


# Estruturas definidas pelo programador



Prof. Bruno Travençolo  
Slides baseados no material do Prof. André Backes

# Variáveis

---

- ▶ As variáveis vistas até agora eram:
  - ▶ simples: definidas por tipos **int**, **float**, **double** e **char**;
  - ▶ compostas homogêneas (ou seja, do mesmo tipo): definidas por **array**.
- ▶ No entanto, a linguagem C permite que se criem novas estruturas a partir dos tipos básicos.
  - ▶ Struct
- ▶ Mas antes, vamos rever alguns conceitos sobre a memória alocada por um programa



# Operador **sizeof**

---

- ▶ Traduzindo: *sizeof*: size (tamanho) of (de)
  - ▶ Retorna o tamanho em bytes ocupado por objetos ou tipos
  - ▶ Exemplo de uso
    - ▶ `printf("\nTamanho em bytes de um char: %u", sizeof(char));`
      - ▶ Retorna 1, pois o tipo char tem 1 byte
    - ▶ `printf("\nTamanho em bytes de um char: %u", sizeof char);`
      - ▶ Também funciona sem o parênteses

Retorna um tipo `size_t`, normalmente `unsigned int`, por isso o `%u` ao invés de `%d`

`unsigned int` - é um número inteiro sem sinal negativo



# Operador **sizeof**

---

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    // descobrindo o tamanho ocupado por diferentes tipos de dados
    printf("\nTamanho em bytes de um char: %u", sizeof(char));
    printf("\nTamanho em bytes de um inteiro: %u", sizeof(int));
    printf("\nTamanho em bytes de um float: %u", sizeof(float));
    printf("\nTamanho em bytes de um double: %u", sizeof(double));

    // descobrindo o tamanho ocupado por uma variável
    int Numero_de_Alunos;
    printf("\nTamanho em bytes de Numero_de_Alunos (int): %u", sizeof Numero_de_Alunos );

    // também é possível obter o tamanho de vetores
    char nome[40];
    printf("\nTamanho em bytes de nome[40]: %u", sizeof(nome));

    double notas[60];
    printf("\nTamanho em bytes de notas[60]: %u", sizeof notas );

    return 0;
}
```

---



# Operador sizeof

```
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
```

```
    // descobrindo o tamanho ocupado por diferentes tipos de dados
```

```
    printf("\nTamanho em bytes de um char: %u", sizeof(char));
    printf("\nTamanho em bytes de um inteiro: %u", sizeof(int));
    printf("\nTamanho em bytes de um float: %u", sizeof(float));
    printf("\nTamanho em bytes de um double: %u", sizeof(double));
```

```
    // descobrindo o tamanho ocupado por uma variável
```

```
    int Numero_de_Alunos;
    printf("\nTamanho em bytes de Numero_de_Alunos (int): %u", sizeof Numero_de_Alunos );
```

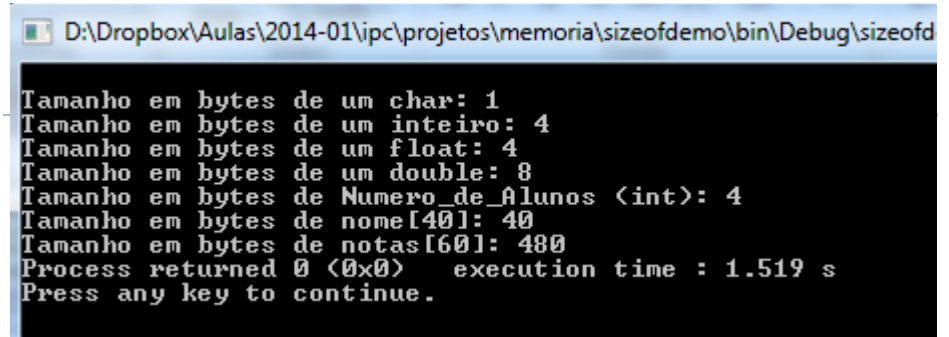
```
    // também é possível obter o tamanho de vetores
```

```
    char nome[40];
    printf("\nTamanho em bytes de nome[40]: %u", sizeof(nome));
```

```
    double notas[60];
    printf("\nTamanho em bytes de notas[60]: %u", sizeof notas );
```

