

Estruturas definidas pelo programador



Prof. Bruno Travençolo

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;
```

```
}
```

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

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 membro1;  
    tipo2 membro2;  
    ...  
    tipoN membroN;  
};
```

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];
```



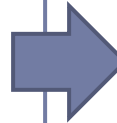
Todas essas informações são da mesma pessoa – podemos agrupá-las. Isso facilita também lidar com dados de outras pessoas no mesmo programa



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

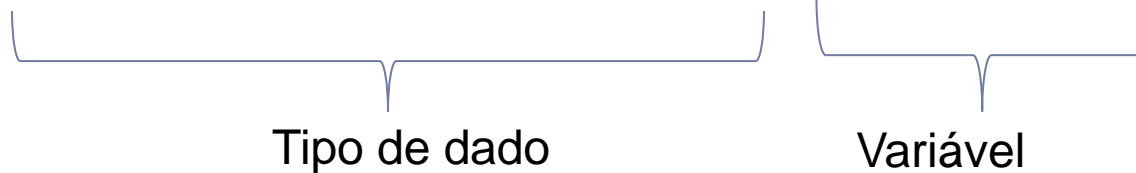
- ▶ Uma vez definida a estrutura, uma **variável** pode ser declarada de modo similar aos tipos já existente:
 - ▶ `struct dados_pacientes` paciente1;
- ▶ Obs: por ser um tipo definido pelo programador, usa-se a palavra **struct** antes do tipo da nova variável



Estruturas – declaração de variáveis

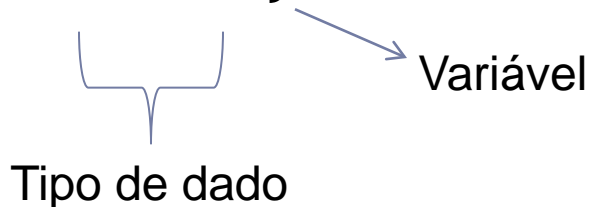
- ▶ Obs: por ser um tipo definido pelo programador, usa-se a palavra **struct** antes do tipo da nova variável

- ▶ `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 membros

- ▶ Como é feito o acesso aos membros da estrutura?
 - ▶ Cada membro 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 membros

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

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

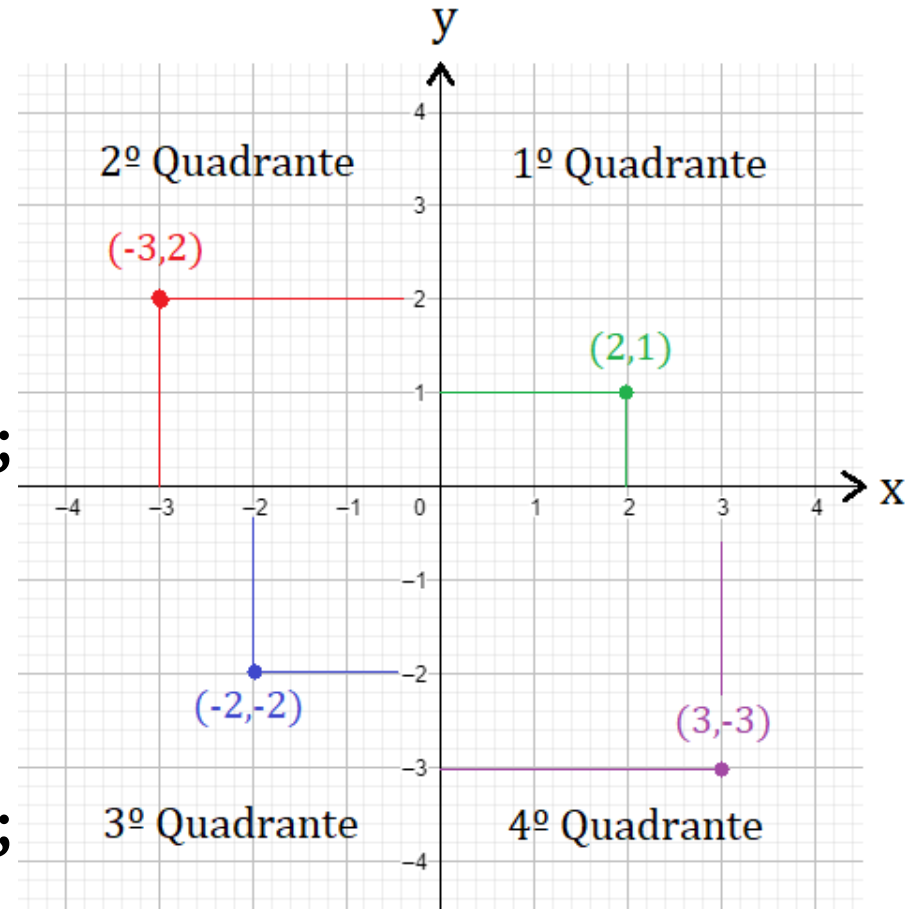


Acesso aos membros

