

---

# Revisão: Tipos de Dados

Parte desse material foi baseado nos slides dos Profs. André Backes e  
Bruno Travençolo

---

# Processamento de Dados

---

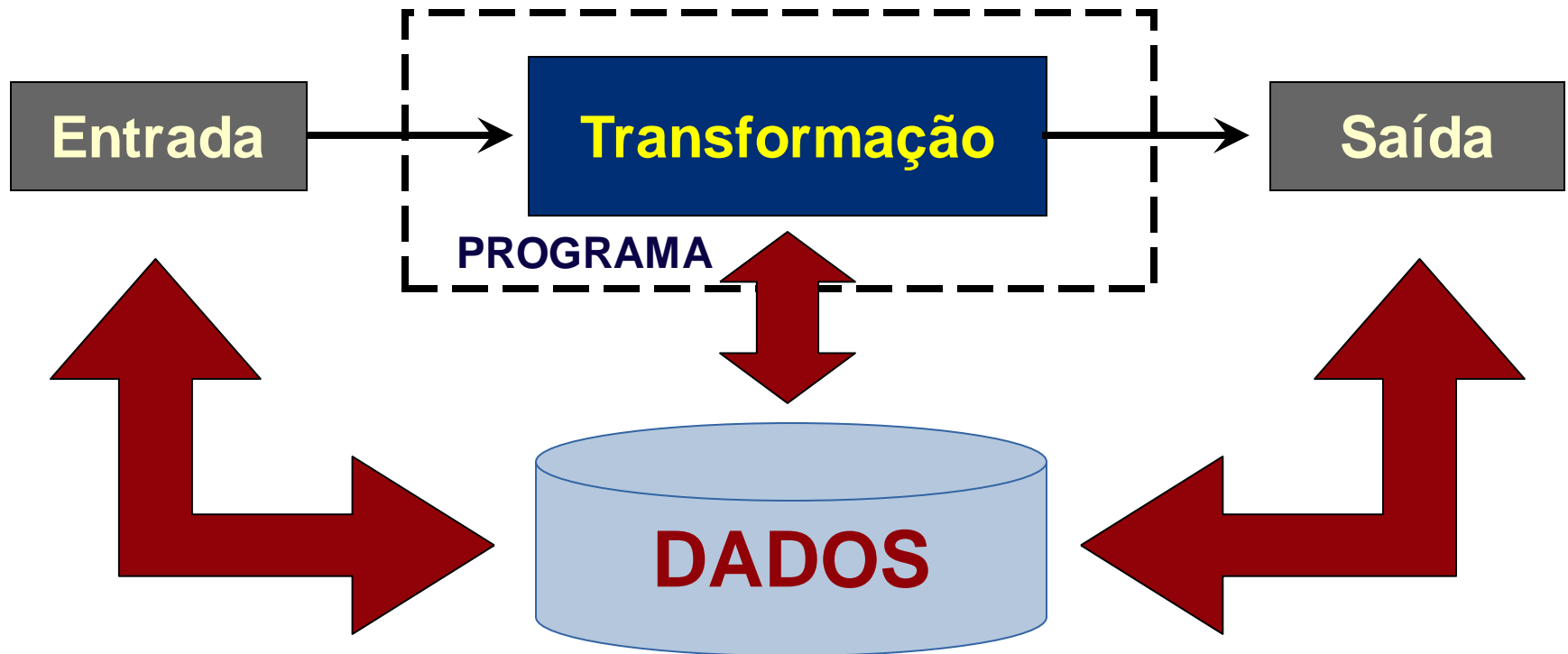


- ▶ O computador é uma **ferramenta** que permite o processamento de dados



# Processamento de Dados

---



- ▶ O computador é uma **ferramenta** que permite o processamento de dados

# Importância dos Dados

---

- ▶ Variáveis são usadas em várias partes do algoritmo:

TIPO	EXEMPLO
Entrada de dados	<code>scanf("%d", &amp;idade);</code>
Saída de dados	<code>printf("Acertos=%d", qtde);</code>
Armazenar resultado/valor (atribuição)	<code>area = base * altura;</code>
Acumuladores/Contadores	<code>cont++; //Conta</code> <code>soma = soma + num; //Acumula</code>
Tomada de decisão (seleção/repetição)	<code>if ( media &gt;= 60 )</code>
Sinalizadores/Flags	<code>Atualizado = 1; // True</code>



# Processamento de Dados

---

- ▶ Programas só reconhecem dados **armazenados na memória principal**
  - Dados estão fixos no código (**constantes**) ou armazenados em alguma posição da memória (**variáveis**)
  - Instruções geralmente envolvem **movimentação e/ou transformação desses dados**
- ▶ Ao conceber um algoritmo, podemos considerar a abstração:  
**Memória = coleção de caixas**
  - Cada caixa possui as seguintes características:
    - Tem um identificador (**nome**)
    - Sempre armazena valor
    - Possui uma **posição exclusiva** na memória



# Abstração da Memória

---

## Abstração da Memória

IDENTIFICADOR	IDADE	NOME	X1
VALOR	18	“João”	2.5

- ▶ As “caixas” de memória (variáveis) seguem as premissas:
  - Ao atribuir um novo valor a uma posição da memória, o seu valor atual será substituído (**perdido**)
  - Mesmo sem qualquer atribuição explícita, uma posição de memória **sempre possui algum conteúdo** (chamado de **“lixo”**)

# Tipos de Dados

---

## ▶ **Tipos primitivos:**

- ▶ São as representações mais elementares (**simples**) dos dados disponibilizados em uma linguagem de programação
- ▶ Armazena um único valor por variável
- ▶ **Ex:** int, char e float

## ▶ **Tipos estruturados:**

- ▶ Possibilita **estruturar dados complexos**
- ▶ **Coleção de dados** relacionados a um único objeto
  - Composta por tipos primitivos e/ou outros tipos estruturados
  - **Ex:** ponto, alunos, notas, faturamento mensal, etc.
- ▶ Podem ser **homogêneas** ou **heterogêneas**



# Tipos Estruturados (**Estruturas Homogêneas**)

---

- ▶ Estruturas de dados compostas por **elementos do mesmo tipo de dado** (arranjos ou *arrays*)
- ▶ **Vetores:** estrutura linear que suporta  $N$  posições distribuídas sequencialmente
  - **Ex:** `char nome[100];`
- ▶ **Matrizes:** estrutura espacial que suporta  $N \times M$  posições
  - **Ex:** `int mat_adj[5][5];`





# Tipos Estruturados (**Estruturas Heterogêneas**)

---

- ▶ Estruturas de dados formadas por  $K$  **elementos de diferentes tipos de dados**
- ▶ Os elementos são denominados **campos da estrutura**
  - Cada campo tem um tipo de dado próprio
- ▶ **Exemplo:**
  - `struct funcionario {`
  - `char nome[100];`
  - `int idade;`
  - `float salario;`
  - `};`



# Alocação dos Dados na Memória

---

- ▶ Como os dados são armazenados na memória?
- ▶ Operador **sizeof** :
  - Tradução: **size** (tamanho) **of** (de)
  - Retorna o tamanho (**em bytes**) ocupado por um objeto ou tipo de dado
- ▶ Exemplo:
  - ▶ `printf("\nTamanho em bytes de um char: %u", sizeof(char));`
    - ▶ Retorna 1, pois o tipo char tem 1 byte
  - ▶ Retorna um tipo **size\_t**, normalmente **unsigned int**, por isso o **%u**  
**unsigned int** - número inteiro sem sinal negativo

