



CENTRO DE CIÊNCIA E TECNOLOGIA  
LABORATÓRIO DE CIÊNCIAS MATEMÁTICAS  
UNIVERSIDADE ESTADUAL DO NORTE FLUMINENSE

# **Linguagem C**

## **Uma breve revisão da Linguagem**

*Disciplina: Estruturas de Dados I*

**Prof. Fermín Alfredo Tang Montané**

**Curso: Ciência da Computação**

# Linguagem C

## *Entrada/Saída*

---

- Temos as conhecidas funções:
  - **scanf().**- lê dados formatados da entrada padrão (teclado); inclusive strings; Na leitura de teclado somente lê até achar o primeiro espaço.
  - **printf().**- imprime dados formatados na saída padrão (monitor); inclusive strings;
- Também temos funções específicas para caracteres:
  - **getc().**- lê um caracter na entrada padrão (teclado).
  - **putc().**- escreve um caráter na saída padrão (monitor);
- Também temos funções específicas para caracteres em arquivo:
  - **fgetc().**- lê um caracter de arquivo.
  - **fputc().**- escreve um caracter em arquivo.
- **Obs. Pesquise sobre o protótipo de cada função.**

# Linguagem C

## *Entrada/Saída*

---

- Também temos funções específicas para strings:
  - **gets().**- lê uma string da entrada padrão (teclado); incluindo espaços até encontrar um “\n” (enter). O enter não fará parte da string.
  - **puts().**- escreve uma string na saída padrão (monitor);
- Também temos funções específicas para strings em arquivo:
  - **fgets().**- lê uma string de um arquivo; Especifica o tamanho da string; Funciona com stdin (teclado), lê espaços até encontrar um “\n” (enter) ou tamanho-l caracteres. O enter fará parte da string.
  - **fputs().**- escreve uma string em arquivo; Funciona com stdout (monitor)
- Obs. Pesquise sobre o protótipo de cada função.

# Linguagem C

## Strings

---

- **Strings.-** Correspondem a uma sequência de caracteres seguidas pelo caractere especial “\0” como indicador do fim da sequência. São basicamente arrays de caracteres mais o caractere especial “\0”. O tamanho da *string* deve levar em consideração o caractere especial “\0”. Como a linguagem C não permite a atribuição de arrays. Também não é possível a atribuição de *strings*.
- Na biblioteca <string.h> temos funções para manipulação de strings:
  - **strlen().-** (abr. *string length*) retorna o comprimento da *string* em caracteres;
  - **strcpy().-** (abr. *string copy*) copia o conteúdo de uma *string* origem para uma *string* destino; retorna o endereço da *string* destino;
  - **strncpy().-** (abr. *string number copy*) copia um número exato de caracteres de uma *string* origem para uma *string* destino; retorna o endereço da *string* destino;
  - **strcat().-** (abr. *string concatenation*) copia o conteúdo de uma *string* origem para o final de uma *string* destino; retorna o endereço da *string* destino;
  - **strcmp().-** (abr. *string comparison*) compara o conteúdo de duas *strings*, caractere a caractere e retorna valor igual a zero, se são iguais, <0 se a primeira é menor que a segunda, >0 se a primeira é maior que a segunda;
- **Obs. Pesquise sobre o protótipo de cada função.**

# Linguagem C

## Alocação Dinâmica

---

- **Alocação Dinâmica.-** A linguagem C permite alocar (reservar) memória dinamicamente (em tempo de execução). Ela é usada quando não se sabe ao certo (no momento em que se escreve o programa) quanto de memória será necessário para armazenar os dados com que se quer trabalhar. O endereço ao início da memória que foi alocada é armazenado em um ponteiro.
- Na biblioteca `<stdlib.h>` (standard library) temos as funções para alocar memória:
  - **malloc().-** aloca uma quantidade de memória especificada em bytes; retorna um ponteiro void;
  - **calloc().-** aloca uma quantidade de memória especificada pelo número de elementos a serem alocados e o tamanho desse elemento em bytes; retorna um ponteiro void;
  - **realloc().-** permite redimensionar um bloco de memória previamente alocado; para isso precisa de um ponteiro com o endereço do bloco alocado; e quantidade de memória em bytes; caso seja necessário, a função pode mover bloco antigo para uma nova posição; retorna um ponteiro void;
- **Obs. Pesquise sobre o protótipo de cada função.**

