Engenharia da Computação

www.eComp.Poli.br

Tipos Avançados de Dados

(Registros/Estruturas de Dados)

Disciplina: **DCExt Programação Imperativa**

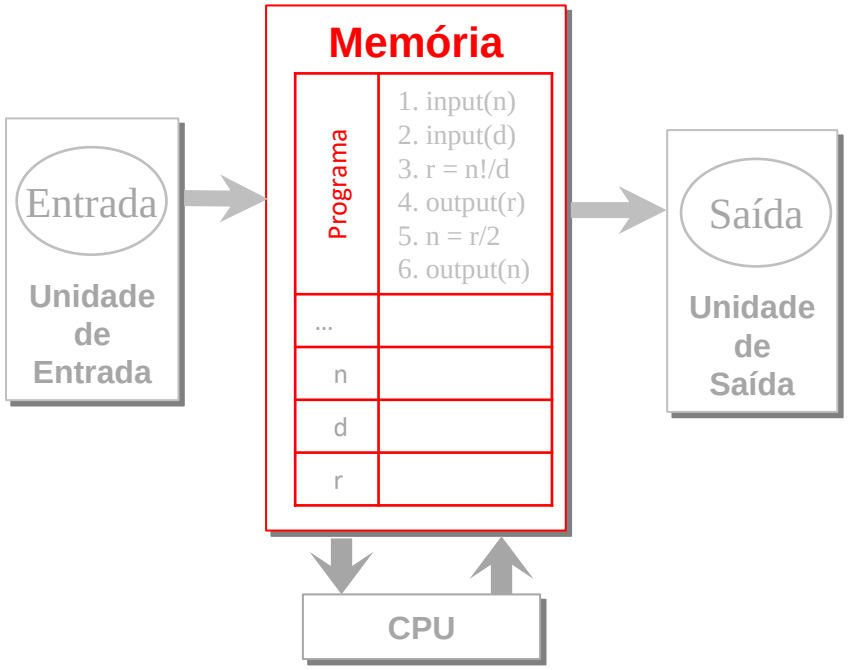
Prof. Hemir Santiago

hcs2@poli.br

Material cedido pelo Prof. Joabe Jesus

typedef

TIPOS NOMEADOS



Tipos **Nomeados**

- Podemos criar novos nomes (*alias*) para os tipos de dados usando o comando **typedef**

```
main( )
{
    typedef unsigned int nat;

    nat x1, x2;

    x1 = 2;
    x2 = 3;
    printf("x1 = %u\n", x1);
    printf("x2 = %u\n", x2);
}
```

Tipos **Nomeados**

- Por que dar novos nomes a tipos?
  - Facilitar escrita/leitura (legibilidade)

- Diferença em relação a #define?
  - #define é uma diretiva de **pré-compilação**, logo, é analisada antes da compilação
  - **typedef** é tratado/interpretado pelo compilador

Exemplo

```
int current_speed;
int high_score;

void congratulate(int
your_score) {
    if (your_score >
high_score) {
        // ...
    }
}
```

```
typedef int km_per_hour;
typedef int points;

km_per_hour current_speed;
points high_score;

void congratulate(points your_score)
{
    if (your_score > high_score) {
        // ...
    }
}
```

