



UNIVERSIDADE MUNICIPAL DE SÃO CAETANO DO SUL

GIOVANNA ANTONIETA LLANOS NARANJO
LETÍCIA FERREIRA MENDONÇA

DESENVOLVIMENTO DE UM JOGO EM LINGUAGEM C

SÃO CAETANO DO SUL

2024

GIOVANNA ANTONIETA LLANOS NARANJO
LETÍCIA FERREIRA MENDONÇA

DESENVOLVIMENTO DE UM JOGO EM LINGUAGEM C

Trabalho de Graduação apresentado à Disciplina
Algoritmo e Estrutura de Dados I no 1º Semestre do
curso Ciência da Computação da Universidade
Municipal de São Caetano do Sul.
Orientador: Prof. Fábio Ferreira de Assis

SÃO CAETANO DO SUL
2024

RESUMO

Este trabalho apresenta o desenvolvimento de um jogo de "Campo Minado" utilizando a linguagem de programação C. O objetivo principal foi criar uma implementação funcional e interativa do jogo, com foco na modularidade do código, permitindo fácil manutenção e expansão. Para melhorar a experiência do usuário, foi utilizada a formatação com cores ANSI, proporcionando uma interface visual mais atraente, embora restrita ao suporte do terminal utilizado. A metodologia envolveu a implementação de funções específicas para gerenciar as ações do jogo, como a revelação de células, a marcação de minas e a verificação de vitória, além da criação de um tabuleiro dinâmico de acordo com as escolhas do usuário. O código foi desenvolvido com a possibilidade de ser facilmente modificado para incluir novos recursos, como níveis de dificuldade ajustáveis, validação de entradas e dicas para iniciantes. Como resultado, obteve-se um jogo funcional, mas com limitações em termos de validação de entrada e portabilidade, que podem ser aprimoradas em versões futuras. As conclusões indicam que o projeto tem grande potencial de evolução, podendo ser expandido com melhorias no design, na jogabilidade e na interface gráfica, além da implementação de novos recursos de interação.

Palavras-chave: *Campo Minado, Linguagem C, Jogabilidade.*

ABSTRACT

This paper presents the development of a "Minesweeper" game using the C programming language. The main goal was to create a functional and interactive implementation of the game, focusing on code modularity to allow for easy maintenance and expansion. To enhance the user experience, ANSI color formatting was used, providing a more attractive visual interface, though limited to the terminal's support. The methodology involved implementing specific functions to manage game actions, such as revealing cells, marking mines, and checking for victory, in addition to creating a dynamic board based on the user's choices. The code was developed with the possibility of being easily modified to include new features, such as adjustable difficulty levels, input validation, and hints for beginners. As a result, a functional game was achieved, but with limitations in terms of input validation and portability, which could be improved in future versions. The conclusions indicate that the project has great potential for evolution, with possibilities for expansion in design, gameplay, and graphical interface, as well as the implementation of new interaction features.

Keywords: Minesweeper, C Programming, Gameplay.

LISTA DE SIGLAS E ABREVIações

ANSI *American National Standards Institute*

SUMÁRIO

INTRODUÇÃO	7
A HISTÓRIA DO CAMPO MINADO	8
COMO JOGAR	8
EXPLICAÇÃO PASSO A PASSO DOS CÓDIGOS	9
Diretivas do pré-processador, declaração de constantes e variáveis globais	9
1. Inclusão de bibliotecas	9
2. Definição de constantes	9
3. Declaração de variáveis globais	10
Funções do Código	11
1. limparTela()	11
2. imprimirRegras()	11
3. inicializarTabuleiro()	13
4. imprimirTabuleiro(int revelarTudo)	14
5. revelar(int linha, int coluna)	17
6. checarVitoria()	18
7. main()	19
1. Configurações iniciais	21
2. Exibição das regras	22
3. Entrada do tamanho do tabuleiro e número de minas	22
4. Inicialização do tabuleiro	22
5. Variáveis para controle do loop principal	23
6. Loop principal do jogo	23
7. Exibição do modo e tabuleiro	23
8. Solicitação de comando do jogador	23
9. Processamento do comando	24
10. Ações de marcar e revelar	24
CÓDIGO COMPLETO DO JOGO	26
CONSIDERAÇÕES FINAIS	27
REFERÊNCIAS BIBLIOGRÁFICAS	28

INTRODUÇÃO

Este trabalho tem como objetivo o desenvolvimento de uma versão do jogo Campo Minado utilizando a linguagem de programação C, com foco em proporcionar uma experiência funcional e interativa ao usuário. O Campo Minado é um jogo clássico de raciocínio e estratégia, amplamente popular, no qual o jogador deve revelar células em um tabuleiro sem desencadear explosões de minas ocultas. Para auxiliar no processo, o jogo oferece pistas numéricas, que indicam a quantidade de minas adjacentes a cada célula. O desafio do jogo está em descobrir quais células contêm minas, com base nessas pistas, sem cometer erros.

