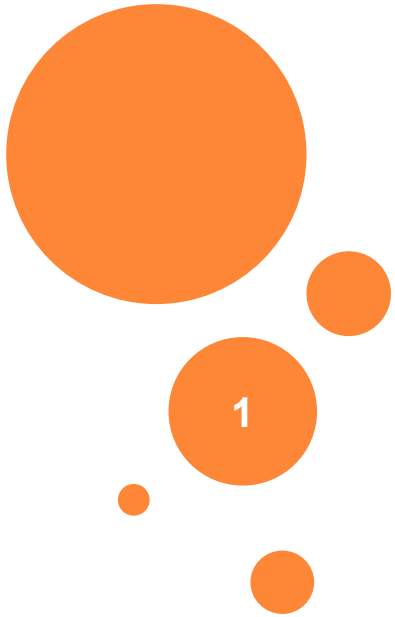


Linguagem C: Estruturas definidas pelo programador

Prof. Humberto Razente

Sala 1B144

VARIÁVEIS

- As variáveis vistas até agora podem ser classificados em duas categorias:
  - simples: definidas por tipos **int**, **float**, **double** e **char**;
  - compostas homogêneas (ou seja, do mesmo tipo): definidas por **vetor**

- No entanto, a linguagem C permite que se criem novas estruturas a partir dos tipos básicos
  - **struct**

ESTRUTURAS

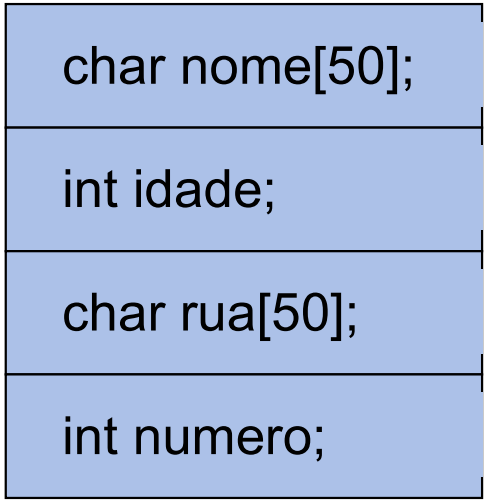
- Uma estrutura pode ser vista como um **novo tipo de dado**, que é formado por composição de variáveis de outros tipos
  - Pode ser declarada em qualquer escopo
  - Ela é declarada da seguinte forma:

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

ESTRUTURAS

- Uma estrutura pode ser vista como um agrupamento de dados (**registro**)
- Ex.: cadastro de pessoas
  - Todas essas informações são da mesma pessoa, logo podemos agrupá-las
  - Isso facilita também lidar com dados de outras pessoas no mesmo programa

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



cadastro

ESTRUTURAS - DECLARAÇÃO

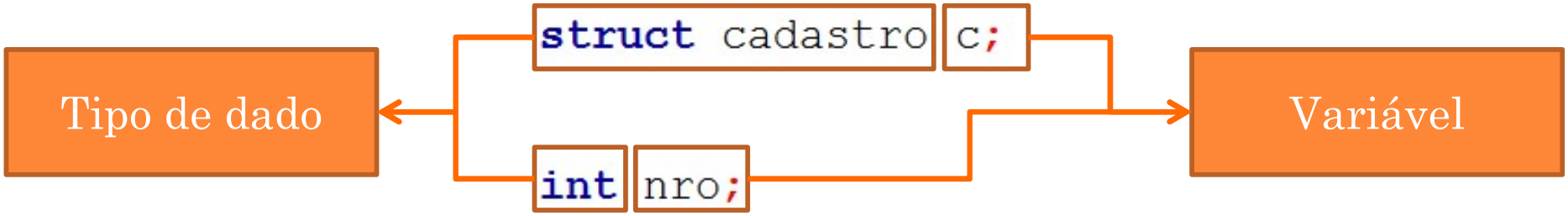
- Uma vez definida a estrutura, uma **variável** pode ser declarada de modo similar aos tipos já existente:

```
struct cadastro c;
```

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

ESTRUTURAS - DECLARAÇÃO

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



Exercício

- Declare uma estrutura capaz de armazenar o número da matrícula (inteiro) e 3 notas para um dado aluno

