 'o';
5      st[1] = 'l';
6      st[2] = 'a';
7
8      printf("%s\n", st);
9      return 0;
10 }
```

---

Mas às vezes será impresso uma palavra diferente, como “ola8uj?”, pois não identificamos o final da string.

# Strings em C

A versão correta do programa seria esta abaixo.

---

```
1  int main() {  
2      char st[80];  
3  
4      st[0] = 'o';  
5      st[1] = 'l';  
6      st[2] = 'a';  
7      st[3] = '\\0';  
8  
9      printf("%s\\n", st);  
10     return 0;  
11 }
```

---

Note que a variável `st` pode armazenar strings com até 79 caracteres, mas neste exemplo só estamos usando 3 (além do '\\0').

# Leitura e escrita de strings

Para ler/imprimir uma string do teclado usamos o operador `%s`.

---

```
1  int main() {
2      char st[80];
3      int id;
4
5      printf("Entre com o nome: ");
6      scanf("%s", st);
7      printf("Entre com a idade:");
8      scanf("%d", &id);
9
10     printf("Digitado: %s e %d\n", st, id);
11     return 0;
12 }
```

---

Note que para strings não é utilizado o `&` antes do identificador da variável no comando `scanf`.

O `scanf` automaticamente coloca um `'\0'` ao final da string lida.

# Leitura e escrita de strings

`scanf` com `%s` termina a leitura em uma quebra de linha ou um espaço.

---

```
1 char st[80];
2 int id;
3 printf("Entre com o nome: ");
4 scanf("%s", st);
5 printf("Entre com a idade: ");
6 scanf("%d", &id);
7 printf("Digitado: %s e %d\n", st, id);
```

---

No exemplo acima, se digitarmos

---

```
1 Joao da Silva
2 19
```

---

será salvo apenas “Joao” em `st`, e um valor diferente de 19 em `id`, porque o `scanf` lê a string até o primeiro espaço, e converte o



# Leitura e escrita de strings

- Para ler strings **incluindo espaços** use o comando `fgets`:

---

```
1 fgets(identificador, limite, stdin);
```

---

onde **identificador** é o nome da variável para onde será lida a string, **limite - 1** é a quantidade máxima de caracteres que poderá ser lida, e **stdin** é uma palavra reservada que indica que a leitura se dará da entrada padrão.

- Serão lidos todos os caracteres até uma quebra de linha, e todos serão armazenados na variável **identificador**, **incluindo o caractere de quebra de linha**, a menos que **limite-1** caracteres tenham sido lidos, caso em que a função para a leitura antes da quebra de linha.
- A função inclui um `'\0'` na posição final, após os caracteres lidos.

# Leitura e escrita de strings

---

```
1 char st[80];
2 int id;
3
4 printf("Entre com o nome: ");
5 fgets(st, 80, stdin);
6 printf("Entre com a idade: ");
7 scanf("%d", &id);
```

---

No exemplo acima se digitarmos

---

```
1 Joao da Silva
2 19
```

---

Será salvo “Joao da Silva\n\0” em **st**, e o valor 19 em **id**.

Como **st** pode armazenar até 80 caracteres, usamos este valor como parâmetro para o limite de caracteres que podem ser lidos do teclado, já que serão lidos até 79, e deve ser incluído o ‘\0’ no final.

## Leitura e escrita de strings

- Em geral é mais seguro usar o `fgets` do que o `scanf`, pois o primeiro especifica o tamanho máximo da string a ser lida.
- Se um usuário digitar uma string maior do que o vetor declarado, o `scanf` pode dar problemas pois irá ler todos caracteres até um espaço ou `'\n'`, sobrescrevendo posições inválidas da memória.
- Existe um ataque conhecido como *buffer overflow* que explora justamente este problema do `scanf`.
- Já o `fgets` sempre lê uma string de até o máximo especificado.

# Inicialização de strings

- Em algumas situações, ao criarmos uma string, pode ser útil atribuir valores já na sua criação.
- No caso de strings, podemos atribuir diretamente uma constante string para a variável.

---

```
1 char st[100] = "sim, isto é possível";
```

---

- O comando de inicialização automaticamente insere o caractere '\0' no final da string.
- Atribuições posteriores à inicialização, no entanto, não são permitidas!

## Problema

Ler uma string de até 79 caracteres (incluindo '\n') e imprimir sua inversa.

# Strings: exemplos

Primeiramente encontramos a posição final da string.

---

```
1  int main() {
2      char st[80], stInv[80];
3      int i, j, fim;
4
5      fgets(st, 80, stdin);
6
7      /* Determinamos o final da string */
8      fim = 0;
9      while (st[fim] != '\0' && st[fim] != '\n')
10         fim++;
11     ...
12 }
```

---

# Strings: exemplos

Depois escrevemos os caracteres em `stInv` na ordem inversa de aparição em `st`.

---