O principal objetivo deste projeto foi criar uma implementação modular e de fácil manutenção do jogo, permitindo uma evolução contínua e a implementação de novas funcionalidades. O código foi estruturado em funções distintas para diferentes ações, como a revelação de células, a marcação de minas e a verificação das condições de vitória, o que facilita a leitura e a manutenção do código. A modularidade também possibilita que futuras modificações sejam feitas com maior agilidade, além de ser uma boa prática para garantir a clareza e a organização do código-fonte.

No aspecto visual, foi utilizado o formato de cores ANSI para melhorar a experiência do usuário. O uso de cores na interface do jogo, como a exibição das minas e dos números de células adjacentes, contribui para uma maior clareza e apelo estético, embora seja necessário um terminal que suporte esse tipo de formatação. Para tornar o jogo ainda mais interativo, a interface foi desenhada para ser amigável, com instruções claras e respostas rápidas às ações do jogador.

Em resumo, este trabalho proporciona uma versão sólida do jogo de Campo Minado, com uma base que permite a expansão e a melhoria contínua. A implementação de novas funcionalidades, a melhoria da interface gráfica e o aprimoramento da jogabilidade são apenas algumas das possibilidades de evolução do projeto, que visa proporcionar uma experiência de jogo desafiadora e divertida para os usuários.

A HISTÓRIA DO CAMPO MINADO

A ideia do Campo Minado tem raízes em jogos de lógica e quebra-cabeças que envolvem dedução e eliminação de possibilidades. Diante disso, ele foi popularizado por apresentar um design simples, com a dificuldade de manter o jogo intacto, sem explosões de minas.

O jogo foi criado em 1989 por Robert Donner e Curt Johnson, sendo lançado inicialmente com o Windows 3.1. Desde então, tornou-se uma parte integrante do Sistema Operacional Windows, com versões lançadas também para outros sistemas operacionais e até dispositivos móveis.

COMO JOGAR

Cada partida começa com um tabuleiro dividido em vários quadrados, todos inicialmente sem marcações. O objetivo do jogador é abrir todas as células que não contenham bombas.

À medida que as células são abertas, números podem aparecer nelas. Esses números variam de 1 a 8 e indicam quantas bombas existem ao redor daquela casa, vertical, horizontal ou diagonalmente. Caso uma célula aberta não contenha números, significa que não há bombas nas casas adjacentes. Nesse caso, todas as casas ao redor dessa área serão abertas automaticamente até encontrar uma que indique um número.

Se o jogador abrir uma casa com uma mina, o jogo termina. Para vencer, é necessário abrir todas as células do tabuleiro que não tenham minas.

Além de abrir as casas, o jogador também pode marcar com bandeiras os locais onde acredita haver minas. Para isso, basta digitar a letra “B” para ativar a função de marcar e digitar as coordenadas (linha e coluna) da célula desejada. O uso das bandeiras é bastante útil para organizar as jogadas e evitar explodir minas acidentalmente.

EXPLICAÇÃO PASSO A PASSO DOS CÓDIGOS

Diretivas do pré-processador, declaração de constantes e variáveis globais

1. Inclusão de bibliotecas

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

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

```
#include <time.h>
```

```
#include <locale.h>
```

- **Descrição:**

Essas diretivas incluem bibliotecas padrão do C, que fornecem funcionalidades essenciais usadas no código.

- **Linha por linha:**

1. `#include <stdio.h>:`

- Permite o uso de funções de entrada e saída, como `printf` e `scanf`.

2. `#include <stdlib.h>:`

- Necessária para funções como `system`, `rand`, e `srand`.
- Também é usada para manipulação de memória dinâmica (não utilizada neste código).

3. `#include <time.h>:`

- Necessária para inicializar o gerador de números aleatórios com base no relógio do sistema por meio da função `time(NULL)`.

4. `#include <locale.h>:`

- Fornece funções para configurar e manipular as configurações regionais de um programa, como idioma, formato de números e datas, ou suporte a caracteres especiais.