Outro Exemplo

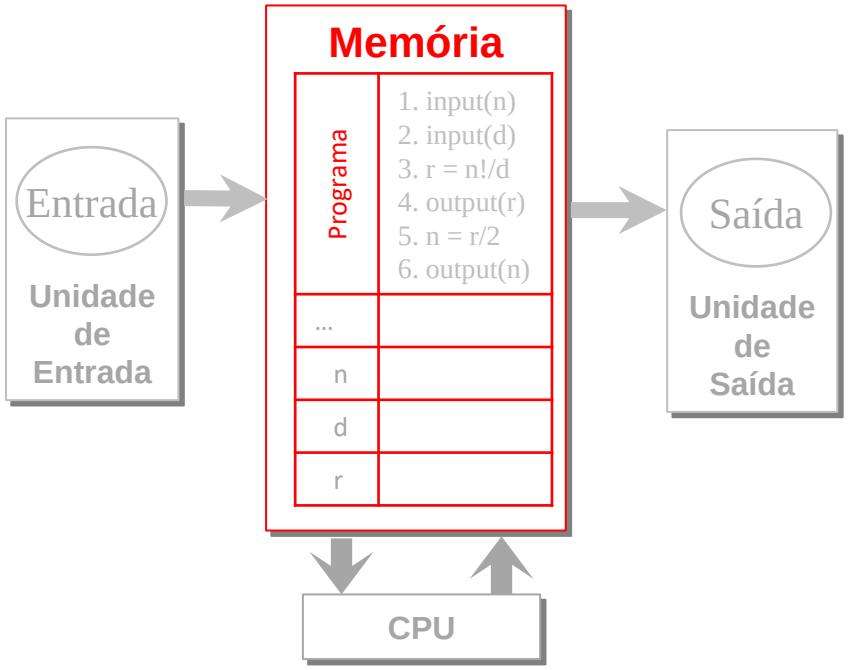
```
typedef unsigned char byte;

byte  b1, b2;

b1 = 0x01; // Hexadecimal 01 = Decimal 1
b2 = 0xFF; // Hexadecimal FF = Decimal 255
```

struct

ESTRUTURAS DE DADOS



Estruturas

- Agrupa um conjunto de **tipos de dados distintos** sob um único nome
- Também chamadas de **Registros**

Cadastro Pessoal

string	Nome
string	Endereço
inteiro	Telefone
inteiro	Idade
inteiro	Ano de Nascimento
float	Peso
float	Altura

```
struct cadastro_pessoal {
        char  nome[50];
        char  endereço[100];
        int   telefone;
        int   idade;
        int   nascimento;
        float peso;
        float altura;
};
```

Estruturas

- **Exemplo**:

```
main( )
{
  struct facil {    // DEFINIÇÃO do novo tipo struct facil
            int num;
            char ch;
  };

  struct facil x;  /* USO: DECLARAÇÃO da variável x do tipo struct facil */
  x.num = 2;
  x.ch = 'Z';
  printf("x.num = %d, x.ch = %c\n", x.num, x.ch);
}
```