# Exemplo

---

```
#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;
}
```

---



# Exemplo

```
#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 );
```

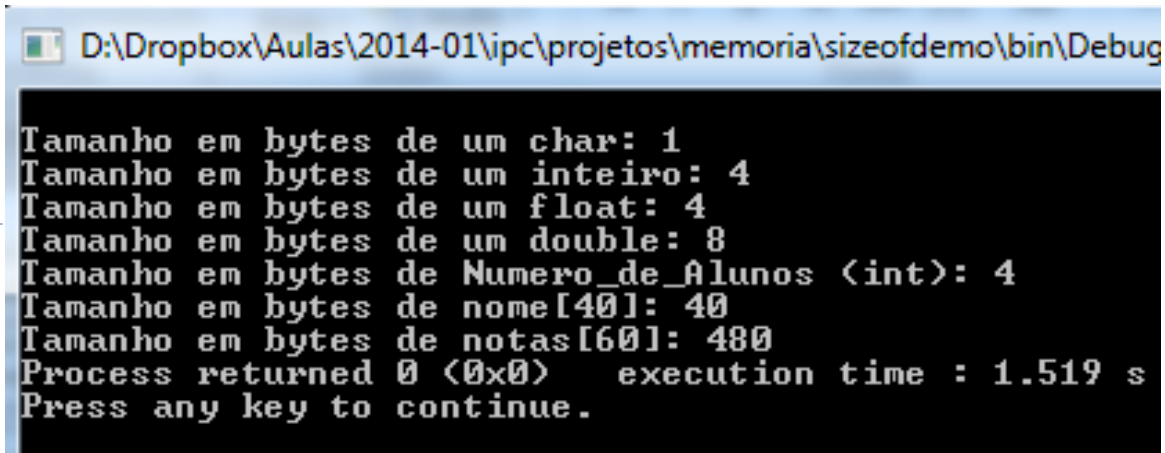
```
    // também é possível obter o tamanho de vetores
```

```
    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;
```

```
}
```



```
D:\Dropbox\Aulas\2014-01\ipc\projetos\memoria\sizeofdemo\bin\Debug
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

- ▶ Pense na memória como uma **sequência linear de bytes**
  - Cada byte possui um endereço
- ▶ Possui **tamanho limitado**
- ▶ Gerenciada pelo S.O.

Blocos				Blocos			
Endereço	(1 byte)	Nome variável	Tipo	Endereço	(1 byte)	Nome variável	Tipo
1				48			
2				49			
3				50			
4				51			
5				52			
6				53			
7				54			
8				55			
9				56			
10				57			
11				58			
12				59			
13				60			
14				61			
15				62			
16				63			
17				64			
18				65			
19				66			
20				67			
21				68			
22				69			
23				70			
24				71			
25				72			
26				73			
27				74			
...				...			
...				4294967294			
...				4294967295			

# Memória

- ▶ Pense na memória como uma **sequência linear de bytes**
  - Cada byte possui um endereço
- ▶ Possui **tamanho limitado**
- ▶ Gerenciada pelo S.O.

▶ **OBSERVAÇÃO:**

- ▶ Esquema simplificado usado para explicar a alocação
- ▶ Funcionamento real é mais complexo e depende de vários fatores: compilador, otimização, S.O., etc.

Blocos				Blocos			
Endereço	(1 byte)	Nome variável	Tipo	Endereço	(1 byte)	Nome variável	Tipo
1				48			
2				49			
3				50			
4				51			
5				52			
6				53			
7				54			
8				55			
9				56			
10				57			
11				58			
12				59			
13				60			
14				61			
15				62			
16				63			
17				64			
18				65			
19				66			
20				67			
21				68			
22				69			
23				70			
24				71			
25				72			
26				73			
27				74			
...				...			
				4294967294			
				4294967295			

# Exemplo de alocação

---

- ▶ Considerando o mapa de memória do slide anterior e a quantidade de bytes que cada variável ocupa, podemos definir um possível estado da memória para o trecho de programa a seguir:

```
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	----	----		obs
1	'M'	nome[0]	char		forma correta: são alocados 10 bytes de memória do tipo char (o tipo char ocupa 1 byte)
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: 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**

0					
1	'M'	nome[10]			erro: nome[10] representa a décima primeira posição do vetor nome, posição esta que não existe! (ele pegaria, neste caso, a posição 22)
2	'a'				
3	'r'				
4	'i'				
5	'a'				
6	'\0'				
7	lx				
8	lx				
9	lx				
10	lx				
11					
12					

# Exemplos de alocação

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

22					
23					
24	'M'	nome[0]	char		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
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					

