Prática: Gerenciamento de Memória Dinâmica em C - Alocação de Matrizes e Análise de Uso de RAM

Disciplina: Programação em C

1. Introdução

1.1 Objetivos

- Compreender os conceitos fundamentais de alocação dinâmica de memória em C
- Implementar e gerenciar matrizes dinâmicas
- Analisar o impacto do uso de memória dinâmica no sistema
- Identificar e evitar vazamentos de memória (memory leaks)
- Praticar boas práticas de gerenciamento de memória

1.2 Pré-requisitos

- Conhecimento básico da sintaxe da linguagem C
- Familiaridade com ponteiros em C
- Acesso a um ambiente de desenvolvimento C no Windows
- Conhecimento básico do Gerenciador de Tarefas do Windows

2. Conceitos Fundamentais

2.1 Memória em C

Em C, existem três tipos principais de alocação de memória:

1. **Alocação Estática**
    - Ocorre em tempo de compilação
    - Tamanho fixo e imutável
    - Exemplo: `int array[100];`
2. **Alocação Automática (Stack)**
    - Ocorre em tempo de execução
    - Gerenciada automaticamente
    - Exemplo: variáveis locais de função
3. **Alocação Dinâmica (Heap)**
    - Ocorre em tempo de execução
    - Gerenciada manualmente pelo programador
    - Exemplo: `malloc()`, `calloc()`, `realloc()`

2.2 Funções de Alocação Dinâmica

Alocação Lazy (Preguiçosa)

- Quando `malloc()` é chamado, o sistema operacional não aloca imediatamente a memória física
- Apenas reserva um espaço no espaço de endereçamento virtual do processo
- A memória física real só é alocada quando o programa acessa essa memória pela primeira vez
- Isso é uma otimização do sistema operacional para:
    - Evitar alocar memória física que pode nunca ser usada
    - Permitir melhor gerenciamento da memória física
    - Reduzir o overhead de alocação inicial
- Em matrizes, este efeito é ainda mais significativo devido ao maior volume de memória envolvido

malloc()

```
void* malloc(size_t size);
```

- Aloca um bloco de memória do tamanho especificado
- Retorna um ponteiro para o início do bloco
- Retorna NULL se a alocação falhar
- Não inicializa a memória alocada

free()

```
void free(void* ptr);
```

- Libera a memória alocada dinamicamente
- Deve ser chamada para cada bloco alocado
- Não faz nada se ptr for NULL
- Não deve ser chamada duas vezes para o mesmo ponteiro

3. Alocação de Matrizes Dinâmicas

3.1 Matriz 2D - Método 1 (Array de Arrays)

```
int **matriz = (int **)malloc(linhas * sizeof(int *));
for(int i = 0; i < linhas; i++) {
    matriz[i] = (int *)malloc(colunas * sizeof(int));
}
```

3.2 Matriz 2D - Método 2 (Array Contíguo)

```c
int *matriz = (int *)malloc(linhas * colunas * sizeof(int));
// Acesso: matriz[i * colunas + j]
```

3.3 Desalocação de Matrizes

```c
// Método 1
for(int i = 0; i < linhas; i++) {
    free(matriz[i]);
}
free(matriz);

// Método 2
free(matriz);
```

4. Boas Práticas

1. **Sempre verifique o retorno das funções de alocação**

   ```c
   if (ptr == NULL) {
       // Tratar erro
   }
   ```

2. **Libere a memória na ordem correta**
   - Primeiro libere as linhas
   - Depois libere o array de ponteiros

3. **Use sizeof() para calcular tamanhos**

   ```c
   int *ptr = malloc(n * sizeof(int));
   ```

4. **Defina ponteiros como NULL após liberar**

   ```c
   free(ptr);
   ptr = NULL;
   ```

5. **Evite vazamentos de memória**
   - Mantenha controle de todas as alocações
   - Libere a memória quando não for mais necessária
   - Use ferramentas como Valgrind para detectar vazamentos

5. Exercícios Práticos

Exercício 1: Implementação Básica

Complete as funções `alocarMatriz()` e `desalocarMatriz()` no código fornecido.

Exercício 2: Matriz Transposta

Implemente uma função que crie uma matriz transposta de uma matriz dinâmica.