Estruturas

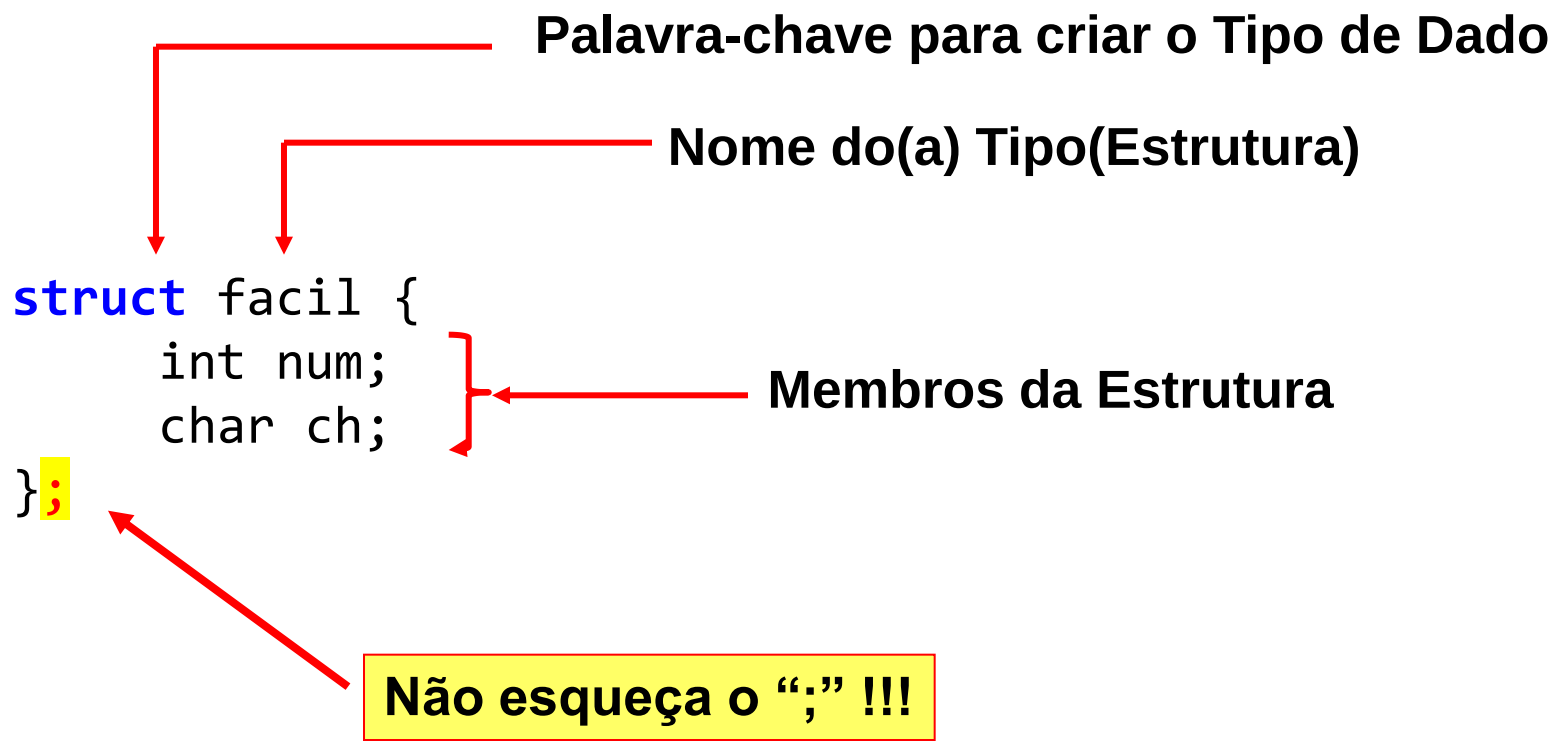
- Outra forma... Variável criada **JUNTO** com a definição

```
main( )
{
  struct facil {    // DEFINIÇÃO do novo tipo struct facil
            int num;
            char ch;
  } x; /* USO: DECLARAÇÃO da variável x do tipo struct facil */


  x.num = 2;
  x.ch = 'Z';
  printf("x.num = %d, x.ch = %c\n", x.num, x.ch);
}
```

Estruturas

- **OBSERVAÇÕES:**



Exemplo com Múltiplas Variáveis

```
main( )
{
  struct facil {
            int num;
            char ch;
  };

  struct facil x1;  /* declara variável x1 do tipo struct facil */
  struct facil x2;  /* declara variável x2 do tipo struct facil */

  x1.num = 2;   x1.ch     = 'Z';
  x2.num = 3;   x2.ch     = 'B';
  printf("x1.num = %d, x1.ch = %c\n", x1.num, x1.ch);
  printf("x2.num = %d, x2.ch = %c\n", x2.num, x2.ch);
}
```

Mais de uma variável

Exemplo com Múltiplas Variáveis

```
main( )
{
  struct facil {
            int num;
            char ch;
  } x1, x2; /* declara variáveis x1 e x2 do tipo struct facil */



  x1.num = 2;   x1.ch     = 'Z';
  x2.num = 3;   x2.ch     = 'B';
  printf("x1.num = %d, x1.ch = %c\n", x1.num, x1.ch);
  printf("x2.num = %d, x2.ch = %c\n", x2.num, x2.ch);
}
```

Estruturas de Dados SEM NOME

- Pode-se definir uma estrutura sem um nome...

```
struct {
    int num;
    char ch;
} x1, x2;
```

