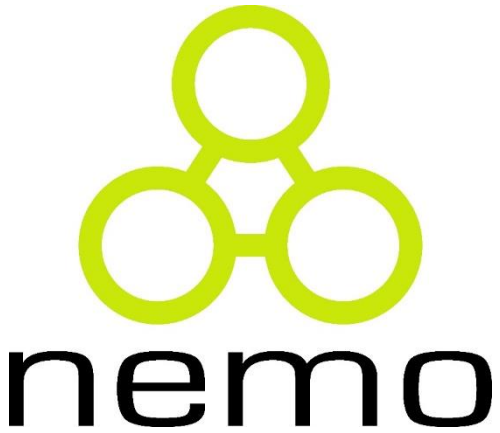


Programação II

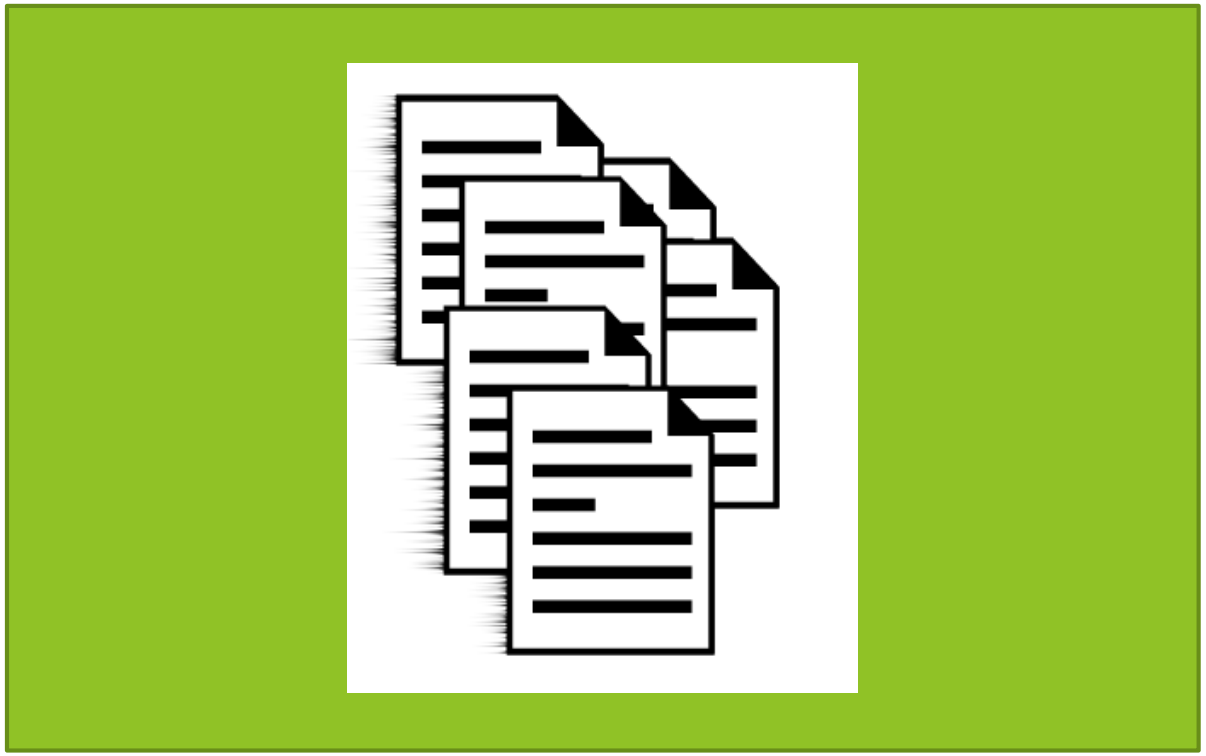
Jordana S. Salamon

jssalamon@inf.ufes.br

DEPARTAMENTO DE INFORMÁTICA

CENTRO TECNOLÓGICO

UNIVERSIDADE FEDERAL DO ESPÍRITO SANTO



Estruturas

nemo

Introdução

- As variáveis estudadas até o momento podem armazenar somente dados simples. Ex: int, float, char;
- Contudo, muitas vezes precisamos armazenar dados compostos, ou seja, várias variáveis que se referem ao mesmo conceito do mundo real.
- Exemplos:
  - Uma **pessoa** que possui nome, idade, peso e altura.
  - Um **aluno** que possui nome, número de matrícula e curso.
  - Um **livro** que possui título, gênero, autor(es), ISBN, etc.
  - Um **jogo** que possui nome, gênero, valor, empresa desenvolvedora.

Estruturas de Dados

- Relembre o exercício em que eram lidas informações de vários pacientes e era calculado o IMC para ver se os pacientes estavam no peso ideal ou acima do peso.
- Para representar um paciente, utilizaríamos as seguintes variáveis para armazenar as informações:
  - **char nome[30];**
  - **int idade;**
  - **float peso, altura;**

Estruturas de Dados

- Agora imagine se tivéssemos que armazenar vários pacientes na memória (ex:5) para depois calcular todos os IMCs de uma vez.
- Para representar um conjunto de pacientes, utilizaríamos as seguintes variáveis para armazenar as informações:
  - **char nome[5][30];**
  - **int idade[5];**
  - **float peso[5], altura[5];**
- O controle das variáveis nesse tipo de programa se torna difícil, uma vez que é necessário controlar vários vetores simultaneamente.

nemo

Estruturas de Dados

- Se pensarmos no mundo real, não temos conjuntos de informações separadas, elas fazem parte de um todo, que nesse exemplo é o Paciente.
- Desta forma seria melhor termos um “vetor de pacientes” ao invés de quatro vetores separados.
