RELATÓRIO TÉCNICO - PROJETO DE CAMPO MINADO EM C

Aluno: João Davi dos Santos Araújo
Professor: Fernando Marques Figueira Filho
Disciplina: Introdução às Técnicas de Programação
Universidade Federal do Rio Grande do Norte
Novembro de 2025

INTRODUÇÃO E CONTEXTO

O projeto Campo Minado, desenvolvido em linguagem C, visa a implementação de um dos jogos clássicos de raciocínio lógico. Na U1, a estrutura básica do jogo foi estabelecida utilizando-se recursos elementares como arrays estáticos. A transição para a U2 marcou uma evolução significativa, focando na aplicação de conceitos mais avançados de C: manipulação de strings, estruturas de dados complexas via ponteiros e gerenciamento de memória através de alocação dinâmica. O principal objetivo da U2 foi reformular a base do código para substituir as matrizes de tamanho fixo por estruturas alocadas dinamicamente. Esta mudança não só aumenta a flexibilidade do jogo (permitindo tamanhos de tabuleiro definidos em tempo de execução) como também consolida o aprendizado sobre o ciclo de vida da memória em C (alocação, uso e liberação).

A versão final do código implementa as seguintes funcionalidades:
– Alocação dinâmica de matrizes: o tabuleiro e as matrizes de estado (revelado e flag) são alocados em tempo de execução.
– Inicialização e posicionamento de minas: distribuição aleatória de 5 minas no tabuleiro.
– Contagem de minas adjacentes: cálculo preciso do número de minas que circundam uma célula.
– Interface de linha de comando (CLI): Interação via comandos r L C (revelar) e f L C (marcar/desmarcar bandeira).
– Condições de vitória/derrota: o jogo termina em derrota ao acertar uma mina ou em vitória ao revelar todas as células seguras.

– Uso de strings: leitura do nome do jogador para personalização da experiência.

METODOLOGIA

O desenvolvimento foi realizado em um ambiente de linha de comando, utilizando o compilador GCC (GNU Compiler Collection) e o editor de código Visual Studio Code. Além disso, foi feito no sistema operacional Windows e com uso das bibliotecas stdio.h, stdlib.h, time.h, locale.h e string.h.

A organização do código em blocos de funções (locação, jogo, principal) facilita imensamente a manutenção. Ao isolar a lógica de gerenciamento de memória em alocaMatriz e liberaMatriz, garante-se que qualquer alteração no tamanho do tabuleiro ou na forma como a memória é tratada possa ser feita em um único local, sem afetar a lógica central do jogo. Similarmente, o motor do jogo (minasAdjacentes) está completamente desacoplado da interface (imprimeCampo).

ANÁLISE DO CÓDIGO

Esta seção detalha a aplicação dos conceitos centrais da U2, que foram fundamentais para a melhoria do projeto.

Strings

Uso no projeto: as strings foram utilizadas para interagir com o jogador. Especificamente, a string nome armazena o nome fornecido pelo usuário, sendo posteriormente utilizada na mensagem inicial e na mensagem de vitória.

Funções de manipulação: a função padrão fgets foi usada para ler o nome, o que garante a leitura segura de strings, incluindo espaços. No entanto, fgets frequentemente captura o caractere de nova linha (\n). Para resolver isso, foi implementada a seguinte linha de manipulação, usando a função strcspn da biblioteca string.h:

```
nome[strcspn(nome, "\n")] = '\0';
```

Estruturas de repetição aninhadas

Aplicação: as estruturas de repetição aninhadas (for dentro de for) são a espinha dorsal da iteração em estruturas 2D (matrizes). Elas foram aplicadas em três funções principais:

1. inicializaCampo: para zerar todas as células do tabuleiro.
2. imprimeCampo: para percorrer e exibir cada célula do tabuleiro na tela.
3. minasAdjacentes: para inspecionar os 8 vizinhos de uma célula específica.

Complexidade dos algoritmos: para o tabuleiro de L linhas e C colunas:

– inicializaCampo e imprimeCampo possuem complexidade de tempo O(L×C), que é $O(N^2)$ se L = C = N. Essas funções percorrem cada célula do tabuleiro exatamente uma vez.

– minasAdjacentes inspeciona no máximo 3 × 3 = 9 células (a célula central e seus 8 vizinhos). Sua complexidade é constante, O(1), pois o número de operações não cresce com o tamanho do tabuleiro.

