Programação de Computadores

Raoni F. S. Teixeira

Aula 14 - Alocação Dinâmica de Matrizes

1 Introdução

Na aula anterior, estudamos ponteiros e alocação dinâmica de vetores unidimensionais. Nesta aula, expandiremos esses conceitos para trabalhar com estruturas mais complexas: matrizes e vetores multidimensionais alocados dinamicamente.

Vetores multidimensionais (especialmente matrizes bidimensionais) são fundamentais para representar:

1. dados tabulares e planilhas,
2. imagens digitais (onde cada pixel é um elemento),
3. sistemas de equações lineares, e
4. grafos representados por matrizes de adjacência.

A alocação dinâmica de matrizes oferece flexibilidade para determinar as dimensões durante a execução do programa, algo impossível com vetores estáticos.

2 Matrizes Estáticas vs. Dinâmicas

2.1 Matrizes Estáticas

Em C, podemos declarar matrizes estáticas cujas dimensões devem ser conhecidas em tempo de compilação:

```
int matriz[3][4];  // matriz 3x4 de inteiros

// Inicializacao
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 4; j++) {
```

```
            matriz[i][j] = i * 4 + j;
        }
}
```

Esta abordagem tem limitações importantes:

- As dimensões devem ser constantes conhecidas em tempo de compilação.
- A matriz é alocada na pilha, que tem tamanho limitado.
- Não há flexibilidade para dimensões determinadas pelo usuário.

2.2 Necessidade de Alocação Dinâmica

Considere um programa que processa imagens. O tamanho da imagem só é conhecido quando o arquivo é aberto. Precisamos alocar a matriz dinamicamente:

```
int linhas, colunas;
printf("Dimensoes da imagem: ");
scanf("%d %d", &linhas, &colunas);

// Precisamos alocar dinamicamente uma matriz
// de tamanho linhas x colunas
```

3 Alocação de Matrizes: Método do Vetor de Ponteiros

A forma mais comum de alocar matrizes dinamicamente é criar um vetor de ponteiros, onde cada ponteiro aponta para uma linha da matriz.

3.1 Estrutura Conceitual

Uma matriz `m` de dimensões $n \times p$ é implementada como:

- Um vetor de n ponteiros (as linhas).
- Cada ponteiro aponta para um vetor de p elementos (as colunas).

A Figura 1 ilustra esta estrutura.

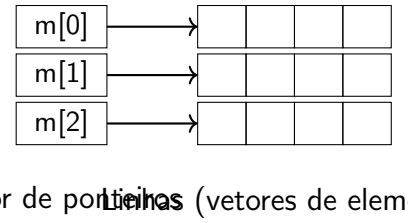

Figura 1: Estrutura de uma matriz alocada como vetor de ponteiros.

3.2 Implementação Passo a Passo

Para alocar uma matriz de `n` linhas e `p` colunas:

Passo 1: Alocar o vetor de ponteiros (um ponteiro para cada linha):

```
int **matriz;
int n = 3, p = 4;

// Aloca vetor de n ponteiros
matriz = malloc(n * sizeof(int*));
if (matriz == NULL) {
    printf("Erro de alocacao\n");
    exit(1);
}
```

Note que `matriz` é do tipo `int**` (ponteiro para ponteiro para inteiro). Estamos alocando espaço para n ponteiros do tipo `int*`.

Passo 2: Alocar cada linha individualmente:

```
for (int i = 0; i < n; i++) {
    matriz[i] = malloc(p * sizeof(int));
    if (matriz[i] == NULL) {
        printf("Erro de alocacao na linha %d\n", i);
        exit(1);
    }
}
```

Agora temos uma matriz completa que pode ser acessada usando `matriz[i][j]`.

