Programação Orientada a Objetos 24/25

Trabalho Prático

Daniel Duarte Silva - 2023144551

Guilherme Alexandre Neves Martins - 2023144573

Conteúdo

Mapa.h

- **Construtor:**
    - Inicializa uma instância da classe. O objetivo é garantir que os atributos sejam inicializados de forma consistente, mas sua implementação específica será encontrada no arquivo .cpp.

```
Mapa();
```

- **Destrutor:**
    - Responsável por liberar a memória alocada dinamicamente para a matriz grelha. Isso evita vazamento de memória, especialmente quando o mapa é destruído ou a aplicação é encerrada.

```
~Mapa();
```

- **Inicializar:**
    - Inicializa o mapa com nl linhas e nc colunas.
    - Gera uma matriz bidimensional (grelha) e a preenche com o caractere '.', que pode representar um espaço vazio ou terreno padrão.

```
void inicializar(int nl, int nc);
```

- **Mostrar Mapa**
    - Exibe o mapa no terminal.
    - Percorre a matriz grelha e imprime cada linha e coluna, provavelmente separadas por espaços ou em formato legível.

```
void mostrarMapa() const;
```

- **Atualizar Posição:**
    - Atualiza uma posição específica no mapa com um símbolo (char).
    - Útil para representar elementos dinâmicos no mapa, como jogadores, inimigos ou objetos.

```
void atualizarPosicao(int linha, int coluna, char simbolo);
```

- **Posição Valida**
    - Verifica se uma posição no mapa é válida.

- A posição é considerada válida se for um “.”.

```
bool posicaoValida(int linha, int coluna) const;
```

- **Obter Conteudo:**
  - Retorna o caractere armazenado em uma posição específica da matriz.
  - Permite consultar o conteúdo do mapa sem modificá-lo.

```
char obterConteudo(int linha, int coluna) const;
```

- **Getters:**
  - Métodos simples que retornam o número de linhas e colunas do mapa.
  - Úteis para obter informações sobre o tamanho do mapa sem acessar diretamente os atributos privados.

```
int getLinhas() const { return linhas; }
int getColunas() const { return colunas; }
char** getGrelha() const { return grelha; }
```

Mapa.cpp

- **Construtor**:
  - linhas e colunas são inicializados com 0.
  - grelha é inicializada com nullptr, indicando que nenhuma matriz está alocada no momento.

```
Mapa::Mapa() : linhas(0), colunas(0), grelha(nullptr) {}
```

- **Destrutor**:
  - Libera a memória alocada dinamicamente para a matriz grelha.
  - Se a grelha não for igual a nullptr ent ele percorre todas as linhas da matriz e liberta utilizando delete[]

```
Mapa::~Mapa() {
    if (grelha != nullptr) {
        for (int i = 0; i < linhas; ++i) {
            delete[] grelha[i];
        }
        delete[] grelha;
    }
}
```

- **Iniciar Mapa**:
    - Função:
        - Configura o mapa com dimensões fornecidas pelo file.txt e preenche todas as posições com ‘.’.
    - Parâmetros:
        - nl (Número de linhas)
        - nc (Número de Colunas)
    - Detalhes de Implementação:
        - Atualiza os atributos linhas e colunas.
        - Liberta qualquer memória anteriormente alocada em grelha, caso tenha sido utilizada.
        - Aloca um novo array bidimensional (char**) com dimensões nl x nc.
        - Preenche cada célula da matriz com '.'.

```
void Mapa::inicializar(int nl, int nc) {
    linhas = nl;
    colunas = nc;

    // Liberar memória anterior, se existir
    if (grelha != nullptr) {
        for (int i = 0; i < linhas; ++i) {
            delete[] grelha[i];
        }
        delete[] grelha;
    }

    // Alocar memória para a matriz
    grelha = new char*[linhas];
    for (int i = 0; i < linhas; ++i) {
        grelha[i] = new char[colunas];
        for (int j = 0; j < colunas; ++j) {
            grelha[i][j] = '.'; // Inicializar com '.'
        }
    }
}
```

- **Mostrar Mapa:**
    - Função:
        - Exibe o conteúdo do mapa no terminal.
    - Detalhes de Implementação:
        - Percorre as linhas e colunas de grelha, imprimindo cada carater.
        - Após cada linha, insere uma quebra de linha (endl).

```
void Mapa::mostrarMapa() const {
    for (int i = 0; i < linhas; ++i) {
        for (int j = 0; j < colunas; ++j) {
            cout << grelha[i][j];
        }
        cout << endl;
    }
}
```

- **Atualizar Posição:**
  - Função:
    - Modifica uma posição especifica da grelha
  - Parâmetros:
    - Linha (índice da linha)
    - Coluna (índice da coluna)
    - Símbolo (carater a ser inserido na posição)
  - Detalhes de Implementação:
    - Verifica se as 2 posições estão dentro do limite da matriz
    - Se for válido, atualiza a posição (linha, coluna) com o símbolo fornecido.

```
void Mapa::atualizarPosicao(int linha, int coluna, char simbolo) {
    if (linha >= 0 && linha < linhas && coluna >= 0 && coluna <
colunas) {
        grelha[linha][coluna] = simbolo;
    }
}
```

- **Posição Válida:**
  - Função:
    - Determina se uma posição da matriz é valida
  - Parâmetros:
    - Linha (índice da linha)
    - Coluna (índice da coluna)
  - Retorno:
    - True, se a posição existir e não contenha o carater '+'
    - False, caso contrario
  - Detalhes de Implementação:
    - Verifica se os índices estão dentro dos limites
    - Valida se a célula não contém carater '+'

```
bool Mapa::posicaoValida(int linha, int coluna) const {
    if (linha < 0 || linha >= linhas || coluna < 0 || coluna >=
colunas) {
        return false; // Fora dos limites
    }
    return grelha[linha][coluna] != '+'; // Montanha é inválida
}
```

- **Obter Conteúdo:**
  - Função:
    - Retorna um carater armazenado em uma posição especifica do mapa
  - Parâmetros:
    - Linha (índice das linhas)
    - Coluna (índice das colunas)
  - Retorno:
    - O carater na posição (linhas e colunas) se os índices forem validos
    - ‘0’ se o índices forem inválidos.
  - Detalhes de Implementação:
    - Valida os índices antes de aceder a matriz

```
char Mapa::obterConteudo(int linha, int coluna) const {
    if (linha >= 0 && linha < linhas && coluna >= 0 && coluna <
colunas) {
        return grelha[linha][coluna];
    }
    return '\0'; // Retorna caractere nulo se fora dos limites
}
```

Item.h

Classe Item

- **Atributos Privados:**
  - Tipo – Representa o tipo do item
  - Linha, coluna - Coordenadas do item no mapa
  - Duração - Representa o tempo de duração do item (em turnos).
- **Construtor:**
  - Inicializa os atributos tipo, linha, coluna, e duração.
  - Serve como base para as subclasses, garantindo que todos os itens compartilhem essas propriedades.

```
Item(const std::string& tipo, int linha, int coluna, int duracao);
```

- **Mostrar Detalhes:**
  - Exibe detalhes sobre o item, como tipo, posição e duração.
  - Permite subclasses forneçam sua própria implementação. (Virtual)

```
virtual void mostrarDetalhes() const;
```

- **Atualizar Duração:**
  - Reduz a duração em cada turno (-1), representando o consumo do item ao longo do tempo.

```
void atualizarDuracao();
```

- **Esta Ativo:**
    - Retorna true se duração for >0
    - Retorna false caso contrário

```
bool estaAtivo() const;
```

- **Aplicar Efeito:**
    - Método virtual, obrigando cada subclasse a implementar um efeito para o item
    - Interage com um objeto da classe caravana/barbaro e modifica o numero de moedas

```
virtual void aplicarEfeitoC(Caravana& caravana, int& moedas) = 0;
virtual void aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas) =
0;
```

- **Getters:**
    - getLinha()/getColuna() – retorna as coordenadas do item
    - getTipo() – retorna o tipo do item

```
int getLinha() const { return linha; }
int getColuna() const { return coluna; }
std::string getTipo() const { return tipo; }
```

Subclasses de Item.h

Classe CaixaPandora:

- **Atributos e construtor:**
    - Herda tipo como “CaixaPandora”
    - É colocado no mapa através da linha e coluna
- **Aplicar Efeito:**
    - Implementa um efeito (reduz a tripulação em 20%)

```
class CaixaPandora : public Item {
public:
    CaixaPandora(int linha, int coluna);

    void aplicarEfeitoC(Caravana& caravana, int& moedas) override;
    void aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas)
override;
};
```

Classe ArcaTesouro:

- **Atributos e construtor:**
  - Herda tipo como “ArcaTesouro”
  - É colocado no mapa através da linha e coluna e cotem uma duração
- **Aplicar Efeito:**
  - Implementa um efeito (aumenta o nº de moedas em 10%)

```
class ArcaTesouro : public Item {
public:
    ArcaTesouro(int linha, int coluna, int duracao);

    void aplicarEfeitoC(Caravana& caravana, int& moedas) override;
    void aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas)
override;

};
```

Classe Jaula:

- **Atributos e construtor:**
  - Herda tipo como “Jaula”
  - É colocado no mapa através da linha e coluna e contem duração
- **Aplicar Efeito:**
  - Implementa um efeito (Junta prisioneiros à tripulação sem exceder o limite)

```
class Jaula : public Item {
public:
    Jaula(int linha, int coluna, int duracao);

    void aplicarEfeitoC(Caravana& caravana, int& moedas) override;
    void aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas)
override;

};
```

Classe Mina:

- **Atributos e construtor:**
  - Herda tipo como “Mina”
  - É colocado no mapa através da linha e coluna e contem duração
- **Aplicar Efeito:**
  - Implementa um efeito (Explode e destrói a caravana)

```
class Mina : public Item {
public:
    Mina(int linha, int coluna, int duracao);

    void aplicarEfeitoC(Caravana& caravana, int& moedas) override;
    void aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas)
override;

};
```

Classe Surpresa:

- **Atributos e construtor:**
  - Herda tipo como “Surpresa”
  - É colocado no mapa através da linha e coluna e contem duração
- **Aplicar Efeito:**
  - aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas):

    Este método é responsável por aplicar um efeito quando o bárbaro interage com a surpresa. O efeito pode ser positivo ou negativo:

    Efeito positivo: O bárbaro recebe 10 turnos adicionais e o jogador perde 5 moedas.

    Efeito negativo: O bárbaro perde 5 tripulantes. Se o número de tripulantes for menor ou igual a 0, o bárbaro é destruído.

  - aplicarEfeitoC(Caravana& caravana, int& moedas):

    Este método aplica um efeito quando a caravana interage com a surpresa. O efeito pode ser:

    Efeito positivo: A caravana reabastece sua água e recebe 2 tripulantes adicionais. Além disso, o jogador ganha 10 moedas.

    Efeito negativo: A caravana perde 20 litros de água, 5 toneladas de carga e 10 moedas.

```
class Surpresa : public Item {
public:
    Surpresa(int linha, int coluna, int duracao);

    void aplicarEfeitoC(Caravana& caravana, int& moedas) override;
    void aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas)
override;

};
```

Item.cpp

- **Construtor:**
  - Inicia os atributos da classe

```
Item::Item(const std::string& tipo, int linha, int coluna, int
duracao)
    : tipo(tipo), linha(linha), coluna(coluna), duracao(duracao) {}
```

- **Mostrar Detalhes:**
  - Função:
    - Exibe informação sobre o item (tipo/posição/nº de turnos restantes)

```
void Item::mostrarDetalhes() const {
    std::cout << "Item: " << tipo << "\nPosição: (" << linha << ", "
```

```
<< coluna
              << ")\nDuração: " << duracao << " turnos restantes.\n";
}
```

- **Atualizar Duração:**
  - Função:
    - Decrementa 1 por cada turno (se for >0)

```
void Item::atualizarDuracao() {
    if (duracao > 0) duracao--;
}
```

- **Esta Ativo:**
  - Função:
    - Retorna a duração se for >0

```
abool Item::estaAtivo() const {
    return duracao > 0;
}
```

Subclasses de item.cpp

- **CaixaPandora:**
  - Efeito:
    - Reduz o número de tripulantes da caravana
  - Detalhes de Implementação:
    - Calcula a perda de 20% dos tripulantes
    - Atualiza o nº de tripulantes.

```
void CaixaPandora::aplicarEfeitoC(Caravana& caravana, int& moedas) {
    int perda = caravana.getTripulantes() * 0.2;
    caravana.adicionaTripulantes(-perda);
    std::cout << "Caixa de Pandora: Caravana perdeu " << perda << "
tripulantes!\n";
}

void CaixaPandora::aplicarEfeitoB(CaravanaBarbara& barbaro, int&
moedas) {
    int perda = barbaro.getTripulantes() * 0.2;
    barbaro.adicionaTripulantes(-perda);
    std::cout << "Caixa de Pandora: Caravana Barbara perdeu " << perda
<< " tripulantes!\n";
}
```

- **ArcaTesouro:**
  - Efeito:
    - Aumenta o nº de moedas em 10%
  - Detalhes de Implementação:
    - Calcula o ganho de 10% em relação ao nº de moedas atuais
    - Adiciona o ganho a moedas

```
void ArcaTesouro::aplicarEfeitoC(Caravana& caravana, int& moedas) {
    int ganho = moedas * 0.1;
    moedas += ganho;
    std::cout << "Arca do Tesouro: Ganhaste " << ganho << "
moedas!\n";
}
void ArcaTesouro::aplicarEfeitoB(CaravanaBarbara& barbaro, int&
moedas) {
    int ganho = moedas * 0.1;
    moedas += ganho;
    std::cout << "Arca do Tesouro: Ganhaste " << ganho << "
moedas!\n";
}
```

- **Jaula:**
  - Efeito:
    - Adiciona tripulantes à caravana
  - Detalhes de Implementação:
    - Adiciona o ganho ao nº de tripulantes (5, definido pelo utilizador)

```
void Jaula::aplicarEfeitoC(Caravana& caravana, int& moedas) {
    int ganho = 5;
    caravana.adicionaTripulantes(ganho);
    std::cout << "Jaula: Adicionou " << ganho << " tripulantes à
caravana!\n";
}

void Jaula::aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas) {
    int ganho = 5;
    barbaro.adicionaTripulantes(ganho);
    std::cout << "Jaula: Adicionou " << ganho << " tripulantes à
Caravana Barbara!\n";
}
```

- **Mina:**
  - Efeito:
    - Destrói uma caravana
  - Detalhes de Implementação:
    - Exibe uma mensagem avisar que a caravana foi destruida

```
void Mina::aplicarEfeitoC(Caravana& caravana, int& moedas) {
    std::cout << "Mina: A caravana foi destruída!\n";
}

void Mina::aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas) {
    barbaro.setDestruida(true);
```

```
    std::cout << "Mina: A caravana foi destruída!\n";
}
```

- Supresa:
    - Efeito:
        - Bárbaros:
          Efeito positivo: O bárbaro recebe 10 turnos adicionais e o jogador perde 5 moedas.
          Efeito negativo: O bárbaro perde 5 tripulantes. Se o número de tripulantes for menor ou igual a 0, o bárbaro é destruído.
        - Caravanas:
          Efeito positivo: A caravana reabastece sua água e recebe 2 tripulantes adicionais. Além disso, o jogador ganha 10 moedas.
          Efeito negativo: A caravana perde 20 litros de água, 5 toneladas de carga e 10 moedas.

```
void Surpresa::aplicarEfeitoC(Caravana& caravana, int& moedas) {
    int efeito = rand() % 2; // Determina se o efeito será positivo ou
negativo

    if (efeito == 0) {
        // Efeito positivo
        cout << "Surpresa! Caravana (ID: " << caravana.getID() << ")
encontrou algo útil.\n";
        caravana.reabastecerAgua();
        caravana.adicionaTripulantes(2); // Adiciona 2 tripulantes
        moedas += 10; // Ganha 10 moedas
        cout << "Caravana reabastecida, +2 tripulantes, +10
moedas.\n";
    } else {
        // Efeito negativo
        cout << "Surpresa! Caravana (ID: " << caravana.getID() << ")
encontrou algo perigoso.\n";
        caravana.setAgua(caravana.getAgua()-20); // Reduz 20 litros de
água
        caravana.adicionaCarga(- 5); // Danifica 5 toneladas de carga
        moedas -= 10; // Perde 10 moedas
        cout << "Caravana perdeu 20 litros de água, 5 toneladas de
carga, e 10 moedas.\n";
    }
}

void Surpresa::aplicarEfeitoB(CaravanaBarbara& barbaro, int& moedas) {
    int efeito = rand() % 2; // Determina se o efeito será útil ou
perigoso

    if (efeito == 0) {
        // Efeito positivoconfig
        cout << "Surpresa! Bárbaro encontrou algo útil.\n";
        barbaro.setTurnosRestantes(barbaro.getTurnosRestantes() + 10);
// Aumenta 10 turnos restantes
        moedas -= 5; // Jogador perde 5 moedas devido ao ganho do
bárbaro
        cout << "Bárbaro ganhou 10 turnos. Jogador perdeu 5
```

```
moedas.\n";
    } else {
        // Efeito negativo
        cout << "Surpresa! Bárbaro encontrou algo perigoso.\n";
        barbaro.adicionaTripulantes(-5); // Reduz 5 bárbaros
        if (barbaro.getTripulantes() <= 0) {
            barbaro.setDestruida(true); // Destrói o bárbaro se não
houver mais tripulantes
            cout << "Bárbaro foi destruído pela surpresa!\n";
        } else {
            cout << "Bárbaro perdeu 5 tripulantes.\n";
        }
    }
}
```