2. Definição de constantes

```
#define MINA -1
```

```
#define REVELADA -2
```

```
#define MARCADA -3
```

```
#define MAX 20
```

- **Descrição:**

Define valores constantes usados em várias partes do programa para melhorar a legibilidade e evitar o uso de números mágicos.

- **Linha por linha:**

1. `#define MINA -1:`

- Representa uma célula que contém uma **mina oculta** no tabuleiro.
- No array `tabuleiro`, se uma posição é **MINA**, significa que o jogador perderá o jogo ao revelar essa célula.
- 2. `#define REVELADA -2`:
 - Representa o estado de uma célula que já foi **revelada pelo jogador**.
 - No array `visualizacao`, se o valor de uma célula é **REVELADA**, significa que o jogador já a abriu no tabuleiro.
- 3. `#define MARCADA -3`:
 - Indica que o jogador marcou uma célula com uma **bandeira** (🚩), presumindo que ali há uma mina.
 - No array `visualizacao`, se uma célula é **MARCADA**, ela está protegida contra revelação acidental, mas isso não garante que contenha realmente uma mina.
- 4. `#define MAX 20`:
 - Define o tamanho **máximo permitido** para o tabuleiro, ou seja, no máximo 20x20 células.
 - Essa constante é usada para limitar os arrays `tabuleiro` e `visualizacao`, garantindo que o programa não ultrapasse o limite de memória alocado estaticamente.
- **Contexto:**
O uso de constantes melhora a legibilidade e facilita alterações futuras no comportamento do programa.

3. Declaração de variáveis globais

```
int LINHAS, COLUNAS, MINAS;

int tabuleiro[MAX][MAX];

int visualizacao[MAX][MAX];

int modoMarcar = 0;
```

- **Descrição:**
Declara variáveis globais que são acessíveis em todo o programa.
- **Linha por linha:**
 1. `int LINHAS, COLUNAS, MINAS`:
 - Variáveis que armazenam o tamanho do tabuleiro (número de linhas e colunas) e a quantidade de minas, definidas pelo jogador no início do jogo.
 2. `int tabuleiro[MAX][MAX]`:
 - Representa o estado real do tabuleiro, com informações como minas (**MINA**) e números indicando a quantidade de minas adjacentes.
 3. `int visualizacao[MAX][MAX]`:

- Armazena o estado visível do tabuleiro para o jogador:
 - 0: Célula não revelada.
 - REVELADA: Célula que o jogador revelou.
 - MARCADA: Célula marcada pelo jogador como possível mina.
- 4. `int modoMarcar = 0;`
 - Controla o modo de interação do jogador:
 - 0: O jogador está no modo **revelar** (abrir células no tabuleiro).
 - 1: O jogador está no modo **marcar** (colocar bandeiras em células que suspeita conterem minas).
 - Essa variável é alterada quando o jogador digita um comando especial (ex.: 'B' para entrar no modo de marcação).
- **Contexto:**
 As dimensões fixas são uma escolha prática para evitar entradas desproporcionais ao terminal, mas podem ser uma limitação se for necessário um tabuleiro maior.

Funções do Código

1. `limparTela()`

```
void limparTela() {
    system("cls");
}
```

- **Descrição:**
 Essa função limpa o terminal, tornando a interface do jogo mais organizada e clara para o jogador. A cada mudança importante no estado do jogo (como revelar ou marcar uma célula), a tela é atualizada.
- **Linha por linha:**
 1. `system("cls");`: Executa o comando `cls` no sistema operacional. Esse comando limpa o terminal em sistemas Windows. Se o jogo for executado em Linux ou Unix-like, seria necessário substituir por `system("clear");`.

2. `imprimirRegras()`

```
void imprimirRegras() {
    printf("\033[1;35m!!!   BEM-VINDO(A)   AO   CAMPO   MINADO\n\n");
    printf("\033[1;34mObjetivo:\033[0m Revelar todas as células sem minas.\n\n");
    printf("\033[1;34mInstruções:\033[0m\n\n");
    printf("\033[1;35m1.\033[0m Escolha o tamanho do tabuleiro e a quantidade de minas.\n");
}
```

```

        printf("\033[1;35m2.\033[0m Digite as coordenadas (linha e
coluna) para revelar ou marcar uma mina.\n");
        printf("\033[1;35m3.\033[0m Se revelar uma mina, o jogo acaba!
Se não, um número indicará quantas minas estão próximas.\n\n\n");
        printf("\033[1;34mBoa sorte!\033[0m\n");

printf("\033[1;35m-----
-----\033[0m\n");
}

```

- **Descrição:**

Essa função exibe as regras e as instruções do jogo, além de dar uma introdução visualmente estilizada usando cores. A função é chamada no início do programa para ajudar o jogador a entender como o jogo funciona.