Exercício - Solução

- Possíveis soluções

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

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

```
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 pessoas diferentes:

```
char nome1[50], nome2[50], nome3[50], nome4[50];
int idade1, idade2, idade3, idade4;
char rua1[50], rua2[50], rua3[50], rua4[50]
int numero1, numero2, numero3, numero4;
```

ESTRUTURAS

- Utilizando uma estrutura, o mesmo pode ser feito da seguinte maneira:

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

//declarando 4 cadastros
struct cadastro c1, c2, c3, c4;
```

Acesso às variáveis

- Como é feito o acesso às variáveis da estrutura?
  - Cada variável da estrutura pode ser acessada com o operador ponto "."
  - Ex.:

```
//declarando a variável
struct cadastro c;

//acessando os seus campos
strcpy(c.nome,"João");
scanf("%d",&c.idade);
strcpy(c.rua,"Avenida 1");
c.numero = 1082;
```

Acesso às variáveis

- Como nos vetores, uma estrutura pode ser previamente iniciada:

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

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

Acesso às variáveis

- E se quiséssemos ler os valores das variáveis da estrutura do teclado?
  - Resposta: basta ler cada variável independentemente, respeitando seus tipos

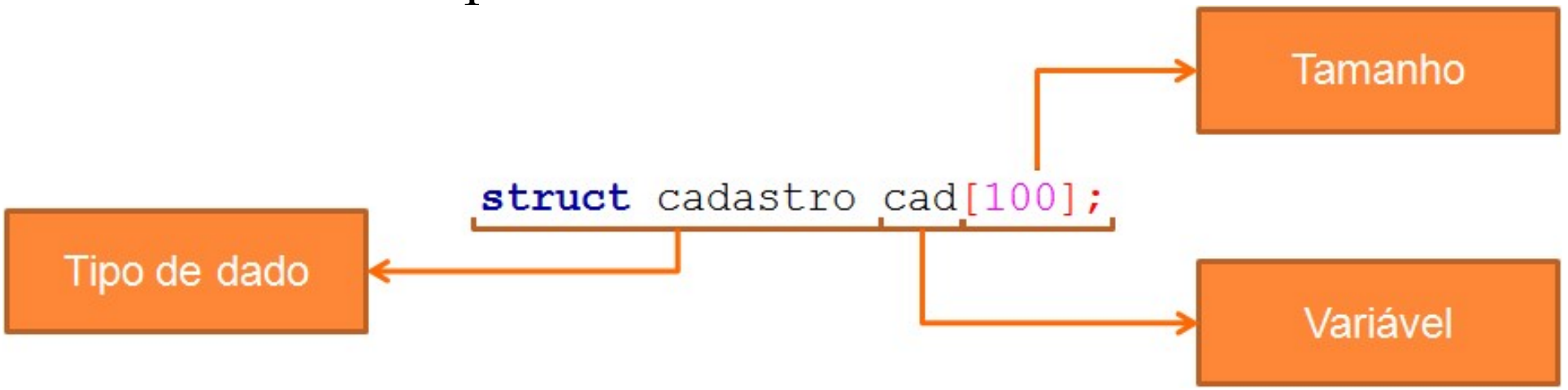
```
struct cadastro c;

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

Estruturas

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

ARRAY DE ESTRUTURAS

- SOLUÇÃO: criar um **vetor de estruturas**
- Sua declaração é similar a declaração de um vetor de um tipo básico



- Desse modo, declara-se um array de 100 posições, onde cada posição é do tipo **struct cadastro**

Array de estruturas

- Lembrando:

  - **struct**: define um conjunto de variáveis que podem ser de tipos diferentes

  - **vetor**: é um conjunto de elementos de mesmo tipo

ARRAY DE ESTRUTURAS

- Em um vetor de estruturas, o operador de ponto (.) vem depois dos colchetes **[ ]** do índice do **vetor**.

```
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 do aluno, as 3 notas e calcular a média de 10 alunos

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

EXERCÍCIO - SOLUÇÃO

- Utilizando a estrutura abaixo, faça um programa para ler o número do aluno, as 3 notas e calcular a média de 10 alunos

```
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
    }
}
```

ATRIBUIÇÃO ENTRE ESTRUTURAS

- Atribuições entre estruturas só podem ser feitas quando as estruturas são **AS MESMAS**, ou seja, quando as estruturas possuem o mesmo nome!