```
    return 0;
```

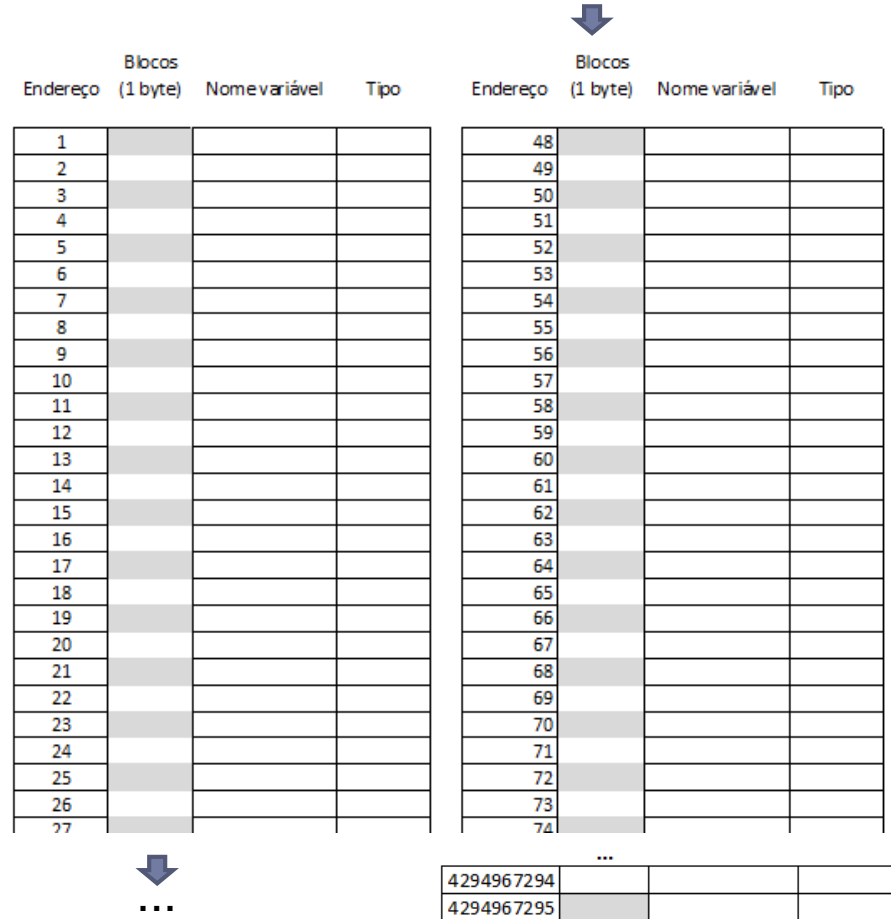
```
}
```



```
Tamanho em bytes de um char: 1
Tamanho em bytes de um inteiro: 4
Tamanho em bytes de um float: 4
Tamanho em bytes de um double: 8
Tamanho em bytes de Numero_de_Alunos (int): 4
Tamanho em bytes de nome[40]: 40
Tamanho em bytes de notas[60]: 480
Process returned 0 (0x0)   execution time : 1.519 s
Press any key to continue.
```

# Memória

- ▶ Podemos pensar na memória como uma sequência linear de bytes, sendo que cada byte possui um endereço.
- ▶ A memória é limitada
- ▶ O Sistema operacional (SO) gerencia a memória
- ▶ Vale observar que esse esquema é usado para entender alocação. A que ocorre de fato depende de vários fatores: so, compilador, otimização, etc.
- ▶ Lembre também da arquitetura de Von Neumann (instruções e dados compartilham o mesmo endereçamento de memória)



The diagram illustrates memory layout with two tables. A downward arrow points from the top to the first table. A second downward arrow points from the first table to the second table. The first table has columns: Endereço, Blocos (1 byte), Nome variável, and Tipo. It contains rows 1 through 27. The second table has the same columns and contains rows 48 through 74. Below the second table, there is an ellipsis (...) and a small table with two rows: 4294967294 and 4294967295.

Endereço	Blocos (1 byte)	Nome variável	Tipo
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			
26			
27			

Endereço	Blocos (1 byte)	Nome variável	Tipo
48			
49			
50			
51			
52			
53			
54			
55			
56			
57			
58			
59			
60			
61			
62			
63			
64			
65			
66			
67			
68			
69			
70			
71			
72			
73			
74			

4294967294			
4294967295			

# Exercício

---

- ▶ Indique, usando o mapa de memória do slide anterior, um possível estado da memória ao fim da execução do seguinte programa.
- ▶ Indique quantos bytes as variáveis declaradas ocupam
- ▶ Continue o programa para mostrar a soma da quantidade de memória ocupada (use a variável `tamanho_total`)

```
int idade;  
char nome[10] = "Maria";  
double peso, altura;  
int casada;  
float grau_miopia[2];  
unsigned int tamanho_total;  
  