3.3 Exemplo Completo

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

int main() {
    int **matriz;
    int n, p;
```

```
    printf("Numero␣de␣linhas:␣");
    scanf("%d", &n);
    printf("Numero␣de␣colunas:␣");
    scanf("%d", &p);

    // Aloca vetor de ponteiros
    matriz = malloc(n * sizeof(int*));
    if (matriz == NULL) {
        printf("Erro␣de␣alocacao\n");
        return 1;
    }

    // Aloca cada linha
    for (int i = 0; i < n; i++) {
        matriz[i] = malloc(p * sizeof(int));
        if (matriz[i] == NULL) {
            printf("Erro␣de␣alocacao\n");
            return 1;
        }
    }

    // Preenche a matriz
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            matriz[i][j] = i * p + j;
        }
    }

    // Imprime a matriz
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < p; j++) {
            printf("%3d␣", matriz[i][j]);
        }
        printf("\n");
    }

    // Libera memoria (importante!)
    for (int i = 0; i < n; i++) {
        free(matriz[i]);
    }
    free(matriz);

    return 0;
}
```

3.4 Liberação de Memória

A liberação deve ser feita na ordem inversa da alocação:

```
// Primeiro libera cada linha
for (int i = 0; i < n; i++) {
    free(matriz[i]);
}

// Depois libera o vetor de ponteiros
free(matriz);
```

Regra importante: Sempre libere a memória na ordem inversa da alocação. Liberar o vetor de ponteiros antes das linhas causaria perda das referências às linhas, resultando em vazamento de memória.

4 Alocação Contígua de Matrizes

Uma alternativa ao método de vetor de ponteiros é alocar toda a matriz como um único bloco contíguo de memória. Este método tem vantagens de desempenho em algumas situações.

4.1 Conceito

Uma matriz $n \times p$ pode ser armazenada como um vetor unidimensional de $n \times p$ elementos. O elemento da linha i e coluna j está na posição $i \times p + j$ do vetor.

4.2 Implementação

```
int *matriz_contigua;
int n = 3, p = 4;

// Aloca n*p elementos em um unico bloco
matriz_contigua = malloc(n * p * sizeof(int));
if (matriz_contigua == NULL) {
    printf("Erro de alocacao\n");
    exit(1);
}

// Acesso ao elemento [i][j]: matriz_contigua[i*p + j]
for (int i = 0; i < n; i++) {
    for (int j = 0; j < p; j++) {
        matriz_contigua[i * p + j] = i * p + j;
    }
}
```

```
// Liberacao simples
free(matriz_contigua);
```

4.3 Vantagens e Desvantagens

Vantagens da alocação contígua:

- Melhor localidade de cache (elementos consecutivos na memória).
- Uma única chamada a `malloc` (mais eficiente).
- Liberação mais simples (um único `free`).

Desvantagens:

- Sintaxe menos intuitiva: `matriz[i*p + j]` ao invés de `matriz[i][j]`.
- Não permite linhas de tamanhos diferentes (matrizes irregulares).

4.4 Função Auxiliar para Alocação Contígua

Podemos criar uma função que encapsula a alocação e retorna um ponteiro duplo para permitir o acesso com `[i][j]`:

```
int** aloca_matriz_contigua(int n, int p) {
    int **matriz = malloc(n * sizeof(int*));
    if (matriz == NULL) return NULL;

    matriz[0] = malloc(n * p * sizeof(int));
    if (matriz[0] == NULL) {
        free(matriz);
        return NULL;
    }

    // Ajusta os ponteiros das linhas
    for (int i = 1; i < n; i++) {
        matriz[i] = matriz[0] + i * p;
    }

    return matriz;
}

void libera_matriz_contigua(int **matriz) {
    free(matriz[0]);  // Libera o bloco de dados
    free(matriz);     // Libera o vetor de ponteiros
}
```

5 Matrizes de Estruturas

Podemos alocar matrizes de qualquer tipo, incluindo estruturas definidas pelo usuário.

5.1 Exemplo: Matriz de Pixels

```
typedef struct {
    unsigned char r, g, b;  // RGB
} Pixel;

Pixel **imagem;
int altura = 480, largura = 640;

// Aloca matriz de pixels
imagem = malloc(altura * sizeof(Pixel*));
for (int i = 0; i < altura; i++) {
    imagem[i] = malloc(largura * sizeof(Pixel));
}

// Define um pixel vermelho em (100, 200)
imagem[100][200].r = 255;
imagem[100][200].g = 0;
imagem[100][200].b = 0;