- Isso é possível utilizando as **estruturas de dados**.
- Para definir uma estrutura de dados utilizamos a palavra-chave **struct**.

Sintaxe - Estruturas de Dados

```
struct <nome> {
    <declaração de variáveis>
    ...
    <declaração de variáveis>
};

<declaração de variáveis> =
    <tipo> <nome>, <nome>, ...;
```

```
struct paciente {
    char nome[20];
    int idade;
    float altura, peso;
};

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

nemo

Declarando e Manipulando uma Estrutura

```
struct paciente{
    char nome[20];
    int idade;
    float altura, peso;
};

int main(){
    struct paciente p;
    strcpy(p.nome,"Joao");
    p.idade = 28;
    p.altura = 1.91;
    p.peso = 88.0;
}
```

nemo

Typedef

```c
struct paciente{
    char nome[20];
    int idade;
    float altura, peso;
};

typedef struct paciente Paciente;

int main(){
    Paciente p;
    strcpy(p.nome,"Joao");
    p.idade = 28;
    p.altura = 1.91;
    p.peso = 88.0;
}
```

nemo

Passando uma estrutura por parâmetro

```c
void imprimePaciente(Paciente pac){
    printf("Dados de Pessoa\n");
    printf("Nome = %s\n", pac.nome);
    printf("Idade = %d\n", pac.idade);
    printf("Altura = %.2f\n", pac.altura);
    printf("Peso = %.2f\n", pac.peso);
}

int main(){
    Paciente p;
    strcpy(p.nome,"Joao");
    p.idade = 28;
    p.altura = 1.91;
    p.peso = 88.0;
    imprimePaciente(p);
}
```

nemo

Retornando uma estrutura

```
Paciente criaPaciente(){
    Paciente p;
    printf("Nome paciente: ");
    scanf("%s", p.nome);
    printf("Idade paciente: ");
    scanf("%d", &p.idade);
    printf("Altura paciente: ");
    scanf("%f", &p.altura);
    printf("Peso paciente: ");
    scanf("%f", &p.peso);
    return p;
}

int main(){
    Paciente p;
    p = criaPaciente();
}
```

nemo

Vetores de Estruturas

