Aula 09:

# Estruturas Heterogêneas

**Prof. Edvard**

**edvard@unifei.edu.br**

Universidade Federal de Itajubá

# Enumerações

- Uma das formas mais simples de se definir um novo tipo de dado em C é a **enumeração**.
- Enumeração é um **conjunto de constantes inteiras** que especifica todos os valores possíveis para uma variável do tipo definido.
  - Pode ser vista simplesmente como uma lista de constantes, em que cada constante possui um nome significativo.
  - Criamos através delas um tipo de dado que contém várias constantes, e uma variável de seu tipo poderá receber como valor apenas uma dessas constantes.
- É definida em C através da palavra **enum**.

# Enumerações

- A forma geral de sua definição é:

```
enum nome_enumeracao { <elem_1>, <elem_2>, <elem_3>, ... , <elem_n> };
```

- Os valores que uma enumeração pode assumir são representados por uma **lista de palavras** separadas por virgula e delimitadas por chaves.
- Cada um de seus elementos representa um **valor inteiro**, e pode ser utilizado em qualquer lugar onde um inteiro poderia ser utilizado.
  - O valor se seu primeiro elemento é, por padrão, 0.
- A definição da enumeração **sempre termina com ponto-e-vírgula**.
- Exemplo:

# Enumerações

```c
#include <stdio.h>

// Definição da enumeração
// (Escopo Global, logo após diretivas #include)
// Valem durante todo o programa
enum e_semana{
    domingo, segunda, terca,
    quarta, quinta, sexta, sabado};

int main()
{
    // Declaraçãod e variável enum
    enum e_semana sem;

    // utilizando cosntantes isoladamente
    printf("Dias = %d %d %d \n", domingo, terca, sabado);

    // utilizando variável do tipo enum e_semana
    sem = sexta;
    printf("Dia = %d \n", sem);
    return 0;
}
```

```
Dias = 0 2 6
Dia = 5
```

# Enumerações

- Para fins de comparação, podemos dizer que as seguintes estruturas são praticamente equivalentes:

```
enum e_semana{
    domingo, segunda, terca,
    quarta, quinta, sexta, sabado};
```

```
#define DOMINGO 0
#define SEGUNDA 1
#define TERCA   2
#define QUARTA  3
#define QUINTA  4
#define SEXTA   5
#define SABADO  6
```

**Observação:**
Uma enumeração nada mais é do que uma estrutura que organiza uma série de constantes inteiras, dando nomes a cada uma delas.
Normalmente, os valores são dados automaticamente, começando em 0.
No entanto, é possível modificar os valores constantes da enum

Desvantagem (#define):
Não podem ser acessadas por um nome único (`e_semana`), e não definem um tipo de dados.

# Enumerações

- Podemos inicializar o primeiro elemento da enum:

```
// Constantes começam em domingo=1 e
// terminam em sabado = 7
enum e_semana{domingo=1, segunda, terca,
    quarta, quinta, sexta, sabado};
```

- Podemos inicializar um dos elementos da enum:

```
// Constantes são 0, 1, 2, 13, 14, 15 e 16
enum e_semana{domingo, segunda, terca,
    quarta=13, quinta, sexta, sabado};
```

- Podemos inicializar cada um dos elementos da enum:

```
// Constantes são potencias de 2
enum e_semana{domingo=1, segunda=2, terca=4,
    quarta=8, quinta=16, sexta=32, sabado=64};
```

# Ao Trabalho!

- Escreva um programa que possua uma enumeração com os meses do ano, onde o **primeiro mês** (janeiro) possua **valor 1**.
- A seguir, escreva um programa que, dado um número inteiro inserido pelo usuário, devolva o nome do mês de seu nascimento.

```c
#include <stdio.h>

// Meses do ano, de 1 a 12
enum meses {janeiro = 1, fevereiro, marco, abril, maio,
            junho, julho, agosto, setembro,
            outubro, novembro, dezembro};

int main()
{
    int mes;
    printf("Entre com o mes de seu nascimento: ");
    scanf("%d", &mes);

    printf("Voce nasceu no mes de ");

    switch(mes)
    {
        case janeiro:
            printf("Janeiro"); break;
        case fevereiro:
            printf("Fevereiro"); break;
        case marco:
            printf("Marco"); break;
        // outros meses
        case dezembro:
            printf("Dezembro"); break;
        default:
            printf("<Desconhecido>"); break;
    }

    printf("!\n");

    return 0;
}
```

Posso utilizar os valores da enum normalmente na estrutura switch, pois eles representam constantes inteiras.

```
Entre com o mes de seu nascimento: 11
Voce nasceu no mes de Novembro!
```

# Criando novos tipos de dados...

- Os tipos de dados que vimos até agora no curso podem ser classificados claramente em duas categorias:
  - **Tipos básicos:** char, int, float, double e void.
  - **Tipos Compostos Homogêneos:** vetores e matrizes
- Dependendo da situação que desejamos simular em nosso programa, esses tipos podem não ser suficientes.
  - Ficamos com a sensação de que uma estrutura um pouco mais complexa nos ajudaria a descrever melhor o mundo real em linguagem de programação. Seria ótimo criar **novos tipos**!
- Por isso, o C permite a **criação de novos tipos de dados** utilizando seus tipos básicos.