```
1  int main() {
2      char st[80], stInv[80];
3      int i, j, fim;
4      ...
5      /* Escrevemos os caracteres na inversa */
6      i = fim - 1;
7      j = 0;
8      while (j < fim) {
9          stInv[j] = st[i];
10         i--;
11         j++;
12     }
13     stInv[fim] = '\0';
14
15     printf("Inversa:\n%s\n", stInv);
16     return 0;
17 }
```

# Strings: exemplos

A mesma coisa mas com laço **for**:

---

```
1  int main() {
2      char st[80], stInv[80];
3      int i, j, fim;
4      fgets(st, 80, stdin);
5
6      /* Determinamos o final da string */
7      for (fim = 0; st[fim] != '\0' && st[fim] != '\n';
8          ↪ fim++);
9
10     /* Escrevemos os caracteres na inversa, stInv */
11     for (i = fim-1, j = 0; j < fim; i--, j++)
12         stInv[j] = st[i];
13     stInv[fim] = '\0';
14
15     printf("Inversa:\n%s\n", stInv);
16     return 0;
```



# Biblioteca `string.h`

- A biblioteca `string.h` possui várias funções úteis para se trabalhar com strings.
- Algumas funções comuns são:
  - `char *strcat(char *s1, const char *s2)` – Para fazer a concatenação de strings.
  - `int strcmp(const char *s1, const char *s2)` – Para fazer a comparação lexicográfica (utilizada em ordenação) de duas strings.
  - `char *strcpy(char *s1, const char *s2)` – Para fazer a cópia de strings.
  - `int strlen(const char *s1)` – Para se determinar o tamanho de uma string.

- Como exemplo de uso de strings vamos implementar duas funcionalidades básicas de processadores de texto:
  1. Contar o número de palavras em um texto.
  2. Fazer a busca de uma palavra em um texto.

# Processamento de texto

Vamos contar o número de palavras em textos sem pontuação:

```
1  int main() {
2      char s[80];
3      int i = 0, n = 0;
4      fgets(s, 80, stdin);
5      /* Enquanto não terminar o texto: */
6      while (s[i] != '\n' && s[i] != '\0') {
7          while (s[i] == ' ') /* Pula espaços */
8              i++;
9          /* Achou o começo de uma palavra ou o fim do texto: */
10         if (s[i] != '\n' && s[i] != '\0') { /* Se achou uma
11             ↪ palavra */
12             n++; /* Incrementa número de palavras */
13             while (s[i] != ' ' && s[i] != '\n' && s[i] != '\0')
14                 i++; /* Passa pela palavra */
15         }
16     }
17     printf("Total de palavras: %d\n", n);
18     return 0;
19 }
```

## Problema

Fazer um programa que acha todas as posições iniciais de ocorrência de uma palavra em um texto (de tam. no máximo 79, incluindo incluindo '\n').

Exemplo:

---

```
1 Texto="a lala lalaland"
2 Palavra="lala"
3
4 A resposta é 2, 7 e 9.
```

---

Ideia do algoritmo:

- Para cada possível posição no texto onde a palavra **pode** iniciar, precisamos checar se a palavra ocorre a partir daquela posição ou não.
- Seja  $\text{tamT}$  (resp.  $\text{tamP}$ ) o tamanho do texto (resp. tamanho da palavra).
- Note que as posições válidas onde a palavra pode iniciar no texto vão de 0 até  $\text{tamT} - \text{tamP}$ .

# Processamento de texto

```
1  int main() {
2      char tex[80], pal[80];
3      int i, j, iguais;
4      int tamP, tamT;
5
6      printf("Digite o texto: ");
7      fgets(tex, 80, stdin);
8      printf("Digite a palavra: ");
9      fgets(pal, 80, stdin);
10
11     tamP = strlen(pal) - 1;
12     tamT = strlen(tex) - 1; /* 0 "- 1" é devido ao \n */
13
14     for (i = 0; i <= tamT - tamP; i++) {
15         /* Dado i, testar se palavra ocorre a partir de
16            ↪ i */
17         ...
18     }
```

## Processamento de texto

Como testar se a palavra ocorre exatamente a partir de uma posição *i*? Checamos se todos os caracteres da palavra são iguais aos do texto a partir de *i*.

---

```
1      /* Dado i, testar se palavra ocorre a partir de
      ↪ i */
2      j = 0;
3      iguais = 1;
4      while (j < tamP && iguais) {
5          if (pal[j] != tex[i+j])
6              iguais = 0;
7          j++;
8      }
9      if (iguais)
10         printf("%d\n", i);
11 }
12 return 0;
13 }
```

# Vetores em funções

---



# Vetores em funções

- Vetores também podem ser passados como parâmetros em funções.
- Ao contrário dos tipos simples, vetores têm um comportamento diferente quando usados como parâmetros de funções.
- Quando uma variável simples é passada como parâmetro, seu valor é atribuído para uma nova variável local da função.
- No caso de vetores, **não é criado** um novo vetor!
- Isto significa que os valores de um vetor **são alterados** dentro de uma função!

# Vetores em funções