Caravana.h

Classe Caravana

- **Atributos:**
  - id – id da caravana
  - tipo – tipo da caravana (comercio/militar…)
  - linha / coluna – posição atual da caravana no mapa
  - tripulantes – Nº atual de tripulantes na caravana
  - agua – Quantidade de água disponível
  - carga – Quantidade de carga transportada
  - maxTripulantes/ maxAgua / maxCarga – Limites máximo para cada atributo
  - instantesSemTripulantes – Quantidade de turnos desde que a caravana ficou sem tripulantes
  - automático – Indica se a caravana opera de forma automática
  - destruída – Indica se a caravana esta destruida
  - ultimaDirecao – Memoriza a última direção de movimento.
  - turnos – Contador de turnos
- **Construtor:**
  - Cria uma caravana com os valores fornecidos, garantindo:
    - A posição inicial esteja dentro dos limites
    - Os recursos (água, carga, etc) comecem vazios ou no máx permitido)

```
Caravana(int id, const string& tipo, int linha, int coluna,
         int maxTripulantes, int maxAgua, int maxCarga);
```

- **Destrutor:**
  - Destrói uma caravana

```
virtual ~Caravana() {}
```

- **Mover:**

- Move a caravana numa determinada direção, garantindo que esta sempre dentro dos limites do mapa e atualiza as posicões
- Permite subclasses forneçam sua própria implementação. (Virtual)

```
virtual void mover(const string& direcao, int linhasMapa, int
colunasMapa, Mapa& mapa);
```

- **Mostrar Detalhes:**
  - Imprime os atributos atuais, como posição, quantidade de recursos, estado da tripulação, etc.
  - Permite subclasses forneçam sua própria implementação. (Virtual)

```
virtual void mostrarDetalhes() const;
```

- **Atualizar Água:**
  - Método abstrato que obriga as subclasses a implementar sua própria lógica de consumo de água

```
virtual void atualizarAgua() = 0;
```

- **Carregar:**
  - Adiciona ou remove carga respeitando os limites máximos

```
void carregar(int quantidade);
```

- **Getters:**
  - Vai buscar os atributos pretendidos

```
int getID() const;
int getLinha() const;
int getColuna() const;
int getAgua() const { return agua; }
bool getAutomatico() const { return automatico; }
int getTripulantes() const { return tripulantes; }
int getMaxTripulantes() const { return maxTripulantes; }
int getCarga() const { return carga; }
int getMaxCarga() const { return maxCarga; }
int getInstantesSemTripulantes() const { return
instantesSemTripulantes; }
int getMaxAgua() const { return maxAgua; }
string getUltDir() const { return ultimaDirecao; }
bool getDestruida() const {return destruida;}
string getTipo() const {return tipo;}
```

- **Setters:**
  - Altera os atributos pretendidos

```
void setDestruida(bool b){destruida = b;}
void setTripulantes(int quantidade) { tripulantes = quantidade; }
void setCarga(int quantidade) { carga = quantidade; }
void setAgua(int quantidade) { agua = quantidade; }
void setPosicao(int l , int c){
```

```
    linha = l;
    coluna = c;
}
```

- **Reabastecer Água:**
    - Restaura o recurso de água ao valor máximo (maxAgua)

```
void reabastecerAgua();
```

- **Altera Automático:**
    - Alterna entre True ou False

```
void alteraAutomatico(bool at) { automatico = at; }
```

- **Adiciona Tripulantes:**
    - Adiciona ou remove tripulantes, verificando o limite maxTripulantes.
    - Retorna false se a operação falhar

```
bool adicionaTripulantes(int quantidade);
```

- **Sem Tripulantes:**
    - Retorna true se o nº de tripulantes for 0
    - Retorna false caso contrario

```
bool semTripulantes() const;
```

- **Adiciona Turno:**
    - Incrementa 1 ao contador de turnos

```
void adicionaTurno() { turnos++; }
```

- **Adiciona Carga:**
    - Adiciona carga à caravana, retorna true se esta operação for concluída
    - Retorna false caso contrario

```
bool adicionaCarga(int quantidade);
```

Subclasse de Caravana

Classe CaravanaComercio:

- **Construtor:**
    - Inicia a caravana com configurações especificas do tipo “Comércio”

```
CaravanaComercio(int id, int linha, int coluna);
```

- **Atualizar Água:**
    - Consome água em uma proporção que reflete a carga e o número de tripulantes.

```
void atualizarAgua() override;
```

- **Tempestade de Areia:**
  - Aplica penalidades, como redução de água e perda de tripulantes

```
void tempestadeAreia() override;
```

- **Movimento sem Tripulantes:**
  - Verifica se a caravana pode se mover sem tripulantes

```
bool movimentoSemTripulantes(int linhasMapa, int colunasMapa, Mapa&
mapa) override;
```

- **Movimento Autónomo:**

```
bool movimentoAutonomo(const vector<CaravanaBarbara*>& barbaros, const
vector<Caravana*>& outrasCaravanas, const vector<Item*>& items, int
linhasMapa, int colunasMapa, Mapa& mapa) override;
```

- **Turnos Excedidos:**

```
bool turnosExcedidos() const override;
```

Classe CaravanaMilitar:

- **Construtor:**
  - Inicia a caravana com configurações específicas para o tipo "Militar".

```
CaravanaMilitar(int id, int linha, int coluna);
```

- **Atualizar Água:**
  - Consome água baseada no número de tripulantes e nos turnos em atividade.

```
void atualizarAgua() override;
```

- **Tempestade de Areia:**
  - Aplica penalidades, como redução de água e perda de tripulantes

```
void tempestadeAreia() override;
```

- **Movimento sem Tripulantes:**
  - Verifica se a caravana pode se mover sem tripulantes

```
bool movimentoSemTripulantes(int linhasMapa, int colunasMapa, Mapa&
mapa) override;
```

- **Movimento Autónomo:**

```
bool movimentoAutonomo(const vector<CaravanaBarbara*>& barbaros, const
vector<Caravana*>& outrasCaravanas, const vector<Item*>& items, int
linhasMapa, int colunasMapa, Mapa& mapa) override;
```

- **Turnos Excedidos:**

```
bool turnosExcedidos() const override;
```

Classe CaravanaSecreta:

- **Construtor:**
  - Inicia a caravana com configurações específicas para o tipo "Militar".

```
CaravanaMilitar(int id, int linha, int coluna);
```

- **Atualizar Água:**
  - Consome água baseada no número de tripulantes e nos turnos em atividade.

```
void atualizarAgua() override;
```

- **Tempestade de Areia:**
  - Aplica penalidades, como redução de água e perda de tripulantes

```
void tempestadeAreia() override;
```

- **Movimento sem Tripulantes:**
  - Verifica se a caravana pode se mover sem tripulantes

```
bool movimentoSemTripulantes(int linhasMapa, int colunasMapa, Mapa&
mapa) override;
```

- **Movimento Autónomo:**

```
bool movimentoAutonomo(const vector<CaravanaBarbara*>& barbaros, const
vector<Caravana*>& outrasCaravanas, const vector<Item*>& items, int
linhasMapa, int colunasMapa, Mapa& mapa) override;
```

- **Turnos Excedidos:**

```
bool turnosExcedidos() const override;
```

Caravana.cpp

Classe Caravana:

- **Construtor:**
  - Atribui valores aos atributos da caravana

```
Caravana::Caravana(int id, const string& tipo, int linha, int coluna,
                   int maxTripulantes, int maxAgua, int maxCarga)
    : id(id), tipo(tipo), linha(linha), coluna(coluna),
      tripulantes(maxTripulantes), agua(maxAgua), carga(0),
      maxTripulantes(maxTripulantes), maxAgua(maxAgua),
maxCarga(maxCarga),
      instantesSemTripulantes(0), automatico(false),
```

```
ultimaDirecao(""),
      turnos(0) {}
```

- **Mostrar Detalhes:**
    - Função:
        - Imprime as informações

```
void Caravana::mostrarDetalhes() const {
    cout << "Caravana (ID: " << id << ")\n";
    cout << "Tipo: " << tipo << "\n";
    cout << "Posição: (" << linha << ", " << coluna << ")\n";
    cout << "Tripulantes: " << tripulantes << "/" << maxTripulantes <<
"\n";
    cout << "Carga: " << carga << "/" << maxCarga << " toneladas\n";
    cout << "Água: " << agua << "/" << maxAgua << " litros\n";
}
```

- **Mover:**

Objetivo:

Esta função permite movimentar a caravana em diferentes direções dentro de um mapa. O movimento pode ser feito nas direções cardeais (cima, baixo, esquerda, direita) e nas diagonais (cima-esquerda, cima-direita, baixo-esquerda, baixo-direita).

Parâmetros:

direcao (string): Comando que define a direção para a qual a caravana se moverá. Pode ser uma das seguintes opções:

"C" - Cima

"B" - Baixo

"E" - Esquerda

"D" - Direita

"CE" - Cima e Esquerda (diagonal)

"CD" - Cima e Direita (diagonal)

"BE" - Baixo e Esquerda (diagonal)

"BD" - Baixo e Direita (diagonal)

linhasMapa (int): O número total de linhas do mapa (utilizado para garantir que o movimento não ultrapasse os limites do mapa).

colunasMapa (int): O número total de colunas do mapa (semelhante ao parâmetro anterior, mas para colunas).

mapa (Mapa&): A referência ao objeto Mapa que contém as informações de posições e obstáculos do mapa, que será utilizado para verificar a validade do movimento e para atualizar as posições no mapa.

Detalhes da Implementação:

Verificação do Comando de Direção:

A função começa identificando qual comando de direção foi passado. Isso é feito comparando a string direcao com as opções possíveis.

Cálculo da Nova Posição:

Com base no comando de direção, a função calcula a nova posição da caravana.

Para direções cardeais:

"C" (Cima): Decrease a linha (linha - 1).

"B" (Baixo): Aumenta a linha (linha + 1).

"E" (Esquerda): Decrease a coluna (coluna - 1).

"D" (Direita): Aumenta a coluna (coluna + 1).

Para direções diagonais, a função ajusta tanto a linha quanto a coluna de acordo com o comando (ex: "CE" move a caravana para cima e para a esquerda, reduzindo tanto a linha quanto a coluna).

Garantia de Movimento Dentro dos Limites do Mapa (Uso de mod):

A operação mod (resto da divisão) é usada para garantir que as novas coordenadas não ultrapassem os limites do mapa. Isso torna o movimento cíclico, ou seja, se a caravana sair por um lado do mapa, ela entra pelo lado oposto.

Verificação de Posição Válida:

Antes de realizar o movimento, a função chama o método mapa.posicaoValida() para verificar se a nova posição calculada é válida (se não há obstáculos ou restrições para o movimento naquele local).

Atualização da Posição no Mapa:

Se a nova posição for válida, a função:

Remove a caravana da posição antiga (marcando-a como vazia '.').

Atualiza a nova posição no mapa, colocando o ID da caravana.

Exibe uma mensagem confirmando a movimentação.

```
• void Caravana::mover(const string& direcao, int linhasMapa, int
  colunasMapa, Mapa& mapa) {
      ultimaDirecao = direcao;

      int novaLinha = linha;
      int novaColuna = coluna;

      // Calculando as novas posições com base na direção
      if (direcao == "C") {
          novaLinha = (linha - 1 + linhasMapa) % linhasMapa;
      } else if (direcao == "B") {
          novaLinha = (linha + 1) % linhasMapa;
      } else if (direcao == "E") {
          novaColuna = (coluna - 1 + colunasMapa) % colunasMapa;
      } else if (direcao == "D") {
          novaColuna = (coluna + 1) % colunasMapa;
      } else if (direcao == "CE") {
          novaLinha = (linha - 1 + linhasMapa) % linhasMapa;
          novaColuna = (coluna - 1 + colunasMapa) % colunasMapa;
      } else if (direcao == "CD") {
          novaLinha = (linha - 1 + linhasMapa) % linhasMapa;
          novaColuna = (coluna + 1) % colunasMapa;
      } else if (direcao == "BE") {
          novaLinha = (linha + 1) % linhasMapa;
          novaColuna = (coluna - 1 + colunasMapa) % colunasMapa;
      } else if (direcao == "BD") {
          novaLinha = (linha + 1) % linhasMapa;
          novaColuna = (coluna + 1) % colunasMapa;
      }

      // Verificar se a nova posição é válida
      if (mapa.posicaoValida(novaLinha, novaColuna)) {
          mapa.atualizarPosicao(linha , coluna , '.');
          linha = novaLinha;
          coluna = novaColuna;
          mapa.atualizarPosicao(linha, coluna, getCharID());
          cout << "Caravana " << id << " movida para (" << linha
  << ", " << coluna << ") na direção " << direcao << ".\n";
      }

  }
      coluna = (coluna + 1) % colunasMapa;
  } else {
      cerr << "Direção inválida: " << direcao << endl;
  }

  cout << "Caravana " << id << " movida para (" << linha << ", " <<
coluna << ") na direção " << direcao << ".\n";
}
```

- **Carregar:**
  - Função:
    - Adiciona carga à caravana, respeitando o limite máximo.
  - Parâmetros:
    - Quantidade – quantidade que irá ser adicionada à carga
  - Detalhes de implementação:
    - Verifica se a carga mais a quantidade ultrapassam o valor máximo:
      - Caso não aconteça, é somado a quantidade a carga
      - Caso aconteça, imprime uma mensagem na consola

```
void Caravana::carregar(int quantidade) {
    if (carga + quantidade <= maxCarga) {
        carga += quantidade;
    } else {
        cout << "Capacidade de carga excedida!\n";
    }
}
```

- **Getters:**
  - Função:
    - Retorna os atributos pretendidos

```
int Caravana::getID() const {
    return id;
}

int Caravana::getLinha() const {
    return linha;
}

int Caravana::getColuna() const {
    return coluna;
}
```

- **Reabastecer:**
  - Função:
    - Reabastece a água para o seu máximo

```
void Caravana::reabastecerAgua() {
    agua = maxAgua;
}
```

- **Adiciona Tripulantes:**
  - Função:
    - Adiciona uma quantidade especifica à tripulação, sem ultrapassar o limite máximo
  - Parâmetros:
    - Quantidade – Nº de tripulantes a serem adicionados
  - Detalhes de implementação:
    - Verifica se a quantidade a adicionar mais o nº de tripulantes existentes não ultrapassa o máximo

```
bool Caravana::adicionaTripulantes(int quantidade) {
    if (tripulantes + quantidade <= maxTripulantes) {
        tripulantes += quantidade;
        if(tripulantes + quantidade  <= 0){tripulantes = 0;}
        return true;
    }

    return false;
}
```

- **Sem Tripulantes:**
  - Função:
    - Só retorna se o nº de tripulantes for 0

```
bool Caravana::semTripulantes() const {
    return tripulantes <= 0;
}
```

- **Adiciona Carga:**
  - Função:
    - Adiciona uma quantidade especifica à carga, sem ultrapassar o limite máximo
  - Parâmetros:
    - Quantidade – quantidade de carga a adicionar
  - Detalhes de Implementação:
    - Verifica se a quantidade mais a carga atual não ultrapassa o valor pretendido:
      - Caso isso aconteça, soma a quantidade à carga atual e retorna true
      - Retorna false caso contrário

```
bool Caravana::adicionaCarga(int quantidade) {
    if (carga + quantidade <= maxCarga) {
        carga += quantidade;
        return true;
    }
    return false;
}
```

Subclasses de Caravana:

Classe CarvanaComercio:

- **Construtor:**
  - Inicia uma caravana do tipo Comercio com alguns com valores já atribuídos

```
CaravanaComercio::CaravanaComercio(int id, int linha, int coluna)
    : Caravana(id, "comercio", linha, coluna, 20, 200, 40) {}
```

- **Atualizar Água:**
  - Função:
    - Atualizar a quantidade de água, por cada turno
  - Detalhes de Implementação:
    - Consome água dependendo da quantidade de tripulantes:
      - Sem tripulantes: não consome água
      - Até a metade da capacidade de tripulantes: Consome 1 litro por turno
      - Mais de metade: Consome 2 litros por turno
      - Quando acaba a água, a caravana perde tripulantes progressivamente.

```
void CaravanaComercio::atualizarAgua() {
    if (getAgua() > 0) {
        if (getTripulantes() == 0) {
            cout << "Caravana de Comércio (ID: " << getID() << ") não
gasta água (sem tripulantes).\n";
        } else if (getTripulantes() <= getMaxTripulantes() / 2) {
            setAgua(getAgua() - 1);
        } else {
            setAgua(getAgua() - 2);
        }
        if (getAgua() < 0) setAgua(0);
    }

    if (getAgua() == 0 && getTripulantes() > 0) {
        adicionaTripulantes(-1);
    }
}
```

- **Tempestade Areia:**
  - Função:
    - Aplicar efeitos durante o jogo
  - Detalhes de Implementação:
    - Se a carga ultrapassa 50% da capacidade, há maior chance (50%) de destruição completa
    - Caso sobreviva, perde 25% da carga

```
void CaravanaComercio::tempestadeAreia() {
    float ocupacaoCarga = static_cast<float>(getCarga()) /
getMaxCarga();
    int probabilidade = (ocupacaoCarga > 0.5) ? 50 : 25;

    int sorte = rand() % 100;

    if (sorte < probabilidade) {
        cout << "Caravana de Comércio (ID: " << getID() << ") foi
destruída por uma tempestade de areia!\n";
        setTripulantes(0);
    } else {
        int perdaCarga = static_cast<int>(getCarga() * 0.25);
        adicionaCarga(-perdaCarga);
        cout << "Caravana de Comércio (ID: " << getID() << ")
sobreviveu à tempestade, mas perdeu "
             << perdaCarga << " toneladas de carga.\n";
    }
}
```

- **Movimento sem Tripulantes:**
  - Função:
    - Quando está sem tripulantes, move-se automaticamente de forma aleatória durante 5 turnos
  - Parâmetros:
    - linhasMapa – índice de linhas atual
    - colunasMapa – índice de colunas atual
  - Detalhes de Implementação:
    - Incrementar o contador de turnos sem tripulantes
    - Sortear um nº de 1 a 8
    - Pegar na posição no array e aplicar a movimentação na caravana

```
bool CaravanaComercio::movimentoSemTripulantes(int linhasMapa, int
colunasMapa, Mapa& mapa) {
    setInstantesSemTripulantes(getInstantesSemTripulantes() + 1);
    int direcao = rand() % 8;
    string direcoes[] = {"C", "B", "E", "D", "CE", "CD", "BE", "BD"};
    mover(direcoes[direcao], linhasMapa, colunasMapa, mapa);
    return getInstantesSemTripulantes() >= 5;
}
```

- **Movimento Autonomo:**

  Objetivo:

  Permite que a caravana de comércio se mova autonomamente em direção a um item próximo, a uma caravana inimiga, ou, caso contrário, faça um movimento aleatório.