- **Linha por linha:**

1. A função **void** é usada para indicar que uma função não retorna nenhum valor ou que ela não recebe parâmetros. Um exemplo disso é a função **imprimirRegras**, que tem o propósito de exibir as regras do jogo. Ela fornece ao jogador informações sobre o objetivo do jogo e explica como interagir com o tabuleiro.
2. Cores no terminal: Ao longo do código, diversas funções utilizam cores para exibir texto no terminal, aproveitando as sequências de controle ANSI. Esses códigos são utilizados para modificar a aparência do texto, como a cor do texto, o fundo, o estilo (negrito, sublinhado). As sequências de controle ANSI começam com o caractere especial “\033”, seguido de um código que especifica o efeito ou a cor desejada. O código é finalizado com o caractere “m”. Para encerrar qualquer formatação de cor ou estilo, é utilizado o código “\033[0m”. Exemplos:
 - \033[1;m: Aplica o estilo negrito ou aumenta a intensidade do texto;
 - \033[30m até \033[37m: Cores de texto, variando de preto a branco;
 - \033[40m até \033[47m: Cores de fundo, variando de preto a branco.

Esses códigos de controle permitem que o texto exibido no terminal seja destacado de maneira visualmente atraente e fácil de interpretar, melhorando a experiência do usuário.

3. **printf(...)**: Exibe o texto explicativo, indicando o objetivo, as instruções e como o jogo funciona.

3. **inicializarTabuleiro()**

```

void inicializarTabuleiro() {
    for (int i = 0; i < LINHAS; i++) {
        for (int j = 0; j < COLUNAS; j++) {
            tabuleiro[i][j] = 0;
            visualizacao[i][j] = 0;
        }
    }

    int minasColocadas = 0;
    while (minasColocadas < MINAS) {
        int linha = rand() % LINHAS;
        int coluna = rand() % COLUNAS;

        if (tabuleiro[linha][coluna] != MINA) {
            tabuleiro[linha][coluna] = MINA;
            minasColocadas++;

            for (int i = linha - 1; i <= linha + 1; i++) {
                for (int j = coluna - 1; j <= coluna + 1; j++) {
                    if (i >= 0 && i < LINHAS && j >= 0 && j <
COLUNAS && tabuleiro[i][j] != MINA) {
                        tabuleiro[i][j]++;
                    }
                }
            }
        }
    }
}

```

- **Descrição:**

A função `inicializarTabuleiro()` tem como objetivo iniciar o estado do tabuleiro do jogo, no caso, o Campo Minado. Ela configura tanto a matriz do tabuleiro real (onde as minas e os números de minas adjacentes estão armazenados) quanto a matriz de visualização (que controla o que o jogador pode ver e interagir, como células reveladas, células marcadas, etc).

- **Linha por linha:**

- Tabuleiro: A matriz `tabuleiro` é inicializada com o valor 0 para todas as células. O valor 0 indica que a célula não contém uma mina e que, inicialmente, não há minas adjacentes;
- Visualização: A matriz “`visualizacao`” também é configurada com 0, o que significa que todas as células estão inicialmente ocultas, e o jogador ainda não interagiu com elas.

- Variável `minasColocadas` é usada para contar quantas minas foram posicionadas.
- `minasColocadas`:
 - O objetivo da função é distribuir as minas aleatoriamente no tabuleiro. A quantidade de minas a ser colocada é determinada pela variável `MINAS`;
 - Para garantir que exatamente o número especificado de minas seja colocado, a função utiliza um loop `while`. Dentro desse loop, são geradas coordenadas aleatórias para as linhas e colunas, com os valores sendo limitados pelos tamanhos do tabuleiro (usando `rand() % LINHAS` para as linhas e `rand() % COLUNAS` para as colunas). Isso garante que as coordenadas geradas estejam sempre dentro dos limites do tabuleiro.
 - Antes de colocar uma mina, o código verifica se a célula já contém uma mina. Caso a célula já tenha uma mina (`tabuleiro[linha][coluna] != MINA`), ele escolhe outra célula aleatoriamente. Isso garante que não haja minas sobrepostas;
 - Se a célula estiver vazia, a mina é colocada e a variável “`minasColocadas`” é incrementada.
- Após colocar uma mina, o código precisa atualizar as células ao redor da mina para indicar quantas minas estão adjacentes a elas;
- Um loop aninhado percorre as 8 células vizinhas (em torno da célula onde a mina foi colocada) e incrementa o valor de cada célula adjacente que não contém uma mina (`tabuleiro[i][j] != MINA`). O valor de cada célula adjacente vai indicar o número de minas que estão ao redor dela.
- Esse processo garante que o número correto de minas adjacentes seja mostrado para o jogador nas células que não contêm minas.
- **Observações:**
 - O `rand()` deve ser inicializado com `srand(time(NULL))` no `main()` para que o posicionamento seja realmente aleatório.