# Linguagem C

## Alocação Dinâmica

---

- O protótipo de `malloc()` é o seguinte:
  - `void *malloc(unsigned int num);`
- O motivo da função retornar um ponteiro genérico é que ela não sabe o tipo de dado que será alocado. Assim, quando a função `malloc()` for usada precisa de uma conversão de tipo (*type cast*). Exemplo:
  - `int *p;`
  - `p = (int *) malloc(5*sizeof(int));`

# Linguagem C

## Alocação Dinâmica

---

- Para calcular o tamanho dos blocos de memória temos a função:
  - **sizeof()** .- que permite calcular o tamanho (em bytes) de um tipo básico ou composto;
- Para liberar memória alocada dinamicamente temos a função:
  - **free()** .- ela recebe um ponteiro com o endereço de um bloco de memória alocado previamente e o libera;
- Obs. Pesquise sobre o protótipo de cada função.

# Estruturas

## Definição

---

- As estruturas permitem colocar em uma única entidade, elementos de tipos diferentes.
- Uma estrutura é um conjunto de uma ou mais variáveis, denominadas de campos ou membros, agrupadas sobre um único nome.
- As estruturas podem conter elementos de tipos básicos, vetores, strings e até outras estruturas.

### Declaração de estrutura.-

```
struct [nome da estrutura]
{
    tipo 1    campo1, campo2;
    ...
    tipo n    campo
}
```

### Exemplo:

```
struct Data
{
    int Dia, Ano;
    char Mes[12];
}
```





# Estruturas

## Declaração de Variáveis

Para declarar uma variável com um tipo de estrutura, basta indicar o nome da estrutura após a palavra reservada **struct** seguido dos nomes das variáveis.

### Declaração de variáveis-

```
struct [nome da estrutura]
{
    tipo l    campo1, campo2;
    ...
    tipo n    campo
} var1, var2, ..., varn;
```

### Exemplos:

```
struct Data
{
    int Dia, Ano;
    char Mes[12];
} d, datas[100], *ptr_data;
```

```
struct Pessoa
{
    int idade;
    char sexo, est_civil,, Nome[60];
    float salario;
} Paulo, Teresa;
```



# Estruturas

## Acesso aos campos de uma estrutura

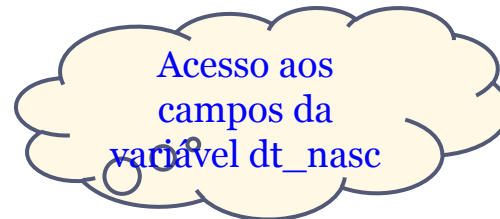
---

Para acessar a um membro y de uma estrutura x, usa-se o operador (.) fazendo-se:  
x.y

### Exemplos:

```
struct Data { int dia, ano; char mes[12]; } dt_nasc;
```

```
dt_nasc.dia = 23;  
dt_nasc.mes = "Janeiro";  
dt_nasc.ano = 1966;  
printf("Data: %d/%s/%d\n", dt_nasc.dia, dt_nasc.mes, dt_nasc.ano);
```



# Estruturas

## Carga Inicial Automática - Inicialização

---

Uma estrutura pode ser iniciada quando declarada usando-se a sintaxe:

```
struct nome da estrutura var = {valor1, valor2, ..., valor n};
```

Entre chaves coloca-se os valores dos campos da estrutura na ordem em que foram definidos.

### Exemplos:

```
struct Data { int dia, ano; char mes[12]; } dt_nasc={23, 1966, "Janeiro"}
```

```
struct Data { int dia, ano; char mes[12]; }  
struct Data dt_nasc={23, 1966, "Janeiro"}
```

```
struct Data v[3]={ {1, 1900, "Janeiro"}, {2, 1920, "Fevereiro"}, {31, 1950, "Dezembro"} };
```

# Estruturas

## Uso do Typedef

---

Permite criar um nome alternativo a tipos existentes, sejam eles tipos básicos , complexos, ou estruturas. Na prática, permitem abreviar a declaração de tipos complexos.

```
typedef struct Pessoa
{
    int idade;
    char sexo, est_civil;
    char Nome[60];
    float salario;
} PESSOA;
```

Podemos declarar as variáveis da forma convencional ou abreviada.

```
struct Pessoa Carlos, Serafim;

ou

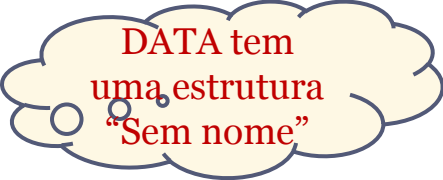
PESSOA Carlos, Serafim;
```