```
1  #include <stdio.h>
2
3  void fun1(int vet[], int tam) {
4      int i;
5      for (i = 0; i < tam; i++)
6          vet[i] = 5;
7  }
8
9  int main() {
10     int x[10], i;
11
12     for (i = 0; i < 10; i++)
13         x[i] = 8;
14
15     fun1(x, 10);
16     for (i = 0; i < 10; i++)
17         printf("%d\n", x[i]);
18
19     return 0;
20 }
```

# Vetores em funções

- Vetores não podem ser devolvidos por funções.
- Mesmo assim, podemos obter um resultado parecido com isso, usando o fato de que vetores são alterados dentro de funções.

---

```
1  int[] leVet() {  
2      int i, vet[100];  
3      for (i = 0; i < 100; i++) {  
4          printf("Digite um numero: ");  
5          scanf("%d", &vet[i]);  
6      }  
7  }
```

---

O código acima **não compila**, pois não podemos retornar um `int[]`.

# Vetores em funções

Como um vetor é alterado dentro de uma função, podemos criar a seguinte função:

---

```
1  #include <stdio.h>
2
3  void leVet(int vet[], int tam) {
4      int i;
5      for (i = 0; i < tam; i++) {
6          printf("Digite numero: ");
7          scanf("%d", &vet[i]);
8      }
9  }
10
11 void escreveVet(int vet[], int tam) {
12     int i;
13     for (i = 0; i < tam; i++)
14         printf("vet[%d] = %d\n", i, vet[i]);
15 }
```

---

# Vetores em funções

---

```
1  int main() {
2      int vet1[10], vet2[20];
3
4      printf(" ----- Vetor 1 -----\\n");
5      leVet(vet1, 10);
6      printf(" ----- Vetor 2 -----\\n");
7      leVet(vet2, 20);
8
9      printf(" ----- Vetor 1 -----\\n");
10     escreveVet(vet1, 10);
11     printf(" ----- Vetor 2 -----\\n");
12     escreveVet(vet2, 20);
13
14     return 0;
15 }
```

---

# Matrizes e vetores multidimensionais

---

# Matrizes e vetores multidimensionais

- Matrizes e vetores multidimensionais são generalizações de vetores simples vistos anteriormente.
- Suponha por exemplo que devemos armazenar as notas de cada aluno em cada laboratório de PE.
- Podemos alocar 15 vetores (um para cada lab) de tamanho 50 (tamanho da turma), onde cada vetor representa as notas de um laboratório específico.

---

```
1  double lab1[50], lab2[50], ..., lab15[50];
```

---

- Matrizes e vetores multidimensionais permitem fazer a mesma coisa, mas com todas as informações sendo acessadas por um único nome (ao invés de 15 nomes distintos).

# Declaração de matrizes

- A criação de uma matriz é feita com a seguinte sintaxe:

---

```
1  tipo nome_da_matriz[linhas][colunas];
```

---

onde **tipo** é o tipo de dados que a matriz armazenará, **linhas** (resp. **colunas**) é um inteiro que especifica o número de linhas (resp. colunas) que a matriz terá.

- A matriz criada equivale a **linhas** × **colunas** variáveis do tipo **tipo**.
- As linhas são numeradas de 0 a **linhas-1**.
- As colunas são numeradas de 0 a **colunas-1**.



## Exemplo de declaração de matriz

```
int matriz[4][4];
```

	0	1	2	3
0				
1				
2				
3				

## Acessando dados de uma matriz

- Em qualquer lugar onde você usaria uma variável no seu programa, você pode usar um elemento específico de uma matriz da seguinte forma:

---

```
1 nome_da_matriz[ind_linha][ind_coluna]
```

---

onde `ind_linha` (resp. `ind_coluna`) é um índice inteiro especificando a linha (resp. coluna) a ser acessada.

- No exemplo abaixo é atribuído para `aux` o valor armazenado na variável da 1ª linha e 11ª coluna da matriz:

---

```
1 int matriz[100][200];  
2 int aux;  
3 ...  
4 aux = matriz[0][10];
```

---

- O compilador não verifica se você utilizou valores válidos para acessar a linha e a coluna!
- Assim como vetores unidimensionais, comportamentos anômalos do programa podem ocorrer em caso de acesso a posições inválidas de uma matriz.

# Declarando vetores multidimensionais

- Para se declarar um vetor com 3 ou mais dimensões usamos a seguinte sintaxe:

---

