

Aula 05:

Vetores e Matrizes

Prof. Edvard

edvard@unifei.edu.br

Universidade Federal de Itajubá

Constantes

- Sabemos que uma **variável** é uma posição na memória onde podemos guardar determinado dado ou valor e modificá-lo ao longo da execução do programa.
- Uma **constante** permite guardar determinado dado ou valor na memória do computador e garante que ele não será alterado durante a execução do programa: Será sempre o mesmo e, portanto, **constante**.
 - Em C, existem duas maneiras de se criar um valor e informar o compilador de que aquele valor não deve ser nunca alterado: o comando **#define** e a palavra reservada **const**.

#define

- Segue a forma geral:
`#define NOME_DA_CONSTANTE valor_da_constante`
- O comando `#define` é uma **diretiva de compilação** (assim como `#include`), que informa ao compilador que ele deve procurar, no código, todas as ocorrências da palavra definida por `NOME_DA_CONSTANTE` e substituir por `valor_da_constante` quando o programa for compilado.
- Vejamos um exemplo:

#define

```
#include <stdio.h>

#define PI 3.141592
#define LARGURA_RAIA 1

int main()
{
    float raio, d;
    int i;

    printf("Entre com o raio ate o centro da primeira raia: ");
    scanf("%f", &raio);

    for(i = 0; i < 5; i++)
    {
        d = 2*PI*(raio + i*LARGURA_RAIA); // utilizo normalmente, como se fosse uma variável.
        printf("Distancia raia %d: %.5f\n", (i+1), d);
    }

    return 0;
}
```

Repare que é possível declarar constantes de qualquer tipo, inclusive **float** e **int**.

Para declarar uma constante do tipo **char**, escreva, por exemplo:

```
#define LETRA 'a'
```

Para o programa, é equivalente a escrever:

```
d = 2*3.141592*(raio + i*1);
```

const

- Uma outra maneira de declarar uma constante é utilizando a palavra reservada `const` seguindo a forma geral:

const tipo_da_constante nome_da_constante = valor_da_constante;

- Note que a declaração de uma constante é, na verdade, uma declaração de variável.
 - A palavra `const` informa ao programa que aquela “variável” não poderá ter seu valor modificado.
- Vejamos o mesmo exemplo anterior, agora com `const`.

const

```
#include <stdio.h>

int main()
{
    // Variáveis
    float raio, d;
    int i;

    // Constantes
    const float pi = 3.141592;
    const int largura_raia = 1;

    printf("Entre com o raio ate o centro da primeira raia: ");
    scanf("%f", &raio);

    for(i = 0; i < 5; i++)
    {
        // utilizo normalmente, como se fosse uma variável.
        d = 2*pi*(raio + i*largura_raia);
        printf("Distancia raia %d: %.5f\n", (i+1), d);
    }

    return 0;
}
```

Existe uma convenção: Normalmente, quando definimos a constante utilizando #define, colocamos seu nome somente com caracteres maiúsculos. Com const, seguimos as regras de nomes de variáveis.

A tentativa de alteração no valor de uma constante gera **erros de compilação** em ambos os casos (#define e const)

Vetores e Matrizes

- As variáveis declaradas até agora em nossos programas são capazes de armazenar um único valor por vez.
 - Quando precisamos atribuir um novo valor para aquela variável, **seu valor anterior é perdido**.
 - Isso ocorre porque cada variável está associada a uma **única posição a memória**, onde é possível guardar um único valor **do tipo especificado** em sua declaração.
- Pensando logicamente:
 - Portanto, para guardar mais valores, precisamos de **mais variáveis!**

Vetores e Matrizes

- Imagine um programa simples para guardar notas de alunos e calcular suas médias. Um programa para realizar esta tarefa para **um único estudante** seria da seguinte maneira (como já vimos exaustivamente!!):

```
#include <stdio.h>

int main()
{
    int nota1, nota2, media;

    printf("Insira as notas do aluno 1: ");
    scanf("%d %d", &nota1, &nota2);

    media = (nota1 + nota2)/2;

    printf("A media eh %d", media);

    return 0;
}
```