# Exemplos de alocação

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

Endereço	Blocos (1 byte)	Nome variável	Tipo		
47					
48	'M'	nome[0]	char		erro: tem que ser 'lx' para as quarto posições, pois são quatro 'char' distintos
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		nome[6]	char		
55	lx	nome[7]	char		
56		nome[8]	char		
57		nome[9]	char		
58					
59					

# Exemplos de alocação

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

58					
59					
60	'M'	nome[1]	char		erro: em C vetor sempre começa na posição zero (0)
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					

# Exemplos de alocação

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

70					
71	'M'	nome[0]	char		erro: faltou colocar o '\0' de fim de string
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					

# Exemplos de alocação

► **double** peso = 10;

Endereço	Blocos (1 byte)	Nome variável	Tipo		
0 / NULL	indefinido	----	----		
1					
2	10	peso	double		Correto. Um tipo double ocupa 8 bytes. Assim, independente do valor atribuído a variável peso (10, 20, 1milhão), serão 8 bytes ocupados
3					
4					
5					
6					
7					
8					
9					
10					

# Exemplos de alocação

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

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

# 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	Correto. Cada elemento do vetor ocupa 4 bytes. O endereço de grau_miopia[0] é 47 e o de grau_miopia[1] é 51
48				
49				
50				
51	2.5	grau_miopia[1]	float	
52				
53				
54				
55				
56				



# 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		Erro. A alocação de dados de vetores é contínua, não há espaço entre um elemento e outro
59					
60					
61					
62					
63	2.5	grau_miopia[1]	float		
64					
65					
66					
67					

```
#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;
```

```
}
```

► O restante do código contabiliza a quantidade total de memória ocupada pelas variáveis do programa

# Estruturas de Dados

---

- ▶ Podem ser vistas como um **novo tipo de dado** formado pela composição de outros tipos (**tipo heterogêneo - struct**)
  - **Representação lógica** de um objeto/elemento do problema
- ▶ Pode ser declarada em qualquer escopo (local ou global)
  - ▶ Declaração = definição do novo tipo
  - ▶ Declaração da struct  $\neq$  declaração da variável (não aloca memória)
- ▶ **Sintaxe da declaração:**
  - ▶ `struct nomestruct {`
    - tipo1 campo1;
    - tipo2 campo2;
    - ...
    - tipoN campoN;
  - ▶ `};`

**Define os dados que compõem a estrutura**

# Estrutura de Dados

---

## ▶ Exemplo de agrupamento de dados:

### ▶ Cadastro de pacientes

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



Todas essas informações são do mesmo paciente, portanto, podemos agrupá-las. Isso facilita também lidar com dados de outros pacientes no mesmo programa (**organização na memória**)

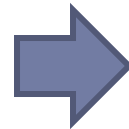
# Estrutura de Dados

---

## ▶ Exemplo de agrupamento de dados:

### ▶ Cadastro de pacientes

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



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

# Declaração de Variáveis

---

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




# Declaração de Variáveis

---

- ▶ Declaração de uma variável do tipo struct:

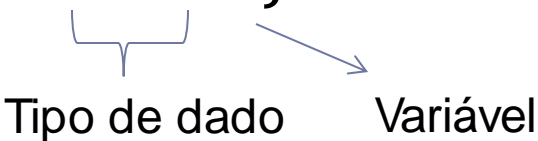
▶ `struct dados_pacientes` `paciente1;`



Tipo de dado                      Variável

- ▶ Declaração de uma variável inteira:

▶ `int` `a;`



Tipo de dado                      Variável



# Exercício

---

- ▶ Declare uma estrutura capaz de armazenar a matrícula (nro inteiro) e 3 notas para um dado aluno.





# Exercício - Solução

---

- Declare uma estrutura capaz de armazenar a matrícula (nro inteiro) e 3 notas para um dado aluno.