O que é uma Matriz Transposta?

A matriz transposta é uma matriz obtida trocando as linhas pelas colunas da matriz original. Em outras palavras, se A é uma matriz m×n, sua transposta AT será uma matriz n×m onde AT[i][j] = A[j][i].

Exemplo:

```
Matriz Original (2×3):     Matriz Transposta (3×2):
[1  2  3]                  [1  4]
[4  5  6]                  [2  5]
                           [3  6]
```

Implementação Sugerida:

```c
int** criarMatrizTransposta(int** matriz, int linhas, int colunas) {
    // Aloca a matriz transposta (colunas x linhas)
    int** transposta = (int**)malloc(colunas * sizeof(int*));

    if (transposta == NULL) {
        printf("Erro: Falha ao alocar matriz transposta\n");
        return NULL;
    }

    // Aloca cada linha da transposta
    for (int i = 0; i < colunas; i++) {
        transposta[i] = (int*)malloc(linhas * sizeof(int));

        if (transposta[i] == NULL) {
            printf("Erro: Falha ao alocar linha %d da transposta\n", i);
            // Libera a memória já alocada
            for (int j = 0; j < i; j++) {
                free(transposta[j]);
            }
            free(transposta);
            return NULL;
        }
    }

    // Preenche a matriz transposta
    for (int i = 0; i < linhas; i++) {
        for (int j = 0; j < colunas; j++) {
            transposta[j][i] = matriz[i][j];
        }
    }

    return transposta;
}
```

Exercício 3: Multiplicação de Matrizes

Implemente uma função que multiplique duas matrizes dinâmicas.

O que é Multiplicação de Matrizes?

A multiplicação de matrizes é uma operação que produz uma matriz a partir de duas matrizes. Para multiplicar duas matrizes, o número de colunas da primeira matriz deve ser igual ao número de linhas da segunda matriz. O resultado será uma matriz com o número de linhas da primeira e o número de colunas da segunda.

Exemplo:

```
Matriz A (2×3):     Matriz B (3×2):     Resultado (2×2):
[1  2  3]          [7   8]             [58   64]
[4  5  6]          [9   10]            [139  154]
                   [11  12]

Cálculo do elemento [0][0] do resultado:
(1×7) + (2×9) + (3×11) = 7 + 18 + 33 = 58
```

Implementação Sugerida:

```c
int** multiplicarMatrizes(int** matrizA, int linhasA, int colunasA,
                          int** matrizB, int linhasB, int colunasB) {
    // Verifica se a multiplicação é possível
    if (colunasA != linhasB) {
        printf("Erro: Dimensões incompatíveis para multiplicação\n");
        return NULL;
    }

    // Aloca a matriz resultado (linhasA x colunasB)
    int** resultado = (int**)malloc(linhasA * sizeof(int*));

    if (resultado == NULL) {
        printf("Erro: Falha ao alocar matriz resultado\n");
        return NULL;
    }

    // Aloca cada linha do resultado
    for (int i = 0; i < linhasA; i++) {
        resultado[i] = (int*)malloc(colunasB * sizeof(int));

        if (resultado[i] == NULL) {
            printf("Erro: Falha ao alocar linha %d do resultado\n", i);
            // Libera a memória já alocada
            for (int j = 0; j < i; j++) {
                free(resultado[j]);
            }
            free(resultado);
            return NULL;
        }
    }

    // Realiza a multiplicação
    for (int i = 0; i < linhasA; i++) {
        for (int j = 0; j < colunasB; j++) {
            resultado[i][j] = 0;
            for (int k = 0; k < colunasA; k++) {
                resultado[i][j] += matrizA[i][k] * matrizB[k][j];
            }
        }
    }

    return resultado;
}
```

Desafios e Considerações:

1. **Gerenciamento de Memória**:
    - É crucial liberar toda a memória alocada
    - Tratar casos de falha na alocação
    - Evitar vazamentos de memória
2. **Validação de Dados**:

- Verificar dimensões das matrizes
- Validar ponteiros nulos
- Tratar erros de forma adequada