Agora, e se for necessário que o programa mantenha as notas de **todos os alunos** de uma turma? Como você faria?
Vamos fazer o programa para uma turma de apenas 10 alunos.


```

#include <stdio.h>

int main()
{
    int nota1_aluno1, nota2_aluno1, media_aluno1;
    int nota1_aluno2, nota2_aluno2, media_aluno2;
    int nota1_aluno3, nota2_aluno3, media_aluno3;
    int nota1_aluno4, nota2_aluno4, media_aluno4;
    int nota1_aluno5, nota2_aluno5, media_aluno5;
    int nota1_aluno6, nota2_aluno6, media_aluno6;
    int nota1_aluno7, nota2_aluno7, media_aluno7;
    int nota1_aluno8, nota2_aluno8, media_aluno8;
    int nota1_aluno9, nota2_aluno9, media_aluno9;
    int nota1_aluno10, nota2_aluno10, media_aluno10;

    // aluno 1
    printf("Insira as notas do aluno 1: ");
    scanf("%d %d", &nota1_aluno1, &nota2_aluno1);
    media_aluno1 = (nota1_aluno1 + nota2_aluno1)/2;
    printf("A media eh %d\n", media_aluno1);

    // aluno 2
    printf("Insira as notas do aluno 2: ");
    scanf("%d %d", &nota1_aluno2, &nota2_aluno2);
    media_aluno2 = (nota1_aluno2 + nota2_aluno2)/2;
    printf("A media eh %d\n", media_aluno2);

    // ...

    // aluno 10
    // aluno 1
    printf("Insira as notas do aluno 10: ");
    scanf("%d %d", &nota1_aluno10, &nota2_aluno10);
    media_aluno10 = (nota1_aluno10 + nota2_aluno10)/2;
    printf("A media eh %d\n", media_aluno10);

    return 0;
}

```

Olhe a quantidade de código repetitivo, e o **elevadíssimo número de variáveis** que você precisará tomar conta em seu código.

Repare que estamos fazendo a **mesma coisa dez vezes seguidas**, e não podemos utilizar um laço de repetição, uma vez que cada vez utilizamos variáveis diferentes.

Vetores e Matrizes

- Como podemos notar, temos uma solução extremamente engessada para o nosso problema.
 - Modificar o número de alunos, por exemplo, para 30 ou 50, significaria reescrever quase que completamente o programa.
 - Grande quantidade de variáveis para gerenciar.
 - Maior probabilidade de erros
 - Muitos nomes, significando todos quase a mesma coisa.
- Surge, portanto, a necessidade de se criar **vetores**, também chamados de **arrays**, em inglês.
 - Como as variáveis do programa **possuem uma relação entre si** (são todas notas de alunos), podemos declará-las com um único nome, e acessá-las através de índices: nota do aluno 1, nota do aluno 2, nota do aluno 27, e assim por diante.
- Vejamos como ficaria:

Vetores e Matrizes

Todos eles têm capacidade para armazenar **NUMERO_ALUNOS** notas, ou seja, 10 notas.

```
#include <stdio.h>

#define NUMERO_ALUNOS 10

int main()
{
    int nota1[NUMERO_ALUNOS], nota2[NUMERO_ALUNOS], media[NUMERO_ALUNOS];
    int i;

    // Ua só estrutura repete o código que é quase identico
    for(i = 0; i < NUMERO_ALUNOS; i++)
    {
        printf("Insira as notas do aluno %d: ", i+1);
        scanf("%d %d", &nota1[i], &nota2[i]);
        media[i] = (nota1[i] + nota2[i])/2;
        printf("A media eh %d\n", media[i]);
    }

    return 0;
}
```

Três vetores são declarados.

- **Um deles guardará todas as notas 1 dos alunos**
- **Outro guardará todas as notas 2**
- **O último guardará as médias de todos os alunos**

Todo o código é condensado em uma estrutura de repetição do tipo **for**, deixando o código mais limpo e conciso.