# Criando novos tipos de dados...

- Se pensarmos bem, veremos que várias entidades do mundo real possuem informações em comum.
  - Todas as **pessoas** possuem um nome, sobrenome, endereço, sexo e idade, por exemplo.
  - Todos os **filmes** possuem um título, um diretor, atores principais, tempo de duração, uma nota dada por você.
    - Imagine que você precisasse criar uma rede social para cinéfilos, como o ***Filmow***. Essas informações seriam muitos úteis!
- Essas informações são chamadas de **atributos**, e cada um deles possui um tipo que o representa melhor.
  - Nome (**string**), idade (**inteiro**), nota (**float**), etc.
  - Uma entidade é **caracterizada** pelos seus atributos.

# Criando novos tipos de dados...

- Precisamos, portanto, aprender como **agrupar os dados** de uma determinada entidade em uma **única estrutura** para utilizá-los de maneira mais direta e simples nos nossos programas.
- Existem mecanismos em C para a criação de novos tipos de dados, compostos por vários atributos, de maneira que cada um pode ter seu próprio tipo. São as **estruturas heterogêneas**.
    - Arrays possuem apenas dados de um tipo: **homogêneos**.

# Estruturas Heterogêneas

- Para modelar atributos de um determinado objeto não é suficiente utilizar *arrays*, que possuem dados de apenas um tipo.
- A solução para esse problema é utilizar as **estruturas**, ou ***structs***.
  - Uma estrutura pode ser vista como um **conjunto de variáveis** referenciadas sob um **mesmo nome**, onde cada uma delas pode ter qualquer tipo (ou até o mesmo tipo).
  - Sua ideia básica é criar um **novo tipo de dado**, que contenha vários membros (seus atributos), que nada mais são do que outras variáveis.

# Estruturas Heterogêneas

- **Struct**: Definição
  - **Coleção** de uma ou mais **variáveis**, possivelmente de **diferentes tipos**, agrupadas sob um **único nome**, para uma **manipulação mais conveniente**.
  - Structs ajudam na organização de dados mais complicados, especialmente em programas maiores, pois permitem que um grupo de variáveis relacionadas entre si seja **tratado como uma unidade**, ao invés de dados separados.
  - **Exemplos:**

```
// Estrutura "ponto"
struct ponto
{
    int x;
    int y;
};
```

```
// Estrutura "filme"
struct filme
{
    char nome[50];
    char diretor[50];
    int ano_producao;
    float nota;
};
```

# Estruturas Heterogêneas

- **Struct**: Definição
  - A **forma geral** da definição de uma nova estrutura utiliza o comando struct:

```
struct nome_estrutura
{
        tipo1 membro1;
        tipo2 membro2;
        tipo3 membro3;
        ...
        tipoN membroN;
};
```

**Observações:**

- Uma estrutura pode conter **quantos membros forem necessários**.
- Cada membro possui um **nome único**.
- Sua definição sempre termina com **ponto-e-vírgula.**
- Precisam ser **declaradas ANTES** de utilizadas.
- Normalmente são declaradas no **escopo global**, logo após as diretivas #include.

# Estruturas Heterogêneas

- **Struct**: Utilização
  - Uma vez definida no código, variáveis do tipo da estrutura podem ser declaradas no programa, de modo similar ao que fazemos com os tipos pré-existentes:

```
// Basta que coloquemos a palavra struct e o
// nome da estrutura definida anteriormente.
// Os nomes das variáveis seguem as regras
// gerais de nomes de variáveis
struct ponto x1, x2;
struct filme jurassic_park;
struct filme indiana_jones, xmen;
```

  - Veja como a utilização de estruturas facilita a escrita do programa. O código para o mesmo programa, **sem structs** ficaria da seguinte maneira:

# Estruturas Heterogêneas

- **Struct**: Utilização

```
// Cada variável teria que ser especificada individualmente,
// gerando um programa cada vez mais confuso
int x1_x, x1_y, x2_x, x2_y;
char nome_jurassic_park[50];
char nome_indiana_jones[50];
char nome_xmen[50];
char diretor_jurassic_park[50], diretor_indiana_jones[50];
char diretor_xmen[50];
int ano_jurassic_park, ano_indiana_jones, ano_xmen;
float nota_jurassic_park, nota_indiana_jones, nota_xmen;
```