# Estruturas

## Estruturas dentro de Estruturas

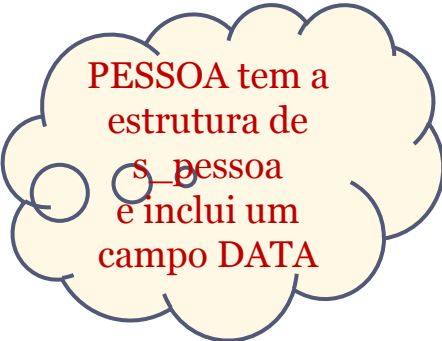
- Uma estrutura pode conter, na sua definição, variáveis simples, vetores, ponteiros e inclusive outras estruturas.
- Todos os tipos e estruturas utilizados na definição de uma nova estrutura devem ter sido previamente definidos.

```
typedef struct
{
    int Dia;
    char Mes[3+1];
    int Ano;
} DATA;
```



DATA tem  
uma estrutura  
"Sem nome"

```
typedef struct s_pessoa
{
    char Nome[100];
    int Idade;
    float Salario;
    DATA dt_Nasc;
} PESSOA;
```



PESSOA tem a  
estrutura de  
s\_pessoa  
e inclui um  
campo DATA

# Estruturas

## Estruturas dentro de Estruturas

---

```
struct s_pessoa homem, mulher[3];
```

ou

```
PESSOA homem, mulher[3];
```

- A inicialização dos campos de uma estrutura composta é realizada da seguinte forma:

```
PESSOA homem      = {"Ze" ,23,123.45,{10,"MAI",1989}},  
mulher[3] = {{ "To" ,51,  0.0,{15,"JUN",1975}},  
             { "Alf",17,  52.0,{22,"JUL",1956}},  
             { "NY" ,22, 25.93,{14,"DEZ",1956}}  
};
```

**Subestrutura  
DATA**

- Para adicionar um dia a data de nascimento do homem.

```
homem.dt_Nasc.Dia++;
```

# Estruturas

## Passagem de Estruturas para Funções

---

A passagem de parâmetros é realizada especificando o tipo da estrutura o seu **typedef**.

Como a passagem é sempre por valor, caso seja necessário modificar algum valor na estrutura, será necessário passar um ponteiro a estrutura.

**Exemplo:** Escrever um programa que:

i) faça o carregamento de (modifique) uma estrutura através de uma função. ii) Mostre o conteúdo da estrutura.

```
Qual o Nome           : Zé Antônio
Qual a Idade          : 25
Qual o Salário         : 45678.6
Qual a Data Nascim.   : 12 3 1952
```

```
Nome      : Zé Antônio
Idade     : 25
Salário   : 45678.60
Dt. Nasc  : 12/3/1952
```

# Estruturas

## Passagem de Estruturas para Funções

prog1102.c

```
1: #include <stdio.h>
2:
3: typedef struct {int Dia,Mes,Ano;} DATA;
4:
5: typedef struct pessoa
6: {
7:     char  Nome[100];
8:     int    Idade;
9:     float  Salario;
10:    DATA  Nasc;
11: } PESSOA;
12:
13: /* Carrega a estrutura passada por parâmetro */
14:
15: void Ler(PESSOA *ptr)
16: {
17:     printf("Qual o Nome           : "); gets((*ptr).Nome);
18:     printf("Qual a Idade          : "); scanf("%d",&(*ptr).Idade);
19:     printf("Qual o Salário         : "); scanf("%f",&(*ptr).Salario);
20:     printf("Qual a Data Nascim.   : ");
21:     scanf("%d %d %d",&(*ptr).Nasc.Dia,&(*ptr).Nasc.Mes,
22:           &(*ptr).Nasc.Ano);
23: }
```

Para modificar  
o conteúdo de  
uma estrutura é  
preciso passar  
um ponteiro  
para a estrutura

Acesso aos  
campos da  
estrutura



# Estruturas

## Acesso aos campos de uma estrutura pelo ponteiro

A explicação da notação: `(*ptr).nome` é a seguinte:

Se `ptr` fosse uma estrutura do tipo `PESSOA`, então a leitura do nome seria realizada com:

```
gets(ptr.Nome);
```

Como `ptr` é um ponteiro, bastará colocar um asterisco antes da variável `ptr`:

```
gets(*ptr.Nome);
```

No entanto, existem dois operadores na expressão `*ptr` — o asterisco e o ponto.

