

# Matrizes, Structs

Prof. Denio Duarte  
Prof. Geomar Schreiner

# Sumário

- Matrizes
- Structs

# Sumário

- **Matrizes**
- Structs

# Matrizes

- Matrizes são vetores de duas dimensões, ou seja, um vetor onde cada uma das posições também é um vetor
- C permite criarmos matrizes de várias dimensões, porém, o mais usual é que uma matriz apenas possua duas dimensões
- Assim como um Vetor uma Matriz é um conjunto de valores de apenas um tipo
- Como declarar:

```
3  int main(){
4      int matriz[10][10];
5      float matriz2[10][10];
6
7      int vetor[1][200];
8
9      return 0;
10 }
```

# Matrizes

- Matrizes são vetores de duas dimensões, ou seja, um vetor onde cada uma das posições também é um vetor
- C permite criarmos matrizes de várias dimensões, porém, o mais usual é que uma matriz apenas possua duas dimensões
- Assim como um Vetor uma Matriz é um conjunto de tipo
- Como declarar:

```
3  int main(){
4      int matriz[10]
5      float matriz2[
6
7      int vetor[1][20
8
9      return 0;
10 }
```



# Matrizes

- Matrizes são vetores de duas dimensões, ou seja, um vetor onde cada uma das posições também é um vetor
- C permite criarmos matrizes de várias dimensões, porém, o mais usual é que uma matriz apenas possua duas dimensões
- Assim como um Vetor uma Matriz é um conjunto de valores de apenas um tipo
- Como declarar:

```
3  int main()
4      int matriz[10][10];
5      float matriz2[10][10];
6
7      int vetor[1][200];
8
9      return 0;
10 }
```

Tipo de  
dado

# Matrizes

- Matrizes são vetores de duas dimensões, ou seja, um vetor onde cada uma das posições também é um vetor
- C permite criarmos matrizes de várias dimensões, porém, o mais usual é que uma matriz apenas possua duas dimensões
- Assim como um Vetor uma Matriz é um conjunto de valores de apenas um tipo
- Como declarar:

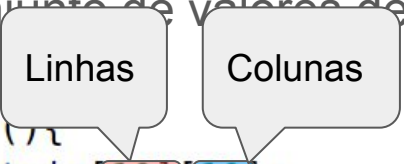
```
3  int main(){
4      int matriz[10][10];
5      float matriz2[10][10];
6
7      int vetor[1][200];
8
9      return 0;
10 }
```

Nome da  
variável

# Matrizes

- Matrizes são vetores de duas dimensões, ou seja, um vetor onde cada uma das posições também é um vetor
- C permite criarmos matrizes de várias dimensões, porém, o mais usual é que uma matriz apenas possua duas dimensões
- Assim como um Vetor uma Matriz é um conjunto de valores de apenas um tipo
- Como declarar:

```
3  int main()  
4      int matriz[10][10];  
5      float matriz2[10][10];  
6  
7      int vetor[1][200];  
8  
9      return 0;  
10 }
```





# Matrizes

- Matrizes são vetores de duas dimensões, ou seja, um vetor onde cada uma das posições também é um vetor
- C permite criarmos matrizes de várias dimensões, porém, o mais usual é que uma matriz apenas possua duas dimensões
- Assim como um Vetor uma Matriz é um conjunto de valores de apenas um tipo
- Como declarar:

```
3  int main  
4      int ma  
5      float  
6  
7      int vetor[1][200];  
8  
9      return 0;  
10 }
```

Mesma coisa que um vetor, já que uma das dimensões é 1

# Matrizes

- Uma matriz pode ser inicializada da seguinte forma:

```
3  int main(){
4      int matriz[3][3] = {
5          |   |   |   |   |   |   |   |   |   {00, 01, 02},
6          |   |   |   |   |   |   |   |   |   {10, 11, 12},
7          |   |   |   |   |   |   |   |   |   {20, 21, 22}
8          |   |   |   |   |   |   |   |   |   };
9
10     return 0;
11 }
```

# Matrizes

- Exercícios
  1. Leia uma matriz 4x4 e imprima a diagonal principal.
  2. Leia uma matriz 4 x 4 e escreva a localização (linha e a coluna) do maior valor.
  3. Declare uma matriz 5 x 5. Preencha com 1 a diagonal principal e com 0 os demais elementos. Escreva ao final a matriz obtida.
  4. Faça um programa que preenche uma matriz 5 x 5 com o produto do valor da linha e da coluna de cada elemento. Em seguida, imprima na tela a matriz.