  Passos da Implementação:

  1. Proximidade de Itens:
     - A caravana verifica se há itens a até 2 casas de distância e move-se para o mais próximo.
  2. Proximidade de Caravanas:
     - Se não houver itens, a caravana encontra a caravana mais próxima e aproxima-se dela.
  3. Movimento Aleatório:
     - Se nem itens nem caravanas estiverem perto, a caravana move-se aleatoriamente em uma das 8 direções possíveis.

  Parâmetros:

  - **barbaros (vector<CaravanaBarbara>&)**: Lista de caravanas de bárbaros.
  - **outrasCaravanas (vector<Caravana>&)**: Lista de outras caravanas no mapa.
  - **items (vector<Item>&)**: Lista de itens no mapa.
  - linhasMapa (int) e colunasMapa (int): Dimensões do mapa.
  - mapa (Mapa&): Objeto que representa o mapa, usado para validar e atualizar posições.

  Descrição:

  1. Itens Próximos: A caravana move-se em direção a um item, se estiver dentro de 2 casas de distância.
  2. Caravanas Inimigas: A caravana aproxima-se da caravana inimiga mais próxima.
  3. Movimento Aleatório: Se não houver itens nem caravanas próximas, a caravana faz um movimento aleatório.

```
Bool CaravanaComercio::movimentoAutonomo(const
vector<CaravanaBarbara*>& barbaros, const vector<Caravana*>&
outrasCaravanas, const vector<Item*>& items, int linhasMapa, int
colunasMapa, Mapa& mapa) {
    // Verificar itens próximos (até 2 posições de distância)
    for (auto& item : items) {
        if (abs(getLinha() - item->getLinha()) <= 2 && abs(getColuna()
- item->getColuna()) <= 2) {
            int novaLinha = getLinha();
            int novaColuna = getColuna();
            int linhaAntiga = getLinha();
            int colunaAntiga = getColuna();

            // Calcular movimento em direção ao item
```

```
            if (item->getLinha() > novaLinha) novaLinha++;
            else if (item->getLinha() < novaLinha) novaLinha--;

            if (item->getColuna() > novaColuna) novaColuna++;
            else if (item->getColuna() < novaColuna) novaColuna--;

            if(mapa.posicaoValida(novaColuna , novaColuna)){
                mapa.atualizarPosicao(linhaAntiga , colunaAntiga
,'.');
                setPosicao(novaLinha, novaColuna);
                mapa.atualizarPosicao(novaLinha , novaColuna
,getCharID());
            }
            return true; // Movimento feito em direção ao item
        }
    }

    // Encontrar a caravana mais próxima
    Caravana* alvo = nullptr;
    int menorDistancia = linhasMapa + colunasMapa; // Distância máxima
inicial

    for (auto& outraCaravana : outrasCaravanas) {
        if (outraCaravana->getID() != getID()) {
            int distancia = abs(getLinha() - outraCaravana-
>getLinha()) + abs(getColuna() - outraCaravana->getColuna());
            if (distancia < menorDistancia) {
                menorDistancia = distancia;
                alvo = outraCaravana;
            }
        }
    }

    // Aproximar-se da caravana mais próxima
    if (alvo) {
        int linhaAntiga = getLinha();
        int colunaAntiga = getColuna();
        int novaLinha = getLinha();
        int novaColuna = getColuna();

        // Calcular movimento em direção à caravana alvo
        if (alvo->getLinha() > novaLinha) novaLinha++;
        else if (alvo->getLinha() < novaLinha) novaLinha--;

        if (alvo->getColuna() > novaColuna) novaColuna++;
        else if (alvo->getColuna() < novaColuna) novaColuna--;

        if(mapa.posicaoValida(novaColuna , novaColuna)){
            mapa.atualizarPosicao(linhaAntiga , colunaAntiga ,'.');
            setPosicao(novaLinha, novaColuna);
            mapa.atualizarPosicao(novaLinha , novaColuna
,getCharID());
        }
        return true; // Movimento feito em direção à caravana
    }

    // Se não houver caravana para se aproximar, faz movimento
aleatório
    string direcoes[] = {"C", "B", "E", "D", "CE", "CD", "BE", "BD"};
    string direcao = direcoes[rand() % 8];
    mover(direcao, linhasMapa, colunasMapa, mapa);
```

```
    return true;
}
```

Classe CaravanaMilitar:

- **Construtor:**
  - Inicia uma caravana do tipo Militar com alguns com valores já atribuídos

```
CaravanaMilitar::CaravanaMilitar(int id, int linha, int coluna)
    : Caravana(id,"militar",linha, coluna, 40, 400, 5) {}
```

- **Atualizar Água:**
  - Função:
    - Atualizar a quantidade de água, por cada turno
  - Detalhes de Implementação:
    - Consome água dependendo da quantidade de tripulantes:
      - Sem tripulantes: não consome água
      - Até a metade da capacidade de tripulantes: Consome 1 litro por turno
      - Mais de metade: Consome 3 litros por turno
      - Quando acaba a água, a caravana perde tripulantes progressivamente.

```
void CaravanaMilitar::atualizarAgua() {
    if (getAgua() > 0) {
        if (getTripulantes() <= getMaxTripulantes() / 2) {
            setAgua(getAgua() - 1);
        } else {
            setAgua(getAgua() - 3);
        }
        if (getAgua() < 0) setAgua(0);
    }

    if (getAgua() == 0 && getTripulantes() > 0) {
        setTripulantes(getTripulantes() - 1);
    }
}
```

- **Tempestade Areia:**
  - Função:
    - Aplicar efeitos durante o jogo
  - Detalhes de Implementação:
    - Perde 10% dos tripulantes em qualquer situação
    - E há uma chance < 33% de a caravana ser distruida

```
void CaravanaMilitar::tempestadeAreia() {
    int perdaTripulantes = static_cast<int>(getTripulantes() * 0.1);
    setTripulantes(getTripulantes() - perdaTripulantes);

    int sorte = rand() % 100;
    if (sorte < 33) {
        cout << "Caravana Militar (ID: " << getID() << ") foi
```

```
destruída por uma tempestade de areia!\n";
        setTripulantes(0);
    } else {
        cout << "Caravana Militar (ID: " << getID() << ") sobreviveu à
tempestade.\n";
    }
}
```

- **Movimento sem Tripulantes:**
  - Função:
    - Quando está sem tripulantes, move-se automaticamente de forma aleatória durante 7 turnos
  - Parâmetros:
    - linhasMapa – índice de linhas atual
    - colunasMapa – índice de colunas atual
  - Detalhes de Implementação:
    - Move-se automaticamente utilizando sempre a última direção

```
bool CaravanaMilitar::movimentoSemTripulantes(int linhasMapa, int
colunasMapa, Mapa& mapa) {
    setInstantesSemTripulantes(getInstantesSemTripulantes() + 1);
    mover(getUltDir(), linhasMapa, colunasMapa, mapa);
    return getInstantesSemTripulantes() >= 7;
}
```

- **Movimento Autonomo:**

Objetivo:

Permitir que a caravana militar se mova autonomamente em direção ao bárbaro mais próximo, se ele estiver dentro de um raio de 6 casas, de forma a realizar um movimento estratégico.

Passos da Implementação:

1. Encontrar o Bárbaro Mais Próximo:
   - A função calcula a distância de Manhattan entre a caravana e cada bárbaro.
   - O objetivo é encontrar o bárbaro mais próximo que esteja dentro de um raio de 6 casas.
2. Mover-se em Direção ao Bárbaro:
   - Se um bárbaro estiver dentro do raio de 6 casas, a caravana se move uma casa na direção do bárbaro (para cima, para baixo, para a esquerda ou para a direita).
3. Verificar Validade do Movimento:
   - O movimento é feito apenas se a nova posição for válida no mapa.
4. Retorno:
   - Se o movimento foi realizado com sucesso, retorna true. Caso contrário, retorna false, indicando que não há bárbaros próximos ou o movimento é inválido.

Parâmetros:

- *barbaros (vector<CaravanaBarbara>&)**: Lista de caravanas de bárbaros.
- *outrasCaravanas (vector<Caravana>&)**: Lista de outras caravanas no mapa (não usada nesta função).
- *items (vector<Item>&)**: Lista de itens no mapa (não usada nesta função).
- linhasMapa (int) e colunasMapa (int): Dimensões do mapa (não usadas diretamente nesta função).
- mapa (Mapa&): Objeto que representa o mapa, utilizado para validar e atualizar a posição da caravana.

```
bool CaravanaMilitar::movimentoAutonomo(const
vector<CaravanaBarbara*>& barbaros, const vector<Caravana*>&
outrasCaravanas, const vector<Item*>& items, int linhasMapa, int
colunasMapa, Mapa& mapa) {
    CaravanaBarbara* barbaroMaisProximo = nullptr;
    int menorDistancia = INT_MAX;

    // Encontrar o bárbaro mais próximo dentro do raio de 6 casas
    for (auto& barbaro : barbaros) {
        int distancia = abs(getLinha() - barbaro->getLinha()) +
abs(getColuna() - barbaro->getColuna());
        if (distancia <= 6 && distancia < menorDistancia) {
            menorDistancia = distancia;
            barbaroMaisProximo = barbaro;
        }
    }

    // Se encontrou um bárbaro dentro do raio
    if (barbaroMaisProximo != nullptr) {

        int novaLinha = getLinha();
        int novaColuna = getColuna();
        int linhaAnt = getLinha();
        int colunaAnt = getColuna();
        // Mover na direção do bárbaro mais próximo
        if (barbaroMaisProximo->getLinha() > novaLinha) {
            novaLinha++; // Move uma casa para baixo
        } else if (barbaroMaisProximo->getLinha() < novaLinha) {
            novaLinha--; // Move uma casa para cima
        }

        if (barbaroMaisProximo->getColuna() > novaColuna) {
            novaColuna++; // Move uma casa para a direita
        } else if (barbaroMaisProximo->getColuna() < novaColuna) {
            novaColuna--; // Move uma casa para a esquerda
        }

        // Verificar se a nova posição é válida no mapa
        if (mapa.posicaoValida(novaLinha, novaColuna)) {
            mapa.atualizarPosicao(linhaAnt , colunaAnt , '.');
            setPosicao(novaLinha, novaColuna);
            mapa.atualizarPosicao(novaLinha , novaColuna
,getCharID());
            return true; // Movimento realizado
        }
```

```
    }

    return false; // Não há bárbaros próximos ou movimento inválido
}
```

Classe CaravanaSecreta:

- **Construtor:**
  - Inicia uma caravana do tipo Militar com alguns com valores já atribuídos

```
CaravanaSecreta::CaravanaSecreta(int id, int linha, int coluna)
    : Caravana(id, "Secreta", linha, coluna, 10, 300, 20) {}
```

- **Atualizar Água:**
  - Função:
    - Atualizar a quantidade de água, por cada turno
  - Detalhes de Implementação:
    - Consome água dependendo da quantidade de tripulantes:
      - Sem tripulantes: não consome água
      - Mais de metade: Consome 3 litros por turno
      - Quando acaba a água, a caravana perde tripulantes progressivamente.

```
void CaravanaSecreta::atualizarAgua() {
    if (getAgua() > 0) {
        if (getTripulantes() == 0) {
            cout << "Caravana Secreta (ID: " << getID() << ") não
gasta água (sem tripulantes).\n";
        } else {
            setAgua(getAgua() - 3); // Consome mais água que outras
caravanas
            cout << "Caravana Secreta (ID: " << getID() << ") consumiu
3 litros de água.\n";
        }

        if (getAgua() < 0) setAgua(0);
    }

    if (getAgua() == 0 && getTripulantes() > 0) {
        adicionaTripulantes(-1); // Perde um tripulante por turno sem
água
    }
}
```

- **Tempestade Areia:**
  - Função:
    - Aplicar efeitos durante o jogo
  - Detalhes de Implementação:
    - Perde 20 litro de água
    - E há uma chance < 10% de a caravana ser destruída

```
void CaravanaSecreta::tempestadeAreia() {
    int sorte = rand() % 100;
    if (sorte < 10) { // Apenas 10% de chance de destruição
```

```
        cout << "Caravana Secreta (ID: " << getID() << ") foi
destruída pela tempestade.\n";
        setTripulantes(0); // Marca como destruída
    } else {
        cout << "Caravana Secreta (ID: " << getID() << ") sobreviveu à
tempestade\n";
        setAgua(getAgua() - 20); // Perde 20 litros de água adicional
        if (getAgua() < 0) setAgua(0);
    }
}
```

- **Movimento sem Tripulantes:**
  - Função:
    - Quando está sem tripulantes, move-se automaticamente de forma aleatória durante 7 turnos
  - Parâmetros:
    - linhasMapa – índice de linhas atual
    - colunasMapa – índice de colunas atual
  - Detalhes de Implementação:
    - Move-se aleatoriamente para uma das 8 posições adjacentes

```
bool CaravanaSecreta::movimentoSemTripulantes(int linhasMapa, int
colunasMapa, Mapa& mapa) {
    setInstantesSemTripulantes(getInstantesSemTripulantes() + 1);
    // Move-se aleatoriamente enquanto não desaparece
    string direcoes[] = {"C", "B", "E", "D", "CE", "CD", "BE", "BD"};
    string direcao = direcoes[rand() % 8];
    mover(direcao, linhasMapa, colunasMapa, mapa);
     return getInstantesSemTripulantes() >= 3;
}
```

- **Movimento Autonomo:**

Objetivo:

Permitir que a caravana secreta se mova autonomamente, priorizando evitar bárbaros, depois se aproximar de itens e, caso nenhuma dessas condições seja atendida, mover-se aleatoriamente.

Passos da Implementação:

1. **Evitar Bárbaros:**
   - A caravana secreta verifica a distância para cada bárbaro. Se um bárbaro estiver a 3 casas de distância ou menos, a caravana vai se mover aleatoriamente para fugir. Este movimento é feito chamando a função mover com uma direção aleatória.
2. **Aproximar-se de Itens:**
   - Se não houver bárbaros próximos, a caravana secreta procura o item mais próximo dentro de um raio de 6 casas. Se encontrar, ela se move em direção ao item.
3. **Verificar Validade do Movimento:**

- Antes de realizar qualquer movimento, a função verifica se a posição calculada é válida no mapa, utilizando o método mapa.posicaoValida().

4. **Movimento Aleatório:**
   - Se não houver nenhum item próximo ou se o movimento em direção ao item for bloqueado, a caravana secreta se move aleatoriamente em uma das quatro direções (Cima, Baixo, Esquerda, Direita).
5. **Retorno:**
   - A função retorna true sempre que a caravana realiza um movimento, seja ele para fugir, se aproximar de um item, ou se mover aleatoriamente.

Parâmetros:

- *barbaros (vector<CaravanaBarbara>&)**: Lista de caravanas de bárbaros.
- *outrasCaravanas (vector<Caravana>&)**: Lista de outras caravanas (não utilizada nesta função).
- *items (vector<Item>&)**: Lista de itens no mapa.
- **linhasMapa (int)** e **colunasMapa (int)**: Dimensões do mapa.
- **mapa (Mapa&)**: Objeto que representa o mapa, utilizado para verificar a validade das posições e atualizar o mapa.

```
bool CaravanaSecreta::movimentoAutonomo(const
vector<CaravanaBarbara*>& barbaros, const vector<Caravana*>&
outrasCaravanas, const vector<Item*>& items, int linhasMapa, int
colunasMapa, Mapa& mapa) {
    // Priorizar evitar bárbaros
    for (auto& barbaro : barbaros) {
        int distancia = abs(barbaro->getLinha() - getLinha()) +
abs(barbaro->getColuna() - getColuna());
        if (distancia <= 3) { // Foge se um bárbaro estiver a 3
posições de distância
            cout << "Caravana Secreta (ID: " << getID() << ") fugiu
de um bárbaro próximo.\n";
             string direcoes[] = {"C", "B", "E", "D", "CE", "CD",
"BE", "BD"};
            mover(direcoes[rand() % 8], linhasMapa, colunasMapa,
mapa); // Move-se aleatoriamente para fugir
            return true;
        }
    }

    // Encontrar o item mais próximo dentro de um raio de 6
posições
    Item* itemMaisProximo = nullptr;
    int menorDistancia = INT_MAX;

    for (auto& item : items) {
        int distancia = abs(item->getLinha() - getLinha()) +
abs(item->getColuna() - getColuna());
        if (distancia <= 6 && distancia < menorDistancia) {
            menorDistancia = distancia;
            itemMaisProximo = item;
        }
```

```
    }

    // Aproximar-se do item mais próximo, se encontrado
    if (itemMaisProximo != nullptr) {
        cout << "Caravana Secreta (ID: " << getID() << ") movendo-
se em direção ao item mais próximo em ("
             << itemMaisProximo->getLinha() << ", " <<
itemMaisProximo->getColuna() << ").\n";

        int novaLinha = getLinha();
        int novaColuna = getColuna();

        if (itemMaisProximo->getLinha() > getLinha()) novaLinha++;
        else if (itemMaisProximo->getLinha() < getLinha())
novaLinha--;

        if (itemMaisProximo->getColuna() > getColuna())
novaColuna++;
        else if (itemMaisProximo->getColuna() < getColuna())
novaColuna--;

        if (mapa.posicaoValida(novaLinha, novaColuna)) {
            mapa.atualizarPosicao(getLinha(), getColuna(), '.'); //
Limpar posição atual no mapa
            setPosicao(novaLinha , novaColuna);
            mapa.atualizarPosicao(novaLinha, novaColuna, 'S'); //
Atualizar o mapa com a identificação da caravana secreta
            return true;
        } else {
            cout << "Movimento bloqueado por obstáculo.\n";
        }
    }

    // Movimento aleatório se nenhum item for encontrado ou
movimento falhar
    string direcoes[] = {"C", "B", "E", "D"};
    string direcao = direcoes[rand() % 4];
    mover(direcao, linhasMapa, colunasMapa, mapa);
    cout << "Caravana Secreta (ID: " << getID() << ") moveu-se
aleatoriamente.\n";
    return true;
}
```