# Estruturas Heterogêneas

- **Struct**: Acesso aos membros
  - Uma vez definida a estrutura, precisamos ter acesso a cada um de seus membros individualmente.
  - Existe um operador próprio para o acesso aos membros da estrutura: o operador ponto (“.”).
  - **Exemplos:**
    - Vamos criar um programa onde irei cadastrar um seriado de televisão em forma de estrutura, com nome (string), número de temporadas (int), rede de televisão (string) e nota do público (float).

```c
#include <stdio.h>
#include <string.h>

struct seriado
{
    char nome[50];
    int temporadas;
    char rede[50];
    float nota_publico;
};

int main()
{
    struct seriado himym;

    // Modificando campos da struct individualmente
    strcpy(himym.nome, "How I Met Your Mother");
    himym.temporadas = 9;
    strcpy(himym.rede, "CBS");
    himym.nota_publico = 9.1;

    // imprimindo informações
    printf("Programa: %s (%d temporadas)\n", himym.nome, himym.temporadas);
    printf("Rede: %s\n", himym.rede);
    printf("Nota do Publico: %.2f\n", himym.nota_publico);

    return 0;
}
```

**Observações:**

- Para acessar um membro da struct, utilizamos o **nome da variável – ponto – nome do membro**.
- Seus valores podem ser utilizados como variáveis comuns (modificados, impressos, alterados, etc).
- **Exemplo**:
  - himym.nome
  - Himym.temporadas

```
Programa: How I Met Your Mother (9 temporadas)
Rede: CBS
Nota do Publico: 9.10
```

# Estruturas Heterogêneas

- **Struct**: Inicialização
  - Além disso, assim como os *arrays*, as estruturas também podem ser inicializadas de maneira especial no momento de sua declaração, independentemente do tipo das variáveis que ela contém.
    - Este tipo de inicialização somente pode ser efetuado no momento de sua declaração.
  - **Exemplo**: Vejamos a inicialização dos dados no mesmo exemplo anterior, mas com outra série:

**Observações:**

- Para inicializar a estrutura, basta definir uma **lista de valores separados por vírgula e delimitados por chaves**.
- Como nos arrays, a **ordem é mantida**.
- Se nem todos os itens forem inicializados, os últimos recebem valores default (0 para membros numéricos e “” para strings)

```c
#include <stdio.h>
#include <string.h>

struct seriado
{
    char nome[50];
    int temporadas;
    char rede[50];
    float nota_publico;
};

int main()
{
    struct seriado bigbang = {"The Big Bang Theory", 7, "Warner Channel", 9.0};

    // imprimindo informações
    printf("Programa: %s (%d temporadas)\n", bigbang.nome, bigbang.temporadas);
    printf("Rede: %s\n", bigbang.rede);
    printf("Nota do Publico: %.2f\n", bigbang.nota_publico);

    return 0;
}
```

```
Programa: The Big Bang Theory (7 temporadas)
Rede: Warner Channel
Nota do Publico: 9.00
```

# Estruturas Heterogêneas

- **Struct**: Utilização
  - Podemos também criar vetores e matrizes de estruturas;

```
// vetores e matrizes
struct seriado comedias[10];
struct seriado grade_programacao[24][10];
```

Para acessar um membro, o operador ponto vem sempre DEPOIS do colchete que especifica o elemento:

```
comedias[0].nota_publico = 7.6;
comedias[7].temporadas = 1;
```

# Estruturas Heterogêneas

- **Struct**: Utilização
  - A atribuição entre duas variáveis de estrutura faz com que os conteúdos de cada uma das variáveis sejam copiados para as variáveis correspondentes da outra estrutura.
  - Somente funciona para atribuições entre variáveis definidas pela MESMA estrutura.
  - Vejamos um exemplo:
    - Utilizando uma estrutura ponto, composta por dois inteiros x e y, atribua seu valor a outra estrutura e imprima os resultados.

```c
#include <stdio.h>

struct ponto
{
    int x, y;
};

int main()
{
    // duas variaveis do tipo struct ponto
    struct ponto p1, p2;

    printf("Entre com o valor x e y de p1: ");
    scanf("%d %d", &p1.x, &p1.y);

    // copia cada valor para variavel correspondente
    // p2.x = p1.x
    // p2.y = p1.y
    // Em um soh comando
    p2 = p1;

    // imprime resposta para verificacao
    printf("p2: x = %d e y = %d \n", p2.x, p2.y);

    return 0;
}
```

```
Entre com o valor x e y de p1: 10 50
p2: x = 10 e y = 50
```

# Ao Trabalho!

- Crie uma estrutura representando um **atleta**. Essa estrutura deve conter o seu nome, o esporte que pratica, sua idade e altura.
- Agora, escreva um programa que leia os dados de **três atletas** e, depois, mostre os nomes do atleta **mais alto** e do **mais jovem**.

# Ao Trabalho!

- Crie uma estrutura representando um **funcionário**. Essa estrutura deve conter o seu nome, o cargo e o salário.
- Agora, escreva um programa que leia os dados de **três funcionários** e verifique quais recebem o mesmo salário.