```c
int main(){
    int i;
    Paciente vetor[10];
    for(i=0;i<10;i++){
        printf("Nome paciente: ");
        scanf("%s", vetor[i].nome);
        printf("Idade paciente: ");
        scanf("%d", &vetor[i].idade);
        printf("Altura paciente: ");
        scanf("%f", &vetor[i].altura);
        printf("Peso paciente: ");
        scanf("%f", &vetor[i].peso);
    }
    for(i=0;i<10;i++){
        imprimePaciente(vetor[i]);
    }
}
```

nemo

Vetores dentro das Estruturas

```c
struct paciente{
    char nome[20];
    int idade;
    float altura, peso;
    char telefones[3][15];
};
typedef struct paciente Paciente;
int main(){
    int i;
    Paciente vetor[2];
    for(i=0;i<2;i++){
        printf("Telefone 1: ");
        scanf("%s", vetor[i].telefones[0]);
        printf("Telefone 2: ");
        scanf("%s", vetor[i].telefones[1]);
        printf("Telefone 3: ");
        scanf("%s", vetor[i].telefones[2]);
    }
}
```

nemo

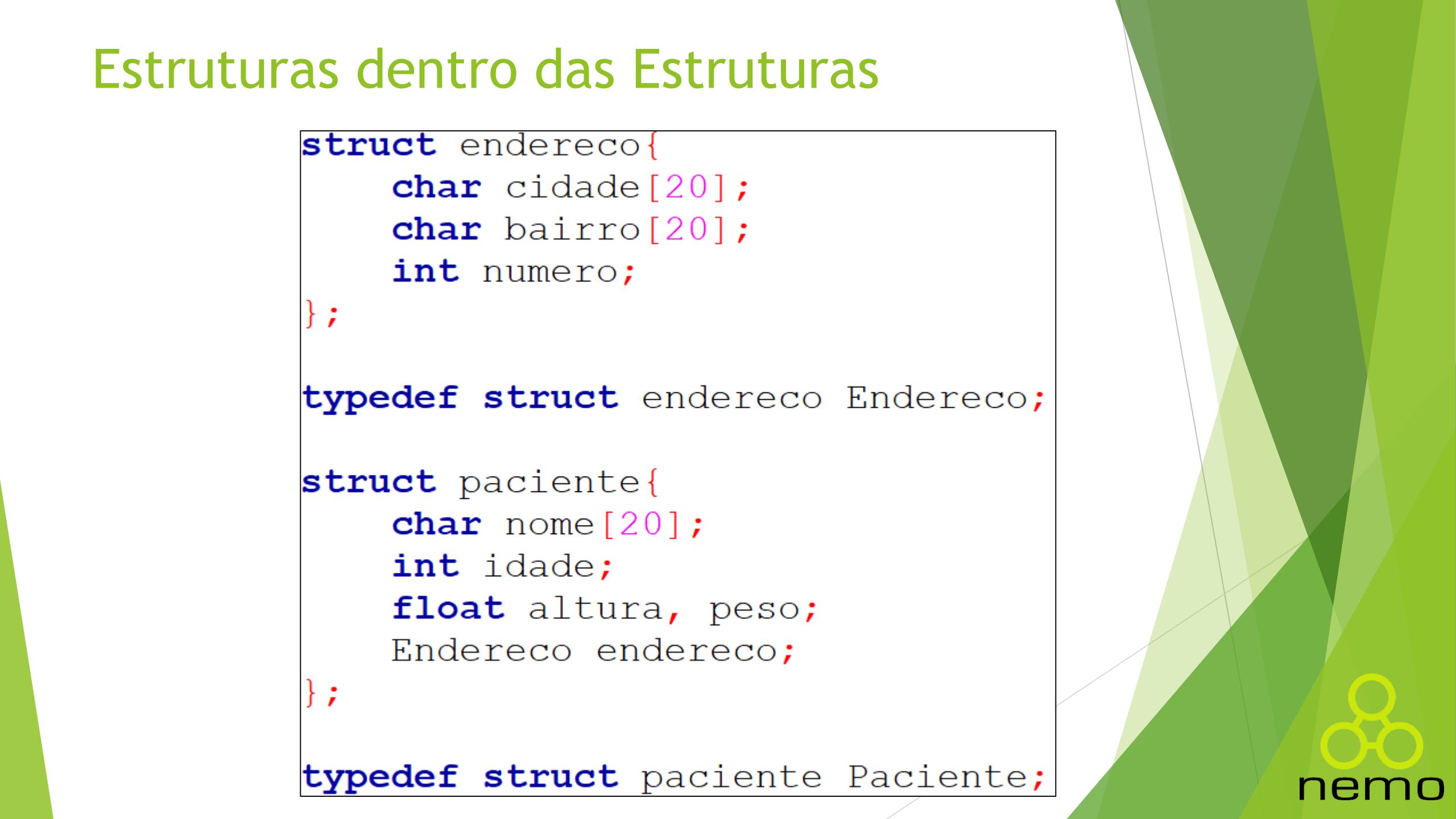
Estruturas dentro das Estruturas