Cidade.h

Classe Cidade:

- **Atribuitos:**
  - nome – Representa o nome da cidade
  - linha/coluna – Coordenadas da cidade no mapa
  - caravanas – Vetor de ponteiros para caravanas, permitindo que Cidade interaja com instâncias de caravana
- **Construtor:**
  - Inicializa a cidade com um nome e uma posição (linha, coluna) no mapa.
  - Vetor caravanas começa vazio

```
Cidade(char nome, int linha, int coluna);
```

- **Getters:**
    - Retorna os atributos pertendidos

```
char getNome() const;
int getLinha() const;
int getColuna() const;
```

- **Mostrar Detalhes:**
    - Apresenta informações detalhadas sobre cidade

```
void mostrarDetalhes() const;
```

- **Adicionar caravana:**
    - Adiciona uma caravana à cidade.
    - A cidade passa a "conhecer" a caravana, permitindo interações futuras.

```
void adicionarCaravana(Caravana* caravana);
```

- **Remover Caravana:**
    - Remove a caravana do vetor, indicando que ela deixou a cidade.

```
void removerCaravana(Caravana* caravana);
```

- **Inspecionar Caravanas:**
    - Exibe os detalhes de todas as caravanas atualmente presentes na cidade.

```
void inspecionarCaravanas() const;
```

- **Comprar Mercadorias:**
    - Permite que uma caravana compre mercadorias da cidade.

```
void comprarMercadorias(Caravana* caravana, int quantidade);
```

- **Vender Mercadorias:**
    - Permite que uma caravana venda mercadorias à cidade.

```
void venderMercadorias(Caravana* caravana, int quantidade);
```

- **Contratar Tripulantes:**
    - Contrata tripulantes para a caravana.

```
void contratarTripulantes(Caravana* caravana, int quantidade);
```

- **Lado Acessível:**
    - Verifica se há pelo menos um lado acessível (livre) ao redor da cidade no mapa.

```
bool ladoAcessivel(const vector<string>& mapa, int linhasMapa, int
colunasMapa) const;
```

Cidade.cpp

- **Construtor:**
  - Inicia a cidade com os atributos defenidos

```
Cidade::Cidade(char nome, int linha, int coluna)
    : nome(nome), linha(linha), coluna(coluna) {}
```

- **Getters:**
  - Retorna os atributos pretendidos

```
char Cidade::getNome() const {
    return nome;
}
int Cidade::getLinha() const {
    return linha;
}
int Cidade::getColuna() const {
    return coluna;
}
```

- **Mostrar Detalhes:**
  - Função:
    - Imprime a informação detalhada na consola

```
void Cidade::mostrarDetalhes() const {
    cout << "Cidade " << nome << " na posição (" << linha << ", " <<
coluna << ")\n";
    cout << "Número de caravanas presentes: " << caravanas.size() <<
"\n";
}
```

- **Adicionar Caravana:**
  - Função:
    - Adiciona uma caravana ao vetor

```
void Cidade::adicionarCaravana(Caravana* caravana) {
    caravanas.push_back(caravana);
    cout << "Caravana " << caravana->getID() << " entrou na cidade "
<< nome << ".\n";
}
```

- **Remover Caravana:**
  - Função:
    - Remove uma caravana do vetor
  - Parâmetros:
    - caravana – Vetor de caravanas
  - Detalhes de implementação:

- Localiza a caravana dentro do vetor
- Remove a caravana do vetor
- Em caso de erro mostra uma mensagem de erro

```
void Cidade::removerCaravana(Caravana* caravana) {
    auto it = find(caravanas.begin(), caravanas.end(), caravana);
    if (it != caravanas.end()) {
        caravanas.erase(it);
        cout << "Caravana " << caravana->getID() << " saiu da cidade "
<< nome << ".\n";
    } else {
        cout << "Caravana não encontrada na cidade " << nome << ".\n";
    }
}
```

- **Inspecionar Caravanas:**
  - Função:
    - Mostra os detalhes as caravanas presentes no vetor

```
void Cidade::inspecionarCaravanas() const {
    cout << "Caravanas na cidade " << nome << ":\n";
    for (const auto& caravana : caravanas) {
        caravana->mostrarDetalhes();
    }
}
```

- **Comprar Mercadorias:**
  - Função:
    - Simula a cidade comprando mercadorias de uma caravana
  - Parâmetros:
    - Caravana – Vetor das caravanas
    - Quantidade – Quantidade de mercadoria a ser vendida
  - Detalhes de Implementação:
    - Remove uma quantidade especifica da carga da caravana
    - Imprime uma mensagem na consola

```
void Cidade::comprarMercadorias(Caravana* caravana, int quantidade) {
    caravana->carregar(-quantidade); // Remove mercadorias da caravana
    cout << "Caravana " << caravana->getID() << " vendeu " <<
quantidade << " toneladas de mercadorias na cidade " << nome << ".\n";
}
```

- **Vender Mercadorias:**
  - Função:
    - Simula a cidade vendendo mercadorias de uma caravana
  - Parâmetros:
    - Caravana – Vetor das caravanas
    - Quantidade – Quantidade de mercadoria a ser comprada
  - Detalhes de Implementação:
    - Adiciona uma quantidade especifica à carga da caravana
    - Imprime uma mensagem na consola

```
void Cidade::venderMercadorias(Caravana* caravana, int quantidade) {
    caravana->carregar(quantidade); // Adiciona mercadorias à caravana
    cout << "Caravana " << caravana->getID() << " comprou " <<
```

```
quantidade << " toneladas de mercadorias na cidade " << nome << ".\n";
}
```

- **Contratar Tripulantes:**
  - Função:
    - Adiciona tripulantes à caravana, garantindo que não ultrapassa o limite
  - Parâmetros:
    - Caravana – Vetor das caravanas
    - Quantidade – Quantidade de tripulantes a juntar
  - Detalhes de Implementação:
    - Tenta adicionar tripulantes
    - Exibe uma mensagem de sucesso ou erro dependendo do espaço disponível na caravana

```
void Cidade::contratarTripulantes(Caravana* caravana, int quantidade)
{
    if (caravana->adicionaTripulantes(quantidade)) {
        cout << "Caravana " << caravana->getID() << " contratou " <<
quantidade << " tripulantes na cidade " << nome << ".\n";
    } else {
        cout << "Caravana " << caravana->getID() << " não pode
adicionar mais tripulantes (capacidade máxima atingida).\n";
    }
}
```

- **Lado Acessível:**
  - Função:
    - Verificar se há pelo menos um lado acessível da cidade no mapa.
  - Parametros:
    - mapa - Vetor de strings representando o mapa
    - linhasMapa/colunasMapa – Dimensões do mapa.
  - Detalhes de Implementação:
    - Define direções adjacentes
    - Calcula as coordenadas da célula adjacente. (para cada direção)
    - Verifica se a célula adjacente contém **'.'**
    - Retorna true se encontrar pelo menos uma célula acessível; caso contrário, false.

```
bool Cidade::ladoAcessivel(const vector<string>& mapa, int linhasMapa,
int colunasMapa) const {
    // Verificar as 4 posições adjacentes (cima, baixo, esquerda,
direita)
    int direcoes[4][2] = {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
    for (const auto& dir : direcoes) {
        int novaLinha = (linha + dir[0] + linhasMapa) % linhasMapa;
        int novaColuna = (coluna + dir[1] + colunasMapa) %
colunasMapa;

        if (mapa[novaLinha][novaColuna] == '.') { // '.' representa
deserto
            return true; // Pelo menos um lado acessível
        }
```

```
    }
    return false; // Nenhum lado acessível
}
```

barbaros.h

Classe CaravanaBarbara:

- **Atributos:**
  - turnosRestantes - Número máximo de turnos que a caravana pode existir.
  - linha/coluna - Coordenadas atuais da caravana no mapa.
  - tripulantes - Número de tripulantes da caravana
  - destruida – indica se foi destruida
- **Construtor:**
  - Inicia a caravana

```
CaravanaBarbara(int linha, int coluna, int turnosMax);
```

- **Destrutor:**
  - Destrói a caravana

```
~CaravanaBarbara(){}
```

- **Getters/Setters:**
  - Retorna o número de tripulantes atuais.

```
int getTripulantes() const {return tripulantes;}
int getLinha() const {return linha;}
int getColuna() const {return coluna;}
int getTurnosRestantes() const {return turnosRestantes;}
bool getDestruida() const {return destruida;}
```

  - Atualiza o número de tripulantes.

```
void setTripulantes(int quantidade){ tripulantes = quantidade;}
void setDestruida(bool b) {destruida = b;}
void setTurnosRestantes(int quant){turnosRestantes = quant;}
```

- **Mover:**
  - Move a caravana em uma direção.

```
void mover(const std::string& direcao, int linhasMapa, int colunasMapa
, Mapa& mapa);
```

- **Mover Autónomo:**
  - Cria a movimentação automática da carava

```
void moverAutonomo(int linhasMapa, int colunasMapa, const
vector<Caravana*>& caravanas, Mapa& mapa);
```

- **Tempestade Areia:**
  - Aplica efeito na carava

```
void tempestadeAreia();
```

- Turnos Excedidos:
  - Retorna true se os turnos restantes forem 0 ou menos, indicando que a caravana deve ser removida.

```
bool turnosExcedidos() const;
```

- **Sem Tripulantes:**
  - Retorna true se o número de tripulantes for 0.

```
bool semTripulantes() const {return tripulantes <= 0;}
```

- **Calcular Distancia:**
  - Calcula a distância entre a posição atual da caravana e um ponto no mapa.

```
int calcularDistancia(int linhaAlvo, int colunaAlvo) const;
```

barbaros.cpp

Classe CaravanaBarbaros:

- **Construtor:**
  - Inicia a Caravana Barbara

```
CaravanaBarbara::CaravanaBarbara(int linha, int coluna, int turnosMax)
    : linha(linha), coluna(coluna) , turnosRestantes(turnosMax) ,
tripulantes(40){}
```

- **Mover:**

  A função é responsável por calcular a nova posição da caravana com base no comando de direção fornecido e garantir que a nova posição seja válida no mapa. Se a posição for válida, a caravana se move para essa posição, e o mapa é atualizado.

  1. **Análise do Comando de Direção:**
     - A função começa verificando qual comando de direção foi inserido no parâmetro direcao. Dependendo do comando, a posição da caravana (linha e coluna) será ajustada.
  2. **Cálculo da Nova Posição:**
     - **Com base na direção fornecida, a função calcula as novas coordenadas da caravana:**
       - "C" (Cima): A linha da caravana diminui em 1.

- "B" (Baixo): A linha da caravana aumenta em 1.
- "E" (Esquerda): A coluna da caravana diminui em 1.
- "D" (Direita): A coluna da caravana aumenta em 1.
- **Direções Diagonais: Para as direções diagonais (ex. "CE", "CD", "BE", "BD"), tanto a linha quanto a coluna são ajustadas simultaneamente:**
  - "CE": A linha diminui e a coluna diminui.
  - "CD": A linha diminui e a coluna aumenta.
  - "BE": A linha aumenta e a coluna diminui.
  - "BD": A linha aumenta e a coluna aumenta.

- **Mapas Cíclicos:** Para garantir que a caravana "não saia" do mapa, utiliza-se a operação de módulo (%) para calcular as novas posições. Isso permite que a caravana, ao alcançar a borda do mapa, "volte" para a outra extremidade (comportamento de mapa cíclico). **Por exemplo:**
  - Se a linha atingir o índice máximo, ela "volta" para 0 (início do mapa).
  - O mesmo acontece com a coluna.

3. **Validação da Nova Posição:**
   - A função verifica se a nova posição é válida no mapa usando um método da classe Mapa (por exemplo, mapa.posicaoValida(novaLinha, novaColuna)). Isso garante que a caravana não se mova para posições inválidas ou fora do limite do mapa.
   - Se a posição for válida, a função prossegue para atualizar a posição da caravana.
4. **Atualização do Mapa:**
   - **Se a nova posição for válida, a função:**
     - Limpa a posição anterior: Atualiza o mapa para remover a caravana da posição anterior (marcando com um ponto ou outro caractere que indica um espaço vazio).
     - Atribui a nova posição à caravana: A nova posição (linha e coluna) é atribuída à caravana.
     - Marca a nova posição no mapa: A posição da caravana no mapa é atualizada, e a caravana é representada por um símbolo específico (por exemplo, '!' ou outro símbolo de identificação).

```
void CaravanaBarbara::mover(const std::string& direcao, int
linhasMapa, int colunasMapa, Mapa& mapa) {

    int novaLinha = linha;
    int novaColuna = coluna;

    // Calculando as novas posições com base na direção
    if (direcao == "C") {
        novaLinha = (linha - 1 + linhasMapa) % linhasMapa;
    } else if (direcao == "B") {
```

```
        novaLinha = (linha + 1) % linhasMapa;
    } else if (direcao == "E") {
        novaColuna = (coluna - 1 + colunasMapa) % colunasMapa;
    } else if (direcao == "D") {
        novaColuna = (coluna + 1) % colunasMapa;
    } else if (direcao == "CE") {
        novaLinha = (linha - 1 + linhasMapa) % linhasMapa;
        novaColuna = (coluna - 1 + colunasMapa) % colunasMapa;
    } else if (direcao == "CD") {
        novaLinha = (linha - 1 + linhasMapa) % linhasMapa;
        novaColuna = (coluna + 1) % colunasMapa;
    } else if (direcao == "BE") {
        novaLinha = (linha + 1) % linhasMapa;
        novaColuna = (coluna - 1 + colunasMapa) % colunasMapa;
    } else if (direcao == "BD") {
        novaLinha = (linha + 1) % linhasMapa;
        novaColuna = (coluna + 1) % colunasMapa;
    }

    // Verificar se a nova posição é válida
    if (mapa.posicaoValida(novaLinha, novaColuna)) {
        mapa.atualizarPosicao(linha , coluna , '.');
        linha = novaLinha;
        coluna = novaColuna;
        mapa.atualizarPosicao(linha, coluna, '!');
    }
}
```

- **Mover Autonomo:**

  Objetivo:

  Realiza o movimento automático da caravana barbara, movendo-a em direção à caravana mais próxima (caso exista) ou de forma aleatória.

  Parâmetros:

  - linhasMapa (int): Número de linhas do mapa.
  - colunasMapa (int): Número de colunas do mapa.
  - caravanas (vector<Caravana*>): Lista das caravanas no mapa, usadas para encontrar a caravana mais próxima.
  - mapa (Mapa&): Instância do mapa onde a caravana será movida e as posições serão atualizadas.

  Detalhes de Implementação:

  1. **Redução dos Turnos Restantes:**
     - O contador turnosRestantes é decrementado em 1, representando a redução do tempo ou número de turnos restantes para o movimento da caravana.
  2. **Encontro da Caravana Mais Próxima:**
     - A função percorre todas as caravanas no vetor caravanas para encontrar a caravana mais próxima dentro de um raio de alcance de 8 posições.
     - A distância entre a caravana Barbara e as outras caravanas é calculada usando a função calcularDistancia(). Se a distância for menor ou igual a 8

e menor que a menor distância registrada, a caravana mais próxima é atualizada.