// Libera
for (int i = 0; i < altura; i++) {
    free(imagem[i]);
}
free(imagem);
```

6 Função `calloc`

A função `calloc` (contiguous allocation) é uma alternativa a `malloc` que:

- Aloca memória para um vetor de elementos.
- Inicializa todos os bytes com zero.

6.1 Sintaxe

```
void* calloc(size_t num_elementos, size_t tamanho_elemento);
```

6.2 Exemplo

```
// Aloca vetor de 100 inteiros, todos inicializados com 0
int *vetor = calloc(100, sizeof(int));

// Equivalente a:
int *vetor = malloc(100 * sizeof(int));
for (int i = 0; i < 100; i++) {
    vetor[i] = 0;
}
```

6.3 Uso com Matrizes

```
int **matriz;
int n = 5, p = 10;

matriz = calloc(n, sizeof(int*));
for (int i = 0; i < n; i++) {
    matriz[i] = calloc(p, sizeof(int));
}

// Todos os elementos ja estao zerados
// matriz[i][j] == 0 para todo i, j
```

7 Função `realloc`

A função `realloc` (reallocation) redimensiona um bloco de memória previamente alocado, preservando o conteúdo existente.

7.1 Sintaxe

```
void* realloc(void *ponteiro, size_t novo_tamanho);
```

7.2 Comportamento

A função `realloc` pode:

1. Expandir o bloco existente se houver espaço adjacente disponível.
2. Alocar um novo bloco, copiar os dados, e liberar o bloco antigo.
3. Retornar NULL se não houver memória suficiente (o bloco original permanece intacto).

7.3 Exemplo: Vetor Dinâmico Expansível

```
int *vetor;
int tamanho = 5;
int capacidade = 5;

vetor = malloc(capacidade * sizeof(int));

// Preenche o vetor
for (int i = 0; i < tamanho; i++) {
    vetor[i] = i;
}

// Precisa adicionar mais elementos
tamanho = 10;
if (tamanho > capacidade) {
    capacidade = tamanho;
    int *temp = realloc(vetor, capacidade * sizeof(int));
    if (temp == NULL) {
        printf("Erro␣ao␣redimensionar\n");
        free(vetor);
        exit(1);
    }
    vetor = temp;
}

// Continua usando o vetor expandido
for (int i = 5; i < 10; i++) {
    vetor[i] = i;
}

free(vetor);
```

Regra importante: Sempre atribua o resultado de `realloc` a uma variável temporária e verifique se não é NULL antes de sobrescrever o ponteiro original. Se `realloc` falhar, o ponteiro original ainda aponta para memória válida que deve ser liberada.

7.4 Casos Especiais

```
// realloc com ponteiro NULL equivale a malloc
int *p = realloc(NULL, 100 * sizeof(int));
// Equivalente a: int *p = malloc(100 * sizeof(int));

// realloc com tamanho 0 equivale a free
realloc(p, 0);
// Equivalente a: free(p);
```

8 Vetores de Strings

Uma string em C é um vetor de caracteres terminado por '\0'. Um vetor de strings é, portanto, uma matriz de caracteres, ou seja, um char**.

8.1 Alocação de Vetor de Strings

```
char **nomes;
int num_nomes = 3;

// Aloca vetor de ponteiros para strings
nomes = malloc(num_nomes * sizeof(char*));

// Aloca cada string individualmente
nomes[0] = malloc(20 * sizeof(char));  // 20 caracteres
nomes[1] = malloc(20 * sizeof(char));
nomes[2] = malloc(20 * sizeof(char));

// Preenche as strings
strcpy(nomes[0], "Alice");
strcpy(nomes[1], "Bob");
strcpy(nomes[2], "Carlos");

// Imprime
for (int i = 0; i < num_nomes; i++) {
    printf("%s\n", nomes[i]);
}

// Libera
for (int i = 0; i < num_nomes; i++) {
    free(nomes[i]);
}
free(nomes);
```

8.2 Leitura Dinâmica de Strings

Podemos ler strings de tamanho arbitrário alocando memória dinamicamente:

```
#include <string.h>

char **ler_nomes(int n) {
    char **nomes = malloc(n * sizeof(char*));
    char buffer[100];

    for (int i = 0; i < n; i++) {
        printf("Nome␣%d:␣", i+1);
```

```
        scanf("%99s", buffer);