```
1  tipo nome_vetor[d_1][d_2]...[d_n];
```

---

onde  $d_i$ , para  $i = 1, \dots, n$ , é um inteiro que especifica o tamanho do vetor na  $i$ -ésima dimensão.

- O vetor criado equivale  $d_1 \times d_2 \times \dots \times d_n$  variáveis do tipo **tipo**.
- Cada dimensão  $i$  pode ser acessada por índices entre 0 e  $d_i - 1$ .

## Declarando vetores multidimensionais

Você pode criar por exemplo uma matriz para armazenar a quantidade de chuva em um dado dia, mês e ano, para cada um dos últimos 3000 anos:

---

```
1 double chuva[3000][12][31];  
2  
3 chuva[1979][3][23] = 6.0;
```

---

## Exemplos com matrizes

Lendo uma matriz  $\ell \times c$  de inteiros:

---

```
1  for (i = 0; i < l; i++)  
2      for (j = 0; j < c; j++)  
3          scanf("%d", &mat[i][j]);
```

---

Imprimindo a matriz:

---

```
1  for (i = 0; i < l; i++) {  
2      for (j = 0; j < c; j++)  
3          printf("%d ", mat[i][j]);  
4      printf("\n");  
5  }
```

---

# Vetores multidimensionais e funções

- Ao passar um **vetor simples** como parâmetro, não é necessário fornecer o seu tamanho na declaração da função.
- Quando o vetor é multidimensional a possibilidade de não informar o tamanho na declaração se restringe à primeira dimensão apenas.

---

```
1 void mostra_matriz(int mat[][10], int n_linhas) {  
2     ...  
3 }
```

---

# Vetores multidimensionais e funções

- Pode-se criar uma função deixando de indicar a primeira dimensão:

---

```
1 void mostra_matriz(int mat[][10], int n_linhas) {  
2     ...  
3 }
```

---

- Ou pode-se criar uma função indicando todas as dimensões:

---

```
1 void mostra_matriz(int mat[5][10], int n_linhas) {  
2     ...  
3 }
```

---

- Mas não pode-se deixar de indicar outras dimensões (exceto a primeira):

---

```
1 void mostra_matriz(int mat[5][], int n_linhas) {  
2     /* ESTE NÃO FUNCIONA */  
3     ...  
4 }
```



# Vetores multidimensionais em funções

```
1 void mostra_matriz(int mat[][10], int n_linhas) {
2     int i, j;
3     for (i = 0; i < n_linhas; i++) {
4         for (j = 0; j < 10; j++)
5             printf("%2d ", mat[i][j]);
6         printf("\n");
7     }
8 }
9 int main() {
10     int mat[][10] = {
11         { 0,  1,  2,  3,  4,  5,  6,  7,  8,  9},
12         {10, 11, 12, 13, 14, 15, 16, 17, 18, 19},
13         {20, 21, 22, 23, 24, 25, 26, 27, 28, 29},
14         {30, 31, 32, 33, 34, 35, 36, 37, 38, 39},
15         {40, 41, 42, 43, 44, 45, 46, 47, 48, 49},
16         {50, 51, 52, 53, 54, 55, 56, 57, 58, 59},
17         {60, 61, 62, 63, 64, 65, 66, 67, 68, 69}};
18     mostra_matriz(mat, 7);
19     return 0;
20 }
```

# Vetores multidimensionais em funções

Lembre-se que vetores (multidimensionais ou não) **são alterados** quando passados como parâmetro em uma função.

---

```
1 void teste(int mat[2][2]) {
2     int i, j;
3     for (i = 0; i < 2; i++) {
4         for (j = 0; j < 2; j++) {
5             mat[i][j] = -1;
6         }
7     }
8 }
9
10 int main() {
11     int mat[2][2] = {{0, 1}, {2, 3}};
12     teste(mat);
13     /* Neste ponto mat tem quais valores em suas posições? */
14     return 0;
15 }
```

---

Criar aplicações com operações básicas sobre matrizes:

- Soma de 2 matrizes com dimensões  $\ell \times c$ .
- Multiplicação de 2 matrizes com dimensões  $\ell \times c$  e  $c \times t$ .

## Exemplo: lendo e imprimindo uma matriz

Código para fazer a leitura e a impressão de uma matriz:

---

```
1  #include <stdio.h>
2  #define MAX_SIZE 100
3
4  void readMat(double mat[MAX_SIZE][MAX_SIZE], int l, int c) {
5      int i, j;
6      /* leitura linha por linha: */
7      for (i = 0; i < l; i++)
8          for (j = 0; j < c; j++)
9              scanf("%lf", &mat[i][j]);
10 }
```

---

- `MAX_SIZE` é uma constante inteira definida com valor 100 (tam. máx. matriz).
- Note porém que o tamanho efetivo da matriz é o número de linhas  $l \leq 100$  e colunas  $c \leq 100$  passado como parâmetros.

## Exemplo: lendo e imprimindo uma matriz

Agora o código da função que faz a impressão de uma matriz:

```
1 void printMat(double mat[MAX_SIZE][MAX_SIZE], int l, int c) {  
2     int i, j;  
3  
4     /* impressão linha por linha: */  
5     for (i = 0; i < l; i++) {  
6         for (j = 0; j < c; j++)  
7             printf("%.2lf \t", mat[i][j]);  
8         printf("\n");  
9     }  
10 }
```

Para imprimir linha por linha, fixada uma linha  $i$ , imprimimos todas colunas  $j$  desta linha e ao final do laço sobre  $j$ , imprimimos uma quebra de linha, para impressão da próxima linha.

## Exemplo: lendo e imprimindo uma matriz

```
1  #include <stdio.h>
2  #define MAX_SIZE 100
3
4  void readMat(double mat[MAX_SIZE][MAX_SIZE], int l, int c);
5  void printMat(double mat[MAX_SIZE][MAX_SIZE], int l, int c);
6
7  int main() {
8      double m1[MAX_SIZE][MAX_SIZE], m2[MAX_SIZE][MAX_SIZE],
9          ↪ m3[MAX_SIZE][MAX_SIZE];
10     int l1, c1, l2, c2, l3, c3;
11
12     scanf("%d %d", &l1, &c1);
13     scanf("%d %d", &l2, &c2);
14     readMat(m1, l1, c1);
15     readMat(m2, l2, c2);
16     printMat(m1, l1, c1);
17     printMat(m2, l2, c2);
18
19     return 0;
20 }
```

## Exemplo: soma de matrizes

Vamos implementar a funcionalidade de soma de matrizes.

A função recebe como parâmetro as matrizes que devem ser somadas e também a matriz resposta `mat3`.

```
1  int soma(double mat1[][MAX_SIZE], int l1, int c1, double  
   ↪ mat2[][MAX_SIZE], int l2, int c2, double mat3[][MAX_SIZE]) {  
2      int i, j;  
3  
4      if (c1 != c2 || l1 != l2)  
5          return 0;  
6  
7      /* Código que soma as matrizes */  
8      return 1;  
9  }
```

A função devolve 1 se a soma foi feita (matrizes de dimensões compatíveis) ou 0 caso contrário (matrizes de dimensões incompatíveis).

## Exemplo: soma de matrizes

Para realizar a soma, para cada posição  $(i,j)$  da matriz resposta fazemos

$\text{mat3}[i][j] = \text{mat1}[i][j] + \text{mat2}[i][j]$ ; de forma o resultado da soma das matrizes estará em  $\text{mat3}$ .

---

```
1  int soma(double mat1[][MAX_SIZE], int l1, int c1, double
   ↪  mat2[][MAX_SIZE], int l2, int c2, double mat3[][MAX_SIZE]) {
2      int i, j;
3
4      if (c1 != c2 || l1 != l2)
5          return 0;
6
7      for (i = 0; i < l1; i++)
8          for (j = 0; j < c1; j++)
9              mat3[i][j] = mat1[i][j] + mat2[i][j];
10     return 1;
11 }
```

---



## Exemplo: soma de matrizes

Com as funções anteriores podemos alterar adicionar à função `main` o seguinte trecho de código:

---

```
1  int main() {
2      double m1[MAX_SIZE][MAX_SIZE],
   ↪  m2[MAX_SIZE][MAX_SIZE], m3[MAX_SIZE][MAX_SIZE];
3      int l1, c1, l2, c2, l3, c3;
4      ...
5      if (soma(m1, l1, c1, m2, l2, c2, m3)) {
6          l3 = l1;
7          c3 = c1;
8          printf("Resultado da soma:\n");
9          printMat(m3, l3, c3);
10     }
11     return 0;
12 }
```

---

## Exemplo: multiplicação de matrizes

- Vamos implementar a funcionalidade de multiplicação de matrizes.
- Vamos multiplicar duas matrizes  $M_1$  e  $M_2$  (de dimensões  $\ell_1 \times c_1$  e  $\ell_2 \times c_2$  com  $c_1 = \ell_2$ ).
- O resultado será uma terceira matriz  $M_3$  (de dimensões  $\ell_1 \times c_2$ ).
- Lembre-se que uma posição  $(i, j)$  de  $M_3$  terá o produto interno do vetor linha  $i$  de  $M_1$  com o vetor coluna  $j$  de  $M_2$ :

$$M_3[i, j] = \sum_{k=0}^{c_1-1} M_1[i, k] \cdot M_2[k, j]$$

## Exemplo: multiplicação de matrizes

O código da multiplicação está abaixo: para cada posição  $(i, j)$  de `mat3` devemos computar

$$\text{mat3}[i, j] = \sum_{k=0}^{c_1-1} \text{mat1}[i, k] \cdot \text{mat2}[k, j]$$

---

```
1  ...
2  for (i = 0; i < l1; i++) {
3      for (j = 0; j < c2; j++) {
4          mat3[i][j] = 0;
5          for (k = 0; k < c1; k++) {
6              mat3[i][j] = mat3[i][j] + (mat1[i][k] *
              ↪ mat2[k][j]);
7          }
8      }
9  }
10 ...
```

---

## Exemplo: multiplicação de matrizes

Abaixo temos a função que devolve 1 caso a multiplicação possa ser feita e 0 caso contrário.