## Vetor

[illegible]

	0	1	2	3	4	5	.....
Matriz							

[illegible]

# Sumário

- Matrizes
- **Structs**
- Ponteiros

# Structs

- Até agora, vimos uma estrutura de dados: vetores
- Propriedades importantes de um vetor:
  - Todos os elementos de um vetor são do mesmo tipo
  - Para selecionar um elemento de um vetor, especificamos a posição (índice) do elemento
- Usamos uma **struct** para armazenar uma coleção de dados de tipos possivelmente diferentes
- Propriedades importantes de uma struct:
  - Os elementos (**membros**) de uma struct podem ser de tipos diferentes
  - Para selecionar um elemento de uma struct, especificamos o nome do elemento

# Structs - declaração de variáveis

- Para declarar variáveis que são structs, podemos escrever

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} data1, data2;
```

```
struct {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} funcionario1, funcionario2;
```

- Representação de `data1` na memória do computador:



- Os nomes dos membros de uma struct não conflitam com outros nomes de fora da struct

# Structs - declaração de variáveis

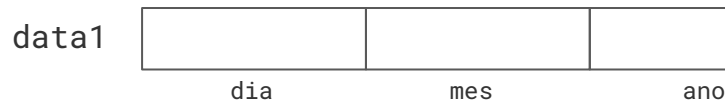
- Para declarar variáveis que são structs, podemos escrever

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} data1, data2;
```

```
str  
i  
c  
d  
}  
f
```



- Representação de data1 na memória



- Os nomes dos membros de uma struct são declarados fora da struct

res de



# Structs - inicialização de variáveis

- Assim como vetores, variáveis que são structs podem ser inicializadas quando declaradas

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} data1 = { 9, 11, 2003 },  
  data2 = { 3, 1, 2008 };
```

```
struct {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} funcionario1 = { 51, "Jose Silva", 5000.00 },  
  funcionario2 = { 89, "Maria Souza", 5000.00 };
```

# Structs - operações

- Para acessar um membro de uma variável que é uma struct, escrevemos o nome da variável seguido de um . seguido do nome do membro

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} data1, data2;
```

```
struct {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} funcionario1, funcionario2;
```

```
printf("Dia: %d\n", data1.dia);  
printf("Nome do funcionario: %s\n", funcionario1.nome);
```

# Structs - operações

- Podemos atribuir valores aos membros de uma variável que é uma struct e usá-los em operações aritméticas (quando cabível)

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} data1, data2;
```

```
struct {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} funcionario1, funcionario2;
```

```
data1.dia = 3;  
media = (funcionario1.salario + funcionario2.salario) / 2;  
  
scanf("%d", &data2.mes);  
scanf("%lf", &funcionario1.salario);
```

# Structs - operações

- Diferente do que vale para vetores, podemos usar o operador = para atribuir uma struct a outra struct - desde que as structs sejam de tipos compatíveis

```
struct {  
    int dia;  
    int mes;  
    int ano;  
} data1, data2;
```

```
struct {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} funcionario1, funcionario2;
```

```
data1 = data2;  
funcionario2 = funcionario1;
```

- O efeito do comando `data1 = data2;` é copiar `data2.dia` para `data1.dia`, `data2.mes` para `data1.mes` e `data2.ano` para `data1.ano`.

# Structs - nomeando tipos

- Para passar uma variável que é uma struct como argumento para uma função, precisamos definir um nome que indique o tipo desta variável
- Opção 1: Definir uma struct tag

```
struct data {  
    int dia;  
    int mes;  
    int ano;  
};
```

```
struct funcionario {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} funcionario1, funcionario2;
```

```
struct data data1, data2; /* não é possível omitir */  
struct funcionario funcionario3, funcionario4; /* a palavra struct! */
```

- Nas declarações acima, não é possível omitir a palavra `struct`!

# Structs - nomeando tipos

- Para passar uma variável que é uma struct como argumento para uma função, precisamos definir um nome que indique o tipo desta variável
- Opção 2: Definir um novo tipo usando `typedef`

```
typedef struct {  
    int dia;  
    int mes;  
    int ano;  
} Data;
```

```
typedef struct funcionario {  
    int id;  
    char nome[TAM_NOME+1];  
    double salario;  
} Funcionario;
```

```
Data data1, data2;  
Funcionario funcionario1, funcionario2;
```

# Structs - como argumentos e retorno de funções

- Funções podem receber structs como argumentos e retornar structs

```
void imprimeData(Data data) {  
    printf("Dia: %d\n", data.dia);  
    printf("Mes: %d\n", data.mes);  
    printf("Ano: %d\n", data.ano);  
}
```

```
Data constroiData(int dia, int mes, int ano) {  
    Data data;  
    data.dia = dia;  
    data.mes = mes;  
    data.ano = ano;  
    return data;  
}
```

```
imprimeData(data1);  
  
data2 = constroiData(9, 11, 2003);
```

# Sumário

- Matrizes
- Structs



TAD

Tipos Abstratos de Dados

# Introdução

- Se considerarmos a definição de estruturas complexas da última aula, podemos criar tipos compostos e que representam de maneira mais fidedigna elementos do mundo real
  - Casa uma destas estruturas é uma discretização de algo no mundo real, sendo assim tem características e comportamentos
- Há diferentes implementações para os comportamentos do mesmo elemento

# Introdução

- Imagine a representação de ponto em um plano de duas dimensões
  - Teremos duas coordenadas (x,y)

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

# Introdução

- Imagine a representação de ponto em um plano de duas dimensões
  - Teremos duas coordenadas (x,y)
    - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

# Introdução

- Imagine a representação de ponto em um plano de duas dimensões
  - Teremos duas coordenadas (x,y)
    - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
int main() {  
    Ponto p1 = {-2,3}, p2 = {-5,-9};  
    double distancia;  
  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
  
    printf("%.5lf \n", distancia);  
  
    return 0;  
}
```