3. **Movimento em Direção à Caravana Mais Próxima:**
    - Se uma caravana próxima for encontrada, a caravana Barbara tenta se mover em direção a ela. A direção do movimento é determinada pela comparação das posições de linha e coluna.
        - Se a linha de Barbara for menor que a linha da caravana mais próxima, a linha de Barbara aumenta (move para baixo).
        - Se a linha de Barbara for maior, a linha diminui (move para cima).
        - O mesmo processo é aplicado para a coluna.
    - Após calcular a nova posição, a função verifica se a posição é válida no mapa usando mapa.posicaoValida().
    - Se for válida, a posição antiga é apagada (representada por '.'), e a caravana Barbara é movida para a nova posição, sendo atualizada no mapa com o símbolo '!'.
4. **Movimento Aleatório:**
    - Caso não haja uma caravana próxima dentro do raio de 8 posições, a caravana Barbara se move aleatoriamente.
    - Uma direção aleatória é escolhida a partir de um conjunto de direções possíveis (C, B, E, D, CE, CD, BE, BD).
    - A função mover() é chamada para realizar o movimento na direção escolhida.

```
void CaravanaBarbara::moverAutonomo(int linhasMapa, int colunasMapa,
const vector<Caravana*>& caravanas, Mapa& mapa) {
    // Reduzir turnos restantes
    turnosRestantes--;

    // Variáveis para armazenar a caravana mais próxima e a sua
distância
    const Caravana* caravanaMaisProxima = nullptr;
    int menorDistancia = INT_MAX;

    // Encontrar a caravana mais próxima dentro do raio de alcance (8
posições)
    for (const auto& caravana : caravanas) {
        int distancia = calcularDistancia(caravana->getLinha(),
caravana->getColuna());
        if (distancia <= 8 && distancia < menorDistancia) {
            caravanaMaisProxima = caravana;
            menorDistancia = distancia;
        }
    }

    // Movimento em direção à caravana mais próxima
    if (caravanaMaisProxima != nullptr) {
        int linhaAlvo = caravanaMaisProxima->getLinha();
        int colunaAlvo = caravanaMaisProxima->getColuna();
        int novaLinha = linha;
```

```
        int novaColuna = coluna;

        // Determinar direção do movimento
        if (linha < linhaAlvo) novaLinha++;
        else if (linha > linhaAlvo) novaLinha--;

        if (coluna < colunaAlvo) novaColuna++;
        else if (coluna > colunaAlvo) novaColuna--;

        // Verificar se a nova posição é válida
        if (mapa.posicaoValida(novaLinha, novaColuna)) {
            mapa.atualizarPosicao(linha, coluna, '.'); // Limpar
posição atual no mapa
            linha = novaLinha;
            coluna = novaColuna;
            mapa.atualizarPosicao(linha, coluna, '!'); // Atualizar
posição no mapa
            cout << "Bárbaro movido para (" << linha << ", " << coluna
<< ") em direção à caravana mais próxima.\n";
        }
    } else {
        // Caso contrário, movimento aleatório
        string direcoes[] = {"C", "B", "E", "D", "CE" , "CD" , "BE" ,
"BD"};
        string direcao = direcoes[rand() % 8]; // Escolher uma direção
aleatória

        // Tentar mover na direção aleatória
        mover(direcao, linhasMapa, colunasMapa, mapa);
        cout << "Bárbaro movido aleatoriamente para (" << linha << ",
" << coluna << ").\n";
    }

}
```

- **Tempestade Areia:**
  - Função:
    - Aplica efeitos na caravana
  - Detalhes de Implementação:
    - Reduz os tripulantes da caravana em 10%
    - Chance de destruir uma caravana (25%)

```
void CaravanaBarbara::tempestadeAreia() {
    // Perde 10% dos tripulantes
    int perdaTripulantes = static_cast<int>(getTripulantes() * 0.1);
    tripulantes -= perdaTripulantes;

    // 25% de chance de destruição total
    int sorte = rand() % 100;
    if (sorte < 25) {
        setDestruida(true);

    }
}
```

- **Turnos Excedidos:**
  - Verifica se os turnos restantes da caravana acabaram.

```
bool CaravanaBarbara::turnosExcedidos() const{
    return turnosRestantes <= 0;
}
```

- **Calcular Distancia:**
    - Função:
        - Calcula a distancia entre posição atual da caravana e o alvo
    - Parametros:
        - linhaAlvo/colunaAlvo – Posição do alvo

```
int CaravanaBarbara::calcularDistancia(int linhaAlvo, int colunaAlvo)
const {
    return abs(linha - linhaAlvo) + abs(coluna - colunaAlvo);
}
```

buffer.h

Classe Buffer:

- **Atributos:**
    - nome – define o nome do buffer
    - linhas/colunas – Dimenções do buffer
    - buffer – Ponteiro para um array unidimensional para armazena os caracteres
    - cursorLinha/cursorColuna – Posição atual do cursor
- **Calcular Índice:**
    - Converte coordenadas bidimensionais (linha, coluna) em um índice para o array unidimensional.

```
int calcularIndice(int linha, int coluna) const;
```

- **Construtor:**
    - Cria um buffer com dimensões especificadas
    - Cria um buffer por copia de outro e com atribuição de um nome

```
Buffer(int linhas, int colunas);
Buffer(const Buffer& outro, const std::string& nm);
```

- **Destrutor:**
    - Liberta a memoria alocada para o array buffer

```
~Buffer();
```

- **Esvaziar:**
    - Esvazia o buffer preenchendo com espaços

```
void esvaziar();
```

- **Mover Cursor:**

- Move o cursor para a posição especificada

```
void moverCursor(int linha, int coluna);
```

- **Escrever Char:**
  - Escreve um único caractere c na posição atual do cursor.

```
void escreverChar(char c);
```

- **Escrever String:**
  - Escreve uma string a partir da posição do cursor.

```
void escreverString(const std::string& str);
```

- **Imprimir:**
  - Mostra o conteúdo do buffer na consola

```
void imprimir() const;
```

- **Operator<<:**
  - Permite inserir strings ou caracteres no buffer

```
Buffer& operator<<(const std::string& str);
Buffer& operator<<(char c);
```

- Operator<<:
  - Permite inserir valor no buffer

```
Buffer& operator<<(int valor);
```

Buffer.cpp

- **Construtor:**
  - Função:
    - Cria o buffer
  - Detalhes de Implementação:
    - Construtor normal:
      - Inicia as variáveis
      - Aloca o buffer num array unidimensional
      - Inicia o buffer com espaços
    - Construtor por cópia:
      - Copia os atributos de outra variável tipo buffer e atribui um nome

```
Buffer::Buffer(int linhas, int colunas)
    : linhas(linhas), colunas(colunas), cursorLinha(0),
cursorColuna(0) , nome(" "){
```

```
    buffer = new char[linhas * colunas]; // Aloca o buffer
unidimensional
    esvaziar(); // Inicializa o buffer com espaços
}

Buffer::Buffer(const Buffer& outro , const std::string& nm) {
    // Copiar as dimensões
    linhas = outro.linhas;
    colunas = outro.colunas;

    // Alocar memória para o novo buffer
    buffer = new char[linhas * colunas];

    // Copiar o conteúdo do buffer
    for (int i = 0; i < linhas * colunas; ++i) {
        buffer[i] = outro.buffer[i];
    }

    // Copiar a posição do cursor
    cursorLinha = outro.cursorLinha;
    cursorColuna = outro.cursorColuna;

    // Copiar o nome
    nome = nm;
}
```

- **Destrutor:**
    - Liberta a memória alocada para o buffer

```
Buffer::~Buffer() {
    delete[] buffer; // Libera a memória alocada para o buffer
}
```

- **Esvaziar:**
    - Função:
        - Preencher o buffer com espaços

```
void Buffer::esvaziar() {
    for (int i = 0; i < linhas * colunas; ++i) {
        buffer[i] = ' ';
    }
}
```

- **Mover Cursor:**
  - Função:
    - Move o cursor dentro do buffer
  - Parâmetros:
    - Linha/Coluna – Posição final do cursor
  - Detalhes de Implementação:
    - Verifica se a posição é valida, se for atualiza a posição do cursor
    - Senão imprime uma mensagem de erro na consola

```
void Buffer::moverCursor(int linha, int coluna) {
    if (linha >= 0 && linha < linhas && coluna >= 0 && coluna <
colunas) {
        cursorLinha = linha;
        cursorColuna = coluna;
    } else {
        std::cerr << "Posição inválida para o cursor!" << std::endl;
    }
}
```

- **Escrever Char:**
  - Função:
    - Escreve um carater dentro do buffer
  - Parâmetros:
    - c – Carater a ser escrito
  - Detalhes de Implementação:
    - Verifica se o cursor está numa posição valida
    - Calcula o índice correspondente no buffer
    - Escreve o carater na posição
    - Move o cursor para a posição seguinte

```
void Buffer::escreverChar(char c) {
    if (cursorLinha >= 0 && cursorLinha < linhas && cursorColuna >= 0
&& cursorColuna < colunas) {
        buffer[calcularIndice(cursorLinha, cursorColuna)] = c;
        cursorColuna++; // Move o cursor para a direita após escrever
    }
}
```

- **Escrever String:**
  - Função:
    - Escreve uma string dentro do buffer
  - Parâmetros:
    - str – string a ser escrita
  - Detalhes de Implementação:
    - Percorre a string e vai escrevendo carater a carater no buffer

```
void Buffer::escreverString(const std::string& str) {
    for (char c : str) {
        escreverChar(c);
    }
}
```

- **Imprimir:**
  - Função:
    - Imprime o buffer
  - Detalhes de Implementação:
    - Percorre as linhas e colunas
    - Converte (linha, coluna) em índice e imprime

```
void Buffer::imprimir() const {
    for (int i = 0; i < linhas; ++i) {
        for (int j = 0; j < colunas; ++j) {
            std::cout << buffer[calcularIndice(i, j)];
        }
        std::cout << std::endl;
    }
}
```

- **Operator<<:**
  - Função:
    - Escreve no buffer
  - Parâmetros:
    - str - string para escrever no buffer
    - C – carater para escrever no buffer
    - Valor – valor a escrever no buffer

```
Buffer& Buffer::operator<<(const std::string& str) {
    escreverString(str);
    return *this;
}
Buffer& Buffer::operator<<(char c) {
    escreverChar(c);
    return *this;
}
Buffer& Buffer::operator<<(int valor) {
    escreverString(std::to_string(valor));
    return *this;
}
```

- **Calcular Indice:**
  - Função:
    - Converte a posição para índice no buffer
  - Parâmetros:
    - Linha/coluna – Posição
  - Detalhes de Implementação:
    - Retorna o índice do buffer

```
int Buffer::calcularIndice(int linha, int coluna) const {
    return linha * colunas + coluna;
}
```

Simulacao.h

Classe Simulação:

- **Atributos Privados:**
  - fase
  - mapa - Mapa do deserto
  - layout
  - caravanas - Lista de caravanas do utilizador
  - cidades - Lista de cidades
  - items – Lista de Items
  - buffers – Lista de buffers
  - barbaros – Lista de Barbaros
  - moedas – Moedas do utilizador
  - turno – Contador de turnos
  - combatesVencidos – Total de combates vencidos
  - instantesEntreItens – Intervalo entre geração de items
  - duracaoItem – Duração Items
  - maxItem – Número máximo de items
  - precoVendaMercadoria – Preço de venda por tonelada
  - precoCompraMercadoria – Preço de compra por tonelada
  - precoCaravana – Preço por caravana
  - instantesEntreBarbaros – Intervalo entre geração de barbaros
  - duracaoBarbaros – Duração máxima de bárbaros
- **Processar Comando:**
  - Interpreta e executa um comando do usuário

```
void processarComando(const string& comando);
```

- **Atualizar Itens:**
  - Gera novos itens no mapa em intervalos regulares e remove itens expirados

```
void atualizarItens();
```

- **Atualizar Bárbaros:**
  - Gera novos bárbaros no mapa em intervalos regulares

```
void atualizarBarbaros();
```

- **Gerir Combates:**
  - Verifica se há combates entre caravanas e bárbaros e processa os resultados

```
void gerirCombates();
```

- **Montar Buffer Layout:**
  - Atualiza o layout visual do jogo no buffer

```
void montarBufferLayout(Buffer& buffer) const;
```

- **Atualizar Caravanas:**
  - Atualiza o estado das caravanas

```
void atualizarCaravanas();
```

- **Construtor:**
  - Inicia os atributos e prepara a simulação

```
Simulacao();
```

- **Destrutores:**
  - Liberta memória alocada

```
~Simulacao();
```

- **Inicia Simulação:**
  - Inicia a simulação principal, processando comandos do usuário

```
void iniciaSimulacao();
```

- **Config:**
  - Interpreta o ficheiro

```
bool config(const string& nomeFicheiro);
```

- **Exec:**
  - Executa o ficheiro

```
void exec(const string& nomeFicheiro);
```

- **Prox:**
  - Avança a simulação por n turnos

```
void prox(int n = 1);
```

- **Comprac:**
  - Compra uma caravana associada a uma cidade e tipo específico

```
void comprac(const char& nomeCidade, char tipo);
```

- **Preços:**
  - Mostra os preços de compra e venda de mercadorias

```
void precos() const;
```

- **Cidade:**
  - Exibe informações sobre uma cidade

```
void cidade(const char nomeCidade) const;
```

- **Caravana:**
  - Exibe detalhes de uma caravana específica

```
void caravana(int idCaravana) const;
```

- **Compra:**
  - Compra mercadorias para uma caravana

```
void compra(int idCaravana, int toneladas);
```

- **Vende:**
  - Vende mercadorias de uma caravana

```
void vende(int idCaravana);
```

- **Move:**
  - Move uma caravana em uma direção específica

```
void move(int idCaravana, const string& direcao);
```

- **Autogestão:**
  - Ativa a movimentação automática de uma caravana

```
void autoGestao(int idCaravana);
```

- **Stop:**
  - Para o movimento automático da caravana

```
void stop(int idCaravana);
```

- **Barbaro:**
  - Cria um bárbaro em uma posição específica

```
void barbaro(int linha, int coluna);
```

- **Areia:**
  - Cria uma tempestade de areia que afeta uma área do mapa

```
void areia(int linha, int coluna, int raio);
```

- **Add Moedas:**
  - Adiciona Moedas

```
void addMoedas(int valor);
```

- **Tripul:**
  - Ajusta o número de tripulantes de uma caravana

```
void tripul(int idCaravana, int quantidade);
```

- **Saves:**
  - Salva o estado atual da simulação em um arquivo

```
void saves(const string& nome);
```

- **Loads:**
  - Carrega o estado de uma simulação salva

```
void loads(const string& nome);
```

- **Lists:**
  - Listagem de saves

```
void lists() const;
```

- **Dels:**
  - Remove um arquivo de simulação salva

```
void dels(const string& nome);
```

- **Terminar:**
  - Finaliza a simulação e exibe um resumo

```
void terminar();
```

Simulacao.cpp

- **Construtor:**
  - Inicializa os atributos da classe com valores padrão

```
Simulacao::Simulacao()
    : fase(1),moedas(0), turno(0), combatesVencidos(0),
      instantesEntreItens(0), duracaoItem(0), maxItens(0),
      precoVendaMercadoria(0), precoCompraMercadoria(0),
      precoCaravana(0), instantesEntreBarbaros(0),
duracaoBarbaros(0){}
```

- **Destrutor:**
  - Liberta a memória alocada para objetos dinâmicos

```
Simulacao::~Simulacao() {
    for (auto caravana : caravanas) delete caravana;
    for (auto cidade : cidades) delete cidade;
    for (auto item : items) delete item;
    for (auto buffer : buffers) delete buffer;
    for (auto barbaro : barbaros) delete barbaro;
}
```

- **Gerir Combates:**
  - Função:
    - Gere os combates que acontecem
  - Detalhes de Implementação:
    - Identifica se alguma caravana está adjacente a um bárbaro
    - Calcula a força de combate das caravanas e bárbaros de forma aleatória, com base no número de tripulantes
    - Resolve o combate:
      - Se a caravana vence:
        - Reduz os tripulantes do bárbaro e da caravana proporcionalmente
        - Remove o bárbaro do mapa e do vetor se ele for eliminado
      - Se o bárbaro vence:
        - Reduz os tripulantes de ambos, mas não remove a caravana
    - Incrementa o contador de combates vencidos caso a caravana triunfe.

```
void Simulacao::gerirCombates() {
    vector<pair<Caravana*, CaravanaBarbara*>> combatesPendentes;

    // Verificar adjacências entre caravanas do utilizador e bárbaros
    for (auto& caravana : caravanas) {
        for (auto& barbaro : barbaros) {
            int deltaLinha = abs(caravana->getLinha() - barbaro-
>getLinha());
            int deltaColuna = abs(caravana->getColuna() - barbaro-
```

```
>getColuna());

            // Verificar se o bárbaro está em uma das 8 posições
adjacentes
            if (deltaLinha <= 1 && deltaColuna <= 1 && !(deltaLinha ==
0 && deltaColuna == 0)) {
                combatesPendentes.emplace_back(caravana, barbaro);
            }
        }
    }


    // Resolver os combates
    for (auto& [caravana, barbaro] : combatesPendentes) {
        int forçaCaravana = rand() % (caravana->getTripulantes() + 1);
        int forçaBarbaro = rand() % (barbaro->getTripulantes() + 1);

        if (forçaCaravana >= forçaBarbaro) {
            combatesVencidos++;
            // Caravana vence
            int perdaBarbaro = forçaCaravana * 2 / 5;  // Perda de
bárbaros
            int perdaCaravana = forçaCaravana / 5;    // Perda de
tripulantes da caravana

            barbaro->setTripulantes(barbaro->getTripulantes() -
perdaBarbaro);
            caravana->adicionaTripulantes(-perdaCaravana);

            if (barbaro->getTripulantes() <= 0) {
                for (auto it = barbaros.begin(); it != barbaros.end();
it++) {
                    if (*it == barbaro) { // Verifica se encontrou o
bárbaro a ser removido
                        mapa.atualizarPosicao((*it)->getLinha() ,
(*it)->getColuna() , '.');
                        barbaros.erase(it); // Remove o elemento do
vetor
                        cout << "Bárbaro destruido!\n";
                        delete *it;        // Libera a memória alocada
para o objeto
                        break;             // Sai do loop após
encontrar e remover o bárbaro
                    }
                }

            }
        } else {
            // Bárbaro vence
            int perdaCaravana = forçaBarbaro * 2 / 5;
            int perdaBarbaro = forçaBarbaro / 5;

            caravana->adicionaTripulantes(-perdaCaravana);
            barbaro->setTripulantes(barbaro->getTripulantes() -
perdaBarbaro);
        }
    }
}
```