altura = 1.65;  
peso = 70;  
casada = 0; // false  
grau_miopia[0] = 2.75; // olho esquerdo  
grau_miopia[1] = 3; // olho direito
```

---



# Exemplos de alocação

► `char nome[10] = "Maria"`

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1	'M'	nome[0]	char
2	'a'	nome[1]	char
3	'r'	nome[2]	char
4	'i'	nome[3]	char
5	'a'	nome[4]	char
6	'\0'	nome[5]	char
7	lx	nome[6]	char
8	lx	nome[7]	char
9	lx	nome[8]	char
10	lx	nome[9]	char
11			

obs

forma correta: são alocados 10 bytes de memória do tipo char (o tipo char ocupa 1 byte)

\*\*\* obs: na verdade as posições de 7 a 10 são inicializadas com `\0`, mas esse comportamento não é padrão em comandos como `gets` e `strcpy`



# Exemplos de alocação

---

► `char nome[10] = "Maria"` – **Forma errada**

11			
12	'M'	nome[10]	
13	'a'		
14	'r'		
15	'i'		
16	'a'		
17	'\0'		
18	lx		
19	lx		
20	lx		
21	lx		
22			
23			

erro: nome[10] representa a décima primeira posição do vetor nome, posição esta que não existe! (ele pegaria, neste caso, a posição 22)



# Exemplos de alocação

---

► `char nome[10] = "Maria"` – **Forma errada**

22			
23			
24	'M'	nome[0]	char
25	'a'	nome[1]	char
26	'r'	nome[2]	char
27	'i'	nome[3]	char
28	'a'	nome[4]	char
29	'\0'	nome[5]	char
30		nome[6]	char
31		nome[7]	char
32		nome[8]	char
33		nome[9]	char
34			

erro: faltou colocar o lixo (lx) de nome[6] até nome[9]. Mesmo que "Maria" não ocupe todo o vetor, ele é alocado. Além disso, como não houve inicialização em parte do vetor, essa parte é lixo

# Exemplos de alocação

► `char nome[10] = "Maria"` – **Forma errada**

Endereço	Blocos (1 byte)	Nome variável	Tipo
47			
48	'M'	nome[0]	char
49	'a'	nome[1]	char
50	'r'	nome[2]	char
51	'i'	nome[3]	char
52	'a'	nome[4]	char
53	'\0'	nome[5]	char
54	lx	nome[6]	char
55		nome[7]	char
56		nome[8]	char
57		nome[9]	char
58			
59			

erro: tem que ser 'lx' para as quarto posições,  
pois são quatro 'char' distintos

# Exemplos de alocação

---

► `char nome[10] = "Maria"` – **Forma errada**

58			
59			
60	'M'	nome[1]	char
61	'a'	nome[2]	char
62	'r'	nome[3]	char
63	'i'	nome[4]	char
64	'a'	nome[5]	char
65	'\0'	nome[6]	char
66		nome[7]	char
67	lx	nome[8]	char
68		nome[9]	char
69		nome[10]	char
70			

erro: em C vetor sempre começa na posição zero (0)



# Exemplos de alocação

► `char nome[10] = "Maria"` – Forma errada

70			
71	'M'	nome[0]	char
72	'a'	nome[1]	char
73	'r'	nome[2]	char
74	'i'	nome[3]	char
75	'a'	nome[4]	char
76	lx	nome[5]	char
77	lx	nome[6]	char
78	lx	nome[7]	char
79	lx	nome[8]	char
80	lx	nome[9]	char
81			

erro: faltou colocar o '\0' de fim de string

# Exemplos de alocação

► **double** peso = 10;

Endereço	Blocos (1 byte)	Nome variável	Tipo
0 / NULL	indefinido	----	----
1			
2	10	peso	double
3			
4			
5			
6			
7			
8			
9			
10			

Correto. Um tipo double ocupa 8 bytes. Assim, independente do valor atribuído a variável peso (10, 20, 1 milhão), serão 8 bytes ocupados

# Exemplos de alocação

---

► `double peso = 10;` - **Forma errada**

10			
11			
12	10	peso	double
13	lx		
14	lx		
15	lx		
16	lx		
17	lx		
18	lx		
19	lx		
20			
21			

Erro. Todos os 8 bytes pertencem à variável. Uma vez atribuído o valor, ele ocupa todos os bits, e não só o primeiro, independente do valor (10, 20 ou 1 milhão)



# Exemplos de alocação

- ▶ `float grau_miopia[2];`
- ▶ `grau_miopia[0] = 3; grau_miopia[1]=2.5;`

Endereço	Blocos (1 byte)	Nome variável	Tipo
47	3	grau_miopia[0]	float
48			
49			
50			
51	2.5	grau_miopia[1]	float
52			
53			
54			
55			
56			

Correto. Cada elemento do vetor ocupa 4 bytes. O endereço de grau\_miopia[0] é 47 e o de grau\_miopia[1] é 51



# Exemplos de alocação

---

- ▶ `float grau_miopia[2];` - **Forma errada**
- ▶ `grau_miopia[0] = 3; grau_miopia[1]=2.5;`

56			
57			
58	3	grau_miopia[0]	float
59			
60			
61			
62			
63	2.5	grau_miopia[1]	float
64			
65			
66			
67			

Erro. A alocação de dados de vetores é contínua, não há espaço entre um elemento e outro

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
    int idade;
    char nome[10] = "Maria";
    double peso, altura;
    int casada;
    float grau_miopia[2];
    unsigned int tamanho_total;

    altura = 1.65;
    peso = 70;
    casada = 0; // false
    grau_miopia[0] = 2.75; // olho esquerdo
    grau_miopia[1] = 3; // olho direito

    // obs: o símbolo \ serve para continuar um comando em
    // uma outra linha.
    tamanho_total = sizeof(nome) + sizeof(altura) + sizeof(peso)+ \
                    sizeof(casada)+sizeof(grau_miopia)+sizeof(idade) + \
                    sizeof(tamanho_total);
    printf("\n Tamanho em bytes ocupado: %u", tamanho_total);

    return 0;
}
```

---



# Variáveis

---

- ▶ As variáveis vistas até agora eram:
  - ▶ simples: definidas por tipos **int**, **float**, **double** e **char**;
  - ▶ compostas homogêneas (ou seja, do mesmo tipo): definidas por **array**.
- ▶ No entanto, a linguagem C permite que se criem novas estruturas a partir dos tipos básicos.
  - ▶ Struct



# Estruturas

---

- ▶ Uma estrutura pode ser vista como um **novo tipo de dado**, que é formado por composição de outros tipos.
  - ▶ Pode ser declarada em qualquer escopo (o conceito de escopo será explicado mais adiante no curso)
  - ▶ Ela é declarada da seguinte forma:

```
struct nomestruct {  
    tipo1 campo1;  
    tipo2 campo2;  
    ...  
    tipoN campoN;  
};
```

Em inglês:

```
struct tag { //structure template  
    declarations //declaration of members  
};
```

# Estruturas

---

- ▶ Uma estrutura pode ser vista como um agrupamento de dados.
  - ▶ Ex.: cadastro de pessoas.

```
int idade;  
char nome[10] = "Maria";  
double peso, altura;  
int estado_civil;  
float grau_miopia[2];
```



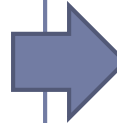
Todas essas informações são da mesma pessoa – podemos agrupá-las. Isso facilita também lidar com dados de outras pessoas no mesmo programa

# Estruturas

---

- ▶ Uma estrutura pode ser vista como um agrupamento de dados.
  - ▶ Ex.: cadastro de pessoas.

```
int idade;  
char nome[10] = "Maria";  
double peso, altura;  
int estado_civil;  
float grau_miopia[2];  
unsigned int tamanho_total;
```



```
struct dados_pacientes {  
    int idade;  
    char nome[10];  
    double peso;  
    double altura;  
    int estado_civil;  
    float grau_miopia[2];  
};
```