3. **Otimização**:
    - A multiplicação de matrizes tem complexidade $O(n^3)$
    - Existem algoritmos mais eficientes (ex: Algoritmo de Strassen)
    - Considerar o uso de técnicas de cache-friendly
4. **Testes**:
    - Testar com matrizes de diferentes tamanhos
    - Verificar casos limite
    - Validar resultados com exemplos conhecidos
5. **Extensões Possíveis**:
    - Implementar multiplicação de matrizes esparsas
    - Adicionar suporte para números de ponto flutuante
    - Implementar operações paralelas para melhor desempenho

Exercício 4: Análise de Memória

1. Execute o programa com diferentes tamanhos de matriz
2. Observe o uso de memória no Gerenciador de Tarefas
3. Documente suas observações

Exercício 5 (Opcional): Demonstração da Alocação Lazy em Matrizes

1. Modifique a função `alocarMatriz()` para incluir uma opção de inicialização:

```
void alocarMatriz(int n, bool inicializar) {
    // Aloca o array de ponteiros
    matriz_atual = (int **)malloc(n * sizeof(int *));

    if (matriz_atual == NULL) {
        printf("Erro: Falha ao alocar array de ponteiros\n");
        return;
    }

    // Aloca cada linha
    for (int i = 0; i < n; i++) {
        matriz_atual[i] = (int *)malloc(n * sizeof(int));

        if (matriz_atual[i] == NULL) {
            printf("Erro: Falha ao alocar linha %d\n", i);

            // Libera a memória já alocada
            for (int j = 0; j < i; j++) {
                free(matriz_atual[j]);
            }
            free(matriz_atual);
            matriz_atual = NULL;
            return;
        }

        // Inicializa a linha se solicitado
        if (inicializar) {
            for (int j = 0; j < n; j++) {
                matriz_atual[i][j] = 0;  // Isso forçará a alocação f
            }
        }
    }

    tamanho_matriz_atual = n;
}
```

2. Execute o programa com uma matriz grande (ex: 1000x1000)
3. Compare o uso de memória no Gerenciador de Tarefas:
    - Sem inicialização: apenas alocação virtual
    - Com inicialização: alocação física real
4. Documente as diferenças observadas no uso de memória
5. Compare com os resultados obtidos na prática de vetores
6. Analise por que o efeito é mais significativo em matrizes

6. Código Base

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

int **matriz_atual = NULL;
int tamanho_matriz_atual = 0;

void alocarMatriz(int n, bool inicializar) {
    // Aloca o array de ponteiros
    matriz_atual = (int **)malloc(n * sizeof(int *));

    if (matriz_atual == NULL) {
        printf("Erro: Falha ao alocar array de ponteiros\n");
        return;
    }

    // Aloca cada linha
    for (int i = 0; i < n; i++) {
        matriz_atual[i] = (int *)malloc(n * sizeof(int));

        if (matriz_atual[i] == NULL) {
            printf("Erro: Falha ao alocar linha %d\n", i);

            // Libera a memória já alocada
            for (int j = 0; j < i; j++) {
                free(matriz_atual[j]);
            }
            free(matriz_atual);
            matriz_atual = NULL;
            return;
        }

        // Inicializa a linha se solicitado
        if (inicializar) {
            for (int j = 0; j < n; j++) {
                matriz_atual[i][j] = 0;  // Isso forçará a alocação física
            }
        }
    }

    tamanho_matriz_atual = n;
}

void desalocarMatriz() {
    if (matriz_atual == NULL) {
        return;
    }

    // Libera cada linha
    for (int i = 0; i < tamanho_matriz_atual; i++) {
        free(matriz_atual[i]);
    }

    // Libera o array de ponteiros
    free(matriz_atual);
```

```
    // Reseta as variáveis
    matriz_atual = NULL;
    tamanho_matriz_atual = 0;
}

int main() {
    int opcao;
    int n;
    bool executando = true;
    bool inicializar;

    while (executando) {
        printf("\n--- Gerenciamento de Matriz ---\n");
        printf("1. Alocar matriz\n");
        printf("2. Desalocar matriz\n");
        printf("3. Imprimir matriz\n");
        printf("4. Criar matriz transposta\n");
        printf("5. Multiplicar matrizes\n");
        printf("6. Sair\n");
        printf("Escolha uma opção: ");
        scanf("%d", &opcao);