4. `imprimirTabuleiro(int revelarTudo)`

```
void imprimirTabuleiro(int revelarTudo) {
    printf("\n      ");
    for (int j = 0; j < COLUNAS; j++) {
        printf("%2d  ", j + 1);
    }
    printf("\n");

    printf("      ");
    for (int j = 0; j < COLUNAS; j++) {
        printf("\033[1;30m---\033[0m");
    }
}
```

```

    }
    printf("\033[1;30m---\033[0m\n");

    for (int i = 0; i < LINHAS; i++) {
        printf("%2d \033[1;30m|\033[0m", i + 1);

        for (int j = 0; j < COLUNAS; j++) {
            if (revelarTudo) {
                if (tabuleiro[i][j] == MINA && visualizacao[i][j] ==
MARCADA) {
                    printf(" \033[1;31mX\033[0m ");
                } else if (visualizacao[i][j] == MARCADA &&
tabuleiro[i][j] != MINA) {
                    printf(" \033[1;34m!\033[0m ");
                } else if (tabuleiro[i][j] == MINA) {
                    printf(" \033[1;31mX\033[0m ");
                } else {
                    printf(" %2d ", tabuleiro[i][j]);
                }
            } else {
                if (visualizacao[i][j] == MARCADA) {
                    printf("  ");
                } else if (visualizacao[i][j] == REVELADA) {
                    if (tabuleiro[i][j] == MINA) {
                        printf(" \033[1;31mX\033[0m ");
                    } else {
                        printf(" %2d ", tabuleiro[i][j]);
                    }
                } else {
                    printf(" \033[1;30m?\033[0m ");
                }
            }
        }
        printf("\033[1;30m |\033[0m\n");
    }

    printf(" ");
    for (int j = 0; j < COLUNAS; j++) {
        printf("\033[1;30m---\033[0m");
    }
    printf("\033[1;30m---\033[0m\n");

```

}

- **Descrição:**

Exibe o tabuleiro no terminal, ajustando a visualização de acordo com o estado do jogo e se deve revelar tudo (fim de jogo).

- **Linha por linha:**

1. Define a função `imprimirTabuleiro`, que recebe um argumento `revelarTudo` para determinar se o tabuleiro deve ser exibido com todas as informações reveladas (quando o argumento é 1) ou ocultas (quando o argumento é 0).
2. As linhas, traços e espaços impressos no terminal, como `printf("\n")` e o último `for` dessa função, são úteis para separar e centralizar conteúdos, tornando a experiência do usuário mais agradável.
3. O primeiro `for` imprime os números das colunas no topo do tabuleiro. Ele percorre todas as colunas (`COLUNAS`) e imprime os números de 1 até o total de colunas (`j + 1`). O `%2d` garante que os números ocupem pelo menos dois espaços, deixando o alinhamento correto.
4. O segundo `for` imprime uma linha horizontal de traços `----` para cada coluna, utilizando a cor cinza para estilizar os traços.
5. O terceiro `for` percorre todas as linhas do tabuleiro (de 0 a `LINHAS-1`), imprimindo o número da linha à esquerda, com a formatação `%2d` (alinhamento de 2 caracteres). Após o número da linha, imprime o separador `|` com a cor cinza escuro.
6. O quarto `for` (encadeado ao terceiro) percorre cada coluna dentro da linha atual (`i`).
 - Se o argumento `revelarTudo` for verdadeiro (1): o tabuleiro será exibido com todos os dados revelados (as minas e os números de minas adjacentes).

```
tabuleiro[i][j] == MINA && visualizacao[i][j] == MARCADA
```

se a célula contém uma mina e foi marcada pelo jogador, é impresso um "X" vermelho.

```
visualizacao[i][j] == MARCADA && tabuleiro[i][j] != MINA
```


se a célula foi marcada pelo jogador mas não contém uma mina, imprime um "!" azul.

```
tabuleiro[i][j] == MINA
```

se a célula contém uma mina, é impresso um "X" vermelho.

Se a célula não contém uma mina, exibe o número de minas adjacentes com a formatação `%2d`, ou seja, o número de minas ao redor dessa célula.