```
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 membros

- ▶ E se quiséssemos ler os valores dos membros da estrutura utilizando o teclado?
 - ▶ Resposta: basta ler cada membro 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 membros

- ▶ Note que cada membro 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



Memória

```
struct dados_pacientes {  
    int idade; //4 bytes  
    char nome[10]; // 10 bytes  
    double peso; // 8 bytes  
    double altura; // 8 bytes  
    int estado_civil; //4 bytes  
    float grau_miopia[2]; //8 bytes  
};
```

4 + 10 + 8 + 8 + 4 + 8 = 42 bytes

sizeof(struct X) = 48

alignof(struct X) = 8

- ▶ Devido a alinhamento de memória, a memória ocupada pela struct não reflete necessariamente a soma dos membros. O alinhamento é um número inteiro não negativo potência de 2
- ▶ É feito um padding (complemento de tamanho) na memória

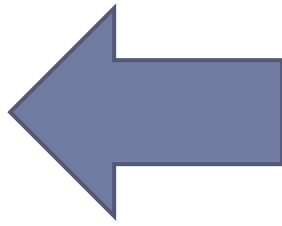


```
int main()
{
    struct dados_pacientes {
        int idade;
        char nome[10];
        double peso;
        double altura;
        int estado_civil;
        float grau_miopia[2];
    };

    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];`

Tipo de dado Variável Tamanho do vetor



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 membros 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

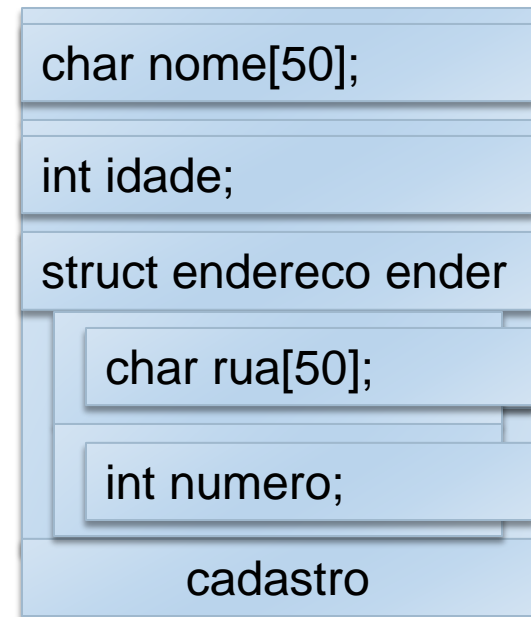
- ▶ No caso de estarmos trabalhando com arrays, a atribuição entre diferentes elementos do array é válida
 - ▶ Ex:
struct cadastro c[10];
c[1] = c[2]; //CORRETO
- ▶ Note que nesse caso, os tipos dos diferentes elementos do array são sempre IGUAIS.



Estruturas de estruturas

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

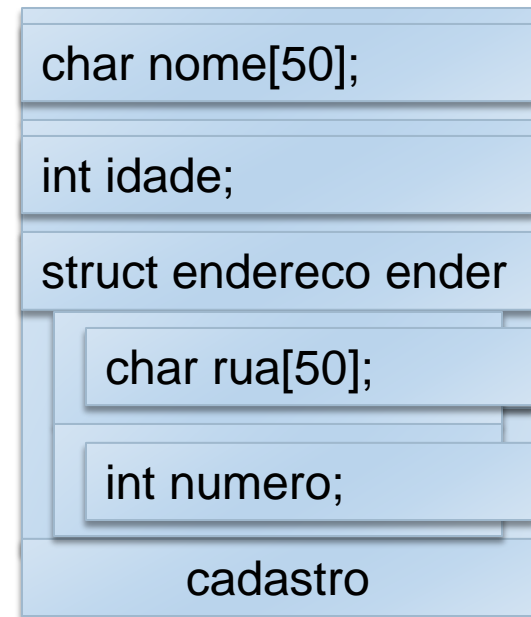
```
struct endereco{  
    char rua[50]  
    int numero;  
};  
struct cadastro{  
    char nome[50];  
    int idade;  
    struct endereco ender;  
};
```



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;
```