# Estruturas – declaração de variáveis

---

- ▶ Uma vez definida a estrutura, uma **variável** pode ser declarada de modo similar aos tipos já existente:
  - ▶ `struct dados_pacientes` paciente1;
- ▶ Obs: por ser um tipo definido pelo programador, usa-se a palavra **struct** antes do tipo da nova variável

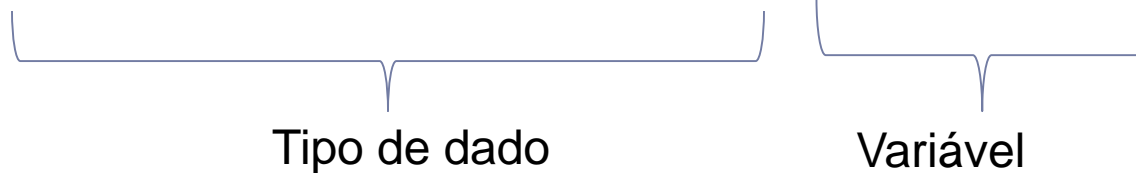


# Estruturas – declaração de variáveis

---

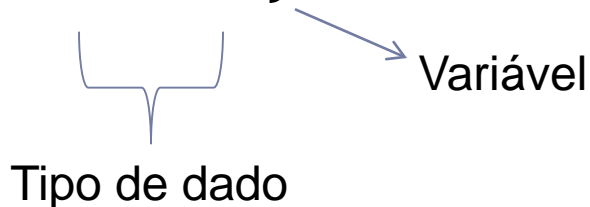
- ▶ Obs: por ser um tipo definido pelo programador, usa-se a palavra **struct** antes do tipo da nova variável

- ▶ `struct dados_pacientes` `paciente1`;



- ▶ Compare com a declaração de uma variável inteira:

- ▶ `int` `a`;





# Exercício

---

- ▶ Declare uma estrutura capaz de armazenar o número inteiro e 3 notas inteiras para um dado aluno.



# Exercício - Solução

---

```
struct aluno {  
    int num_aluno;  
    int nota1;  
    int nota2;  
    int nota3;  
};
```

ou

```
struct aluno {  
    int num_aluno;  
    int nota1, nota2, nota3;  
};
```

ou

```
struct aluno {  
    int num_aluno;  
    int nota[3];  
};
```

---



# Estruturas

---

- ▶ O uso de estruturas facilita na manipulação dos dados do programa. Imagine declarar 4 cadastros, para 4 pacientes diferentes:

```
char nome1[10], nome2[10], nome3[10], nome4[10];  
int idade1, idade2, idade3, idade4;  
double grau_miopia1[2],grau_miopia2[2],grau_miopia3[2],grau_miopia4[2];
```

Ou

```
char nome1[4][10];  
int idade[4];  
double grau_miopia1[4][2];
```



# Estruturas

---

- ▶ Utilizando uma estrutura, o mesmo pode ser feito da seguinte maneira:
  - ▶ Declarando variáveis para 4 pacientes e um cliente especial

```
struct dados_pacientes {  
    int idade;  
    char nome[10];  
    double peso;  
    double altura;  
    int estado_civil;  
    float grau_miopia[2];  
};
```

```
struct dados_pacientes paciente1, paciente2, \  
                                paciente3, paciente4;  
struct dados_pacientes cliente_especial;
```



# Acesso aos elementos

---

- ▶ Como é feito o acesso aos elementos da estrutura?
  - ▶ Cada elemento da estrutura pode ser acessado com o operador ponto “.”
  - ▶ Ex.:

```
// declarando a variável da struct  
struct dados_pacientes cliente_especial;
```

```
// acessando os elementos da struct  
scanf("%d",&cliente_especial.idade);  
scanf("%lf",&cliente_especial.peso);  
gets(cliente_especial.nome);
```



# Acesso aos elementos

---

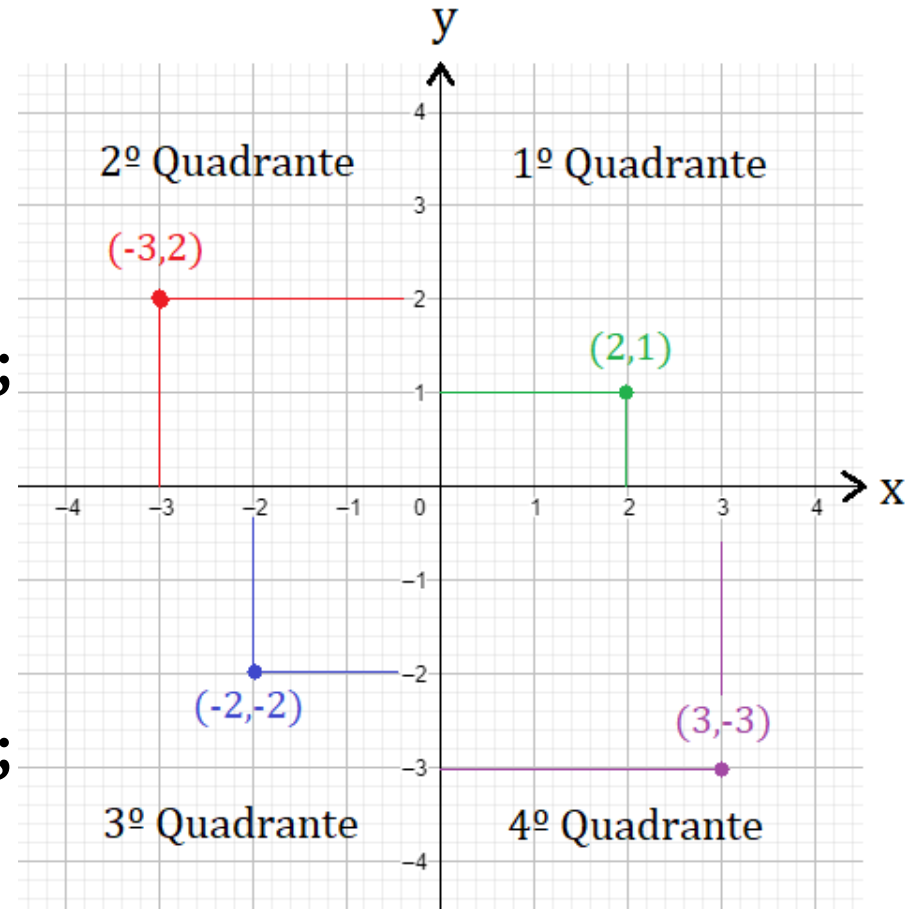
- ▶ Como nos arrays, uma estrutura pode ser previamente inicializada:

```
struct ponto {  
    int x;  
    int y;  
};  
struct ponto p1 = { 220, 110 };
```