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

ou

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

ou, ainda

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



# Estruturas

---

- ▶ O uso de estruturas facilita na manipulação dos dados do programa
- ▶ **Ex:** declaração do cadastro de 4 pacientes diferentes poderia ser feita por:

```
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 nome[4][10];  
int idade[4];  
double grau_miopia[4][2];
```

---



# Estruturas

---

- Usando struct, a declaração pode ser feita por:

```
// Declaração do tipo de dados (struct)
struct dados_pacientes {
    int idade;
    char nome[10];
    double peso;
    double altura;
    int estado_civil;
    float grau_miopia[2];
};
```

```
// Declaração da variável (vetor de pacientes)
struct dados_pacientes pacientes[4];
```



# Acesso aos Campos da Estrutura

---

- ▶ Como é feito o acesso aos campos de uma variável do tipo struct?



# Acesso aos Campos da Estrutura

---

- ▶ Como é feito o acesso aos campos de uma variável do tipo struct?
- ▶ **R:** usar o operador ponto “.” para indicar o campo a ser acessado

- ▶ **Exemplo:**

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

// acessando os campos da struct
cliente_especial.idade = 18;
cliente_especial.peso= 80.5;
strcpy(cliente_especial.nome, “João”);
```



# Inicialização de Variáveis Estruturadas

---

- ▶ Assim como nos arrays, uma variável struct pode ser inicializada na sua declaração:

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

```
struct ponto p1 = { 220, 110 };
```



# Leitura de Variáveis Estruturadas

---

- ▶ Como ler os valores dos campos de uma variável struct do teclado?



# Leitura de Variáveis Estruturadas

---

- ▶ Como ler os valores dos campos de uma variável struct do teclado?
- ▶ **R:** Ler cada variável independentemente, respeitando seus respectivos tipos
- ▶ **Exemplo:**

```
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
```





# Leitura de Variáveis Estruturadas

---

- ▶ Cada campo dentro de uma estrutura pode ser acessado como uma variável independente
  - Seu uso não sofre interferência dos demais campos da estrutura
  - **Ex:** ler o campo `paciente_especial.idade` não me obriga a ler o campo `paciente_especial.peso`



# Exemplo de um Programa

---

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

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

int main() {

    struct dados_pacientes paciente;

    // lembre que string é um vetor, não pode atribuir direto
    strcpy(paciente.nome, "Jose");
    paciente.altura = 1.25;
    paciente.peso = 73;
    paciente.e_civil = 1; // 0:solteiro, 1:casado, 2:outro
    paciente.grau_miopia[0] = 1.75; // olho esquerdo
    paciente.grau_miopia[1] = 0; // olho direito
}
```

---



# Exemplo de um Programa

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

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

```
int main() {
    struct dados_pacientes paciente;
```

```
// lembre que string é um vetor, não pode atribuir direto
strcpy(paciente.nome, "Jose");
paciente.altura = 1.25;
paciente.peso = 73;
paciente.e_civil = 1; // 0:solteiro, 1:casado, 2:outro
paciente.grau_miopia[0] = 1.75; // olho esquerdo
paciente.grau_miopia[1] = 0; // olho direito
```

```
}
```

- ▶ A struct pode ser declarada fora da `main()`
- ▶ Isso é o mais comum e será importante quando a struct for usada por outras funções no programa

# Exercício

---

I- Considerando o programa do slide anterior e sabendo que os elementos de um struct são alocados sequencialmente na memória, faça o mapa de memória para o seu código.



# Dúvida

---

Considerando a `struct dados_pacientes`, como podemos alterar o código para fazer o cadastro de 100 pacientes?



# Vetor de Estruturas

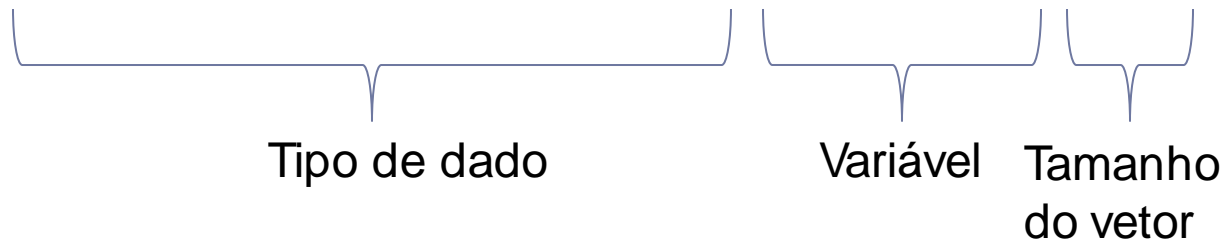
---

**Dúvida:** considerando a `struct dados_pacientes`, como podemos alterar o código para fazer o cadastro de 100 pacientes?