- Se o argumento `revelarTudo` for falso (0): o tabuleiro deve ocultar informações não reveladas e apenas mostrar marcas de bandeira ou células reveladas.

`visualizacao[i][j] == MARCADA` se a célula foi marcada pelo jogador, imprime um ícone de bandeira .

`visualizacao[i][j] == REVELADA` se a célula foi revelada e for uma mina, imprime um "X" vermelho; caso contrário, imprime o número de minas adjacentes.

Se a célula não foi marcada nem revelada, imprime um "?" cinza, indicando que a célula está oculta.

5. `revelar(int linha, int coluna)`

```
int revelar(int linha, int coluna) {
    if (linha < 0 || linha >= LINHAS || coluna < 0 || coluna >=
COLUNAS) {
        printf("Coordenadas inválidas!\n");
        return 0;
    }

    if (visualizacao[linha][coluna] == REVELADA) {
        printf("Essa posição já foi revelada!\n");
        return 0;
    }

    visualizacao[linha][coluna] = REVELADA;

    if (tabuleiro[linha][coluna] == MINA) {
        return -1;
    }

    if (tabuleiro[linha][coluna] == 0) {
        for (int i = linha - 1; i <= linha + 1; i++) {
            for (int j = coluna - 1; j <= coluna + 1; j++) {
                if (i >= 0 && i < LINHAS && j >= 0 && j < COLUNAS &&
visualizacao[i][j] != REVELADA) {
                    revelar(i, j);
                }
            }
        }
    }
}
```

```

    }
}
return 1;
}

```

- **Descrição:**

Essa função revela uma célula no tabuleiro, lidando com a lógica de abrir espaços vazios (propagação recursiva). Se a célula contiver uma mina, indica o fim do jogo.

- **Linha por linha:**

- Se as coordenadas estão fora dos limites do tabuleiro ou se a célula já foi revelada, a função retorna sem realizar alterações.
- `visualizacao[linha][coluna] = REVELADA`, marca a célula como revelada no array `visualizacao`.
- `tabuleiro[linha][coluna] == MINA`, verifica se a célula contém uma mina e retorna `-1`, indicando que o jogador perdeu.
- Caso a célula seja um espaço vazio (`0`):
 - Percorre todas as células adjacentes (incluindo diagonais).
 - Chama recursivamente `revelar()` para cada célula adjacente, propagando a abertura dos espaços vazios.
- Retorna `0` se a célula revelada não for uma mina.

- **Observações:**

- A lógica recursiva é fundamental para o comportamento clássico do Campo Minado, onde abrir uma célula vazia revela automaticamente outras áreas.

6. `checarVitoria()`

```

int checarVitoria() {
    for (int i = 0; i < LINHAS; i++) {
        for (int j = 0; j < COLUNAS; j++) {
            if (tabuleiro[i][j] != MINA && visualizacao[i][j] !=
REVELADA) {
                return 0;
            }
        }
    }
    return 1;
}

```

- **Descrição:**

Verifica se o jogador venceu o jogo. A vitória ocorre quando todas as células sem minas foram reveladas.

- **Linha por linha:**
 - Percorre todo o tabuleiro.
 - Para cada célula:
 - Se a célula não contém uma mina (`tabuleiro[i][j] != MINA`) e não foi revelada (`visualizacao[i][j] != REVELADA`), a função retorna `0`, indicando que o jogo ainda não terminou.
 - Se todas as células sem minas foram reveladas, retorna `1`, sinalizando a vitória.
- **Observações:**
 - A lógica garante que a vitória depende exclusivamente da revelação correta das células sem minas, ignorando bandeiras marcadas erroneamente.