# Acesso aos elementos

```
struct ponto {  
    int x;  
    int y;  
};  
struct ponto pontQuad1;  
pontQuad1.x = 2;  
pontQuad1.y = 1;  
  
struct ponto pontQuad2;  
pontQuad2.x = -3;  
pontQuad2.y = 2;
```



# Acesso aos elementos

---

- ▶ E se quiséssemos ler os valores dos elementos da estrutura utilizando o teclado?
  - ▶ Resposta: basta ler cada elemento independentemente, respeitando seus tipos.

```
gets(cliente_especial.nome); //string  
scanf("%d",&cliente_especial.idade); //int  
scanf("%f",&cliente_especial.grau_miopia[0]); //float  
scanf("%f",&cliente_especial.grau_miopia[1]); //float
```





# Acesso aos elementos

---

- ▶ Note que cada elemento dentro da estrutura pode ser acessado como se apenas ele existisse, não sofrendo nenhuma interferência dos outros.
- ▶ Uma estrutura pode ser vista como um simples agrupamento de dados.
- ▶ Se faço um `scanf` para `estrutura.idade` não me obriga a fazer um `scanf` para `estrutura.peso`



# Memória

---

- ▶ Faça o mapa de memória para o seguintes código, sabendo que os elementos de um struct são alocados sequencialmente na memória

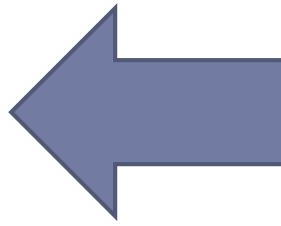


```
int main()
{
    struct dados_pacientes {
        int idade;
        char nome[10];
        double peso;
        double altura;
        int estado_civil;
        float grau_miopia[2];
    };

    unsigned int tamanho_da_struct;
    struct dados_pacientes paciente1, paciente2 ;


    // lembre que string é um vetor, não posso fazer
    // paciente1.nome = "José"
    // o correto é usar strcpy -string copy
    strcpy(paciente1.nome, "Jose");
    paciente1.altura = 1.25;
    paciente1.peso = 73;
    paciente1.estado_civil = 1; // 0 para solteiro
    paciente1.grau_miopia[0] = 1.75; // olho esquerdo
    paciente1.grau_miopia[1] = 0; // olho direito
}
```

```
struct dados_pacientes {  
    int idade;  
    char nome[10];  
    double peso;  
    double altura;  
    int estado_civil;  
    float grau_miopia[2];  
};
```



Podemos declarar a struct fora do main()

Isso na verdade é o mais comum, e será importante quando a struct for usada por outras funções do programa

```
int main()   
{  
    unsigned int tamanho_da_struct;  
    struct dados_pacientes paciente1, paciente2 ;  
  
    // lembre que string é um vetor, não posso fazer  
    // paciente1.nome = "José"  
    // o correto é usar strcpy -string copy  
    strcpy(paciente1.nome, "Jose");  
    paciente1.altura = 1.25;  
    paciente1.peso = 73;  
    paciente1.estado_civil = 1; // 0 para solteiro  
    paciente1.grau_miopia[0] = 1.75; // olho esquerdo  
    paciente1.grau_miopia[1] = 0; // olho direito  
}
```

# Estruturas

---

- ▶ Voltando ao exemplo anterior, se, ao invés de 4 cadastros, quisermos fazer 100 cadastros de pacientes?



# Array de estruturas

---

- ▶ SOLUÇÃO: criar um **array de estruturas**.
- ▶ Sua declaração é similar a declaração de uma array de um tipo básico
  - ▶ `struct dados_pacientes pacientes[100];`
  - ▶ Desse modo, declara-se um array de 100 posições, onde cada posição é do tipo `struct dados_pacientes`
  - ▶ `struct dados_pacientes pacientes[100];`



# Array de estruturas

---

- ▶ SOLUÇÃO: criar um **array de estruturas**.
- ▶ Sua declaração é similar a declaração de uma array de um tipo básico
  - ▶ `struct dados_pacientes pacientes[100];`
- ▶ Desse modo, declara-se um array de 100 posições, onde cada posição é do tipo `struct dados_pacientes`

▶ `struct dados_pacientes pacientes[100];`

Tipo de dado      Variável      Tamanho do vetor

Quanto bytes ocupa a variável pacientes?