```
1  int mult(double mat1[][MAX_SIZE], int l1, int c1, double
   ↪ mat2[][MAX_SIZE], int l2, int c2, double mat3[][MAX_SIZE]) {
2      int i, j, k;
3      if (c1 != l2)
4          return 0;
5
6      for (i = 0; i < l1; i++) {
7          for (j = 0; j < c2; j++) {
8              mat3[i][j] = 0;
9              for (k = 0; k < c1; k++)
10                 mat3[i][j] = mat3[i][j] + (mat1[i][k] *
   ↪ mat2[k][j]);
11         }
12     }
13     return 1;
14 }
```

## Exemplo: multiplicação de matrizes

Com as funções anteriores podemos alterar adicionar à função main o seguinte trecho de código:

---

```
1  int main() {
2      double m1[MAX_SIZE][MAX_SIZE], m2[MAX_SIZE][MAX_SIZE],
   ↪    m3[MAX_SIZE][MAX_SIZE];
3      int l1, c1, l2, c2, l3, c3;
4
5      ...
6
7      if (mult(m1, l1, c1, m2, l2, c2, m3)) {
8          l3 = l1;
9          c3 = c2;
10         printf("Resultado da multiplicacao:\n");
11         printMat(m3, l3, c3);
12     }
13     return 0;
14 }
```

---

## Informações extras: inicialização de matrizes

- No caso de matrizes, usa-se chaves para delimitar as linhas:

### Exemplo

```
int vet[2][5] = {  
    {10, 20, 30, 40, 50},  
    {60, 70, 80, 90, 100 } };
```

- No caso tridimensional, cada índice da primeira dimensão se refere a uma matriz inteira:

### Exemplo

```
int v3[2][3][4] = {  
    {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},  
    {{0, 0, 0, 0}, {5, 6, 7, 8}, {0, 0, 0, 0}}};
```

## Informações extras: inicialização de matrizes

---

```
1  int main() {
2      int i, j, k;
3      int v1[5] = {1,2,3,4,5};
4      int v2[2][3] = {{1,2,3}, {4,5,6}};
5      int v3[2][3][4] = {
6          {{1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12}},
7          {{0, 0, 0, 0}, {5, 6, 7, 8}, {0, 0, 0, 0}}};
8      ...
9  }
```

---

+Recursão

---



## Exemplo: soma de elementos de um vetor

### Problema

Dado vetor  $v$  de tamanho  $tam$ , calcular a soma dos elementos da posição 0 até  $tam - 1$ .

- Como podemos descrever este problema de forma recursiva? Isto é, como podemos descrever este problema em função de si mesmo?
- Vamos denotar por  $S(n)$  a soma dos elementos das posições 0 até  $n$  do vetor. Portanto, devemos achar  $S(tam - 1)$ .
- O valor de  $S(n)$  pode ser calculado com a seguinte definição recursiva:
  - Se  $n = 0$ , então  $S(0) = v[0]$ .
  - Se  $n > 0$ , então  $S(n) = v[n] + S(n - 1)$ .

---

```
1  int soma(int v[], int n) {  
2      if (n == 0)  
3          return v[0];  
4      else  
5          return v[n] + soma(v, n-1);  
6  }
```

---

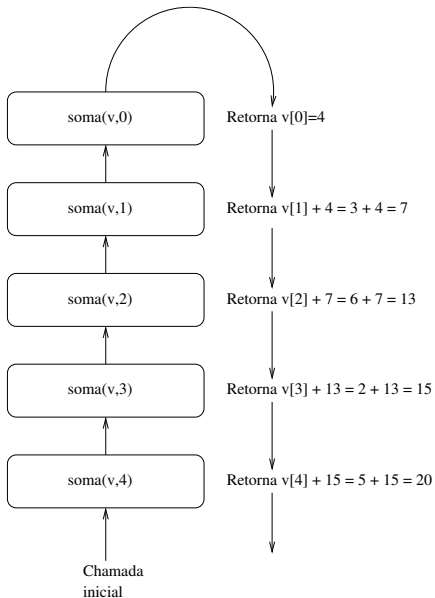
## Algoritmo em C – exemplo de uso