        switch (opcao) {
            case 1:
                if (matriz_atual != NULL) {
                    printf("Já existe uma matriz alocada. Desaloque-a prime
                } else {
                    printf("Digite o tamanho N para a matriz N x N: ");
                    scanf("%d", &n);
                    if (n > 0) {
                        printf("Deseja inicializar a matriz? (1-Sim/0-Não)
                        scanf("%d", &inicializar);
                        alocarMatriz(n, inicializar);
                        if (matriz_atual != NULL) {
                            printf("Matriz %d x %d alocada com sucesso.\n"
                            if (inicializar) {
                                printf("Matriz inicializada com zeros.\n");
                            }
                        } else {
                            printf("Falha ao alocar a matriz.\n");
                        }
                    } else {
                        printf("Tamanho inválido.\n");
                    }
                }
                break;
            case 2:
                if (matriz_atual == NULL) {
                    printf("Nenhuma matriz alocada para desalocar.\n");
                } else {
                    desalocarMatriz();
                    printf("Matriz desalocada com sucesso.\n");
                }
                break;
```

```
            case 3:
                if (matriz_atual == NULL) {
                    printf("Nenhuma matriz alocada para imprimir.\n");
                } else {
                    printf("Funcionalidade ainda não implementada.\n");
                    printf("Esta opção permitirá imprimir a matriz atual.\
                }
                break;
            case 4:
                if (matriz_atual == NULL) {
                    printf("Nenhuma matriz alocada para transpor.\n");
                } else {
                    printf("Funcionalidade ainda não implementada.\n");
                    printf("Esta opção permitirá criar a matriz transposta
                }
                break;
            case 5:
                if (matriz_atual == NULL) {
                    printf("Nenhuma matriz alocada para multiplicação.\n");
                } else {
                    printf("Funcionalidade ainda não implementada.\n");
                    printf("Esta opção permitirá multiplicar a matriz atua
                }
                break;
            case 6:
                if (matriz_atual != NULL) {
                    printf("Desalocando matriz antes de sair...\n");
                    desalocarMatriz();
                }
                executando = false;
                printf("Saindo do programa.\n");
                break;
            default:
                printf("Opção inválida.\n");
        }
    }

    return 0;
}
```

7. Solução Sugerida

Função alocarMatriz()

```c
void alocarMatriz(int n, bool inicializar) {
    // Aloca o array de ponteiros
    matriz_atual = (int **)malloc(n * sizeof(int *));

    if (matriz_atual == NULL) {
        printf("Erro: Falha ao alocar array de ponteiros\n");
        return;
    }

    // Aloca cada linha
    for (int i = 0; i < n; i++) {
        matriz_atual[i] = (int *)malloc(n * sizeof(int));

        if (matriz_atual[i] == NULL) {
            printf("Erro: Falha ao alocar linha %d\n", i);

            // Libera a memória já alocada
            for (int j = 0; j < i; j++) {
                free(matriz_atual[j]);
            }
            free(matriz_atual);
            matriz_atual = NULL;
            return;
        }

        // Inicializa a linha se solicitado
        if (inicializar) {
            for (int j = 0; j < n; j++) {
                matriz_atual[i][j] = 0;  // Isso forçará a alocação física
            }
        }
    }

    tamanho_matriz_atual = n;
}
```

Função desalocarMatriz()

```c
void desalocarMatriz() {
    if (matriz_atual == NULL) {
        return;
    }

    // Libera cada linha
    for (int i = 0; i < tamanho_matriz_atual; i++) {
        free(matriz_atual[i]);
    }

    // Libera o array de ponteiros
    free(matriz_atual);

    // Reseta as variáveis
    matriz_atual = NULL;
    tamanho_matriz_atual = 0;
}
```

8. Análise de Desempenho

8.1 Uso de Memória

- Execute o programa com diferentes tamanhos de matriz
- Observe o uso de memória no Gerenciador de Tarefas
- Compare o uso de memória entre alocação e desalocação

8.2 Testes de Estresse

1. Tente alocar uma matriz muito grande
2. Observe o comportamento do sistema
3. Documente quando ocorrem falhas de alocação

9. Referências

1. C Programming Language
2. Dynamic Memory Allocation in C
3. Memory Management in C