

Uso avançado de ponteiros

Graziela Araújo

Faculdade de Computação
Universidade Federal de Mato Grosso do Sul

Algoritmos e Programação II

Conteúdo da aula

- 1 Introdução
- 2 Alocação dinâmica de memória
- 3 Ponteiros para ponteiros
- 4 Exercícios

- ▶ já vimos formas importantes de uso de ponteiros como parâmetros de funções simulando passagem por referência e também como elementos da linguagem C que podem acessar indiretamente outros compartimentos de memória, seja uma variável, uma célula de um vetor ou de uma matriz ou ainda um campo de um registro, usando, inclusive uma aritmética específica para tanto
- ▶ nesta aula veremos outros usos para ponteiros: como auxiliares na alocação dinâmica de espaços de memória e como ponteiros para outros ponteiros

- ▶ já vimos formas importantes de uso de ponteiros como parâmetros de funções simulando passagem por referência e também como elementos da linguagem C que podem acessar indiretamente outros compartimentos de memória, seja uma variável, uma célula de um vetor ou de uma matriz ou ainda um campo de um registro, usando, inclusive uma aritmética específica para tanto
- ▶ nesta aula veremos outros usos para ponteiros: como auxiliares na alocação dinâmica de espaços de memória e como ponteiros para outros ponteiros

Alocação dinâmica de memória

- ▶ estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo
- ▶ para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente
- ▶ a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa
- ▶ dessa forma, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa
- ▶ a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas

Alocação dinâmica de memória

- ▶ estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo
- ▶ para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente
- ▶ a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa
- ▶ dessa forma, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa
- ▶ a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas

Alocação dinâmica de memória

- ▶ estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo
- ▶ para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente
- ▶ a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa
- ▶ dessa forma, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa
- ▶ a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas

Alocação dinâmica de memória

- ▶ estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo
- ▶ para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente
- ▶ a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa
- ▶ dessa forma, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa
- ▶ a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas

Alocação dinâmica de memória

- ▶ estruturas de armazenamento de informações na memória principal da linguagem C têm, em geral, tamanho fixo
- ▶ para alterar a capacidade de armazenamento de uma estrutura de tamanho fixo, é necessário alterar seu tamanho no arquivo-fonte e compilar esse programa novamente
- ▶ a linguagem C permite **alocação dinâmica de memória**, que é a habilidade de reservar espaços na memória principal durante a execução de um programa
- ▶ dessa forma, podemos projetar estruturas de armazenamento que crescem ou diminuem quando necessário durante a execução do programa
- ▶ a alocação dinâmica é usada em geral com variáveis compostas homogêneas e heterogêneas

Alocação dinâmica de memória

- ▶ para que os dados de entrada sejam armazenados em vetores e matrizes com dimensão(ões) adequadas, é necessário saber antes essa(s) dimensão(ões)
- ▶ um limitante que indica a capacidade máxima de armazenamento dessas estruturas deve ser informado e, muitas vezes, o total de espaço alocado na memória não é usado durante a execução do programa
- ▶ **alocação estática de memória:** antes da execução, o compilador reserva na memória um número fixo de compartimentos correspondentes à declaração (espaço é fixo e não pode ser alterado durante a execução do programa)
- ▶ memória é um recurso limitado e programas maiores que armazenam muitas informações em memória têm de usá-la de maneira eficiente, economizando compartimentos sempre que possível

Alocação dinâmica de memória

- ▶ para que os dados de entrada sejam armazenados em vetores e matrizes com dimensão(ões) adequadas, é necessário saber antes essa(s) dimensão(ões)
- ▶ um limitante que indica a capacidade máxima de armazenamento dessas estruturas deve ser informado e, muitas vezes, o total de espaço alocado na memória não é usado durante a execução do programa
- ▶ **alocação estática de memória:** antes da execução, o compilador reserva na memória um número fixo de compartimentos correspondentes à declaração (espaço é fixo e não pode ser alterado durante a execução do programa)
- ▶ memória é um recurso limitado e programas maiores que armazenam muitas informações em memória têm de usá-la de maneira eficiente, economizando compartimentos sempre que possível