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

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

ATRIBUIÇÃO ENTRE ESTRUTURAS

- No caso da utilização de vetores, a atribuição entre diferentes elementos do vetor é válida

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

- Note que nesse caso, os tipos dos diferentes elementos do vetor são sempre IGUAIS

Atribuição entre estruturas

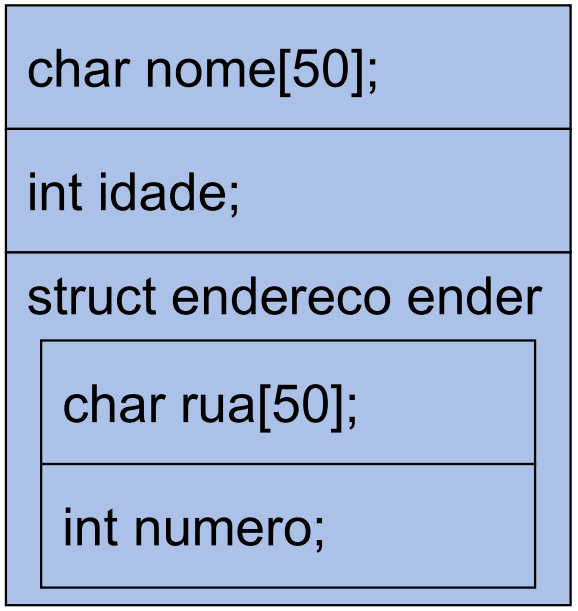
- E se a estrutura tiver um ponteiro?

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

int main() {
    struct pessoa_estatica { int cpf; char nome[20]; };

    struct pessoa_estatica p1 = { 123456789, "Maria" }, p2;
    p2 = p1;

    printf("\n\n P1 %d %s", p1.cpf, p1.nome);
    printf("\n\n P2 %d %s", p2.cpf, p2.nome);

    struct pessoa_dinamica { int cpf; char *nome; };

    struct pessoa_dinamica p3 = { 987654321, NULL }, p4;
    p3.nome = (char *) malloc(20);
    strcpy(p3.nome,"Ana");

    // cuidado: o endereço apontado por char*nome será copiado para p4
    // ou seja, ambos p3 e p4 apontarão para a mesma área alocada pelo malloc
    p4 = p3;

    printf("\n\n P3 %d %s", p3.cpf, p3.nome);
    printf("\n\n P4 %d %s", p4.cpf, p4.nome);

    return 0;
}
```

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;
};
```



cadastro

Estruturas de estruturas

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

```
struct cadastro c;

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

//atribuição
strcpy(c.nome,"João");
c.idade = 34;
strcpy(c.ender.rua,"Avenida 1");
c.ender.numero = 131;
```

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}};
```

COMANDO TYPEDEF

- A linguagem C permite que o programador defina os seus próprios tipos com base em outros tipos de dados existentes

- Para isso, utiliza-se o comando ***typedef***, cuja forma geral é:
  - **`typedef tipo_existente novo_nome;`**

COMANDO TYPEDEF

- Exemplo
  - Note que o comando **typedef** não cria um novo tipo chamado **inteiro**. Ele apenas cria um sinônimo (**inteiro**) para o tipo **int**

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

typedef int inteiro;

int main(){
    int x = 10;
    inteiro y = 20;
    y = y + x;
    printf("Soma = %d\n",y);

    return 0;
}
```

COMANDO TYPEDEF

- O **typedef** é muito utilizado para definir nomes mais simples para estrutura, evitando carregar a palavra **struct** sempre que referenciamos a estrutura

```
struct cadastro{
    char nome[300];
    int idade;
};
// redefinindo o tipo struct cadastro
typedef struct cadastro CadAlunos;

int main(){
    struct cadastro aluno1;
    CadAlunos aluno2;

    return 0;
}
```