**SOLUÇÃO:** criar um **vetor de struct**

► Declaração similar a um array de tipo básico

► `struct dados_pacientes pacientes[100];`



► A variável `pacientes` contém 100 posições, onde cada uma é do tipo `struct dados_pacientes`



# Vetor de Estruturas

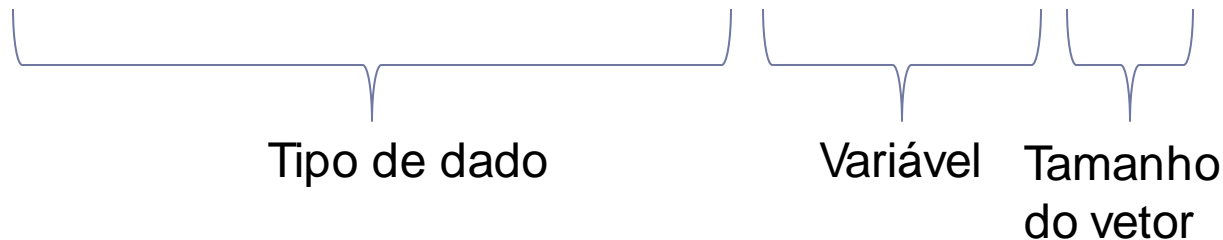
---

**Dúvida:** considerando a `struct dados_pacientes`, como podemos alterar o código para fazer o cadastro de 100 pacientes?

**SOLUÇÃO:** criar um **vetor de struct**

► Declaração similar a um array de tipo básico

► `struct dados_pacientes pacientes[100];`



Quanto bytes ocupa a variável pacientes?

► A variável `pacientes` contém 100 posições, onde cada uma é do tipo `struct dados_pacientes`

# Vetor de Estruturas

---

## ▶ Lembrando:

- ▶ **struct:** define um “conjunto” de campos que podem ser de tipos diferentes
- ▶ **Array:** é uma “lista” de elementos de mesmo tipo





# Vetor de Estruturas

---

## ▶ Lembrando:

- ▶ **struct:** define um “conjunto” de campos que podem ser de tipos diferentes
  - Deve somar o tamanho de todos os campos para obter o tamanho de um elemento do tipo struct
- ▶ **Array:** é uma “lista” de elementos de mesmo tipo
  - Deve multiplicar o tamanho de um elemento do tipo struct pela quantidade de elementos da lista
- ▶ *Essa conta fica como exercício e pode ser facilmente verificada usando o operador sizeof*



# Vetor de Estruturas

---

- ▶ O acesso aos campos da estrutura ainda é feito pelo operador de ponto (.) que vem depois dos colchetes ([ ]) do índice do vetor

- ▶ **Ex:**

```
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 1

---

- 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 1 – Solução (sem printf)

---

```
▶ 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 2

---

- ▶ Modifique o exercício anterior para considerar a estrutura abaixo
- ▶ Note que temos um vetor dentro da estrutura

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



## Exercício 2 – 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 2 estruturas só podem ocorrer se os **campos forem IGUAIS**

- ▶ **Exemplo 1:**

```
struct cadastro c1, c2;  
c1 = c2; // CORRETO!
```

- ▶ **Exemplo 2:**

```
struct cadastro c1;  
struct ficha c2;  
c1 = c2; // ERRADO! (TIPOS DIFERENTES)
```



# Atribuição entre Estruturas

---

- ▶ A atribuição entre diferentes elementos do array de estruturas é válida
  - Elementos de um mesmo array são sempre IGUAIS
  - **Exemplo:**  

```
struct cadastro c[10];  
c[1] = c[2]; // CORRETO!
```