Matrizes e ponteiros

Implementação de matrizes: as matrizes (campo, revelado, flag) não foram implementadas como arrays estáticos. Em vez disso, foram construídas sendo representadas por um ponteiro para um ponteiro (int **).

Estratégias de percorrimento: para acessar a célula na linha i e coluna j, utiliza-se a sintaxe de colchetes campo[i][j]. A grande diferença é que, internamente, campo[i] acessa o ponteiro para a i-ésima linha, e campo[i][j] desreferencia o j-ésimo inteiro dessa linha.

Uso de ponteiros: os ponteiros foram utilizados primariamente na simulação das matrizes e na passagem de argumentos para funções.

Alocação dinâmica e gerenciamento de memória

Necessidade: a alocação dinâmica, realizada na função alocaMatriz, foi necessária para que o tamanho das matrizes pudesse ser definido em tempo de execução, em vez de ser fixado em tempo de compilação (#define na U1). Isso confere flexibilidade ao código.

Estratégia de gerenciamento: a memória é gerenciada em duas etapas dentro de alocaMatriz:

1. Alocação do vetor de ponteiros: int **m = malloc(lin * sizeof(int *));

2. Alocação de cada linha: m[i] = malloc(col * sizeof(int));

Tratamento de falhas: a função alocaMatriz inclui tratamento de falha de alocação:

```
if (m == NULL) {
    printf("Erro: memória insuficiente.\n");
    exit(1);
}
```

Caso malloc retorne NULL, o programa exibe uma mensagem de erro e é encerrado com exit(1).

Estratégia contra memory leaks: a estratégia para garantir a liberação total da memória alocada é a função void liberaMatriz.

```
void liberaMatriz(int **m, int lin) {
    for (int i = 0; i < lin; i++)
        free(m[i]); // Libera cada linha
        free(m); // Libera o vetor de ponteiros
}
```

O princípio é que para cada malloc, deve haver um free correspondente. Como há 1(vetor de ponteiros) + lin(linhas) chamadas a malloc na alocação, a liberaMatriz executa lin chamadas a free para as linhas e mais 1 chamada final para o vetor de ponteiros. Isso garante que o código não terá vazamentos de memória.

DIFICULDADES E SOLUÇÕES

O maior desafio técnico na transição da U1 para a U2 centrou-se na compreensão e implementação correta da alocação dinâmica de matrizes usando ponteiros duplos (int **). A dificuldade inicial foi diferenciar a alocação do vetor de ponteiros (sizeof(int *)) da alocação das linhas (sizeof(int)).

– Memory leaks (vazamentos de memória): o risco de esquecer de liberar uma das partes da matriz alocada dinamicamente é alto, levando a vazamentos de memória.

A superação desses desafios veio através da criação de pares de funções de gerenciamento. A solução foi:

1. Isolamento: criar as funções alocaMatriz e liberaMatriz para centralizar e encapsular toda a lógica de memória.

2. Seguir rigorosamente a regra: para cada malloc, deve haver um free correspondente.
3. Checagem de falhas: incluir testes explícitos para NULL após cada malloc para garantir robustez.

O uso de ponteiros duplos demonstrou o poder de C de manipular estruturas de dados complexas. O gerenciamento de memória se tornou um foco principal, transformando o projeto de um simples exercício de lógica para uma aplicação robusta com atenção ao ciclo de vida da memória.

CONCLUSÃO

A reformulação do projeto Campo Minado na Unidade 2 representou um salto qualitativo. Na U1, o código era simples, seguro, mas ainda inflexível devido ao uso de constantes (#define) e arrays estáticos. O código da U2 é mais complexo, exigindo um entendimento mais profundo de ponteiros e alocação dinâmica, mas em troca oferece uma estrutura flexível e escalável. A capacidade de criar estruturas de dados em tempo de execução é um diferencial crucial para a programação de sistemas. O projeto demonstra a aplicação bem-sucedida de todos os conceitos chaves da U2, conforme detalhado na seção de análise do código.

Para eventuais mudanças futuras, o projeto pode ser expandido com algumas funcionalidades mais avançadas:
1 - Implementar uma função recursiva que, ao revelar uma célula sem minas adjacentes (valor 0), revele automaticamente todas as células vizinhas, simulando a funcionalidade clássica do Campo Minado.
2 - Permitir que o usuário defina o tamanho do tabuleiro (lin e col) no início do jogo, aproveitando a alocação dinâmica já implementada.