```
1  #include <stdio.h>
2
3  int soma(int v[], int n);
4
5  int main() {
6      int vet[5] = {4, 3, 6, 2, 5};
7      printf("%d\n", soma(vet, 4));
8      return 0;
9  }
10
11 int soma(int v[], int n) {
12     if (n == 0)
13         return v[0];
14     return v[n] + soma(v, n-1);
15 }
```

Na chamada, o segundo parâmetro é o índice da última posição do vetor.

# Exemplo de execução

$V = (4, 3, 6, 2, 5)$



Neste problema, a solução iterativa seria melhor (não há criação de variáveis das chamadas recursivas):

---

```
1 int calcula_soma(int[] v, int n) {  
2     int soma = 0, i;  
3     for (i = 0; i <= n; i++)  
4         soma = soma + v[i];  
5     return soma;  
6 }
```

---

Informações extras: biblioteca  
`string.h`

---

# Biblioteca `string.h`

A função `strcat` faz concatenação de strings.

Ela recebe duas strings como parâmetro e concatena a string dada no segundo parâmetro no final da string dada no primeiro parâmetro.

Deve haver espaço suficiente na primeira string, caso contrário ocorrerá um erro.

---

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char s1[80] = "ola ", s2[80] = "turma de PE!";
6      /* concatena s2 no final de s1: */
7      strcat(s1, s2);
8      printf("%s\n", s1);
9      return 0;
10 }
```

---

A função `strcmp` compara duas strings.

Ela recebe duas strings `s1` e `s2` como parâmetro e devolve:

- 0, caso as duas strings sejam iguais.
- um valor menor que 0, caso `s1` seja lexicograficamente menor que `s2`.
- um valor maior que 0, caso `s1` seja lexicograficamente maior que `s2`.



# Biblioteca string.h

---

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char s1[80] = "aab", s2[80] = "aac";
6      int r;
7      r = strcmp(s1, s2);
8      if (r < 0)
9          printf("%s vem antes que %s\n", s1, s2);
10     else if (r > 0)
11         printf("%s vem antes que %s\n", s2, s1);
12     else
13         printf("sao iguais\n");
14     return 0;
15 }
```

---

# Biblioteca `string.h`

A função `strcpy` faz cópia de strings.

Ela recebe duas strings como parâmetro e copia a string dada no segundo parâmetro na string dada no primeiro parâmetro.

---

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char s1[80], s2[80] = "ola pessoal";
6      strcpy(s1, s2);
7      printf("%s\n", s1);
8      return 0;
9  }
```

---

A saída será

---

```
1  ola pessoal
```

---

# Biblioteca string.h

A função `strlen` calcula o tamanho de uma string.

Ela recebe uma string como parâmetro e devolve o número de caracteres na string até o `'\0'`.

---

```
1  #include <stdio.h>
2  #include <string.h>
3
4  int main() {
5      char s1[80] = "ola pessoal";
6      int t = strlen(s1);
7      printf("%d\n", t);
8      return 0;
9  }
```

---

A saída será

---

```
1  11
```

---

Informações extras: representação  
de matrizes por linearização

---

- Podemos usar sempre vetores simples para representar matrizes (na prática o compilador faz isto por você).
- Ao declarar uma matriz como `int mat[3][4]`, sabemos que serão alocadas 12 posições de memória associadas com a variável `mat`.
- Poderíamos simplesmente criar `int mat[12]`, mas perderíamos a simplicidade de uso dos índices em forma de matriz.
  - Você não mais poderá escrever `mat[1][3]`, por exemplo.

- A *linearização de índices* é justamente a representação de matrizes usando-se um vetor simples.
- Mas devemos ter um padrão para acessar as posições deste vetor como se sua organização fosse na forma de matriz.

- Considere o exemplo:

```
int mat[12]; /* em vez de int mat[3][4] */
```

- Fazemos a divisão por linhas como segue:

- Primeira linha: `mat[0]` até `mat[3]`
- Segunda linha: `mat[4]` até `mat[7]`
- Terceira linha: `mat[8]` até `mat[11]`

- Para acessar uma posição `[i][j]` usamos:

```
mat[i * 4 + j];
```

onde  $0 \leq i \leq 2$  e  $0 \leq j \leq 3$ .

## Informações extras: linearização de índices

- De forma geral, seja matriz `mat[n * m]`, representando `mat[n][m]`.
- Para acessar a posição correspondente à `[i][j]` usamos:  
`mat[i * m + j];`  
onde  $0 \leq i \leq n - 1$  e  $0 \leq j \leq m - 1$ .
- Note que  $i$  “pula” blocos de tamanho  $m$ , e  $j$  indexa a posição dentro de um bloco.