- **Atualizar Itens:**
  - Função:
    - Atualiza o estado de cada item no mapa
  - Detalhes de Implementação:
    - Reduz a duração restante de cada item
    - Remove itens cuja duração tenha expirado e atualiza o mapa
    - Cria itens em posições aleatórias do mapa, respeitando um limite máximo configurado
    - Verifica se itens foram coletados por caravanas ou bárbaros e é aplicado aos respetivos

```
void Simulacao::atualizarItens() {
    // Reduzir a duração dos itens existentes
    for (auto it = items.begin(); it != items.end();) {
        (*it)->atualizarDuracao();
        if (!(*it)->estaAtivo()) {
            if(mapa.obterConteudo((*it)->getLinha() , (*it)-
>getColuna()) == 'I'){
                mapa.atualizarPosicao((*it)->getLinha(), (*it)-
>getColuna(), '.');
            }
            delete *it;
            it = items.erase(it); // Remove o item
        } else {
            ++it;
        }
    }

    // Criar novos itens se abaixo do limite configurável
    if (items.size() < maxItens && turno % instantesEntreItens == 0 &&
turno > 0) {
        int linha, coluna;
        do {
            linha = rand() % mapa.getLinhas();
            coluna = rand() % mapa.getColunas();
        } while (mapa.obterConteudo(linha, coluna) != '.'); // Apenas
em posições vazias

        // Criar um item aleatório
        int tipo = rand() % 5; // Tipo entre 0 e 4
        Item* novoItem;
        switch (tipo) {
            case 0: novoItem = new CaixaPandora(linha, coluna,
duracaoItem); break;
            case 1: novoItem = new ArcaTesouro(linha, coluna,
duracaoItem); break;
            case 2: novoItem = new Jaula(linha, coluna, duracaoItem);
break;
            case 3: novoItem = new Mina(linha, coluna, duracaoItem);
break;
            case 4: novoItem = new Surpresa(linha, coluna,
duracaoItem); break;
        }
        items.push_back(novoItem);
        mapa.atualizarPosicao(linha, coluna, 'I'); // 'I' representa
um item
    }

    // Verificar se itens foram apanhados por caravanas
```

```
    for (auto& caravana : caravanas) {
        for (auto it = items.begin(); it != items.end();) {
            if (abs(caravana->getLinha() - (*it)->getLinha()) <= 1 &&
                abs(caravana->getColuna() - (*it)->getColuna()) <= 1)
{
                (*it)->aplicarEfeitoC(*caravana, moedas); // Aplica
efeito do item
                if(mapa.obterConteudo((*it)->getLinha() , (*it)-
>getColuna()) == 'I'){
                    mapa.atualizarPosicao((*it)->getLinha(), (*it)-
>getColuna(), '.');
                }
                delete *it;
                it = items.erase(it);
            } else {
                ++it;
            }
        }
    }

    // Verificar se itens foram apanhados por bárbaros
    for (auto& barbaro : barbaros) {
        for (auto it = items.begin(); it != items.end();) {
            if (abs(barbaro->getLinha() - (*it)->getLinha()) <= 1 &&
                abs(barbaro->getColuna() - (*it)->getColuna()) <= 1) {
                (*it)->aplicarEfeitoB(*barbaro, moedas); // Aplica
efeito do item
                if(mapa.obterConteudo((*it)->getLinha() , (*it)-
>getColuna()) == 'I'){
                    mapa.atualizarPosicao((*it)->getLinha(), (*it)-
>getColuna(), '.');
                }
                delete *it;
                it = items.erase(it);
            } else {
                ++it;
            }
        }
    }
}
```

- **Atualizar Bárbaros:**
  - Função:
    - Gera novos bárbaros no mapa em intervalos regulares
  - Detalhes de Implementação:
    - Remove bárbaros que excederam sua duração ou foram destruídos
    - Gera novos bárbaros em posições aleatórias, respeitando o intervalo configurado
    - Move os bárbaros existentes de forma autônoma em direção às caravanas ou outras metas

```
void Simulacao::atualizarBarbaros() {

    for (auto it = barbaros.begin(); it != barbaros.end();) {
        if ((*it)->turnosExcedidos() || (*it)->getDestruida()){
            mapa.atualizarPosicao((*it)->getLinha(), (*it)-
>getColuna(), '.');
            delete *it;
```

```
            it = barbaros.erase(it); // Remove o item
        } else {
            ++it;
        }
    }

    if (turno % instantesEntreBarbaros == 0 && turno > 0) {
        int linha, coluna;
        do {
            linha = rand() % mapa.getLinhas();
            coluna = rand() % mapa.getColunas();
        } while (mapa.obterConteudo(linha, coluna) != '.'); // Apenas
em posições vazias

        CaravanaBarbara* novoBarbara = new CaravanaBarbara(linha,
coluna, duracaoBarbaros);
        barbaros.push_back(novoBarbara);
        mapa.atualizarPosicao(linha, coluna, '!');
        cout << "Uma nova caravana bárbara apareceu em (" << linha <<
", " << coluna << ")!\n";
    }

    // Movimentar os bárbaros existentes
    for (auto& barbaro : barbaros) {
        barbaro->moverAutonomo(mapa.getLinhas(), mapa.getColunas(),
caravanas , mapa);
    }
}
```

- **Montar Buffer Layout:**
  - Função:
    - Prepara uma representação textual do estado atual do jogo para exibição
  - Parâmetros:
    - buffer
  - Detalhes de Implementação:
    - Mostra:
      - O mapa
      - Informações como moedas disponíveis, número de itens e bárbaros no mapa, e combates vencidos.
      - Detalhes das caravanas e cidades.

```
void Simulacao::montarBufferLayout(Buffer& buffer) const {
    buffer.esvaziar(); // Esvaziar o buffer antes de começar

    // Adicionar o mapa ao buffer com controle de cursor
    for (int i = 0; i < mapa.getLinhas(); ++i) {
        for (int j = 0; j < mapa.getColunas(); ++j) {
            buffer.moverCursor(i, j);                    // Mover o
cursor para (linha, coluna)
            buffer.escreverChar(mapa.obterConteudo(i, j)); // Escrever
o símbolo no buffer
        }
    }

    // Adicionar uma linha em branco após o mapa
    buffer.moverCursor(mapa.getLinhas(), 0); // Mover o cursor para a
```

```
primeira posição após o mapa
    buffer.escreverString("\n");

    // Adicionar os dados configuráveis ao buffer
    int linhaAtual = mapa.getLinhas() + 2; // Linha inicial para os
dados
    buffer.moverCursor(linhaAtual++, 0);
    buffer.escreverString("Informacoes:");
    buffer.moverCursor(linhaAtual++, 0);
    buffer.escreverString("Moedas: " + std::to_string(moedas));
    buffer.moverCursor(linhaAtual++, 0);
    buffer.escreverString("Numero Items: " +
std::to_string(items.size()));
    buffer.moverCursor(linhaAtual++, 0);
    buffer.escreverString("Numero bárbaros: " +
std::to_string(barbaros.size()));
    buffer.moverCursor(linhaAtual++, 0);
    buffer.escreverString("Combates Vencidos: " +
std::to_string(combatesVencidos));

    // Adicionar os detalhes das caravanas
    buffer.moverCursor(++linhaAtual, 0);
    buffer.escreverString("Caravanas:");
    for (const auto& e : caravanas) {
        buffer.moverCursor(++linhaAtual, 0);
        buffer.escreverString("ID: " + std::to_string(e->getID()) +
                              " Posição: (" + std::to_string(e-
>getLinha()) +
                              ", " + std::to_string(e->getColuna()) +
")");
    }

    // Adicionar os detalhes das cidades
    buffer.moverCursor(++linhaAtual, 0);
    buffer.escreverString("\nCidades:");
    for (const auto& e : cidades) {
        buffer.moverCursor(++linhaAtual, 0);
        buffer.escreverString("Nome: " + std::string(1, e->getNome())
+
                              " Posição: (" + std::to_string(e-
>getLinha()) +
                              ", " + std::to_string(e->getColuna()) +
")");
    }

    // Adicionar os detalhes dos bárbaros
    buffer.moverCursor(++linhaAtual, 0);
    buffer.escreverString("\nBárbaros:");
    for (const auto& e : barbaros) {
        buffer.moverCursor(++linhaAtual, 0);
        buffer.escreverString("Posição: (" + std::to_string(e-
>getLinha()) +
                              ", " + std::to_string(e->getColuna()) +
                              ") Duração: " + std::to_string(e-
>getTurnosRestantes()) +
                              " Tripulantes: " + std::to_string(e-
>getTripulantes()));
    }
}
```

- **Config:**
  - Função:
    - Interpreta o ficheiro
  - Parâmetros:
    - nomeFicheiro – Nome do ficheiro que contem as informações inicias
  - Detalhes de Implemetação:
    - Carrega um ficheiro de configuração para inicializar o estado do jogo
    - Lê dimensões do mapa, símbolos que representam diferentes elementos (cidades, caravanas, bárbaros, etc.) e configurações como intervalos e preços
    - Garante que cidades sejam posicionadas corretamente (verificando se o lado acessível está válido)
    - Inicializa o mapa, insere elementos nele e configura parâmetros do jogo
    - Mostra o estado inicial do mapa e prepara o buffer para exibição

```
bool Simulacao::config(const string& nomeFicheiro) {
    ifstream ficheiro(nomeFicheiro);
    if (!ficheiro.is_open()) {
        cerr << "Erro ao abrir o ficheiro: " << nomeFicheiro << endl;
        return false;
    }

    string linha;
    int linhas, colunas;

    string palavra;

    // Ler "linhas" e o número correspondente
    ficheiro >> palavra;
    if (palavra != "linhas") {
        std::cerr << "Formato inválido: esperado 'linhas'" <<
std::endl;
        exit(1);
    }
    ficheiro >> linhas;

    // Ler "colunas" e o número correspondente
    ficheiro >> palavra;
    if (palavra != "colunas") {
        std::cerr << "Formato inválido: esperado 'colunas'" <<
std::endl;
        exit(1);
    }
    ficheiro >> colunas;
    ficheiro.ignore();

    mapa.inicializar(linhas , colunas);

    // Ler o conteúdo do mapa
    for (int i = 0; i < linhas; ++i) {
        getline(ficheiro, linha);
        for (int j = 0; j < colunas; ++j) {
            char simbolo = linha[j];
```

```
            switch (simbolo) {
                case '.':
                    mapa.atualizarPosicao(i, j, '.'); // Espaço vazio
                    break;
                case '+':
                    mapa.atualizarPosicao(i, j, '+'); // Obstáculo
                    break;
                case 'a' ... 'z': {
                    // Criar uma cidade com o nome do símbolo
                    Cidade* novaCidade = new Cidade(simbolo, i, j);
                    if(novaCidade->ladoAcessivel(mapa, i , j)){
                        cidades.push_back(novaCidade);
                        mapa.atualizarPosicao(i, j, simbolo);
                    }else{
                        cout << "Cidade " << simbolo << "nao está bem
posicionado!!" << endl;
                    }
                    break;
                }
                case '1' ... '9': {
                    int id = simbolo - '0';
                    int tipoCaravana = rand() % 3 + 1;
                    Caravana* novaCaravana = nullptr;
                    switch(tipoCaravana){
                        case 1:  novaCaravana = new
CaravanaComercio(id, i, j); break;
                        case 2:  novaCaravana = new
CaravanaMilitar(id, i, j); break;
                        case 3:  novaCaravana = new
CaravanaSecreta(id, i, j); break;
                    }

                    caravanas.push_back(novaCaravana);
                    mapa.atualizarPosicao(i, j, simbolo);
                    break;
                }
                case '!': {
                    // Criar um bárbaro
                    CaravanaBarbara* novoBarbara = new
CaravanaBarbara(i, j, duracaoBarbaros);
                    barbaros.push_back(novoBarbara);
                    mapa.atualizarPosicao(i, j, simbolo);
                    break;
                }
                default:
                    cerr << "Símbolo desconhecido no mapa: " <<
simbolo << endl;
            }
        }
    }

    // Ler os valores configuráveis
    while (ficheiro >> linha) {
        if (linha == "moedas") {
            ficheiro >> moedas;
        } else if (linha == "instantes_entre_novos_itens") {
            ficheiro >> instantesEntreItens;
        } else if (linha == "duração_item") {
            ficheiro >> duracaoItem;
        } else if (linha == "max_itens") {
            ficheiro >> maxItens;
```

```
        } else if (linha == "preço_venda_mercadoria") {
            ficheiro >> precoVendaMercadoria;
        } else if (linha == "preço_compra_mercadoria") {
            ficheiro >> precoCompraMercadoria;
        } else if (linha == "preço_caravana") {
            ficheiro >> precoCaravana;
        } else if (linha == "instantes_entre_novos_barbaros") {
            ficheiro >> instantesEntreBarbaros;
        } else if (linha == "duração_barbaros") {
            ficheiro >> duracaoBarbaros;
        }
    }

    for(int i = 0 ; i < barbaros.size() ; i++){
        barbaros.at(i)->setTurnosRestantes(duracaoBarbaros);
    }

    mapa.mostrarMapa(); // Mostrar o estado inicial do mapa
    montarBufferLayout(layout);
    cout << "Configuração carregada com sucesso.\n";
    return true;
}
```

- **Caravana:**
  - Função:
    - Mostra detalhes de uma caravana especifica
  - Parâmetros:
    - idCaravana – id da caravana pretendida
  - Detalhes de Implementação:
    - Percorre o array caravanas até encontrar o mesmo id
    - Apresenta detalhes da mesma

```
void Simulacao::caravana(int idCaravana) const {
    for(auto &e : caravanas){
        if(e->getID() == idCaravana){
            e->mostrarDetalhes();
        }
    }
}
```

- **Cidade:**
  - Função:
    - Exibe detalhes e inspeciona caravanas em uma cidade específica
  - Parâmetros:
    - nomeCidade – Nome da cidade a procurar
  - Detalhes de Implementação:
    - Percorre a lista de cidades
    - Se encontra a cidade cujo nome coincide com o argumento nomeCidade, exibe detalhes da cidade e inspeciona as caravanas associadas

```
void Simulacao::cidade(const char nomeCidade) const {
    for(auto &e : cidades){
        if(e->getNome() == nomeCidade){
            e->mostrarDetalhes();
```

```
            e->inspecionarCaravanas();
        }
    }
}
```

- **Preços:**
    - Função:
        - Exibe os preços das mercadorias e caravanas
    - Detalhes de Implementação:
        - Imprime os valores relacionados com a venda e compra de mercadoria e caravana

```
void Simulacao::precos() const {
    cout << "Preco por venda Mercadoria: " << precoVendaMercadoria <<
endl
    << "Preco por compra Mercadoria: " << precoCompraMercadoria <<
endl
    << "Preco por Caravana: " << precoCaravana << endl;
}
```

- **Autogestão**:
    - Função:
        - Ativa o modo automático de gestão para uma caravana específica
    - Parâmetros:
        - idCaravana – Id da caravana que se pretende alternar o modo automático
    - Detalhes de Implementação:
        - Encontra a caravana com o ID especificado em caravanas
        - Ativa o modo automático da mesma

```
void Simulacao::autoGestao(int idCaravana) {
    for(auto &e: caravanas){
        if(e->getID() == idCaravana){
            e->alteraAutomatico(true);
        }
    }
}
```

- **Stop:**
    - Função:
        - Desativa o modo automático de gestão para uma caravana específica
    - Parâmetros:
        - idCaravana – Id da caravana que se pretende alternar o modo automático
    - Detalhes de Implementação:
        - Encontra a caravana com o ID especificado em caravanas
        - Desativa o modo automático da mesma

```
void Simulacao::stop(int idCaravana) {
    for(auto &e: caravanas){
        if(e->getID() == idCaravana){
```

```
            e->alteraAutomatico(false);
        }
    }
}
```

- **Add Moedas:**
    - Função:
        - Adiciona moedas ao total do jogador
    - Parâmetros:
        - Valor – Valor a adicionar às moedas do jogador
    - Detalhes de Implementação:
        - Adiciona às moedas existente o valor
        - Exibe o novo total de moedas

```
void Simulacao::addMoedas(int valor) {
    moedas += valor;
    cout << "Moedas atualizadas para " << moedas << endl;
}
```

- **Bárbaro**:
    - Função:
        - Cria uma caravana bárbara numa posição no mapa
    - Parâmetros:
        - linha/coluna- Posição onde a caravana deve aparecer
    - Detalhes de Implementação:
        - Verifica se a posição fornecida é válida no mapa
        - Se válida, cria uma instância de CaravanaBarbara, adiciona à lista de bárbaros e atualiza o mapa com o símbolo “!”
        - Caso contrário, exibe uma mensagem de erro.

```
void Simulacao::barbaro(int linha, int coluna) {
    if(mapa.posicaoValida(linha ,coluna)){
        CaravanaBarbara* novoBarbara = new CaravanaBarbara(linha ,
coluna , duracaoBarbaros);
        barbaros.push_back(novoBarbara);
        mapa.atualizarPosicao(linha, coluna, '!');
        mapa.mostrarMapa();
    }else{
        cout << "Posicao nao valida" << endl;
    }
}
```

- **Compra C:**
    - Função:
        - Compra e cria uma caravana de um tipo específico em uma cidade
    - Parâmetros:
        - nomeCidade – Nome da cidade onde irá realizar a compra
        - tipo – Tipo de caravana
    - Detalhes de Implementação:
        - Verifica se a cidade existe na lista cidades
        - Garante que o limite de caravanas não foi atingido.