# Array de estruturas

---

## ▶ Lembrando:

- ▶ struct: define um “conjunto” de variáveis que podem ser de tipos diferentes;
- ▶ array: é uma “lista” de elementos de mesmo tipo.





# Array de estruturas

---

- ▶ Num array de estruturas, o operador de ponto (.) vem depois dos colchetes ([ ]) do índice do array.

```
int main(){
    struct cadastro c[4];
    int i;
    for(i=0; i<4; i++){
        gets(c[i].nome);
        scanf("%d",&c[i].idade);
        gets(c[i].rua);
        scanf("%d",&c[i].numero);
    }
    system("pause");
    return 0;
}
```



# Exercício

---

- Utilizando a estrutura abaixo, faça um programa para ler o número e as 3 notas de 10 alunos. Calcule a média para cada aluno e armazene na estrutura.

```
struct aluno {  
    int num_aluno;  
    float nota1, nota2, nota3;  
    float media;  
};
```



# Exercício – Solução (sem printf's)

---

```
► struct aluno {
    int num_aluno;
    float nota1, nota2, nota3;
    float media;
};

int main(){

    struct aluno a[10];
    int i;
    for (i=0;i<10;i++){
        scanf("%d",&a[i].num_aluno);
        scanf("%f",&a[i].nota1);
        scanf("%f",&a[i].nota2);
        scanf("%f",&a[i].nota3);
        a[i].media = (a[i].nota1 + a[i].nota2 + a[i].nota3)/3.0
    }
}
```

---

# Exercício

---

- ▶ Utilizando a estrutura abaixo, faça um programa para ler o número e as 3 notas de 10 alunos. Calcule a média para cada aluno e armazene na estrutura.
- ▶ Note que temos um vetor dentro da estrutura

```
struct aluno {  
    int num_aluno;  
    float nota[3];  
    float media;  
};
```



# Exercício – Solução (sem printf)

---

```
int main(){
    struct aluno a[10];
    int i;
    for (i=0;i<10;i++){
        scanf("%d",&a[i].num_aluno);
        scanf("%f",&a[i].nota[0]);
        scanf("%f",&a[i].nota[1]);
        scanf("%f",&a[i].nota[2]);
        a[i].media = (a[i].nota[0] + a[i].nota[1] + a[i].nota[2])/3.0
    }
}
```



# Atribuição entre estruturas

---

- ▶ Atribuições entre estruturas só podem ser feitas quando os campos são IGUAIS!

- ▶ Ex:

```
struct cadastro c1,c2;
```

```
c1 = c2; //CORRETO
```

- ▶ Ex:

```
struct cadastro c1;
```

```
struct ficha c2;
```

```
c1 = c2; //ERRADO!! TIPOS DIFERENTES
```



# Atribuição entre estruturas

---

- ▶ No caso de estarmos trabalhando com arrays, a atribuição entre diferentes elementos do array é válida
  - ▶ Ex:  
struct cadastro c[10];  
c[1] = c[2]; //CORRETO
- ▶ Note que nesse caso, os tipos dos diferentes elementos do array são sempre IGUAIS.

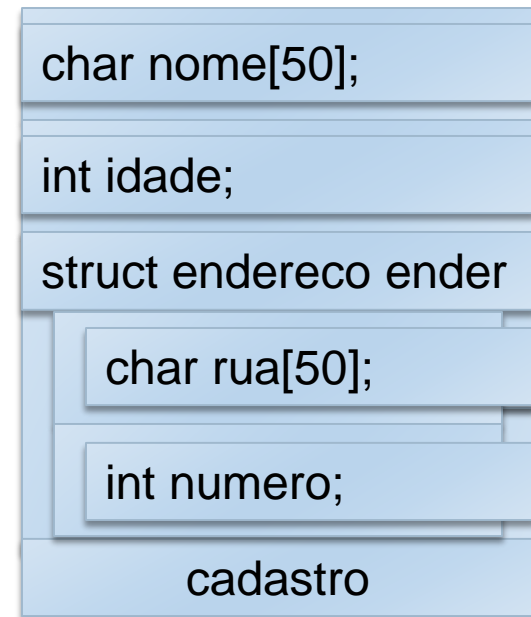


# Estruturas de estruturas

---

- ▶ Sendo uma estrutura um tipo de dado, podemos declarar uma estrutura que utilize outra estrutura previamente definida:

```
struct endereco{  
    char rua[50]  
    int numero;  
};  
struct cadastro{  
    char nome[50];  
    int idade;  
    struct endereco ender;  
};
```

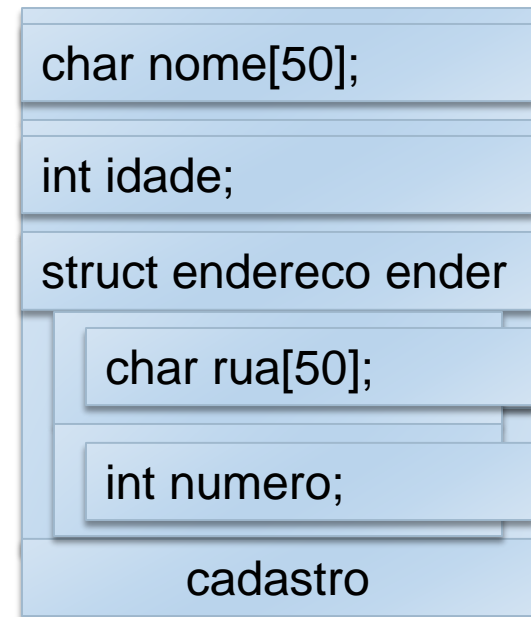




# Estruturas de estruturas

- ▶ Sendo uma estrutura um tipo de dado, podemos declarar uma estrutura que utilize outra estrutura previamente definida:

```
struct endereco{  
    char rua[50]  
    int numero;  
};  
struct cadastro{  
    char nome[50];  
    int idade;  
    struct endereco ender;  
};
```