Vetores e Matrizes

- **Vetores**, ou *arrays*, são a forma mais simples de estrutura de dados em linguagem C e podem, também, ser chamados de **estruturas de dados homogêneas**.
- São, na verdade, um conjunto de variáveis do **mesmo tipo** associadas a um **mesmo nome**, mas acessadas através de **índices diferentes**, colocados entre colchetes.
 - **Homogêneas?** Sim! Sempre o mesmo tipo de dado.
 - Alocados sequencialmente na memória, ou seja, lado a lado.
- Sua declaração segue a forma geral:

tipo_dado nome_vetor[tamanho];

```
// exemplos
int notas[100];
float distancias[20];
double concentracoes[10];
char texto_descricao[1000]; // uma string!
```

O tamanho do vetor deve ser SEMPRE uma constante do tipo **inteiro**.

Vetores e Matrizes

- Como a variável que armazena todos os valores de uma série possui um único nome, seus elementos precisam ser **acessados individualmente através de índices**, colocados entre colchetes na frente do nome da variável.
 - Nomes de vetores seguem as mesmas regras de nomes de variáveis (letras, números e *underscores*, mas sem começar com números).
- **Exemplo:**

```
// declaração
int notas[5];

// acesso
notas[0] = 90;
notas[1] = 82;
notas[2] = 56;
notas[3] = 47;
notas[4] = 79;
```

Em C, o vetor sempre se inicia no índice 0. Ao especificar o índice 1, estamos na verdade referenciando seu segundo elemento. No exemplo, se tentássemos acessar o índice 5, nosso programa dá erro de execução.

Veja como ficaria sua **disposição na memória:**

0	1	2	3	4
90	82	56	47	79

Vetores e Matrizes

- Portanto, repare como existem **duas situações** onde iremos utilizar os **colchetes**:
 1. Na **declaração**: significa o tamanho do vetor.
 - Se declararmos um vetor de tamanho 10, ele terá índices entre 0 e 9.
 2. Na **utilização**: significa o índice do elemento que queremos acessar.
 - Em C, o primeiro índice é SEMPRE igual a 0.

```
// declaração
float temperaturas[10]; // 10 é a quantidade de elementos

// acesso
temperaturas[0] = 985.23; // 0 é o primeiro elemento
...
temperaturas[9] = 790.60; // 9 é o décimo elemento
```

Vetores e Matrizes

- Deve-se sempre **atribuir valores** aos elementos de um vetor antes de utilizá-los no código, assim como fazemos como variáveis comuns.
 - Devemos enxergar **cada elemento** do vetor como uma **variável simples**, que faz parte de uma estrutura que a relaciona diretamente a alguns outros valores.
- Como atribuímos valor?
 - Podemos atribuir valor no **decorrer do nosso programa**, utilizando o operador de atribuição (=) e o índice do elemento;
 - Ou atribuir valores já no momento da declaração. Este procedimento é chamado de “**inicialização do vetor**”.

Vetores e Matrizes

- Para atribuir valores no decorrer do programa, basta acessarmos o elemento de interesse através do **índice**.

Exemplos:

```
coordenada_x[20] = 22.4567; // atribuição de valor literal
coordenada_y[18] = x; // atribuição de valor de outra variável
distancia[15] = coordenada_c[15] * 2 + coordenada_y[15]; // atribuição de valor com expressão
```

- Quando temos muitos elementos, o método de atribuição individual pode não ser o mais eficiente.
- Para tanto, podemos inicializar o vetor, no momento de sua declaração, com TODOS os valores do vetor, separados por vírgula. Exemplo:

```
// inicialização de vetor unidimensional
int fibonacci[10] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
```

Somente funciona na inicialização!