- Gera um ID único para a nova caravana.
- Verifica se há moedas suficientes para a compra
- Cria a caravana do tipo especificado (C / M / S)
- Adiciona a nova caravana à lista e deduz moedas do total

```
void Simulacao::comprac(const char& nomeCidade, char tipo) {
    // Verificar se a cidade existe
    Cidade* cidadeEncontrada = nullptr;
    for (auto& cidade : cidades) {
        if (cidade->getNome() == nomeCidade) { // Supondo que a cidade
tem o método getNome()
            cidadeEncontrada = cidade;
            break;
        }
    }

    if (!cidadeEncontrada) {
        cout << "Erro: Cidade " << nomeCidade << " não encontrada.\n";
        return;
    }

    // Verificar se o limite de caravanas foi atingido
    if (caravanas.size() >= 9) {
        cout << "Erro: Não é possível criar mais caravanas. Limite
máximo de 9 atingido.\n";
        return;
    }

    // Gerar um ID único para a nova caravana
    int novoID = -1;
    for (int id = 1; id <= 9; ++id) {
        bool idEmUso = false;
        for (const auto& caravana : caravanas) {
            if (caravana->getID() == id) {
                idEmUso = true;
                break;
            }
        }
        if (!idEmUso) {
            novoID = id;
            break;
        }
    }

    if (novoID == -1) {
        cout << "Erro: Não foi possível gerar um ID único para a nova
caravana.\n";
        return;
    }

    // Verificar se há moedas suficientes
    if (moedas < precoCaravana) {
        cout << "Erro: Moedas insuficientes para comprar uma caravana.
Custo: " << precoCaravana << ", Moedas disponíveis: " << moedas <<
".\n";
        return;
    }

    // Criar a caravana com base no tipo
    Caravana* novaCaravana = nullptr;
```

```
    switch (tipo) {
        case 'C': // Caravana de Comércio
            novaCaravana = new CaravanaComercio(novoID,
cidadeEncontrada->getLinha(), cidadeEncontrada->getColuna());
            cidadeEncontrada->adicionarCaravana(novaCaravana);
            break;

        case 'M': // Caravana Militar
            novaCaravana = new CaravanaMilitar(novoID,
cidadeEncontrada->getLinha(), cidadeEncontrada->getColuna());
            cidadeEncontrada->adicionarCaravana(novaCaravana);
            break;
        case 'S':
            novaCaravana = new CaravanaSecreta(novoID,
cidadeEncontrada->getLinha(), cidadeEncontrada->getColuna());
            cidadeEncontrada->adicionarCaravana(novaCaravana);
            break;
        default:
            cout << "Erro: Tipo de caravana inválido.\n";
            return;
    }

    // Adicionar a caravana à lista de caravanas e atualizar moedas
    if (novaCaravana) {
        caravanas.push_back(novaCaravana);
        moedas -= precoCaravana; // Reduzir o número de moedas
        cout << "Caravana do tipo " << tipo << " criada com sucesso na
cidade " << nomeCidade << " com ID " << novoID << ".\n";
        cout << "Moedas restantes: " << moedas << ".\n";
    }
}
```

- **Compra:**
    - Função:
        - Compra mercadorias para uma caravana
    - Parâmetros:
        - idCaravana – id da caravana
        - toneladas – quantidade adicionar à carga
    - Detalhes de Implementação:
        - Verifica se a caravana existe e está localizada em uma cidade
        - Calcula o custo total com base no preço de compra e a quantidade
        - Verifica se há moedas suficientes
        - Se houver, adiciona as mercadorias à caravana e reduz as moedas

```
void Simulacao::compra(int idCaravana, int toneladas) {
    // Verificar se a caravana existe
    Caravana* caravanaEncontrada = nullptr;
    for (auto& caravana : caravanas) {
        if (caravana->getID() == idCaravana) {
            caravanaEncontrada = caravana;
            break;
        }
    }

    if (!caravanaEncontrada) {
        cout << "Erro: Caravana com ID " << idCaravana << " não
```

```
encontrada.\n";
        return;
    }

    // Verificar se a caravana está numa cidade
    Cidade* cidadeEncontrada = nullptr;
    for (auto& cidade : cidades) {
        if (cidade->getLinha() == caravanaEncontrada->getLinha() &&
            cidade->getColuna() == caravanaEncontrada->getColuna()) {
            cidadeEncontrada = cidade;
            break;
        }
    }

    if (!cidadeEncontrada) {
        cout << "Erro: Caravana não está numa cidade.\n";
        return;
    }

    // Calcular custo total da compra
    int custoTotal = toneladas * precoCompraMercadoria;

    // Verificar se há moedas suficientes
    if (moedas < custoTotal) {
        cout << "Erro: Moedas insuficientes. Custo da compra: " <<
custoTotal
             << ", Moedas disponíveis: " << moedas << ".\n";
        return;
    }

    // Realizar a compra
    if (caravanaEncontrada->adicionaCarga(toneladas)) { // Supondo que
existe um método para adicionar mercadorias
        moedas -= custoTotal;
        cout << "Compra realizada com sucesso! " << toneladas
             << " toneladas de mercadorias adicionadas à caravana "
             << idCaravana << ".\n";
        cout << "Moedas restantes: " << moedas << ".\n";
    } else {
        cout << "Erro: A caravana não pode transportar mais
mercadorias.\n";
    }
}
```

- **Vende:**
  - Função:
    - Vende todas as mercadorias de uma caravana
  - Parâmetros:
    - idCaravana – id da caravana
  - Detalhes de Implementação:
    - Verifica se a caravana existe e está localizada em uma cidade
    - Calcula o lucro com base no preço de venda e a quantidade de mercadoria disponivel
    - Remove as mercadorias da caravana, adiciona o lucro ao total de moedas, e exibe os resultados

```cpp
void Simulacao::vende(int idCaravana) {
    // Verificar se a caravana existe
    Caravana* caravanaEncontrada = nullptr;
    for (auto& caravana : caravanas) {
        if (caravana->getID() == idCaravana) {
            caravanaEncontrada = caravana;
            break;
        }
    }

    if (!caravanaEncontrada) {
        cout << "Erro: Caravana com ID " << idCaravana << " não
encontrada.\n";
        return;
    }

    // Verificar se a caravana está numa cidade
    bool estaNaCidade = false;
    for (auto& cidade : cidades) {
        if (cidade->getLinha() == caravanaEncontrada->getLinha() &&
            cidade->getColuna() == caravanaEncontrada->getColuna()) {
            estaNaCidade = true;
            break;
        }
    }

    if (!estaNaCidade) {
        cout << "Erro: A caravana deve estar numa cidade para vender
mercadorias.\n";
        return;
    }

    // Vender mercadorias
    int mercadorias = caravanaEncontrada->getCarga();
    if (mercadorias <= 0) {
        cout << "Erro: A caravana não possui mercadorias para
vender.\n";
        return;
    }

    int lucro = mercadorias * precoVendaMercadoria; //
`precoVendaMercadoria` é o preço por tonelada
    moedas += lucro; // Atualiza as moedas do jogador
    caravanaEncontrada->adicionaCarga(-mercadorias); // Remove todas
as mercadorias da caravana

    cout << "Venda realizada com sucesso! " << mercadorias << "
toneladas de mercadorias vendidas.\n";
    cout << "Lucro obtido: " << lucro << " moedas.\n";
    cout << "Moedas totais: " << moedas << ".\n";
}
```

- **Tripul:**
  - Função:
    - Contrata tripulantes para uma caravana
  - Parâmetros:
    - idCaravana – id da caravana
    - quantidade - Quantidade de tripulantes a contratar

- Detalhes de Implementação:
  - Verifica se a caravana existe e está localizada em uma cidade
  - Calcula o custo total com base na quantidade de tripulantes solicitada
  - Verifica se há moedas suficientes
  - Se a caravana pode adicionar os tripulantes, deduz as moedas e atualiza a tripulação

```
void Simulacao::tripul(int idCaravana, int quantidade) {
    // Verificar se a caravana existe
    Caravana* caravanaEncontrada = nullptr;
    for (auto& caravana : caravanas) {
        if (caravana->getID() == idCaravana) {
            caravanaEncontrada = caravana;
            break;
        }
    }

    if (!caravanaEncontrada) {
        cout << "Erro: Caravana com ID " << idCaravana << " não
encontrada.\n";
        return;
    }

    // Verificar se a caravana está numa cidade
    bool estaNaCidade = false;
    for (auto& cidade : cidades) {
        if (cidade->getLinha() == caravanaEncontrada->getLinha() &&
            cidade->getColuna() == caravanaEncontrada->getColuna()) {
            estaNaCidade = true;
            break;
        }
    }

    if (!estaNaCidade) {
        cout << "Erro: A caravana deve estar numa cidade para
contratar tripulantes.\n";
        return;
    }

    // Calcular o custo e tentar adicionar tripulantes
    int custoTotal = quantidade * 1;
    if (moedas < custoTotal) {
        cout << "Erro: Moedas insuficientes para contratar " <<
quantidade << " tripulantes.\n";
        return;
    }

    if (caravanaEncontrada->adicionaTripulantes(quantidade)) {
        moedas -= custoTotal; // Deduzir moedas apenas se a adição foi
bem-sucedida
        cout << "Sucesso: " << quantidade << " tripulantes adicionados
à caravana " << idCaravana << ".\n";
        cout << "Custo total: " << custoTotal << " moedas.\n";
        cout << "Moedas restantes: " << moedas << ".\n";
    }else{
         cout << "Aviso: Excede o limite de tripulantes. \n";
    }
}
```

- **Areia:**
  - Função:
    - Simula uma tempestade de areia que afeta caravanas e bárbaros
  - Parâmetros:
    - linha/coluna – Coordenadas centrais da tempestade
    - raio - Alcance da tempestade
  - Detalhes de Implementação:
    - Percorre uma área retangular centrada em (linha, coluna) e verifica cada posição
    - Se uma caravana for encontrada dentro da área afetada, o método tempestadeAreia é chamado
    - Caso a caravana seja destruída, remove a caravana do mapa e da cidade associada, atualizando o mapa e liberando a memória
    - Exibe o mapa atualizado

```
void Simulacao::areia(int linha, int coluna, int raio) {
    cout << "Criando tempestade de areia na posição (" << linha << ",
" << coluna
         << ") com raio " << raio << ".\n";

    // Percorrer todas as posições dentro do quadrado de lado 2*raio +
1
    for (int i = linha - raio; i <= linha + raio; ++i) {
        for (int j = coluna - raio; j <= coluna + raio; ++j) {
            if (i >= 0 && i < mapa.getLinhas() && j >= 0 && j <
mapa.getColunas()) {
                // Verificar se há uma caravana do utilizador na
posição
                for (int k = 0; k < caravanas.size(); ++k) {
                    Caravana* caravana = caravanas[k];
                    if (caravana->getLinha() == i && caravana-
>getColuna() == j) {
                        caravana->tempestadeAreia(); // Aplicar o
efeito da tempestade

                        // Se a caravana foi destruída
                        if (caravana->getDestruida()) {
                            cout << "Caravana " << caravana->getID()
<< " foi destruída pela tempestade.\n";
                            char simb = '.';
                             for (auto& cidade : cidades) {
                                if (cidade->possuiCaravana(caravana-
>getID())) {
                                    cidade->removerCaravana(caravana);
// Remove a caravana da cidade
                                    simb = cidade->getNome();
                                    break;
                                }
                            }

                             mapa.atualizarPosicao(caravana-
>getLinha(), caravana->getColuna(), simb);
                            delete caravana;

                            // Substituir o elemento removido pelo
último
                            caravanas[k] = caravanas.back();
                            caravanas.pop_back(); // Remover o último
```

```
elemento
                            continue; // Evitar incrementar o índice
manualmente
                        }
                    }

                }

                // Verificar se há um bárbaro na posição
                for (int k = 0; k < barbaros.size(); ++k) {
                    CaravanaBarbara* barbaro = barbaros[k];
                    if (barbaro->getLinha() == i && barbaro-
>getColuna() == j) {
                        barbaro->tempestadeAreia(); // Aplicar o
efeito da tempestade

                        // Se o bárbaro foi destruído
                        if (barbaro->getDestruida()) {

                            mapa.atualizarPosicao(barbaro->getLinha(),
barbaro->getColuna(), '.'); // Atualizar o mapa
                            delete barbaro; // Liberar memória

                            // Substituir o elemento removido pelo
último
                            barbaros[k] = barbaros.back();
                            barbaros.pop_back(); // Remover o último
elemento
                            continue; // Evitar incrementar o índice
manualmente
                        }
                    }
                }
            }
        }
    }

    cout << "Tempestade de areia concluída.\n";
    mapa.mostrarMapa();
}
```

- **Move:**
  - Função:
    - Movimenta uma caravana em uma direção especificada
  - Parâmetros:
    - idCaravana – Id da caravana a ser movida
    - direcao - Direção do movimento
  - Detalhes de Implementação:
    - Busca a caravana correspondente pelo idCaravana
    - Calcula a nova posição com base na direção informada
    - Realiza várias verificações:
      - Se a posição é uma montanha, impede o movimento
      - Remove a caravana de uma cidade se ela estiver em uma
      - Se a nova posição for uma cidade, a caravana é movida para ela

- Impede movimentos para posições ocupadas por outras caravanas ou bárbaros

- Atualiza a posição da caravana e o mapa caso o movimento seja válido
- Incrementa os turnos da caravana e a reabastece.

```
void Simulacao::move(int idCaravana, const string& direcao){
    for (auto& caravana : caravanas) {
        if (caravana->getID() == idCaravana) {
            if (caravana->turnosExcedidos()) {
                int linhaAtual = caravana->getLinha();
                int colunaAtual = caravana->getColuna();
                int novaLinha = linhaAtual;
                int novaColuna = colunaAtual;

                // Determinar a nova posição com base na direção
                if (direcao == "D") {
                    novaColuna = (colunaAtual + 1) %
mapa.getColunas();
                } else if (direcao == "E") {
                    novaColuna = (colunaAtual - 1 + mapa.getColunas())
% mapa.getColunas();
                } else if (direcao == "C") {
                    novaLinha = (linhaAtual - 1 + mapa.getLinhas()) %
mapa.getLinhas();
                } else if (direcao == "B") {
                    novaLinha = (linhaAtual + 1) % mapa.getLinhas();
                } else if (direcao == "CE") {
                    novaLinha = (linhaAtual - 1 + mapa.getLinhas()) %
mapa.getLinhas();
                    novaColuna = (colunaAtual - 1 + mapa.getColunas())
% mapa.getColunas();
                } else if (direcao == "CD") {
                    novaLinha = (linhaAtual - 1 + mapa.getLinhas()) %
mapa.getLinhas();
                    novaColuna = (colunaAtual + 1) %
mapa.getColunas();
                } else if (direcao == "BE") {
                    novaLinha = (linhaAtual + 1) % mapa.getLinhas();
                    novaColuna = (colunaAtual - 1 + mapa.getColunas())
% mapa.getColunas();
                } else if (direcao == "BD") {
                    novaLinha = (linhaAtual + 1) % mapa.getLinhas();
                    novaColuna = (colunaAtual + 1) %
mapa.getColunas();
                } else {
                    cout << "Erro: Direção inválida.\n";
                    return;
                }

                // Verificar se a nova posição é válida
                char conteudo = mapa.obterConteudo(novaLinha,
novaColuna);
                if (conteudo == '+') {
                    cout << "Erro: A posição (" << novaLinha << ", "
<< novaColuna << ") é uma montanha. Movimento não permitido.\n";
                    return;
                }
```

```
                // Verificar se a caravana está em uma cidade e
remover da cidade
                for (auto& cidade : cidades) {
                    if (cidade->possuiCaravana(idCaravana)) {
                        cidade->removerCaravana(caravana); // Remove a
caravana da cidade
                        cout << "Caravana " << idCaravana << " saiu da
cidade " << cidade->getNome() << ".\n";
                        caravana->setPosicao(novaLinha, novaColuna);
// Atualizar posição da caravana
                        mapa.atualizarPosicao(novaLinha, novaColuna,
'1' + idCaravana - 1); // Atualizar o mapa
                        cout << "Caravana " << idCaravana << " movida
para (" << novaLinha << ", " << novaColuna << ") na direção " <<
direcao << ".\n";
                        caravana->adicionaTurno();
                        caravana->reabastecerAgua();
                        return;
                    }
                }


                // Verificar se a nova posição é uma cidade
                for (auto& cidade : cidades) {
                    if (cidade->getLinha() == novaLinha && cidade-
>getColuna() == novaColuna) {
                        cidade->adicionarCaravana(caravana); //
Adicionar a caravana à nova cidade
                        mapa.atualizarPosicao(linhaAtual, colunaAtual,
'.'); // Limpar posição antiga no mapa
                        cout << "Caravana " << idCaravana << " entrou
na cidade " << cidade->getNome() << ".\n";
                        caravana->setPosicao(novaLinha,novaColuna);
                        caravana->adicionaTurno();
                        caravana->reabastecerAgua();
                        return;
                    }
                }

                 // Verificar se já existe uma caravana na nova
posição
                for (const auto& outraCaravana : caravanas) {
                    if (outraCaravana->getLinha() == novaLinha &&
outraCaravana->getColuna() == novaColuna) {
                        cout << "Erro: A posição (" << novaLinha << ",
" << novaColuna << ") já está ocupada por outra caravana.\n";
                        return;
                    }
                }

                // Verificar se a nova posição está ocupada por um
bárbaro
                for (const auto& barbaro : barbaros) {
                    if (barbaro->getLinha() == novaLinha && barbaro-
>getColuna() == novaColuna) {
                        cout << "Erro: A posição (" << novaLinha << ",
" << novaColuna << ") está ocupada por um bárbaro.\n";
                        return;
                    }
                }
```

```
                // Mover a caravana para a nova posição
                mapa.atualizarPosicao(linhaAtual, colunaAtual, '.');
// Limpar posição antiga no mapa
                caravana->setPosicao(novaLinha, novaColuna); //
Atualizar posição da caravana
                mapa.atualizarPosicao(novaLinha, novaColuna, '1' +
idCaravana - 1); // Atualizar o mapa
                cout << "Caravana " << idCaravana << " movida para ("
<< novaLinha << ", " << novaColuna << ") na direção " << direcao <<
".\n";
                caravana->adicionaTurno();
                return;
            } else {
                cout << "Erro: A caravana excedeu o limite de
movimentos permitidos.\n";
                return;
            }
        }
    }

    cout << "Erro: Caravana com ID " << idCaravana << " não
encontrada.\n";
}
```

- **Lists:**
  - Função:
    - Lista todas as cópias de buffer existentes na simulação
  - Detalhes de Implementação:
    - Se o vetor buffers estiver vazio, exibe uma mensagem indicando que não há buffers
    - Caso contrário, itera sobre os buffers e imprime seus nomes

```
void Simulacao::lists() const {
    if (buffers.empty()) {
        cout << "Nenhuma cópia de buffer encontrada.\n";
        return;
    }

    cout << "Cópias do buffer existentes:\n";
    for (const auto& buffer : buffers) {
        cout << "- " << buffer->getNome() << "\n";
    }
}
```

- **Dels:**
  - Função:
    - Remove uma cópia de buffer pelo nome
  - Parâmetros:
    - Nome – nome do buffer a ser excluído
  - Detalhes de Implementação:
    - Itera sobre o vetor buffers para encontrar um buffer com o nome correspondente.
    - Se encontrado, deleta o buffer da memória, remove-o do vetor e exibe uma mensagem de sucesso.
    - Se não encontrado, exibe um erro

```
void Simulacao::dels(const string& nome) {
    bool encontrado = false;

    // Percorre todos os buffers para encontrar o que tem o nome
correspondente
    for (auto it = buffers.begin(); it != buffers.end(); ++it) {
        if ((*it)->getNome() == nome) {
            delete *it;  // Apaga a memória alocada
            buffers.erase(it);  // Remove o ponteiro do vetor
            cout << "Cópia do buffer '" << nome << "' apagada com
sucesso.\n";
            encontrado = true;
            break;  // Sai do loop após excluir o buffer
        }
    }

    if (!encontrado) {
        cout << "Erro: Cópia do buffer '" << nome << "' não
encontrada.\n";
    }
}
```