# Estruturas de estruturas

---

- ▶ Nesse caso, o acesso aos dados do **endereço** do cadastro é feito utilizando novamente o operador “.”.

```
struct cadastro c;
```

```
gets(c.nome);
```

```
scanf("%d",&c.idade);
```

```
gets(c.ender.rua);
```

```
scanf("%d",&c.ender.numero);
```



# Estruturas de estruturas

---

## ► Outros exemplos

```
struct cadastro c;
```

```
strcpy(c.nome, "João");
```

```
c.idade = 30;
```

```
strcpy(c.ender.rua, "Avenida 1");
```

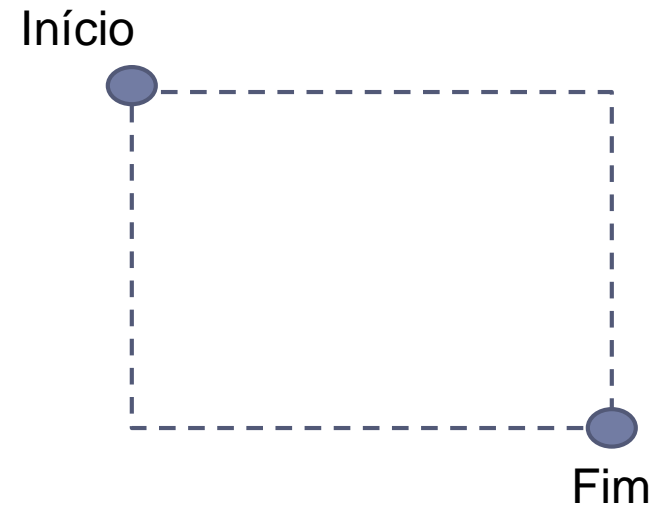
```
c.ender.numero = 45;
```



---

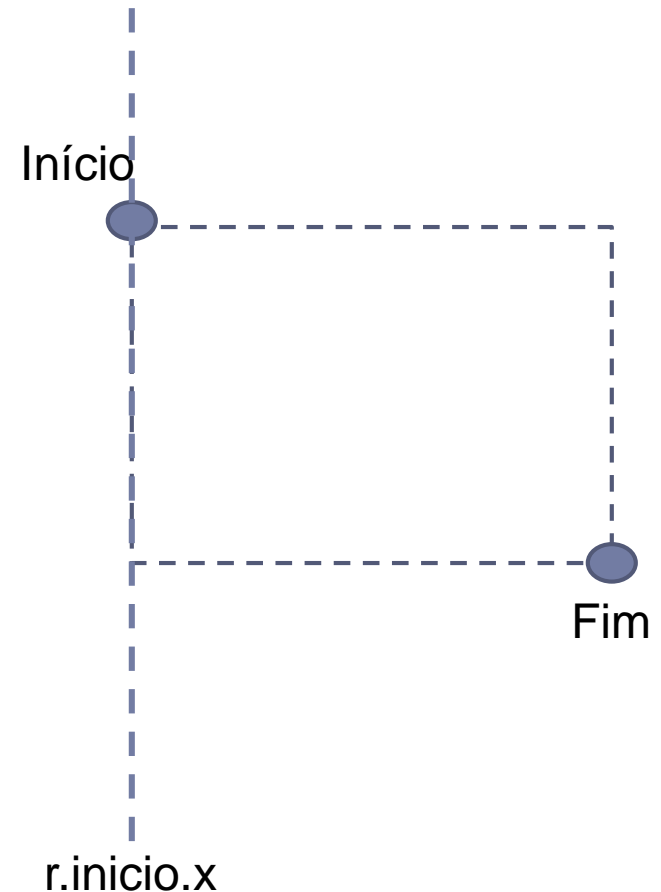
## ► Lendo um retângulo

```
struct ponto {  
    int x, y;  
};  
  
struct retangulo {  
    struct ponto inicio, fim;  
};  
  
struct retangulo r;  
  
scanf("%d",&r.inicio.x);  
scanf("%d",&r.inicio.y);  
scanf("%d",&r.fim.x);  
scanf("%d",&r.fim.y);
```



## ► Lendo um retângulo

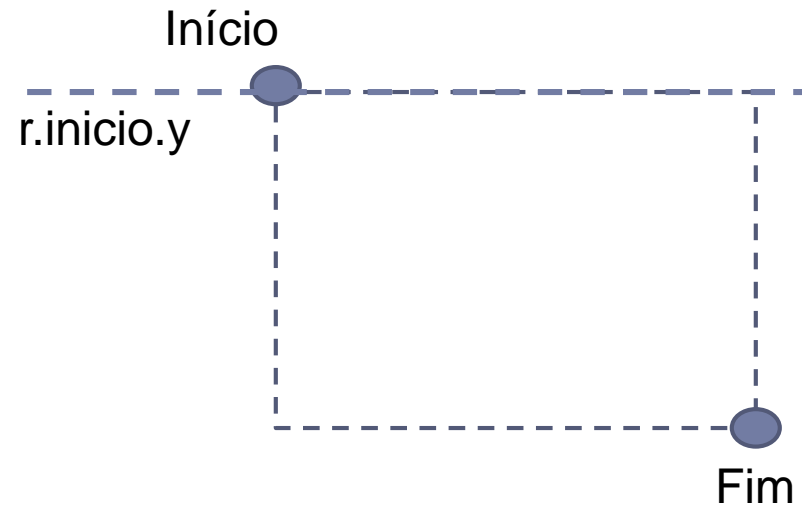
```
struct ponto {  
    int x, y;  
};  
  
struct retangulo {  
    struct ponto inicio, fim;  
};  
  
struct retangulo r;  
  
scanf("%d",&r.inicio.x);  
scanf("%d",&r.inicio.y);  
scanf("%d",&r.fim.x);  
scanf("%d",&r.fim.y);
```