7. `main()`

```
int main() {
    setlocale(LC_ALL, "pt_BR.UTF-8");
    srand(time(NULL));

    imprimirRegras();
    printf("Pressione Enter para continuar...");
    getchar();
    limparTela();

    printf("Digite o \033[1;34mtamanho\033[0m do tabuleiro desejado
(máximo %d): ", MAX);
    scanf("%d", &LINHAS);
    COLUNAS = LINHAS;

    printf("Digite quantas \033[1;35mminas\033[0m deseja esconder:
");
    scanf("%d", &MINAS);

    inicializarTabuleiro();
    limparTela();

    int linha, coluna, resultado;
    char comando;

    while (1) {

        printf("\n");
```

```

        printf("Modo: %s\n", modoMarcar ? "\033[1;34mMarcar\033[0m"
: "\033[1;35mRevelar\033[0m");
        imprimirTabuleiro(0);
        printf("\n");
        printf("Para marcar uma possível mina, digite 'B'.\nDigite
as coordenadas (linha e coluna): ");
        scanf(" %c", &comando);
        limparTela();

        if (comando == 'B' || comando == 'b') {
            modoMarcar = 1;
            limparTela();
            printf("Modo de marcação ativado. Digite as coordenadas
para marcar uma possível mina com uma bandeira.\n");
        } else {

            ungetc(comando, stdin);
            if (scanf("%d %d", &linha, &coluna) != 2) {
                limparTela();
                printf("Coordenadas inválidas!\n");
                getchar();
                continue;
            }

            linha--; coluna--;

            if (modoMarcar) {
                if (visualizacao[linha][coluna] == REVELADA) {
                    printf("Essa posição já foi revelada, não pode
ser marcada!\n");
                } else {
                    visualizacao[linha][coluna] = MARCADA;
                    modoMarcar = 0;
                }
            } else {
                resultado = revelar(linha, coluna);

                if (resultado == -1) {
                    limparTela();
                    imprimirTabuleiro(1);
                }
            }
        }
    }
}

```

```

        printf("\033[1;31mVocê revelou uma mina! Game
Over!\033[0m\n\n");
        break;
    } else if (checarVitoria()) {
        limparTela();
        imprimirTabuleiro(1);
        printf("\033[1;33mPARABÉNS! Você revelou todas
as células sem minas!\033[0m\n");
        break;
    }
}
}
}

return 0;
}

```

- **Descrição:**
Função principal que conecta todas as funcionalidades do jogo e gerencia o loop de jogabilidade.
- **Observações:**
 - O loop principal é o coração do jogo, permitindo a interação contínua do jogador.
- **Linha por linha:**

1. Configurações iniciais

```
setlocale(LC_ALL, "pt_BR.UTF-8");
```

- Configura a localidade para português brasileiro (para que acentuação e caracteres especiais funcionem corretamente no terminal).

```
srand(time(NULL));
```

- Inicializa o gerador de números aleatórios com a semente do tempo atual, garantindo que os números gerados sejam diferentes a cada execução.

2. Exibição das regras

```
imprimirRegras();
```

- Chama a função que imprime as regras do jogo.

```
printf("Pressione Enter para continuar...");
```

```
getchar();
```

- Exibe uma mensagem solicitando que o usuário pressione Enter para continuar e aguarda essa ação.

```
limparTela();
```

- Limpa o terminal (ou console) para preparar para a próxima etapa.

3. Entrada do tamanho do tabuleiro e número de minas

```
scanf("%d", &LINHAS);
```

- Solicita ao usuário que digite o número de linhas do tabuleiro e armazena na variável `LINHAS`.

```
COLUNAS = LINHAS;
```

- Define o número de colunas igual ao número de linhas, criando um tabuleiro quadrado.

```
scanf("%d", &MINAS);
```

- Solicita ao usuário o número de minas que devem ser colocadas no tabuleiro e armazena o valor em `MINAS`.

4. Inicialização do tabuleiro

```
inicializarTabuleiro();
```

- Chama a função que inicializa o tabuleiro, coloca as minas aleatoriamente e calcula os números de minas ao redor.

5. Variáveis para controle do loop principal

```
int linha, coluna, resultado;
```

- `linha, coluna`: Variáveis que armazenam as coordenadas fornecidas pelo usuário para revelar ou marcar uma célula.
- `resultado`: Variável que armazena o resultado da função `revelar()`, que indica se o jogo acabou ou se a célula foi revelada com sucesso.

```
char comando;
```

- `comando`: Variável para armazenar o comando digitado pelo usuário (por exemplo, 'B' para marcar uma mina).

6. Loop principal do jogo

```
while (1)
```

- Loop infinito que continuará até que o jogo termine, seja por vitória ou derrota.

7. Exibição do modo e tabuleiro

```
printf("Modo: %s", modoMarcar ? "Marcar" : "Revelar");
```

- `modoMarcar ? ... : ...`: Imprime qual modo está ativo, ou "Marcar" ou "Revelar", com cores diferentes dependendo do modo.

```
imprimirTabuleiro(0);
```

- Chama a função para exibir o estado atual do tabuleiro. O parâmetro `0` indica que não deve revelar todas as células (apenas as visíveis ao jogador).

8. Solicitação de comando do jogador

```
scanf(" %c", &comando);
```

- Solicita ao usuário um comando, sendo 'B' para marcar uma mina. O espaço antes de `%c` garante que qualquer caractere de nova linha deixado pelo `scanf` anterior seja ignorado.