### O problema é que:

O operador ponto (`.`) tem precedência sobre o asterisco (`*`)

Dessa forma, o compilador vai interpretar a expressão `*ptr.Nome` como

```
*(ptr.Nome)
```

Erro!

O objetivo era ter acesso ao campo `Nome` que é apontado por `ptr`, isto é:

```
(*ptr).Nome
```

Acesso aos  
campos da  
estrutura

# Estruturas

## Acesso aos campos de uma estrutura pelo ponteiro

---

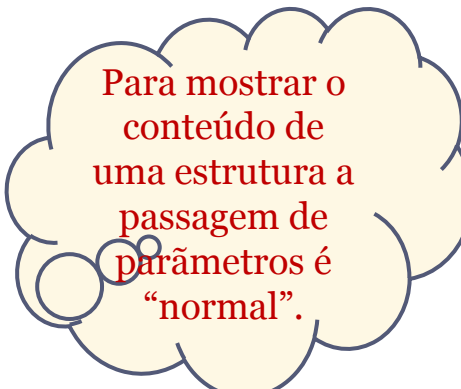
A expressão `(*ptr).nome` também pode ser escrita como:

`(*ptr).nome`            `ptr->nome`

# Estruturas

## Passagem de Estruturas para Funções

```
25: /* Mostra a estrutura passada por parâmetro */
26:
27: void Mostrar(struct pessoa x)
28: {
29:     printf("Nome      : %s\n",x.Nome);
30:     printf("Idade     : %d\n",x.Idade);
31:     printf("Salário    : %.2f\n",x.Salario);
32:     printf("Dt. Nasc   : %d/%d/%d\n",x.Nasc.Dia,x.Nasc.Mes,
33:           x.Nasc.Ano);
34: }
35:
36: main()
37: {
38:     struct pessoa p = {"Carlos",23,12345.67,{23,5,1954}};
39:
40:     Mostrar(p);
41:     puts("\n");
42:     Ler(&p);
43:     puts("\n");
44:     Mostrar(p);
45: }
```



Para mostrar o conteúdo de uma estrutura a passagem de parâmetros é “normal”.

# Tópicos Diversos

## Tópicos Diversos

---

Tipos enumerados.- Mediante a palavra reservada **enum**, podem-se definir constantes numéricas, numeradas sequencialmente.

```
enum dias=(segunda, terca, quarta, quinta, sexta, sabado);
```

# Tópicos Diversos

## Tópicos Diversos

---

Estrutura union.- Estruturas especiais de notação semelhante as struct, porém, neste caso, os membros da estrutura compartilham o mesmo espaço de memória. Conhecidas como record variant. São pouco utilizadas.

```
union conjunto
```

```
{ char c;  
  int n;  
  float x;  
}
```

```
union conjunto cl;
```

```
cl.x=12345.6;
```

# Tópicos Diversos

## Tópicos Diversos

---

- Divisão de projetos em vários arquivos.- Apenas um dos arquivos terá a função `main()`, outros arquivos serão compilados de maneira independente, gerando o arquivo `.obj` que depois serão integrados em um único `.exe` após a linkagem.
- Variáveis globais entre arquivos.- Declarada como global em um arquivo, pode ser acessada em outros arquivos mediante a palavra reservada **extern**.  
**extern int** var;  
**extern char** ch;
- Funções **static**.- Permite definir funções que somente serão reconhecidas no arquivo onde foram escritas. Permite assim, definir funções com o mesmo nome.
- Variáveis **static**.- Se declarada dentro de uma função, não é destruída após a função.

# Linguagem C

## Padrão C99

---

- O padrão C99 é uma revisão do padrão ANSI 1989 da Linguagem C, foi reconhecido em 2000. Adicionando uma serie de recursos úteis a linguagem C, dentre eles temos por exemplo:
  - **Declarações de Variaveis.-** permite declarar variáveis em qualquer lugar do programa inclusive dentro de um comando for.
  - **Tipo bool.-** que admite valores **true** e **false**;
  - **Tipo long long int.-** que é um inteiro de 64 bits;
  - **Arrays de Comprimento Variável.-** Neste tipo de array, o número de elementos que o array poderá possuir pode ser especificado em tempo de execução do programa.
- Obs.Vale observar que já existe o padrão C11 é uma revisão do padrão C99 da Linguagem C, foi reconhecido em 2011.

# Linguagem C

## Referências

---

- Linguagem C . Completa e Descomplicada. Andres Backes, Segunda Edição. 2019. Editora Elsevier.
- Linguagem C. Luis Damas. Décima Edição. Editora LTC. 2007.