Vetores e Matrizes

- Quando se inicializa um vetor no momento de sua declaração, seu tamanho pode ser determinado automaticamente pela quantidade de itens da inicialização.
 - Não é necessário especificar o tamanho.
 - O próprio compilador contará o número de valores dentro das chaves e **declarará um vetor de tamanho adequado**.

```
// inicialização sem tamanho especificado
int fibonacci[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
int n[] = {10, 20, 30, 40, 50}; // 5 posições
float k[] = {2.45, 32.1, 54.6}; // 3 posições
```

Vetores e Matrizes

- Os vetores de caracteres possuem algumas características próprias e são chamados de **strings**, ou **cadeias de caracteres**.
- Sua inicialização pode ser feita das seguintes maneiras:

```
// inicialização de cadeias de caracteres
char universidade[] = {'U', 'N', 'I', 'F', 'E', 'I'};
char universidade[] = "UNIFEI"; // Aspas duplas
```

- Informações Importantes:
 - A utilização de vetores de dados permite utilizar **comandos de repetição** sobre tarefas que sejam intrinsecamente repetitivas, e devem ser realizadas de forma idêntica para cada posição do vetor, apenas modificando-se o índice.
 - Cada posição do *array* é uma variável, identificada por um índice. **O tempo para acessar qualquer posição do array é o mesmo.**

Vetores e Matrizes

- **Exemplo:**

- Faça um programa que receba uma série de 10 números inteiros positivos, guarde em um vetor e identifique qual, entre eles, é o **maior**.

```
#include <stdio.h>

int main()
{
    int vetor[10], i, maior = -1;

    // laço de repetição variando entre 0 e 9
    for(i = 0; i < 10; i++)
    {
        printf("Entre com um valor inteiro positivo: ");
        scanf("%d", &vetor[i]); // utilização com & comercial

        // se o valor digitado for maior do que o maior deles até agora, troque
        if(vetor[i] > maior)
            maior = vetor[i];
    }

    printf("O maior numero digitado foi %d! \n\n", maior);

    return 0;
}
```

Hands-On!

- **Exercício:**

- Faça um programa que receba uma série de 10 números inteiros positivos e os coloque em um vetor. Em seguida, calcule a **soma de todos os elementos** daquele vetor.

```
#include <stdio.h>

int main()
{
    int vetor[10], i, soma = 0;

    // laço de repetição variando entre 0 e 9
    for(i = 0; i < 10; i++)
    {
        printf("Entre com um valor inteiro positivo: ");
        scanf("%d", &vetor[i]); // utilização com & comercial
    }

    for(i = 0; i < 10; i++)
        soma += vetor[i];

    printf("A soma dos numeros do vetor é igual a %d! \n\n", soma);

    return 0;
}
```

Vetores e Matrizes

- **Exemplo:**

- Faça um programa que receba uma série de 10 caracteres e guarde-os em um vetor. Em seguida, **imprima-os em ordem inversa.**

```
#include <stdio.h>

int main()
{
    char caracteres[10], i;

    // laço de repetição variando entre 0 e 9
    for(i = 0; i < 10; i++)
    {
        printf("Entre com um caractere: ");
        scanf("%c", &caracteres[i]); // utilização com & comercial
        fflush(stdin); // Limpa buffer de entrada
    }

    printf("\nCaracteres digitados, em ordem inversa: \n");
    for(i = 9; i >= 0; i--)
        printf("%c ", caracteres[i]);

    printf("\n\n");
    return 0;
}
```

Hands-On!

- **Exercício:**

- Entre com um vetor de float de cinco posições. A seguir, calcule a **média de todos os seus valores**.

```
#include <stdio.h>

int main()
{
    float vetor[5], soma = 0, media;
    int i;

    // laço de repetição variando entre 0 e 4
    for(i = 0; i < 5; i++)
    {
        printf("Entre com um valor real: ");
        scanf("%f", &vetor[i]); // utilização com & comercial
    }

    for(i = 0; i < 5; i++)
        soma += vetor[i];

    media = soma / 5;

    printf("A media dos numeros do vetor é igual a %.2f! \n\n", media);

    return 0;
}
```

Vetores e Matrizes

- Os arrays vistos até aqui possuem apenas uma dimensão e, portanto, são tratados como uma **lista sequencial de variáveis**, como na figura abaixo:

25.1	34.2	5.25	7.45	6.09	7.54
0	1	2	3	4	5

- No entanto, há casos onde uma estrutura com duas dimensões pode ser útil.
 - Nesse caso, utilizamos um arranjo de linhas e colunas, parecido com uma tabela.
- Um array com uma única dimensão é chamado de **vetor**.
- Um array com duas dimensões é chamado de **matriz**.