## Informações extras: linearização de índices

- Podemos estender para mais dimensões.
- Seja matriz `mat[n * m * q]`, representando `mat[n][m][q]`.
  - As posições de 0 até  $(m * q) - 1$  são da primeira matriz.
  - As posições de  $(m * q)$  até  $(2 * m * q) - 1$  são da segunda matriz.
  - etc...
- Para acessar a posição correspondente à `[i][j][k]` usamos:  
`mat[i * m * q + j * q + k];`

## Informações extras: linearização de índices

```
1  int main() {  
2      int mat[40]; /* representando mat[5][8] */  
3      int i, j;  
4  
5      for (i = 0; i < 5; i++)  
6          for(j = 0; j < 8; j++)  
7              mat[i * 8 + j] = i * j;  
8  
9      for (i = 0; i < 5; i++) {  
10         for(j = 0; j < 8; j++)  
11             printf("%d, ", mat[i * 8 + j]);  
12         printf("\n");  
13     }  
14     return 0;  
15 }
```

# Exercícios

---

## Exercício 1

Escreva um programa que lê 10 números inteiros e os salva em um vetor. Em seguida o programa deve encontrar a posição do maior elemento do vetor e imprimir esta posição.

## Exercício 2

Escreva um programa que lê 10 números ponto flutuante e os salva em um vetor. Em seguida o programa deve calcular a média dos valores armazenados no vetor e imprimir este valor.

## Exercício 3

Escreva um programa que lê 10 números inteiros e os salva em um vetor. Em seguida o programa deve ler um outro número inteiro  $C$ . O programa deve então encontrar dois números em posições distintas do vetor cuja multiplicação seja  $C$  e imprimí-los. Caso não existam tais números, o programa deve informar isto.

**Exemplo:** Se  $\text{vetor} = (2, 4, 5, -10, 7)$  e  $C = 35$ , então o programa deve imprimir “5 e 7”. Se  $C = -1$ , então o programa deve imprimir “Não existem tais números”.

## Exercício 4

Escreva um programa que lê uma string de até 50 caracteres, e imprime “Palindromo” caso a string seja um palindromo e “Nao Palindromo” caso contrário.

Obs.: Um palíndromo é uma palavra ou frase que é igual quando lida da esquerda para a direita ou da direita para a esquerda (espaços em brancos são descartados).

Assuma que as palavras são todas escritas em letras minúsculas e sem acentos.

Exemplo de palíndromo: “saudavel leva duas”.

## Exercício 5

Refaça o exemplo visto em aula de inversão de uma string de tal forma que não seja utilizado nenhum vetor adicional!

Isto é, devemos computar a inversa no próprio vetor onde a string foi lida.



## Exercício 6

Faça um programa para realizar operações com matrizes que tenha as seguintes funcionalidades:

- Um menu para escolher a operação a ser realizada:
  1. Leitura de uma matriz  $M_1$ .
  2. Leitura de uma matriz  $M_2$ .
  3. Impressão das matrizes  $M_1$  e  $M_2$ .
  4. Impressão da soma de  $M_1$  com  $M_2$ .
  5. Impressão da multiplicação de  $M_1$  com  $M_2$ .
  6. Impressão da subtração de  $M_2$  de  $M_1$ .
  7. Impressão da transposta de  $M_1$  e  $M_2$ .

## Exercício 7

Escreva um programa que leia todas as posições de uma matriz  $10 \times 10$ .

O programa deve então exibir o número de posições não nulas na matriz.

## Exercício 8

Escreva um programa que lê todos os elementos de uma matriz  $4 \times 4$  e mostra a matriz e a sua transposta na tela.

Matriz	Transposta
$\begin{bmatrix} 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 2 \\ 0 & 1 & 0 & 2 \end{bmatrix}$	$\begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 2 & 2 & 2 & 2 \end{bmatrix}$

## Exercício 9

Escreva um programa que lê uma matriz do teclado e então imprime os elementos com menor e maior frequência de ocorrência na matriz.

## Exercício 10

Escreva um algoritmo iterativo que, dado um inteiro  $n$  e um vetor de inteiros  $v$  de tamanho  $t$ , devolve um inteiro  $i$  tal que  $v[i] == n$  ou  $-1$  caso  $n$  não esteja presente em  $v$ .

Reescreva o algoritmo acima de maneira recursiva.

## Exercício 11

Escreva um programa que lê uma palavra do teclado e então imprime todas as permutações desta palavra.

Se por exemplo for digitado “abca” o seu programa deveria imprimir:

---

1	aabc
2	aacb
3	abac
4	abca
5	acab
6	acba
7	baac
8	bach
9	bcac
10	caab
11	caba
12	cbaa

---

## Exercício 12

Mostre a execução da função `imprime`. O que será impresso?

---

```
1  #include <stdio.h>
2
3  void imprime(int v[], int i, int n);
4
5  int main() {
6      int vet[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
7      imprime(vet, 0, 9);
8      printf("\n");
9      return 0;
10 }
11 void imprime(int v[], int i, int n) {
12     if (i == n) {
13         printf("%d, ", v[i]);
14     } else {
15         imprime(v, i+1, n);
16         printf("%d, ", v[i]);
17     }
18 }
```