```
struct endereco{
    char cidade[20];
    char bairro[20];
    int numero;
};

typedef struct endereco Endereco;

struct paciente{
    char nome[20];
    int idade;
    float altura, peso;
    Endereco endereco;
};

typedef struct paciente Paciente;
```



Estruturas dentro das Estruturas

- Podemos representar um círculo como

```
struct circulo {
  float x, y;  //centro do círculo
  float r;  //raio
}
```

- Como já temos o tipo Ponto definido:

```
 struct circulo {
  Ponto p;
  float r;
}
```

```
typedef struct circulo Circulo;
```

Estruturas dentro das Estruturas

- Para implementar uma função que determinar se um dado ponto está dentro de um círculo
  - Podemos usar a função da distância, visto que usamos o tipo ponto na definição do círculo

```
int interior (Circulo* c, Ponto* p)
{
   float d = distancia (&c->p, p);
   return (d<c->r);
}
```

nemo

Ponteiro para Estruturas

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

- Do mesmo modo que podemos declarar variáveis do tipo estrutura:
  - struct ponto p;
- Podemos também declarar variáveis do tipo ponteiro para estrutura:
  - struct ponto *pp;

Ponteiro para Estruturas

- Se a variável pp armazena o endereço (&) de uma estrutura, podemos acesso os campos dessa estrutura indiretamente, por meio de seu ponteiro:
  - struct ponto p;
  - struct ponto *pp;
  - pp = &p;
  - (*pp).x = 12.0;
  - (*pp).y = 5.0;

- Obs: Nesse caso, os parênteses são indispensáveis, pois o operador “conteúdo de” (*) tem precedência menor do que operador de acesso (.). Ou seja, se utilizarmos simplesmente *pp.x estamos acessando o conteúdo do valor de x e não de pp.

Operador ->

- O acesso a campos de estruturas é tão comum em programas C que a linguagem oferece outro operador de acesso, que permite acessar campos a partir do ponteiro da estrutura.
- Esse operador é composto por um traço seguido de um sinal de maior, formando uma seta (->).
- Portanto podemos reescrever a atribuição anterior da seguinte maneira:
  - struct ponto p;
  - struct ponto *pp;
  - pp = &p;
  - pp->x = 12.0;
  - pp->y = 5.0;

Passagem de Estruturas por cópia

- Assim como em tipos primitivos da linguagem (int, float), variáveis do tipo estrutura também podem ser passadas para serem processadas em funções. Ex:

```
void imprime(struct ponto p) {
   printf("O ponto fornecido foi: (%.2f, %.2f)\n", p.x, p.y);
}
```

- Contudo, da forma como está escrita no código acima, a função recebe uma estrutura inteira como parâmetro.
- Portanto, faz-se uma cópia de toda a estrutura e a função acessa essa cópia. Ou seja, se a estrutura ocupa 40 bytes, ao passa-la para a função, teremos 80 bytes.

Problemas da Passagem de Estruturas por cópia

- Primeiro, como em toda passagem por cópia, a função não tem como alterar os valores dos elementos da estrutura original (na função *imprime* isso realmente não é necessário, mas seria numa função de leitura).
- Segundo, visto que copiar uma estrutura inteira pode ser uma operação custosa (principalmente se a estrutura for muito grande), isso afeta a eficiência do programa.
  - tanto pelo tempo gasto em realizar a cópia;
  - quanto pelo gasto desnecessário de memória.
- Qual a solução para esse problema?

Passagem de Estruturas por referência

- A ideia de utilizarmos ponteiro para estrutura em passagens para funções, não diz respeito apenas na necessidade de alterarmos os valores do elemento dentro da função;
- Mas é muito importante pois não é necessário copiar toda a estrutura.
- Geralmente um ponteiro ocupa em geral 4 bytes, enquanto uma estrutura pode ser definida com um tamanho arbitrariamente grande.
- Assim, uma segunda (e mais adequada) alternativa para escrever a função *imprime* é:

```
void imprime(struct ponto* pp) {
    printf("O ponto fornecido foi: (%.2f, %.2f)\n", pp->x, pp->y);
}
```

nemo

Exemplo de ponteiro para estruturas

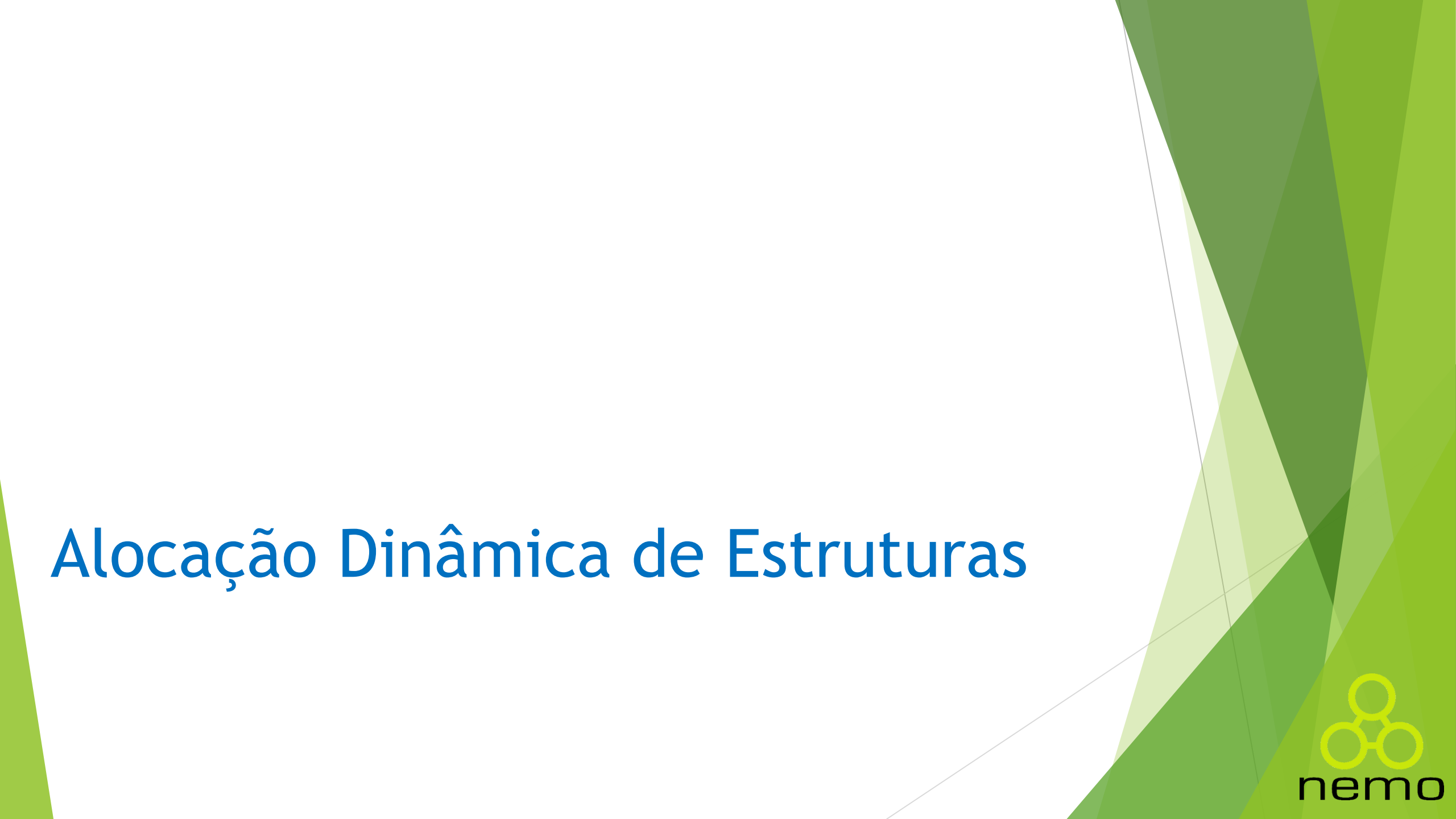
```
void imprime(struct ponto* pp) {
    printf("O ponto fornecido foi: (%.2f, %.2f)\n", pp->x, pp->y);
}