Vetores e Matrizes

- A melhor forma de pensar em uma **matriz** é comparando-a com uma **tabela**. Nela, um determinado elemento será acessado através de dois índices, um para linha e outro para o número da coluna.
- Veja a figura:

	0	1	2	3
0	51	52	53	53
1	54	55	56	56
2	51	52	53	53

- Sua declaração segue a forma geral:
tipo_dado nome_vetor[n_linhas][n_colunas];

O número de linhas e de colunas deve ser SEMPRE uma constante do tipo **inteiro**.

Vetores e Matrizes

- Informações importantes:
 - Uma matriz apenas guarda valores de um mesmo tipo. Também é uma **estrutura de dados homogênea**.
 - O número de linhas e de colunas **não precisa ser sempre o mesmo**.
 - Os índices de linha e coluna, assim como nos vetores unidimensionais, **inicia-se em 0**.
- Exemplos de declaração:

```
// declaração de matrizes
int tabuleiro[10][10]; // 10 linhas e 10 colunas
float plant[3][5]; // 3 linhas e 5 colunas
char mat[2][10]; // 2 linhas e 10 colunas
```

Vetores e Matrizes

- O **acesso aos elementos da matriz** é sempre realizado através de dois índices – **linha** e **coluna** – como vemos abaixo:

`a[1][4] = 10;`

	0	1	2	3	4	5
0						
1					10	

`mat[2][1] = 3.45;`

	0	1	2
0			
1			
2		3.45	

- Sempre utilize **dois pares de colchetes** para acessar um elemento individual da matriz, na forma geral:
 - `matriz[indice_linha][indice_coluna]`

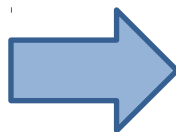
Vetores e Matrizes

- As matrizes também podem ser **inicializadas** no momento de sua declaração, da seguinte maneira:

```
// Inicialização de matrizes
int matriz[2][3] = { {1, 2, 3}, {4, 5, 6} };
int matriz[2][3] = { {1, 2, 3},
                     {4, 5, 6} };
```

- Colocamos os elementos de cada uma das linhas separados por vírgulas dentro de chaves, e depois separamos as linhas também com vírgulas.
 - A segunda maneira é equivalente à primeira, mas já nos provê com uma representação da matriz mais próxima da que imaginamos:

```
{ {1, 2, 3},
  {4, 5, 6} }
```



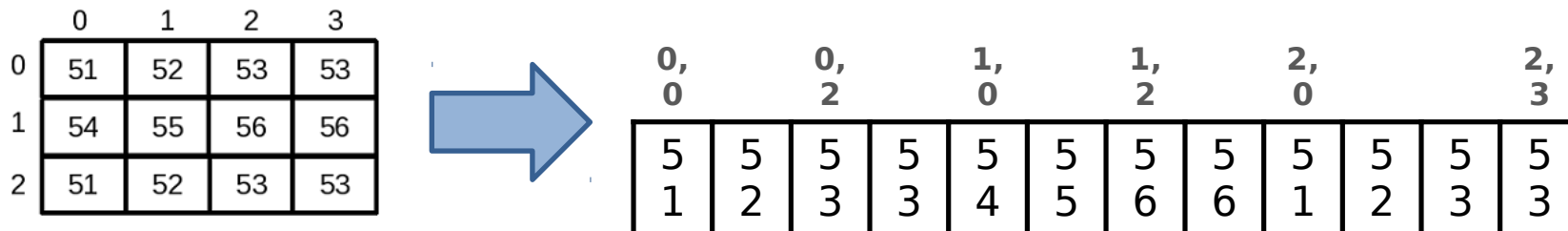
	0	1	2
0	1	2	3
1	4	5	6

Vetores e Matrizes

- No caso da matriz, é sempre necessário definir a quantidade de colunas no ato da declaração. Apenas a **primeira dimensão pode ficar sem tamanho indefinido**.

```
// Inicialização de matrizes sem numero de linhas  
int matriz[][3] = { {1, 2, 3}, {4, 5, 6} };  
int array2[][2] = {{10, 20}, {30, 40}, {50, 60}};
```