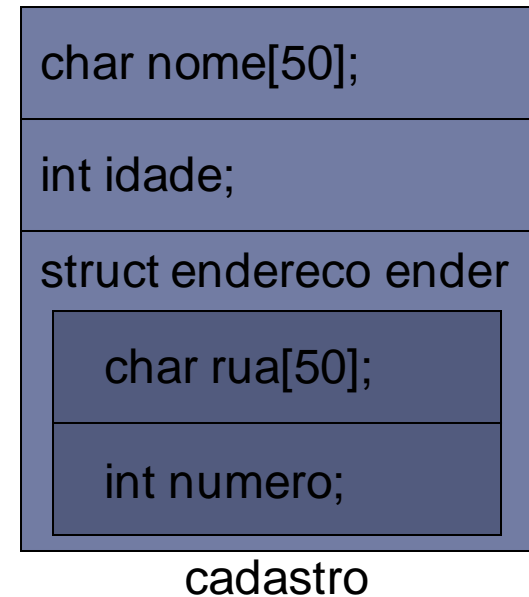
# Estrutura de Estruturas

---

- Considerando que uma estrutura é um tipo de dado, podemos declarar uma estrutura que utilize outra estrutura previamente definida

## Exemplo:

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



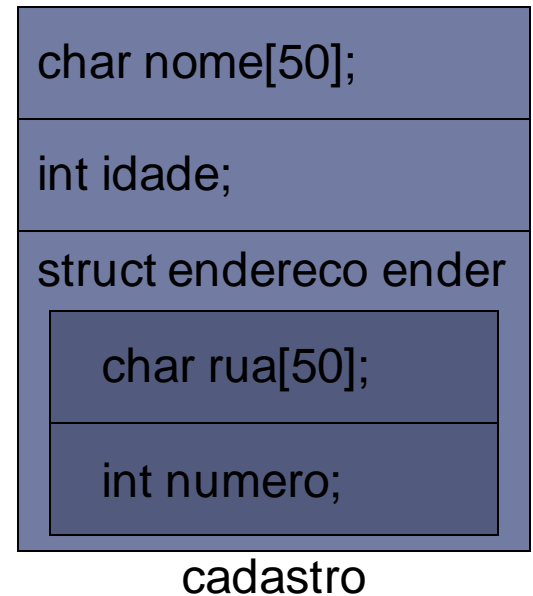
# Estrutura de Estruturas

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

- ▶ **Exemplo:**

```
struct cadastro c;
```

```
gets(c.nome);  
scanf("%d",&c.idade);  
gets(c.ender.rua);  
scanf("%d",&c.ender.numero);
```

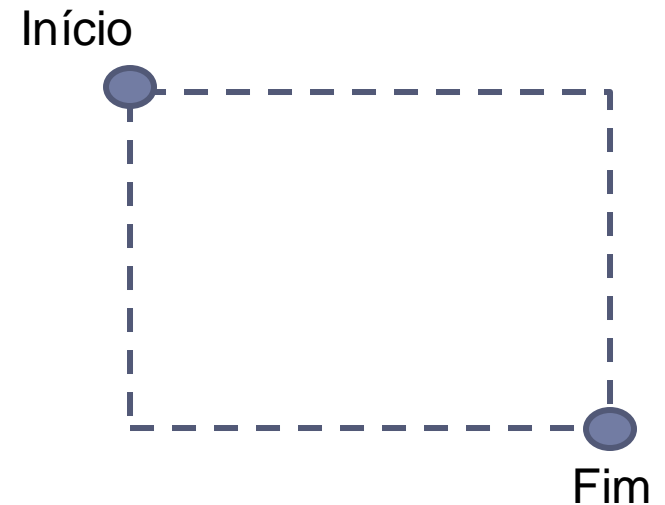


# Estrutura de Estruturas

---

## ► Ex: Representação de 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);
```



# Estrutura 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
```



# Exercício

---

- ▶ Considerando a `struct retangulo` definida no slide anterior, faça um programa que leia as coordenadas dos pontos que definem um retângulo e retorne a sua área.
- ▶ Ao final, teste o seu programa para as coordenadas:

1ª execução: (10,20) e (30,40)

2ª execução: (30,40) e (10,20)