9. Processamento do comando

```
if (comando == 'B' || comando == 'b')
```

```
    modoMarcar = 1;
```

- Se o comando for 'B', ativa o modo de marcação.

```
else
```

```
    ungetc(comando, stdin);
```

- Se o comando não for 'B', o código volta o caractere de comando para o fluxo de entrada e lê as coordenadas da célula onde o jogador quer agir (revelar ou marcar).

```
    if (scanf("%d %d", &linha, &coluna) != 2) {
```

```
        printf("Coordenadas inválidas!\n");
```

- Caso as coordenadas sejam inválidas, uma mensagem de erro é exibida e o loop recomeça.

```
    linha--; coluna--;
```

- Ajusta para o índice correto.

10. Ações de marcar e revelar

```
if (modoMarcar)
```

```
    if (visualizacao[linha][coluna] == REVELADA)
```

```
    else
```

```
        visualizacao[linha][coluna] = MARCADA;
```

```
        modoMarcar = 0;
```

- **Se no modo de marcar:** Verifica se a célula já foi revelada. Caso contrário, marca a célula como "MARCADA" e desativa o modo de marcação.

```
else
```

```
    resultado = revelar(linha, coluna);
```

```
    if (resultado == -1)
```



```
        imprimirTabuleiro(1);

        printf("Você revelou uma mina! Game Over!");

        break;

    else if (checarVitoria())

        imprimirTabuleiro(1);

        printf("PARABÉNS!");

        break;
```

- **Se no modo de revelar:** Chama a função `revelar()`. Se o resultado for -1 (minas reveladas), o jogo termina com derrota e exibe o tabuleiro final. Se o jogador revelar todas as células sem minas, chama `checarVitoria()` para verificar a vitória e encerra o jogo.

CÓDIGO COMPLETO DO JOGO

Todo o código-fonte deste projeto está disponível publicamente no GitHub, plataforma de armazenamento de códigos de programação, rede de compartilhamento e troca de conhecimento entre os profissionais e estudantes da área: <https://github.com/LeeMendonca/Campo-Minado.git>

CONSIDERAÇÕES FINAIS

O código foi estruturado com funções específicas para cada tarefa, o que melhora a clareza e facilita a manutenção. Essa separação de responsabilidades torna o código mais legível e compreensível, além de permitir ajustes futuros sem afetar outras partes do sistema. A experiência do jogador também foi aprimorada com o uso de cores ANSI, que adicionam um toque visual ao jogo. No entanto, é importante notar que essa funcionalidade pode depender de configurações específicas do terminal utilizado, o que pode limitar a portabilidade em alguns casos.

Uma das limitações do código é a dependência de entradas válidas do jogador, pois o programa não conta com uma validação robusta para garantir que as coordenadas inseridas sejam corretas, o que pode gerar erros se o usuário fornecer dados inválidos. Como melhorias, seria interessante adicionar uma funcionalidade de dica, onde o jogador pudesse revelar uma célula segura, o que facilitaria a jogabilidade para iniciantes. Além disso, seria vantajoso incluir variações de dificuldade, como fácil, médio e difícil, que poderiam ser definidas com base no tamanho do tabuleiro, permitindo ao jogador escolher o grau de desafio desejado. Outra possível melhoria seria permitir que o usuário escolhesse as dimensões do tabuleiro, tornando-o retangular, em vez de fixá-lo como quadrado, oferecendo mais flexibilidade e personalização para o jogador. Essas melhorias poderiam tornar o jogo mais interativo e acessível a diferentes tipos de jogadores, além de aumentar a diversão e o desafio.

REFERÊNCIAS BIBLIOGRÁFICAS

FATECH. Campo Minado - Origem do Jogo. Disponível em: <<https://fatechgirls.com.br/wp-content/uploads/regrasCampoMinado.pdf>>. Acesso em: 26 nov. 2024.

ARKADE. A origem do jogo campo minado e sua evolução. Disponível em: <https://arkade.com.br/a-origem-do-jogo-campo-minado-e-sua-evolucao/#google_vignette>. Acesso em: 26 nov. 2024.

WIKIHOW. Como Jogar Campo Minado. Disponível em: <<https://pt.wikihow.com/Jogar-Campo-Minado>>. Acesso em: 26 nov. 2024.

IBM. Diretivas do pré-processador. Disponível em: <<https://www.ibm.com/docs/pt-br/i/7.5?topic=reference-preprocessor-directives>>. Acesso em: 26 nov. 2024.