- Em matrizes, a utilização de colchetes nos dá a ilusão de estarmos trabalhando em um espaço de memória bidimensional. No entanto, **isso não ocorre**. Todos os itens de uma matriz são alocados sequencialmente na memória, linha após linha. **Abstração!**



Vetores e Matrizes

- **Exemplo: Soma de Matrizes**

- Ler duas matrizes de inteiros A e B, cada uma de duas dimensões com 2 linhas e 3 colunas. Construir uma matriz C de mesma dimensão, onde C é formada pela soma dos elementos da matriz A com os elementos da matriz B.

- **Observação**

- Interessante notar que, enquanto um vetor (unidimensional) demandava a utilização de apenas uma estrutura do tipo “for” para ser percorrido completamente, uma matriz (bidimensional) precisa que utilizemos **dois laços aninhados**: **Um para contar as linhas e outro para contar as colunas.**

```
printf("Matriz A: \n");
for(i = 0; i < 2; i++) // um for irá percorrer as linhas
{
    for(j = 0; j < 3; j++) // para cada linha, outro for percorrerá as colunas
    {
        printf("Entre com o elemento na linha %d e coluna %d: ", i, j);
        scanf("%d", &matA[i][j]);
    }
}
```

Hands-On!

- **Exercício:**

- Faça um programa que declare e inicialize uma **matriz identidade de ordem 10**. A seguir, imprima-a na tela.

[illegible]

Hands-On!


```
#include <stdio.h>

int main()
{
    int id[10][10];
    int i, j;

    printf("Matriz identidade: \n");
    for(i = 0; i < 10; i++) // um for irá percorrer as linhas
    {
        for(j = 0; j < 10; j++) // para cada linha, outro for percorrerá as colunas
        {
            if(i == j)
                id[i][j] = 1;
            else
                id[i][j] = 0;
            printf("%d ", id[i][j]);
        }
        printf("\n"); // Pula uma linha após último elemento na linha
    }

    return 0;
}
```

Substituição em Vetor I

Adaptado por Neilor Tonin, URI  Brasil

Timelimit: 1

Faça um programa que leia um vetor $X[10]$. Substitua a seguir, todos os valores nulos e negativos do vetor X por 1. Em seguida mostre o vetor X .

Entrada


A entrada contém 10 valores inteiros, podendo ser positivos ou negativos.

Saída

Para cada posição do vetor, escreva " $X[i] = x$ ", onde i é a posição do vetor e x é o valor armazenado naquela posição.

Exemplo de Entrada	Exemplo de Saída
0	$X[0] = 1$
-5	$X[1] = 1$
63	$X[2] = 63$
0	$X[3] = 1$
...	...

Preenchimento de Vetor II

Adaptado por Neilor Tonin, URI  Brasil

Timelimit: 1

Faça um programa que leia um valor **T** e preencha um vetor **N[1000]** com a sequência de valores de 0 até **T-1** repetidas vezes, conforme exemplo abaixo. Imprima o vetor **N**.

Entrada

A entrada contém um valor inteiro **T** ($2 \leq T \leq 50$).

Saída

Para cada posição do vetor, escreva "**N[i] = x**", onde **i** é a posição do vetor e **x** é o valor armazenado naquela posição.

Exemplo de Entrada	Exemplo de Saída
3	N[0] = 0 N[1] = 1 N[2] = 2 N[3] = 0 N[4] = 1 N[5] = 2 N[6] = 0 N[7] = 1 N[8] = 2 ...

Dúvidas?

Abstração:

Capacidade de representar cenários complexos utilizando termos mais simples.