- **Terminar:**
  - Função:
    - Finaliza a simulação e reinicia os valores e estruturas
  - Detalhes de Implementação:
    - Exibe a pontuação final da simulação
    - Dá reset variáveis de controle
    - Apaga e limpa todos os objetos dinâmicos alocados

```
void Simulacao::terminar() {
    // Exibir a pontuação final
    cout << "Simulação terminada. Pontuação final: " <<
combatesVencidos << endl;

    // Reiniciar a simulação (fazendo a limpeza e reinicializando os
valores)
    // Reinicia os parâmetros e variáveis da simulação
    // Exemplo de redefinir variáveis:
    moedas = 0;
    turno = 0;
    combatesVencidos = 0;
    instantesEntreItens = 0;
    duracaoItem = 0;
    maxItens = 0;
    precoVendaMercadoria = 0;
    precoCompraMercadoria = 0;
    precoCaravana = 0;
    instantesEntreBarbaros = 0;
    duracaoBarbaros = 0;

    // Apagar todos os objetos criados (caravanas, cidades, itens,
etc.)
    for (auto& caravana : caravanas) {
        delete caravana;
    }
    caravanas.clear();

    for (auto& cidade : cidades) {
```

```
        delete cidade;
    }
    cidades.clear();

    for (auto& item : items) {
        delete item;
    }
    items.clear();

    for (auto& buffer : buffers) {
        delete buffer;
    }
    buffers.clear();

    for (auto& barbaro : barbaros) {
        delete barbaro;
    }
    barbaros.clear();

    fase = 1;
    iniciaSimulacao();
}
```

- **Loads:**
  - Função:
    - Carrega um buffer utilizando o nome como pesquisa
  - Parâmetros:
    - nome – Nome do buffer a procurar
  - Detalhes de Implementação:
    - Itera sobre o vetor para encontrar um buffer com o nome especificado
    - Se encontrado, exibe uma mensagem de sucesso e chama o método imprimir() do buffer para mostrar seu estado
    - Caso contrário, exibe uma mensagem de erro.

```
void Simulacao::loads(const string& nome) {
    // Itera sobre o vetor de buffers manualmente
    for (Buffer* buffer : buffers) {
        // Comparando o nome do buffer com o nome fornecido
        if (buffer->getNome() == nome) {
            // Se o buffer for encontrado, exibe as informações dele
            cout << "Buffer '" << nome << "' recuperado com
sucesso.\n";
            buffer->imprimir();  // Assumindo que você tem uma função
para mostrar o estado
            return;  // Sai da função após encontrar o buffer
        }
    }

    // Caso não tenha encontrado o buffer
    cout << "Erro: Cópia do buffer '" << nome << "' não
encontrada.\n";
}
```

- **Saves:**
  - Função:
    - Salva o estado atual em um novo buffer
  - Parâmetros:
    - nome – Nome do buffer a ser salvo
  - Detalhes de Implementação:
    - Verifica se já existe um buffer com o nome especificado
    - Se não existir:
      - Cria um objeto Buffer com o estado atual e o nome fornecido
      - Adiciona o novo buffer ao vetor buffers
      - Exibe uma mensagem de sucesso
    - Caso contrario exibe uma mensagem de erro indicando que nome já está em uso

```
void Simulacao::saves(const string& nome) {
    // Verifica se o nome do buffer já existe
    for (const auto& buffer : buffers) {
        if (buffer->getNome() == nome) {
            cout << "Erro: Já existe um buffer com o nome '" << nome
<< "'.\n";
            return;
        }
    }

    // Cria uma nova cópia do buffer visual
    Buffer* novoBuffer = new Buffer(layout , nome);
    cout << novoBuffer->getNome() << endl;

    // Armazena a cópia do buffer na lista de buffers
    buffers.push_back(novoBuffer);
    cout << "Cópia do estado visual do buffer '" << nome << "'
armazenada com sucesso.\n";
```

- **Atualizar Caravanas:**
  - Função:
    - Atualiza o estado de todas as caravanas do jogo
  - Detalhes de Implementação:
    - Cada caravana reinicia os turnos e consome água
    - Se uma caravana for destruída, é removida do mapa e da lista de caravanas
    - Se a caravana está em modo automático e tem tripulantes, executa movimentos autônomos.
    - Se não há tripulantes, tenta se mover como caravana vazia. Se falhar, a caravana desaparece

```
void Simulacao::atualizarCaravanas() {
    for (auto it = caravanas.begin(); it != caravanas.end(); ++it) {
        auto& caravana = *it;

        caravana->resetTurnos();
        caravana->atualizarAgua();

        if(caravana->getDestruida()){
```

```
            mapa.atualizarPosicao(caravana->getLinha(), caravana-
>getColuna(), '.');
            delete caravana;
            it = caravanas.erase(it); // Remover e avançar o iterador
        }

        if (caravana->getAutomatico()) {
            if (caravana->getTripulantes() > 0) {
                // Movimento autónomo (modo automático)
                caravana->movimentoAutonomo(barbaros, caravanas,
items, mapa.getLinhas(), mapa.getColunas(), mapa);
                ++it; // Avançar se a caravana não for removida
            } else {
                // Movimento sem tripulantes
                if (caravana-
>movimentoSemTripulantes(mapa.getLinhas(), mapa.getColunas(), mapa)) {
                    mapa.atualizarPosicao(caravana->getLinha(),
caravana->getColuna(), '.');
                    cout << "Caravana " << caravana->getID() << "
desapareceu por falta de tripulantes.\n";
                    delete caravana;
                    it = caravanas.erase(it); // Remover e avançar o
iterador
                } else {
                    ++it; // Apenas avançar se a caravana não for
removida
                }
            }
        }
    }
}
```

- **Prox:**
  - Função:
    - Avança a simulação em n turnos
  - Parâmetros:
    - n – Quantidade de vezes que se avança a simulação
  - Detalhes de Implementação:
    - Atualiza caravanas, itens, bárbaros, e gerência combates
    - Gera o estado atual do mapa e o exibe a cada turno
    - Incrementa o contador de turnos

```
void Simulacao::prox(int n) {
    if (n <= 0) {
        cout << "Erro: O número de instantes deve ser maior que 0.\n";
        return;
    }

    cout << "Avançando " << n << " instantes...\n";

    for (int i = 0; i < n; ++i) {
        cout << "Turno " << (turno + 1) << ":\n";

        // Atualizar caravanas (modo automático ou sem tripulantes)
        atualizarCaravanas();

        // Atualizar os itens no mapa
        atualizarItens();
```

```
        // Gerar e mover bárbaros
        atualizarBarbaros();

        // Gerir combates entre caravanas e bárbaros
        gerirCombates();

        // Atualizar o buffer/layout do mapa
        montarBufferLayout(layout);

        // Mostrar o estado do mapa e informações adicionais
        layout.imprimir();

        // Incrementar o contador de turnos
        turno++;

        // Separador entre turnos (para melhor visualização)
        if (i < n - 1) {
            cout << "============================\n";
        }
    }

    cout << "Avanço concluído.\n";
}
```

- **Exec:**
  - Função:
    - Lê um arquivo contendo comandos e os executa
  - Parâmetros:
    - nomeFicheiro – Nome do ficheiro a ser lido
  - Detalhes de Implementação:
    - Cada linha é processada
    - Mostra mensagens de erro caso o arquivo não seja encontrado ou os comandos estejam incorretos

```
void Simulacao::exec(const string& nomeFicheiro) {
    ifstream ficheiro(nomeFicheiro);
    if (!ficheiro.is_open()) {
        cout << "Erro: Não foi possível abrir o ficheiro '" <<
nomeFicheiro << "'.\n";
        return;
    }

    cout << "Executando comandos do ficheiro '" << nomeFicheiro <<
"':\n";

    string linha;
    int linhaAtual = 1; // Contador de linhas
    while (getline(ficheiro, linha)) {
        if (!linha.empty()) {
            cout << "Linha " << linhaAtual << ": " << linha << "\n";
            try {
                processarComando(linha); // Reutiliza o método para
interpretar o comando
            } catch (const exception& e) {
                cout << "Erro ao processar o comando na linha " <<
linhaAtual << ": " << e.what() << "\n";
            }
```

```
        }
        linhaAtual++;
    }

    ficheiro.close();
    cout << "Execução concluída.\n";
}
```

- **Processar Comando:**
  - Função:
    - Interpreta e executa comandos passados pelo utilizador
  - Parâmetros:
    - comando – Comando a ser executado
  - Detalhes de Implementação:
    - Procura o comando a ser executado e executa-o

```
void Simulacao::processarComando(const string& comando) {
    // Dividir o comando em palavras
    istringstream ss(comando);
    vector<string> args;
    string arg;
    while (ss >> arg) {
        args.push_back(arg);
    }

    if (args.empty()) return;


    if(fase == 1){
        if (args[0] == "config") {
            if (args.size() == 2) {
                config(args[1]);
                fase = 2;
            } else {
                cout << "Erro: Argumentos inválidos para comando
'config'.\n";
            }
        }
        else if (args[0] == "sair") {
            // Encerra o programa
            cout << "Saindo do programa...\n";
            exit(0);
        }else {
            cout << "Comando desconhecido: " << args[0] << "\n";
        }
    } else if(fase == 2){
        if(args[0] == "caravana"){
            if (args.size() == 2) {
                // Convertendo o argumento de string para inteiro
                try {
                    int idCaravana = stoi(args[1]);
                    caravana(idCaravana);  // Passa o id como inteiro
                } catch (const invalid_argument& e) {
                    cout << "Erro: ID da caravana inválido.\n";
                }
            } else {
                cout << "Erro: Argumentos inválidos para comando
'caravana'.\n";
            }
```

```
        }
        else if(args[0] == "cidade"){
            if (args.size() == 2) {
                char c = args[1][0];
                cidade(c);
            }else {
                cout << "Erro: Argumentos inválidos para comando
'cidade'.\n";
            }
        }
        else if(args[0] == "precos"){
            precos();
        }
        else if(args[0] == "auto"){
            if(args.size() == 2){
                try {
                    int idCaravana = stoi(args[1]);
                    autoGestao(idCaravana);  // Passa o id como
inteiro
                } catch (const invalid_argument& e) {
                    cout << "Erro: ID da caravana inválido.\n";
                }
            }else {
                cout << "Erro: Argumentos inválidos para comando
'auto'.\n";
            }
        }
        else if(args[0] == "stop"){
            if(args.size() == 2){
                try {
                    int idCaravana = stoi(args[1]);
                    stop(idCaravana);  // Passa o id como inteiro
                } catch (const invalid_argument& e) {
                    cout << "Erro: ID da caravana inválido.\n";
                }
            }else {
                cout << "Erro: Argumentos inválidos para comando
'stop'.\n";
            }
        }
        else if(args[0] == "moedas"){
            if(args.size() == 2){
                int quantMoedas = stoi(args[1]);
                addMoedas(quantMoedas);  // Passa o id como inteiro
            }else {
                cout << "Erro: Argumentos inválidos para comando
'moedas'.\n";
            }
        }
        else if(args[0] == "barbaro"){
            if(args.size() == 3){
                barbaro(stoi(args[1]) , stoi(args[2]));
            }else {
                cout << "Erro: Argumentos inválidos para comando
'barbaro'.\n";
            }
        }
        else if (args[0] == "comprac") {
            if (args.size() == 3) {
                char cidade = args[1][0];  // Obter a cidade
                char tipoCaravana = args[2][0];  // Obter o tipo de
```

```
caravana

                // Validar o tipo de caravana
                if (tipoCaravana == 'C' || tipoCaravana == 'M') {
                    comprac(cidade, tipoCaravana);
                } else {
                    cout << "Erro: Tipo de caravana inválido. Use 'C',
'M' ou 'S'.\n";
                }
            } else {
                cout << "Erro: Argumentos inválidos para comando
'comprac'.\n";
            }
        }
        else if (args[0] == "compra") {
            if (args.size() == 3) {
                try {
                    int idCaravana = stoi(args[1]); // Converte o ID
da caravana para inteiro
                    int toneladas = stoi(args[2]); // Converte o
número de toneladas para inteiro

                    if (toneladas <= 0) {
                        cout << "Erro: O número de toneladas deve ser
maior que zero.\n";
                        return;
                    }

                    compra(idCaravana, toneladas); // Chama a função
de compra
                } catch (const invalid_argument& e) {
                    cout << "Erro: Argumentos inválidos para o comando
'compra'. Certifique-se de usar números inteiros.\n";
                }
            } else {
                cout << "Erro: Uso correto do comando: compra
<ID_Caravana> <Toneladas>\n";
            }
        }
        else if (args[0] == "vende") {
            if (args.size() == 2) {
                try {
                    int idCaravana = stoi(args[1]); // Converte o ID
da caravana para inteiro
                    vende(idCaravana); // Chama a função de venda
                } catch (const invalid_argument& e) {
                    cout << "Erro: ID da caravana inválido.\n";
                }
            } else {
                cout << "Erro: Uso correto do comando: vende
<ID_Caravana>\n";
            }
        }
        else if (args[0] == "tripul") {
            if (args.size() == 3) {
                try {
                    int idCaravana = stoi(args[1]);  // Convertendo ID
da caravana
                    int quantidade = stoi(args[2]);  // Convertendo a
quantidade de tripulantes
                    tripul(idCaravana, quantidade);  // Chama a função
```

```
de adicionar tripulantes
                } catch (const invalid_argument& e) {
                    cout << "Erro: Argumentos inválidos para comando
'tripul'.\n";
                }
            } else {
                cout << "Erro: Argumentos inválidos para comando
'tripul'.\n";
            }
        }
        else if (args[0] == "areia") {  // Adicionado o comando areia
            if (args.size() == 4) {
                try {
                    int linha = stoi(args[1]); // Linha da tempestade
                    int coluna = stoi(args[2]); // Coluna da
tempestade
                    int raio = stoi(args[3]); // Raio da tempestade

                    if (raio < 0) {
                        cout << "Erro: O raio da tempestade deve ser
não negativo.\n";
                    } else {
                        areia(linha, coluna, raio); // Chama o método
correspondente
                    }
                } catch (const invalid_argument& e) {
                    cout << "Erro: Argumentos inválidos para comando
'areia'.\n";
                }
            } else {
                cout << "Erro: Uso correto do comando: areia <linha>
<coluna> <raio>\n";
            }
        }
        else if (args[0] == "move") {
            if (args.size() == 3) {
                try {
                    int idCaravana = stoi(args[1]); // Obter o ID da
caravana
                    string direcao = args[2];       // Obter a direção
de movimento

                    // Validar a direção
                    if (direcao == "D" || direcao == "E" || direcao ==
"C" || direcao == "B" ||
                        direcao == "CE" || direcao == "CD" || direcao
== "BE" || direcao == "BD") {
                        move(idCaravana, direcao); // Chama o método
de movimentação
                        mapa.mostrarMapa();
                    } else {
                        cout << "Erro: Direção inválida. Use D, E, C,
B, CE, CD, BE, BD.\n";
                    }
                } catch (const invalid_argument& e) {
                    cout << "Erro: Argumentos inválidos para comando
'move'.\n";
                }
            } else {
                cout << "Erro: Uso correto do comando: move
<ID_Caravana> <Direção>\n";
```

```
            }
        }
        else if (args[0] == "saves") {
            if (args.size() == 2) {
                saves(args[1]);  // Chama o método saves para criar
uma cópia do estado visual do buffer
            } else {
                cout << "Erro: Argumentos inválidos para comando
'saves'.\n";
            }
        }
        else if(args[0] == "loads") {
            if (args.size() == 2) {
                loads(args[1]);  // Chama o método loads para
recuperar o buffer
            } else {
                cout << "Erro: Argumentos inválidos para comando
'loads'.\n";
            }
        }

        else if (args[0] == "lists") {
            lists();  // Chama o método lists
        }
        else if (args[0] == "dels") {
            if (args.size() == 2) {
                dels(args[1]);  // Chama o método dels
            } else {
                cout << "Erro: Argumentos inválidos para comando
'dels'.\n";
            }
        }
        else if (args[0] == "prox") {
            if (args.size() == 1) {
                // Se nenhum valor for especificado, avança 1 turno
por padrão
                prox(1);
            } else if (args.size() == 2) {
                try {
                    int n = stoi(args[1]); // Converte o argumento
para inteiro
                    if (n > 0) {
                        prox(n); // Chama o método prox com o valor
especificado
                    } else {
                        cout << "Erro: O número de turnos deve ser
maior que 0.\n";
                    }
                } catch (const invalid_argument&) {
                    cout << "Erro: Argumento inválido para comando
'prox'. Certifique-se de usar um número inteiro.\n";
                }
            } else {
                cout << "Erro: Uso correto do comando 'prox <n>'.\n";
            }
        }
        else if (args[0] == "exec") {
            if (args.size() == 2) {
                exec(args[1]); // Chama o método `exec` com o nome do
ficheiro
            } else {
```

```
                cout << "Erro: Uso correto do comando 'exec
<nomeFicheiro>'.\n";
            }
        }

        else if(args[0] == "terminar"){
            terminar();
        }
        else {
            cout << "Comando desconhecido: " << args[0] << "\n";
        }
    }

    montarBufferLayout(layout);
}
```

- **Inicia Simulação:**
  - Função:
    - Inicia a Simulação
  - Detalhes de Implementação:
    - Lê comandos do utilizador
    - Executa o comando

```
void Simulacao::iniciaSimulacao() {
    string cmd;
    while (1) {
        getline(cin, cmd);
        processarComando(cmd);
    }
}
```

Main.cpp

- Configura o gerador de números aleatórios
- Cria uma instância da classe Simulacao
- Inicia o loop interativo para permitir ao jogador interagir com a simulação.
- O programa continua a executar até o jogador decidir terminar

```
int main() {
    srand(time(0));
    Simulacao simulacao;

    simulacao.iniciaSimulacao();

    return 0;
}
```