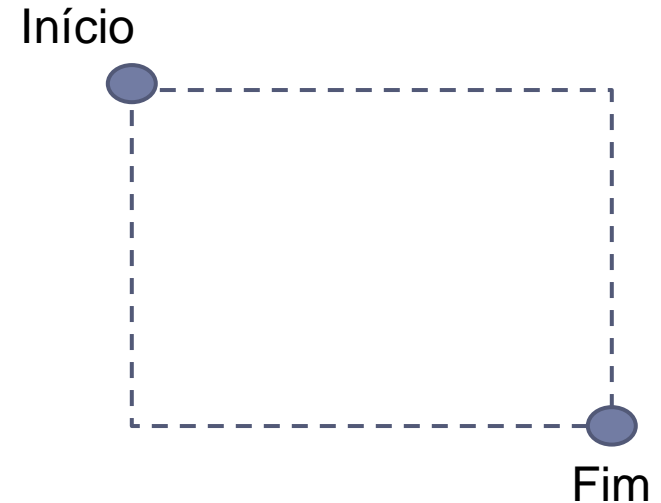
```
strcpy(c.ender.rua, "Avenida 1");
```

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



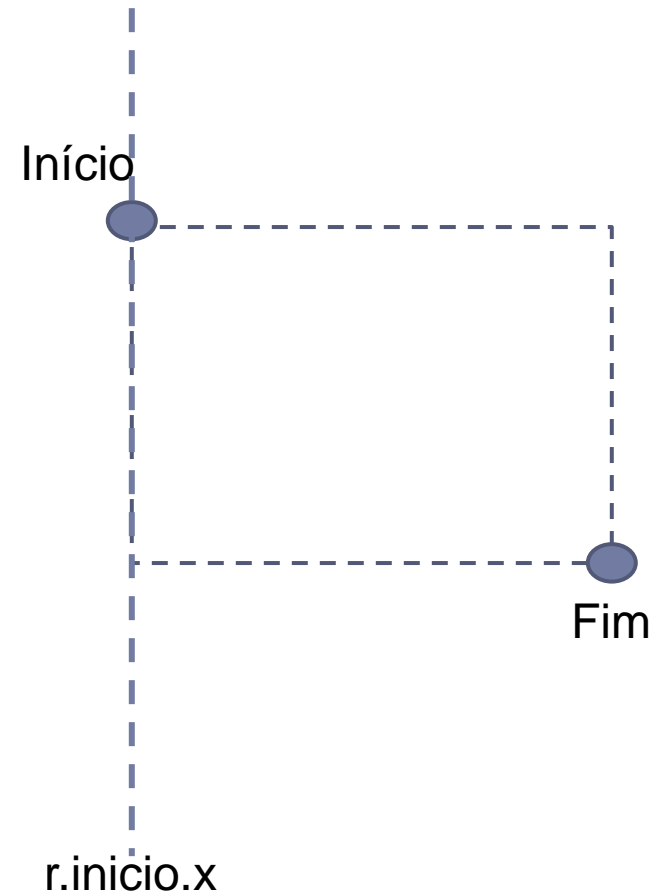
► Lendo um retângulo

```
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);
```