Estruturas de Dados SEM NOME e typedef

- Podemos usar o comando **typedef** para simplificar o trabalho com estruturas:

```
main( )
{
    typedef
    struct {
            int num;
            char ch;
    } tFacil;

    tFacil x1, x2;
    x1.num = 2;    x1.ch = 'Z';
    x2.num = 3;    x2.ch = 'B';
    printf("x1.num = %d, x1.ch = %c\n", x1.num, x1.ch);
    printf("x2.num = %d, x2.ch = %c\n", x2.num, x2.ch);
}
```

Estruturas de Dados e typedef

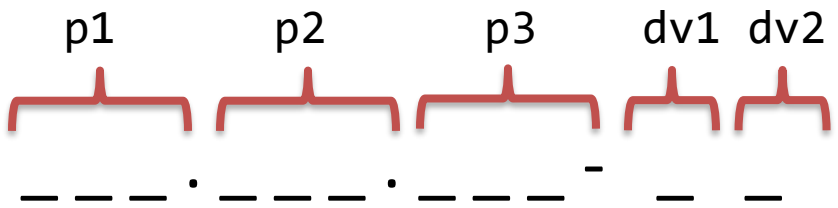
- O código anterior é similar ao código a seguir:

```
main( )
{
    struct facil {
            int num;
            char ch;
    };
    typedef struct facil tFacil;

    tFacil x1, x2;
    x1.num = 2;    x1.ch = 'Z';
    x2.num = 3;    x2.ch = 'B';
    printf("x1.num = %d, x1.ch = %c\n", x1.num, x1.ch);
    printf("x2.num = %d, x2.ch = %c\n", x2.num, x2.ch);
}
```

Outros Exemplos

- **CPF**
  - Podemos dividir em 5 partes:




```
struct CPF {
      unsigned short p1, p2, p3;
      char dv1, dv2;
};
struct CPF cpf1 = { 111, 111, 111, 1, 1 };
struct CPF cpf9 = { 999, 999, 999, 9, 9 };
struct CPF x;

x.p1 = 123; x.p2 = 456; x.p3 = 789;
x.dv1 = 1;
x.dv2 = 2;
```

p1, p2 e p3 assumem valores de 000 a 999. dv1 e dv2 assumem valores de 0 a 9

Outros Exemplos

- Criando uma Lista de Livros
  - Passo inicial: 2 Livros

```
struct livro {
      char titulo[40];
      int  regnum;
};

struct livro livro1 =
    {“Treinamento em Linguagem C - I”, 124};
struct livro livro2 =
    {“Treinamento em Linguagem C - II”, 125};
```

Atribuições e Estruturas

- Na **versão original do C**, definida por **Kernighan** e **Ritchie**, era impossível atribuir o valor de uma variável estrutura a outra do mesmo tipo usando uma simples expressão de atribuição.

~~`livro2 = livro1;`~~

- Nas **versões modernas** de C, esta forma **é possível**

```
livro2 = livro1;
```

Atribuições e Estruturas - Exemplo

```
struct livro {
     char titulo[40];
     int  regnum;
};

struct livro livro1 =
    {“Treinamento em Linguagem C - I”, 124};
struct livro livro2 =
    {“Treinamento em Linguagem C - II”, 125};

livro2 = livro1;
```

Estruturas Aninhadas

- Assim como é possível ter vetores de vetores, pode-se criar estruturas que contém outras estruturas.

```
struct professor {
      char nome[50];
      char disciplina[20];
      int  carga_horaria;
};
struct aluno {
       char nome[50];
       int matricula;
};
struct cadastro_escolar {
     struct professor docentes[10];
     struct aluno discentes[100];
};
```

Unions

Enums

Bit Fields

ESTRUTURAS AVANÇADAS

Estruturas Avançadas

- Além do uso de *struct*, a linguagem C permite a criação de tipos avançados (estruturas mais complexas ou mais abstratas) como:
  - Uniões Disjuntas (*Unions*)
  - Enumerações (*enums*)
  - Campos Binários (*Bit Fields*)

- Essas estruturas serão **brevemente** apresentadas a seguir.