        // Aloca apenas o espaco necessario
        nomes[i] = malloc((strlen(buffer) + 1) * sizeof(char));
        strcpy(nomes[i], buffer);
    }

    return nomes;
}
```

9 Matrizes Tridimensionais

Para aplicações como processamento de vídeo ou simulações 3D, precisamos de matrizes tridimensionais.

9.1 Alocação

```
int ***cubo;
int d1 = 10, d2 = 20, d3 = 30;

// Aloca a primeira dimensao
cubo = malloc(d1 * sizeof(int**));

// Aloca a segunda dimensao
for (int i = 0; i < d1; i++) {
    cubo[i] = malloc(d2 * sizeof(int*));

    // Aloca a terceira dimensao
    for (int j = 0; j < d2; j++) {
        cubo[i][j] = malloc(d3 * sizeof(int));
    }
}

// Acesso: cubo[i][j][k]
cubo[5][10][15] = 42;

// Liberacao (ordem inversa)
for (int i = 0; i < d1; i++) {
    for (int j = 0; j < d2; j++) {
        free(cubo[i][j]);
    }
    free(cubo[i]);
}
free(cubo);
```

10 Boas Práticas e Armadilhas Comuns

10.1 Verificação de Erros

Sempre verifique o retorno de `malloc`, `calloc` e `realloc`:

```
int *p = malloc(100 * sizeof(int));
if (p == NULL) {
    fprintf(stderr, "Erro: memoria insuficiente\n");
    exit(EXIT_FAILURE);
}
```

10.2 Liberação Adequada

Para estruturas complexas, libere na ordem inversa da alocação:

```
// Alocacao: matriz -> linhas
// Liberacao: linhas -> matriz

for (int i = 0; i < n; i++) {
    free(matriz[i]);  // Primeiro as linhas
}
free(matriz);  // Depois a estrutura principal
```

10.3 Ponteiros Pendentes

Após liberar memória, o ponteiro ainda contém o endereço antigo (ponteiro pendente). É boa prática atribuir NULL:

```
free(p);
p = NULL;  // Previne uso acidental
```

10.4 Dupla Liberação

Nunca chame `free` duas vezes no mesmo ponteiro:

```
free(p);
free(p);  // ERRO! Comportamento indefinido
```

10.5 Vazamento de Memória em Loops

Cuidado ao alocar dentro de loops:

```c
// ERRADO - vazamento de memoria
for (int i = 0; i < 1000; i++) {
    int *p = malloc(100 * sizeof(int));
    // ... usa p ...
    // Esqueceu de liberar!
}

// CORRETO
for (int i = 0; i < 1000; i++) {
    int *p = malloc(100 * sizeof(int));
    // ... usa p ...
    free(p);
}
```

11 Resumo

Nesta aula, aprendemos sobre alocação dinâmica de estruturas multidimensionais:

- **Matrizes dinâmicas** podem ser alocadas como vetor de ponteiros (permite `[i][j]`) ou como bloco contíguo (mais eficiente).
- `calloc` aloca e inicializa com zero.
- `realloc` redimensiona blocos de memória preservando o conteúdo.
- **Vetores de strings** são matrizes de caracteres (`char**`).
- **Liberação** deve ser feita na ordem inversa da alocação.

11.1 Funções de Alocação

Função	Uso
`malloc(n)`	Aloca n bytes não inicializados
`calloc(n, tam)`	Aloca n elementos de tamanho `tam`, inicializados com 0
`realloc(p, n)`	Redimensiona bloco apontado por `p` para n bytes
`free(p)`	Libera memória apontada por `p`

11.2 Checklist de Boas Práticas

1. Sempre verificar se a alocação foi bem-sucedida (ponteiro $\neq$ NULL).
2. Usar `sizeof` para calcular tamanhos, nunca valores fixos.

3. Liberar toda memória alocada (para cada `malloc`, um `free`).
4. Liberar na ordem inversa da alocação em estruturas aninhadas.
5. Não usar ponteiros após liberá-los.
6. Atribuir NULL a ponteiros após `free` quando apropriado.
7. Evitar vazamentos de memória, especialmente em loops e funções.