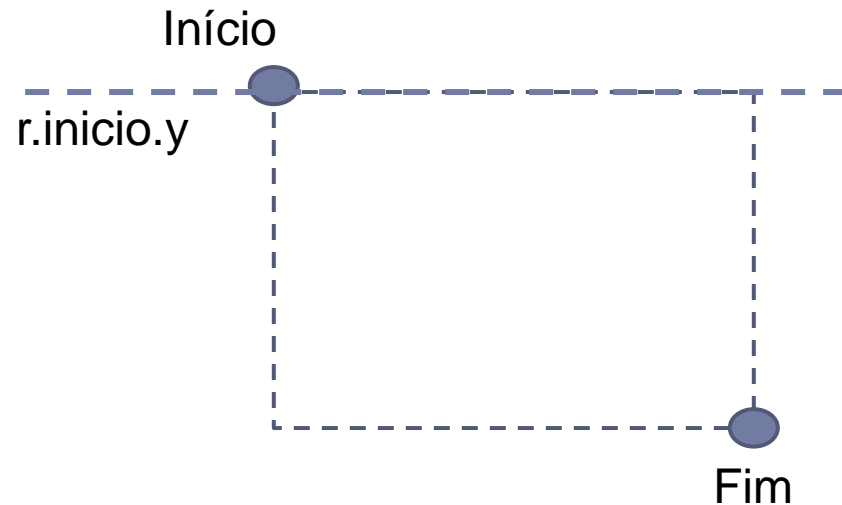
► Lendo um retângulo

```
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);
```



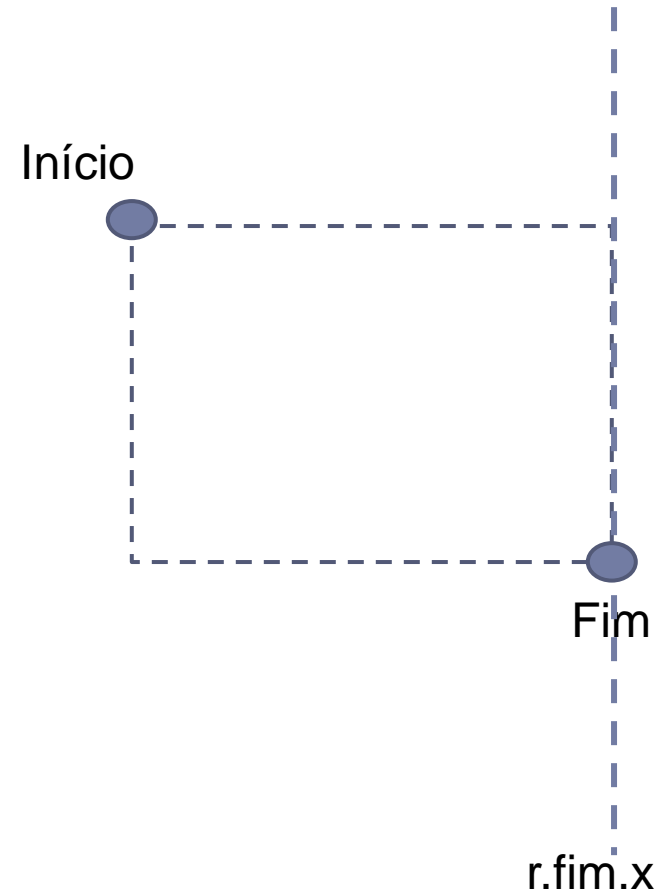
► Lendo um retângulo

```
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);
```



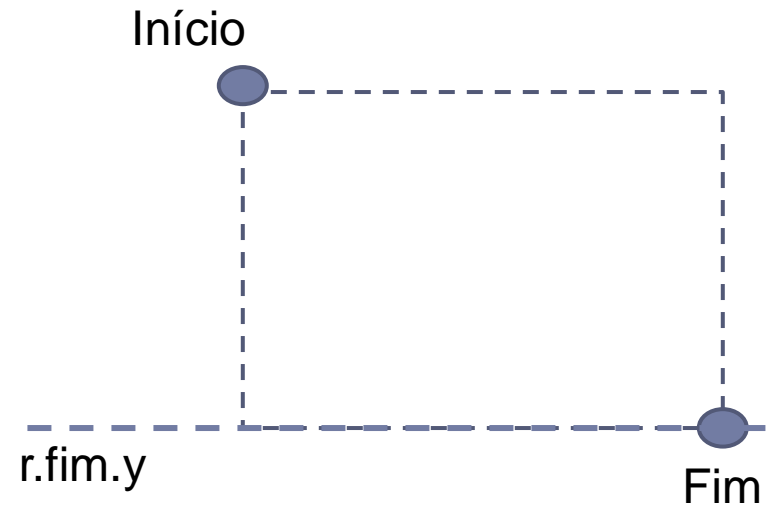
► Lendo um retângulo

```
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