void captura(struct ponto* pp) {
     printf("Digite as coordenadas do ponto (x, y): ");
     scanf("%f %f", &pp->x, &pp->y);
}

int main (){
     struct ponto p;
     captura (&p);
     imprime (&p);
     return 0;
     }
```

nemo

Alocação Dinâmica de Estruturas



Malloc

- Assim como os vetores e matrizes, as estruturas podem ser alocadas dinamicamente. Ex:

```
struct ponto* pp;
pp = (struct ponto*) malloc (sizeof (struct ponto));
```

- Nesse fragmento de código, alocamos, de modo dinâmico, uma única estrutura e armazenamos o endereço da área alocada em p.
- O tamanho do espaço de memória alocado de forma dinâmica é dado pelo operador **sizeof** aplicado sobre a estrutura (sizeof (struct ponto)).
- Ou seja, corresponde ao espaço necessário para armazenar uma estrutura.
- A função *malloc* retorna o endereço do espaço alocado, que é então convertido para o tipo ponteiro da estrutura ponto (struct ponto *).

nemo

Free

- Para liberar um espaço de memória alocado dinamicamente, usamos a função **free** (stdlib.h).
- Essa função recebe como parâmetro a referencia do ponteiro da memória a ser liberada.
- Assim, para liberar a estrutura ponto p, fazemos:
  - free (p);
- Só podemos passar para a função **free** um endereço de memória que tenha sido alocado dinamicamente.
- Devemos lembrar ainda que não podemos acessar o espaço da memória depois de liberado.

Definição de novos tipos

- Podemos criar nomes de tipos em C
  - typedef float Real;
  - Real pode ser usado como mnemônico de float

```
typedef unsigned char UChar;
typedef int* PInt;
typedef float Vetor[4];
```

- Podemos declarar as seguintes variáveis:

```
Vetor v;
...
v[0] = 3;
...
```

nemo

Exercício

Struct Matriz (m por n)

- Definir uma estrutura para armazenar uma matriz e seus valores de linhas e colunas; definir operações básicas para manipulação de elementos (i,j), consulta de linhas e colunas, alocação e liberação de memória.

That’s all Folks!