Alocação dinâmica de memória

- ▶ para que os dados de entrada sejam armazenados em vetores e matrizes com dimensão(ões) adequadas, é necessário saber antes essa(s) dimensão(ões)
- ▶ um limitante que indica a capacidade máxima de armazenamento dessas estruturas deve ser informado e, muitas vezes, o total de espaço alocado na memória não é usado durante a execução do programa
- ▶ **alocação estática de memória:** antes da execução, o compilador reserva na memória um número fixo de compartimentos correspondentes à declaração (espaço é fixo e não pode ser alterado durante a execução do programa)
- ▶ memória é um recurso limitado e programas maiores que armazenam muitas informações em memória têm de usá-la de maneira eficiente, economizando compartimentos sempre que possível

Alocação dinâmica de memória

- ▶ para que os dados de entrada sejam armazenados em vetores e matrizes com dimensão(ões) adequadas, é necessário saber antes essa(s) dimensão(ões)
- ▶ um limitante que indica a capacidade máxima de armazenamento dessas estruturas deve ser informado e, muitas vezes, o total de espaço alocado na memória não é usado durante a execução do programa
- ▶ **alocação estática de memória**: antes da execução, o compilador reserva na memória um número fixo de compartimentos correspondentes à declaração (espaço é fixo e não pode ser alterado durante a execução do programa)
- ▶ memória é um recurso limitado e programas maiores que armazenam muitas informações em memória têm de usá-la de maneira eficiente, economizando compartimentos sempre que possível

Alocação dinâmica de memória

- ▶ é possível alocar dinamicamente um ou mais blocos de memória na linguagem C
- ▶ **alocação dinâmica de memória** significa que um programa solicita ao sistema computacional, durante a sua execução, blocos da memória principal que estejam disponíveis para uso

Alocação dinâmica de memória

- ▶ é possível alocar dinamicamente um ou mais blocos de memória na linguagem C
- ▶ **alocação dinâmica de memória** significa que um programa solicita ao sistema computacional, durante a sua execução, blocos da memória principal que estejam disponíveis para uso

Alocação dinâmica de memória

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

int main(void)
{
    int i, n, *vetor, *pt;

    scanf("%d", &n);
    vetor = (int *) malloc(n * sizeof(int));
    if (vetor != NULL) {
        for (i = 0; i < n; i++)
            scanf("%d", (vetor + i));
        for (pt = vetor; pt < (vetor + n); pt++)
            printf("%d ", *pt);
        printf("\n");
        for (i = n - 1; i >= 0; i--)
            printf("%d ", vetor[i]);
        printf("\n");
        free(vetor); /*desalocando o vetor*/
    }
    else
        printf("Impossível alocar espaço\n");
    return 0;
}
```


Alocação dinâmica de memória

- ▶ a função `malloc`, declarada no arquivo-cabeçalho `stdlib.h`, tem a seguinte interface:

```
void *malloc(size_t tamanho)
```

- ▶ a função `malloc` reserva uma certa quantidade específica de memória e devolve um ponteiro do tipo `void`
- ▶ no programa anterior, reservamos n compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`
- ▶ o operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão
- ▶ esse número é multiplicado por n , o número de compartimentos que desejamos para armazenar números inteiros

Alocação dinâmica de memória

- ▶ a função `malloc`, declarada no arquivo-cabeçalho `stdlib.h`, tem a seguinte interface:

```
void *malloc(size_t tamanho)
```

- ▶ a função `malloc` reserva uma certa quantidade específica de memória e devolve um ponteiro do tipo `void`
- ▶ no programa anterior, reservamos n compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`
- ▶ o operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão
- ▶ esse número é multiplicado por n , o número de compartimentos que desejamos para armazenar números inteiros

Alocação dinâmica de memória

- ▶ a função `malloc`, declarada no arquivo-cabeçalho `stdlib.h`, tem a seguinte interface:

```
void *malloc(size_t tamanho)
```

- ▶ a função `malloc` reserva uma certa quantidade específica de memória e devolve um ponteiro do tipo `void`
- ▶ no programa anterior, reservamos n compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`
- ▶ o operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão
- ▶ esse número é multiplicado por n , o número de compartimentos que desejamos para armazenar números inteiros

Alocação dinâmica de memória

- ▶ a função `malloc`, declarada no arquivo-cabeçalho `stdlib.h`, tem a seguinte interface:

```
void *malloc(size_t tamanho)
```

- ▶ a função `malloc` reserva uma certa quantidade específica de memória e devolve um ponteiro do tipo `void`
- ▶ no programa anterior, reservamos n compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`
- ▶ o operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão
- ▶ esse número é multiplicado por n , o número de compartimentos que desejamos para armazenar números inteiros

Alocação dinâmica de memória

- ▶ a função `malloc`, declarada no arquivo-cabeçalho `stdlib.h`, tem a seguinte interface:

```
void *malloc(size_t tamanho)
```

- ▶ a função `malloc` reserva uma certa quantidade específica de memória e devolve um ponteiro do tipo `void`
- ▶ no programa anterior, reservamos n compartimentos contínuos que podem armazenar números inteiros, o que se reflete na expressão `n * sizeof(int)`
- ▶ o operador unário `sizeof` devolve como resultado o número de bytes dados pelo seu operando, que pode ser um tipo de dados ou uma expressão
- ▶ esse número é multiplicado por n , o número de compartimentos que desejamos para armazenar números inteiros

Alocação dinâmica de memória

- ▶ o endereço da primeira posição de memória onde encontram-se esses compartimentos é devolvido pela função `malloc`
- ▶ essa função devolve um ponteiro do tipo `void` e, por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um ponteiro para um número inteiro
- ▶ depois, esse endereço é armazenado em `vetor`, que foi declarado como um ponteiro para números inteiros
- ▶ a partir daí, podemos usar `vetor` da forma como preferirmos, como um ponteiro ou como um vetor

Alocação dinâmica de memória

- ▶ o endereço da primeira posição de memória onde encontram-se esses compartimentos é devolvido pela função `malloc`
- ▶ essa função devolve um ponteiro do tipo `void` e, por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um ponteiro para um número inteiro
- ▶ depois, esse endereço é armazenado em `vetor`, que foi declarado como um ponteiro para números inteiros
- ▶ a partir daí, podemos usar `vetor` da forma como preferirmos, como um ponteiro ou como um vetor

Alocação dinâmica de memória

- ▶ o endereço da primeira posição de memória onde encontram-se esses compartimentos é devolvido pela função `malloc`
- ▶ essa função devolve um ponteiro do tipo `void` e, por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um ponteiro para um número inteiro
- ▶ depois, esse endereço é armazenado em `vetor`, que foi declarado como um ponteiro para números inteiros
- ▶ a partir daí, podemos usar `vetor` da forma como preferirmos, como um ponteiro ou como um vetor

Alocação dinâmica de memória

- ▶ o endereço da primeira posição de memória onde encontram-se esses compartimentos é devolvido pela função `malloc`
- ▶ essa função devolve um ponteiro do tipo `void` e, por isso, usamos o modificador de tipo `(int *)` para indicar que o endereço devolvido é de fato um ponteiro para um número inteiro
- ▶ depois, esse endereço é armazenado em `vetor`, que foi declarado como um ponteiro para números inteiros
- ▶ a partir daí, podemos usar `vetor` da forma como preferirmos, como um ponteiro ou como um vetor

Alocação dinâmica de memória

- ▶ funções `malloc`, `calloc` e `realloc` solicitam blocos de memória de um espaço de armazenamento conhecido também como **heap** ou ainda **lista de espaços disponíveis**
- ▶ a chamada freqüente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um ponteiro nulo
- ▶ pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário

```
p = malloc(...);  
q = malloc(...);  
p = q;
```

- ▶ a função `free` é usada para ajudar a resolver o problema de geração de lixo na memória durante a execução de programas

Alocação dinâmica de memória

- ▶ funções `malloc`, `calloc` e `realloc` solicitam blocos de memória de um espaço de armazenamento conhecido também como **heap** ou ainda **lista de espaços disponíveis**
- ▶ a chamada freqüente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um ponteiro nulo
- ▶ pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário

```
p = malloc(...);  
q = malloc(...);  
p = q;
```

- ▶ a função `free` é usada para ajudar a resolver o problema de geração de lixo na memória durante a execução de programas

Alocação dinâmica de memória

- ▶ funções `malloc`, `calloc` e `realloc` solicitam blocos de memória de um espaço de armazenamento conhecido também como **heap** ou ainda **lista de espaços disponíveis**
- ▶ a chamada freqüente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um ponteiro nulo
- ▶ pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário

```
p = malloc(...);  
q = malloc(...);  
p = q;
```

- ▶ a função `free` é usada para ajudar a resolver o problema de geração de lixo na memória durante a execução de programas

Alocação dinâmica de memória

- ▶ funções `malloc`, `calloc` e `realloc` solicitam blocos de memória de um espaço de armazenamento conhecido também como **heap** ou ainda **lista de espaços disponíveis**
- ▶ a chamada freqüente dessas funções pode exaurir o *heap* do sistema, fazendo com que essas funções devolvam um ponteiro nulo
- ▶ pior ainda, um programa pode alocar blocos de memória e perdê-los de algum modo, gastando espaço desnecessário

```
p = malloc(...);  
q = malloc(...);  
p = q;
```

- ▶ a função `free` é usada para ajudar a resolver o problema de geração de lixo na memória durante a execução de programas

Alocação dinâmica de memória

- ▶ a função **free** tem a seguinte interface:

```
void free(void *pt)
```

- ▶ devemos lhe passar à função **free** um ponteiro para um bloco de memória que não mais necessitamos:

```
p = malloc(...);  
q = malloc(...);  
free(p);  
p = q;
```

- ▶ uma chamada à função **free** devolve o bloco de memória apontado por *p* para o *heap*
- ▶ o argumento da função **free** deve ser um ponteiro que foi previamente devolvido por uma função de alocação de memória

Alocação dinâmica de memória

- ▶ a função **free** tem a seguinte interface:

```
void free(void *pt)
```

- ▶ devemos lhe passar à função **free** um ponteiro para um bloco de memória que não mais necessitamos:

```
p = malloc(...);  
q = malloc(...);  
free(p);  
p = q;
```

- ▶ uma chamada à função **free** devolve o bloco de memória apontado por *p* para o *heap*
- ▶ o argumento da função **free** deve ser um ponteiro que foi previamente devolvido por uma função de alocação de memória

Alocação dinâmica de memória

- ▶ a função **free** tem a seguinte interface:

```
void free(void *pt)
```

- ▶ devemos lhe passar à função **free** um ponteiro para um bloco de memória que não mais necessitamos:

```
p = malloc(...);  
q = malloc(...);  
free(p);  
p = q;
```

- ▶ uma chamada à função **free** devolve o bloco de memória apontado por *p* para o *heap*
- ▶ o argumento da função **free** deve ser um ponteiro que foi previamente devolvido por uma função de alocação de memória

Alocação dinâmica de memória

- ▶ a função **free** tem a seguinte interface:

```
void free(void *pt)
```

- ▶ devemos lhe passar à função **free** um ponteiro para um bloco de memória que não mais necessitamos:

```
p = malloc(...);  
q = malloc(...);  
free(p);  
p = q;
```

- ▶ uma chamada à função **free** devolve o bloco de memória apontado por *p* para o *heap*
- ▶ o argumento da função **free** deve ser um ponteiro que foi previamente devolvido por uma função de alocação de memória

Alocação dinâmica de memória

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i, j, m, n, **matriz, **pt;
    scanf("%d%d", &m, &n);
    matriz = (int **) malloc(m * sizeof(int *));
    if (matriz == NULL)
        return 0;
    for (i = 0; i < m; i++) {
        matriz[i] = (int *) malloc(n * sizeof(int));
        if (matriz[i] == NULL)
            return 0; }
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &matriz[i][j]);
    pt = matriz;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("%d ", *(pt+i)+j));
        printf("\n"); }
    return 0;
}
```

Alocação dinâmica de memória

```
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    int i, j, m, n, **matriz, **pt;
    scanf("%d%d", &m, &n);
    matriz = (int **) malloc(m * sizeof(int *));
    if (matriz == NULL)
        return 0;
    for (pt = matriz, i = 0; i < m; i++, pt++) {
        *pt = (int *) malloc(n * sizeof(int));
        if (*pt == NULL)
            return 0; }
    for (i = 0; i < m; i++)
        for (j = 0; j < n; j++)
            scanf("%d", &matriz[i][j]);
    pt = matriz;
    for (i = 0; i < m; i++) {
        for (j = 0; j < n; j++)
            printf("%d ", *(pt+i)+j));
        printf("\n"); }
    return 0;
}
```

Ponteiros para ponteiros

- ▶ uma variável que é um ponteiro para um compartimento de memória que contém um outro ponteiro é chamada de **ponteiro para um ponteiro**
- ▶ podemos estender indireções com a multiplicidade que desejarmos como, por exemplo, indireção dupla, indireção tripla, indireção quádrupla
- ▶ ponteiros para ponteiros têm diversas aplicações na linguagem C, especialmente no uso de matrizes

