
# Structs

Linguagem de programação

# Introdução

- **Structs** (estruturas) são conjuntos de variáveis agrupadas que possuem um mesmo *nome*
  - as variáveis são as partes de um todo (Struct)
  - podem conter variáveis de vários tipo: `int`, `char`, `double`, `int*`, `char[10]`...
- Um “tipo” de variável criado pelo programador
- São utilizadas na criação de estruturas mais complexas de dados, como:
  - listas encadeadas;
  - filas;
  - pilhas; e
  - árvores

# Definições

- Considere a seguinte definição de struct:

```
struct carta {
 char face[25];
 char naipe[25];
};
```

- **`struct`**: palavra reservada para se criar uma estrutura
- **`carta`**: nome da struct criada. Usada para declarar variáveis do tipo da struct
- **`face[25]`** e **`naipe[25]`** são membros da struct

Obs.: cada definição deve ser encerrada com ; (ponto-e-vírgula)

# Declaração

- Uma variável poder declarada ou juntamente com a struct

```
struct ponto {
  int x, y;
} p1;
```

- ou como em tipos básicos do C

```
struct ponto {
  int x, y;
};
int main() {
  struct ponto p1; }
```

# Inicialização

- Membros não pode ser inicializados junto com a declaração da struct

```
struct ponto {
  int x = 0;  // ERRO DE COMPILACAO
  int y = 0;  // ERRO DE COMPILACAO
};
```

  - não há memória alocada para a atribuição de valores

- Exemplo de declaração válida:
  - `struct ponto p1 = {0, 1};`

# Acesso a membros da *struct*

- Membros de estruturas são acessadas utilizando o operador ponto (.)

```c
struct ponto{
  int x, y;
};
 int main() {
  struct ponto p1 = {0, 1};
 // Acessando membros do ponto p1
  p1.x = 20;
  printf ("x = %d, y = %d", p1.x, p1.y);
  return 0;
}
```

- Saída:
  - x = 20, y = 1

# Vetores de structs

- Assim como outros tipos primitivos, podem criar vetores de *structs*

```c
struct ponto {
  int x, y;
};
int main() {
  // Cria um vetor de Ponto
  struct ponto arr[10];
   // Acessando os membros do 1ª posição do vetor
  arr[0].x = 10;
  arr[0].y = 20;
   printf("%d %d", arr[0].x, arr[0].y);
return 0; }
```

- Saída:
  - 10 20

# Ponteiro de struct

- Como em tipos primitivos, podemos ter ponteiros para *struct*
  - nesse caso, os membros são acessados pelo operador **(->)**
  - **OU** pelo operador (.), desde que faça (*ptr).membro
- No exemplo anterior:
  - **`(*p2).x`** equivale a **`p2->x`**

```c
struct ponto{
  int x, y;
};

int main() {
 struct ponto p1 = {1, 2};

 // p2 é um ponteiro para struct p1
 struct ponto *p2 = &p1;

 // acessando membros da struct usando ponteiro
 printf("%d %d", p2->x, p2->y);
 return 0; }
```

# Typedef

- É um comando que cria um “sinônimo” ou “apelido” para tipos de dados existentes
- Renomeia um tipo de dado, que pode facilitar a organização e o entendimento do código
- Sintaxe: **typedef** <nome do tipo de dado> <novo nome>;

# Typedef: exemplo

```c
struct ponto{
 int x, y;
};

typedef struct ponto
Ponto;

int main() {
 Ponto p1 = {1, 2};
return 0;
}
```

OU

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

int main() {
 Ponto p1 = {1, 2};
return 0;
}
```

- Não se faz necessário o uso de `struct` toda vez que for usar a estrutura
- A tag `ponto` da struct é opcional

# Exercício

- Defina um tipo de estrutura para representar um ponto através de suas coordenadas cartesianas. Em seguida, crie uma função (`double distancia(Ponto q, Ponto p)`) para calcular e retornar a distância euclidiana entre dois pontos fornecidos como entrada.

  Dica: a distância entre dois pontos P e Q, é dada pela seguinte fórmula:

$$d_{qp} = \sqrt{(x_q - x_p)^2 + (y_q - y_p)^2}$$

# Exercício

1. Crie uma struct para representar uma Pessoa, com nome (até 100 caracteres) e telefone; em seguida, crie uma agenda para armazenar o contato de até **n** pessoas (espaço da agenda deverá ser alocado dinamicamente).

2. Crie uma função `void addPessoa(Pessoa *agenda)` que adicione pessoas à agenda;

3. Adicione **p** pessoas (p <= n), e mostre o conteúdo da agenda na main

# Exercício

- A partir do exercício anterior, crie uma função para buscar uma pessoa na agenda (pelo nome) e retornar o número do seu telefone.