# Introdução

- Imagine a representação de ponto em um plano de duas dimensões
  - Teremos duas coordenadas (x,y)
    - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana( Ponto p1, Ponto p2){  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```

# Introdução

- Imagine a representação de ponto em um plano de duas dimensões
  - Teremos duas coordenadas (x,y)
    - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana( Ponto p1, Ponto p2){  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```

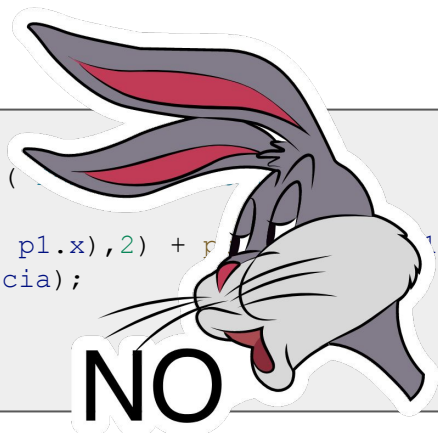
Mas eu tenho que implementar a função todas as vezes?

# Introdução

- Imagine a representação de ponto em um plano de duas dimensões
  - Teremos duas coordenadas (x,y)
    - Como calcular a distancia euclidiana?

```
typedef struct {  
    int x;  
    int y;  
} Ponto;
```

```
double distanciaEuclidiana(  
    double distancia;  
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);  
    distancia = sqrt(distancia);  
    return distancia;  
}
```



Mas eu tenho que implementar a função todas as vezes?

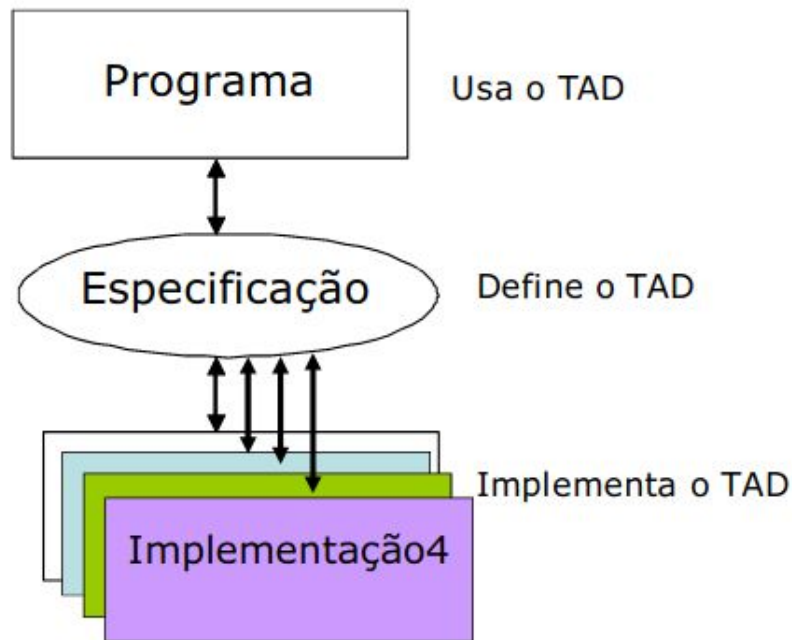


# TAD

- Utilizamos nesses casos os Tipos Abstratos de Dados (TAD)
  - TAD especifica o tipo de dado (domínio de operações) sem referência a detalhes da implementação
  - Permite maior flexibilidade no desenvolvimento, principalmente na manutenção do código
  - O programador não sabe como o TAD foi implementado, as implementações ficam “escondidas”
- O TAD especifica tudo o que precisa saber para usar um determinado tipo
- O TAD divide o sistema em:
  - Programas de Usuário
  - Implementação

# TAD

- Utilizamos nesse contexto:
  - TAD específico para cada implementação
  - Permite maior reutilização de código
  - O programador não precisa lidar com detalhes “escondidos”
- O TAD especifica a interface a ser utilizada
- O TAD divide o trabalho em partes menores:
  - Programas de aplicação
  - Implementações



TAD)

ência a detalhes da

a manutenção do código  
implementações ficam

n determinado tipo

# TAD

- O TAD é representado por dois documentos
  - Especificação
    - Chamado de arquivo de cabeçalho em C
  - Implementação
    - Efetivamente implementa as funções declaradas no cabeçalho

# TAD

- Cabeçalho

planoCartesiano.h

```
typedef struct {  
    int x;  
    int y;  
} Ponto;  
  
double distanciaEuclidiana( Ponto p1, Ponto p2);
```

# TAD

- implementação

planoCartesiano.c

```
#include <stdlib.h>
#include <math.h>
#include "planoCartesiano.h"

double distanciaEuclidiana( Ponto p1, Ponto p2){
    double distancia;
    distancia = pow((p2.x - p1.x),2) + pow((p2.y - p1.y),2);
    distancia = sqrt(distancia);
    return distancia;
}
```