Ponteiros para ponteiros

- ▶ uma variável que é um ponteiro para um compartimento de memória que contém um outro ponteiro é chamada de **ponteiro para um ponteiro**
- ▶ podemos estender indireções com a multiplicidade que desejarmos como, por exemplo, indireção dupla, indireção tripla, indireção quádrupla
- ▶ ponteiros para ponteiros têm diversas aplicações na linguagem C, especialmente no uso de matrizes

Ponteiros para ponteiros

- ▶ uma variável que é um ponteiro para um compartimento de memória que contém um outro ponteiro é chamada de **ponteiro para um ponteiro**
- ▶ podemos estender indireções com a multiplicidade que desejarmos como, por exemplo, indireção dupla, indireção tripla, indireção quádrupla
- ▶ ponteiros para ponteiros têm diversas aplicações na linguagem C, especialmente no uso de matrizes

Ponteiros para ponteiros

```
#include <stdio.h>

int main(void)
{
    int x, y, *pt1, *pt2, **ptpt1, **ptpt2;

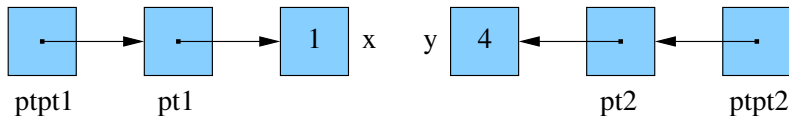
    x = 1;
    y = 4;
    printf("x=%d y=%d\n", x, y);

    pt1 = &x;
    pt2 = &y;
    printf("*pt1=%d *pt2=%d\n", *pt1, *pt2);

    ptpt1 = &pt1;
    ptpt2 = &pt2;
    printf("**ptpt1=%d **ptpt2=%d\n", **ptpt1, **ptpt2);

    return 0;
}
```

Ponteiros para ponteiros



- 16.1 Dados dois vetores x e y , ambos com n elementos, $1 \leq n \leq 100$, determinar o produto escalar desses vetores. Use alocação dinâmica de memória.

- 16.2** Dizemos que uma seqüência de n elementos, com n par, é **balanceada** se as seguintes somas são todas iguais:
- a soma do maior elemento com o menor elemento;*
 - a soma do segundo maior elemento com o segundo menor elemento;*
 - a soma do terceiro maior elemento com o terceiro menor elemento;*
 - e assim por diante ...*

Exemplo:

2 12 3 6 16 15 é uma seqüência balanceada, pois $16 + 2 = 15 + 3 = 12 + 6$.

Dados n (n par e $0 \leq n \leq 100$) e uma seqüência de n números inteiros, verificar se essa seqüência é balanceada. Use alocação dinâmica de memória.

- 16.3 Dada uma cadeia de caracteres com no máximo 100 caracteres, contar a quantidade de letras minúsculas, letras maiúsculas, dígitos, espaços e símbolos de pontuação que essa cadeia possui. Use alocação dinâmica de memória.
- 16.4 Dada uma matriz de números reais A com m linhas e n colunas, $1 \leq m, n \leq 100$, e um vetor de números reais v com n elementos, determinar o produto de A por v . Use alocação dinâmica de memória.

- 16.3 Dada uma cadeia de caracteres com no máximo 100 caracteres, contar a quantidade de letras minúsculas, letras maiúsculas, dígitos, espaços e símbolos de pontuação que essa cadeia possui. Use alocação dinâmica de memória.
- 16.4 Dada uma matriz de números reais A com m linhas e n colunas, $1 \leq m, n \leq 100$, e um vetor de números reais v com n elementos, determinar o produto de A por v . Use alocação dinâmica de memória.

- 16.5 Dizemos que uma matriz quadrada de números inteiros distintos é um **quadrado mágico** se a soma dos elementos de cada linha, a soma dos elementos de cada coluna e a soma dos elementos da diagonal principal e secundária são todas iguais.

Exemplo:

A matriz

$$\begin{pmatrix} 8 & 0 & 7 \\ 4 & 5 & 6 \\ 3 & 10 & 2 \end{pmatrix}$$

é um quadrado mágico.

Dada uma matriz quadrada de números inteiros $A_{n \times n}$, com $1 \leq n \leq 100$, verificar se A é um quadrado mágico. Use alocação dinâmica de memória.