STRUCT COMO PARÂMETRO DE FUNÇÕES

- Podemos passar uma struct por valor ou por referência

- Possibilidades

  - Passar por valor apenas um campo específico da struct

  - Passar por valor toda a struct

  - Passar por referência o endereço de uma struct de ou de um vetor de struct

STRUCT COMO PARÂMETRO DE FUNÇÕES

- Passar por valor apenas um campo específico da struct
  - Valem as mesmas regras vistas até o momento
  - Cada campo da struct é como uma variável independente. Ela pode, portanto, ser passada individualmente por ***valor*** ou por ***referência***

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

int quadrado (int n) { return n * n; }

int main() {

    struct ponto { int x; int y; } p1 = { 5, 10 };

    p1.x = quadrado(p1.x);
    p1.y = quadrado(p1.y);

    printf("%d %d", p1.x, p1.y);

    return 0;

}
```

STRUCT COMO PARÂMETRO DE FUNÇÕES

- Passar por parâmetro toda a struct

- Passagem por valor
  - Valem as mesmas regras vistas até o momento
  - A struct é tratada com uma variável qualquer e seu valor é copiado para dentro da função

- Passagem por referência
  - Valem as regras de uso do asterisco "*" e operador de endereço "&"
  - Usar o ***operador seta "->"***

PASSAGEM POR VALOR

- Passagem por valor

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

struct aluno {
    int numero;
    char nome[20], curso[20];
};

void imprime (struct aluno a) {
    printf("\n %d %s %s", a.numero, a.nome, a.curso);
}

int main() {
    struct aluno b = { 12345, "jose", "eng quim" };
    struct aluno c = { 12346, "maria", "eng quim" };

    imprime(b);
    imprime(c);

    return 0;
}
```

PONTEIRO PARA STRUCT

- Existem duas abordagens para acessar o conteúdo de um ponteiro para uma struct
- Abordagem 1
  - Devemos acessar o conteúdo do ponteiro para struct para somente depois acessar os seus campos e modificá-los.
- Abordagem 2
  - Podemos usar o ***operador seta "->"***
  - **`ponteiro->nome_campo`**

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

struct ponto q;
struct ponto *p;

p = &q;

(*p).x = 10;
p->y = 20;
```

Ponteiro para Vetor Dinâmico de struct

```c
int main()
{
    struct cadastro {
        char nome[50];
        int idade;
    };

    struct cadastro *cad = (struct cadastro *) malloc(sizeof(struct cadastro));
    strcpy(cad->nome,"Maria");
    cad->idade = 30;

    struct cadastro *vcad = (struct cadastro *) malloc(10 * sizeof(struct cadastro));

    strcpy(vcad[0].nome,"Maria");
    vcad[0].idade = 30;

    strcpy(vcad[1].nome,"Cecilia");
    vcad[1].idade = 10;

    strcpy(vcad[2].nome,"Ana");
    vcad[2].idade = 10;

    return 0;
}
```

Struct como parâmetro de Funções por Referência

Usando *

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

void atribui(struct ponto *p){
    (*p).x = 10;
    (*p).y = 20;
}

struct ponto p1;

atribui(&p1);
```

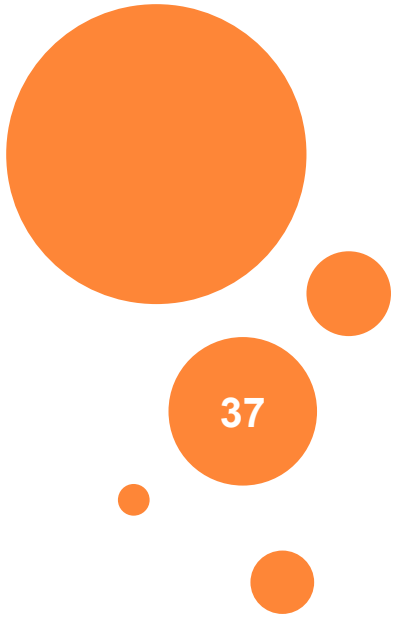
Usando ->

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

void atribui(struct ponto *p){
    p->x = 10;
    p->y = 20;
}

struct ponto p1;

atribui(&p1);
```

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
  - Aula 42: Typedef
  - Aula 50: Função – Struct como parâmetro

  - https://programacaodescomplicada.wordpress.com/indice/linguagem-c/



Linguagem C: Estruturas definidas pelo programador

Contém slides originais gentilmente disponibilizados pelo Prof. André R. Backes (UFU)