Unions

- Uma declaração **union** determina uma **única** localização de memória para **várias variáveis diferentes**. Apenas uma delas deve ser usada por vez.

```
union angulo {
  float graus;
  float radianos;
};
void main() {
  union angulo ang;
  char op;
  printf("\nGraus ou radianos (G/R)?");
  scanf("%c",&op);
  printf("\nAngulo:");
  if (op == 'G'){
    ang.graus = 180;
    printf("%f\n",ang.graus);
  } else if (op == 'R') {
    ang.radianos = 3.1415;
    printf("%f\n",ang.radianos);
  } else printf("\nInvalido!\n");
}
```

Enumerações (Enums)

- Numa **enumeração** podemos dizer ao compilador quais os valores que uma determinada variável pode assumir. Sua forma geral é:

```
enum nome_do_tipo_da_enumeração {
    lista_de_valores
} lista_de_variáveis;
```

  - Cada valor está associado a um número, começando em 0. Logo as variáveis são do tipo int.

Enumerações (Enums)

- Exemplo:

```
enum dias_da_semana {segunda, terca, quarta, quinta, sexta, sabado, domingo};
void main ()
{
    enum dias_da_semana d1 = segunda, d2 = sexta;
    if (d1 == d2)
        printf ("O dia é o mesmo.");
    else
        printf ("São dias diferentes.");
}
```

Bit Fields

- Útil quando temos pouca memória
- Permite **compactar** os dados da estrutura, isto é, compactar vários valores em apenas uma palavra (word) da máquina

```
struct pacote {
  unsigned int f1:1;
  unsigned int f2:1;
  unsigned int f3:1;
  unsigned int f4:1;
  unsigned int type:4;
  unsigned int my_int:9;
} p;
```

Observe que após cada : há um número de bits desejado para o membro da estrutura. Assim, neste exemplo, a variável ***p*** ocupará aproximadamente 17 bits.

- Em cada computador poderá ter um pouco mais de bits dependendo do tamanho de uma word.

Exercício 01

- Considere que foi definida a seguinte estrutura:

```
struct fracao {
    int numerador, denominador;
};
```

- Escreva um programa em C que calcule as quatro operações (+, -, *, /) usando frações definidas com a estrutura acima. O programa deve ler duas frações e imprimir o resultado de cada uma das quatro operações.
- O resultado DEVE ser como fração E como ponto flutuante.

Exercício 01

```
//Algoritmo de Euclides iterativo
int mdc(int a, int b){
    while(b != 0){
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
```

```
//Algoritmo do MMC
int mmc(int a, int b){
    return a * (b / mdc(a, b));
}
```

Exercício 02

- Utilizar *struct* para implementar um programa que possua um cadastro de usuários com as seguintes informações: login, senha, nome, endereço, telefone, data de nascimento. Além de realizar o cadastro, o usuário também pode atualizá-lo, mas para isso deve validar seu login e sua senha. Fluxo do programa:

1. Usuário realiza um ou mais cadastros;
2. Após realizar o(s) cadastro(s) o usuário pode atualizar algum deles ou sair do programa.

	DATA	AULA
1	22/08/2024	Apresentação da disciplina | Introdução à Programação Imperativa
2	29/08/2024	Introdução à Linguagem de Programação C
3	05/09/2024	Conceitos Fundamentais
4	12/09/2024	Tipos de Dados Especiais em C
5	19/09/2024	Estruturas Condicionais e de Repetição
6	26/09/2024	Pré-processamento
7	03/10/2024	Registros/Estruturas de Dados
8	10/10/2024	Ponteiros
9	17/10/2024	1º Exercício Escolar

Plano de Aulas

	DATA	AULA
10	24/10/2024	Arquivos
11	31/10/2024	Acompanhamento de projetos
12	07/11/2024	Acompanhamento de projetos
13	14/11/2024	Acompanhamento de projetos
14	21/11/2024	Acompanhamento de projetos
15	28/11/2024	Apresentação parcial
16	05/12/2024	Apresentação de projetos
17	12/12/2024	Avaliação Final

Plano de Aulas