---

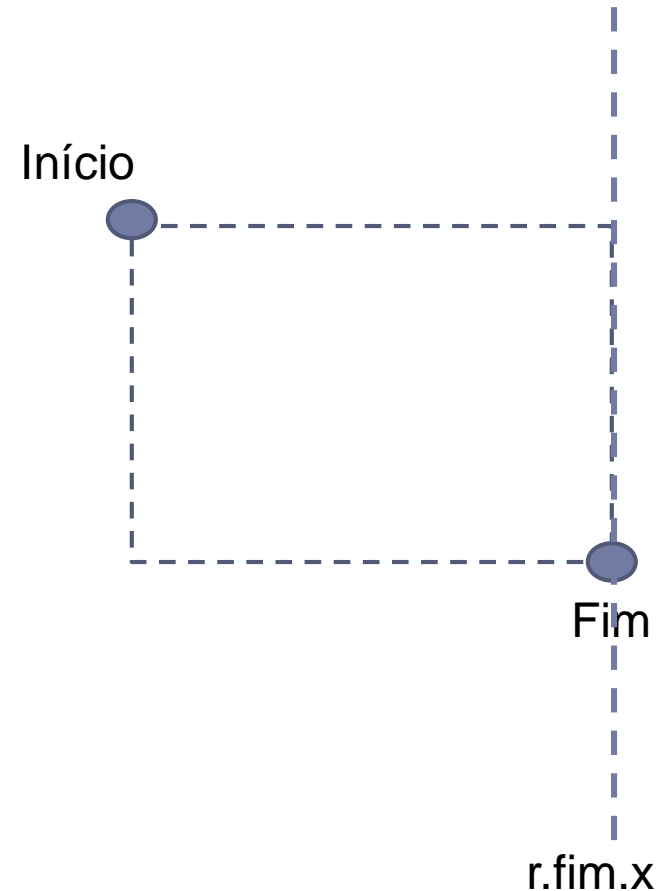
## ► Lendo um retângulo

```
struct ponto {  
    int x, y;  
};  
  
struct retangulo {  
    struct ponto inicio, fim;  
};  
  
struct retangulo r;  
  
scanf("%d",&r.inicio.x);  
scanf("%d",&r.inicio.y);  
scanf("%d",&r.fim.x);  
scanf("%d",&r.fim.y);
```



## ► Lendo um retângulo

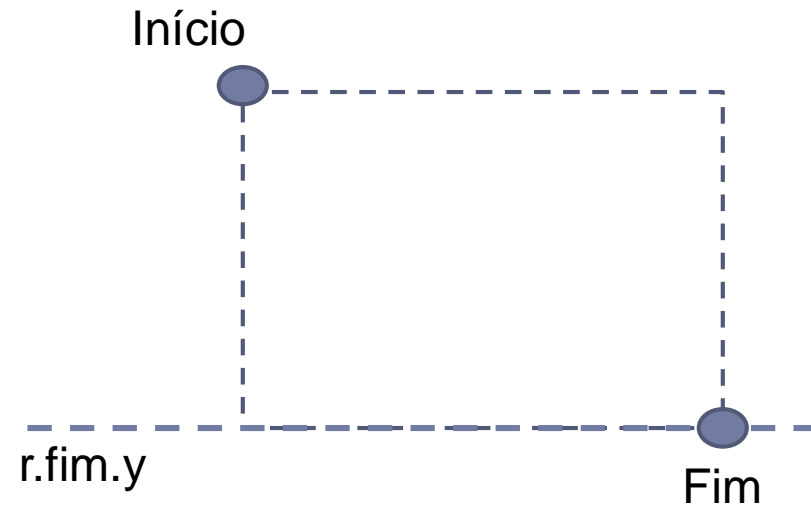
```
struct ponto {  
    int x, y;  
};  
  
struct retangulo {  
    struct ponto inicio, fim;  
};  
  
struct retangulo r;  
  
scanf("%d",&r.inicio.x);  
scanf("%d",&r.inicio.y);  
scanf("%d",&r.fim.x);  
scanf("%d",&r.fim.y);
```



---

## ► Lendo um retângulo

```
struct ponto {  
    int x, y;  
};  
  
struct retangulo {  
    struct ponto inicio, fim;  
};  
  
struct retangulo r;  
  
scanf("%d",&r.inicio.x);  
scanf("%d",&r.inicio.y);  
scanf("%d",&r.fim.x);  
scanf("%d",&r.fim.y);
```





# Estruturas de estruturas


---

- Inicialização de uma estrutura de estruturas:

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

```
struct retangulo {  
    struct ponto inicio, fim;  
};
```

```
struct retangulo r = {{10,20},{30,40}};
```

  
                    inicio                fim



# Material Complementar

---

## ▶ Vídeo Aulas

- ▶ Aula 35: Struct: Introdução
- ▶ Aula 36: Struct: Trabalhando com Estruturas
- ▶ Aula 37: Struct: Arrays de Estruturas
- ▶ Aula 38: Struct: Aninhamento de Estruturas

