Universidade Federal de Viçosa
Departamento de Informática
INF 112 - Programação II
2º Semestre de 2025
Professor: Julio Cesar Soares dos Reis
Estagiários de Ensino: Carolina Sarah Pacceli Ferreira e Samuel Raimundo Lopes Pinto
Valor: 6 Pontos

Trabalho Prático 2

Motivação

A motivação deste trabalho é rever e aplicar conceitos básicos de programação orientada a objetos, herança, polimorfismo, etc.

Descrição/Objetivo

Neste trabalho, você irá desenvolver um sistema que simule o Jogo de tabuleiro War [1]. Lançado em 1972 pela empresa *Grow* e baseado no jogo estadunidense *Risk de Albert Lamorisse*, War é um jogo de estratégia e sorte onde os jogadores tentam alcançar a dominação mundial através da conquista militar e da gestão de exércitos.

Figura 1: Tabuleiro do War.

Detalhes do Jogo e Principal Objetivo

- O jogo é composto por um **mapa-múndi** dividido em 6 continentes: América do Norte, América do Sul, Europa, África, Ásia e Oceania, que juntos totalizam **42 territórios**.
- O **objetivo** é definido de forma aleatória. No início da partida, cada jogador recebe um objetivo (ex.: conquistar 24 territórios ou destruir outro jogador).
- O primeiro jogador a cumprir seu objetivo **vence** o jogo.
- O jogo começa com a distribuição aleatória dos 42 territórios entre os jogadores. Na distribuição inicial, os jogadores devem colocar **1 exército** em cada território que receberam.

Como Jogar

O jogo se desenvolve em várias rodadas até que um jogador atinja seu objetivo. Cada rodada se divide em 3 ações obrigatórias para o jogador da vez:

[1] Materiais com informações adicionais: Wikipédia (`https://pt.wikipedia.org/wiki/War`), War Net (`https://warnet.com.br/`)

1. **Fase de Reforço:** O jogador calcula a quantidade de exércitos de reforço a serem distribuídos seguindo a seguinte fórmula:

$$\text{Exércitos a serem distribuídos} = \left\lfloor \frac{\text{Número de territórios do jogador}}{2} \right\rfloor$$

   Estes exércitos devem ser posicionados estrategicamente em territórios dominados.

2. **Fase de Ataque:** O jogador pode realizar N ataques consecutivos contra territórios inimigos, seguindo as regras de combate.

3. **Fase de Movimentação:** O jogador pode movimentar seus exércitos entre seus territórios adjacentes.

Formas de Combate e Exércitos (Polimorfismo)

O sistema deve aplicar as seguintes regras de combate, que utilizam o conceito de Polimorfismo:

- **Restrição de Exércitos:** O território atacante **não pode** iniciar um ataque se possuir menos de **2 exércitos** (um exército deve sempre permanecer no território de origem).
- **Combate Simplificado (1 *vs.* 1):** O ataque e a defesa são resolvidos com **1 dado contra 1 dado**. O dado com o maior valor vence. Em caso de empate, o defensor vence. O perdedor perde 1 exército do território envolvido (se o atacante perde, perde 1 no seu território de origem; se o defensor perde, perde 1 no território atacado).
- **Exército Terrestre:** Só pode atacar territórios que sejam **fronteiras diretas** do território de origem.
- **Exército Aéreo: Não existe limitação de fronteira**. Pode atacar **qualquer território** no mapa, desde que o território de origem pertença ao atacante.

Especificação das Funcionalidades Gerais Básicas

Visão Geral

O sistema deverá ser capaz de simular uma partida do jogo War, começando com a preparação (definição do número de jogadores, distribuição de territórios e objetivos) até o fim do jogo (quando um jogador alcança seu objetivo).

- **Preparação:** O jogo se inicia com a leitura do mapa (a partir da leitura do arquivo de configuração `.txt`) e a definição do número de jogadores (mínimo 2 e máximo 6).
- **Distribuição dos territórios:** Os **42 territórios** do mapa devem ser distribuídos igualmente e de forma aleatória entre os jogadores.
- **Distribuição dos objetivos:** Cada jogador receberá um **objetivo** definido aleatoriamente.
- **Adição de exércitos:** Em cada rodada, um jogador recebe uma quantidade de exércitos equivalente a: $\left\lfloor \frac{\text{Número de territórios do jogador}}{2} \right\rfloor$. Estes exércitos devem ser posicionados em seus territórios.
- **Rodadas do jogo:** Cada rodada consiste em: 1) Adição de exércitos, 2) Ataque e 3) Movimentação dos exércitos.

Regras de Ataque e Combate Simplificado

Para o ataque, você deve considerar as seguintes restrições:

- **Exército Aéreo:** Pode atacar **qualquer território** no mapa, independentemente de fronteira.
- **Exército Terrestre:** O ataque só pode ocorrer em territórios que possuam **fronteira** com o território de origem.
- **Limite de Ataque:** O território de origem **só pode atacar** se o número de exércitos for ≥ 2 (Observação: sempre precisa restar um exército no território de origem).
- **Combate (1 *vs.* 1):** Considere, por simplicidade, apenas 1 dado de ataque e 1 dado de defesa. O dado com o **maior valor vence**. Em caso de empate, o defensor vence. O jogador que perdeu no dado perde **1** exército do território em questão.
- **Conquista:** Caso o território atacado fique sem exércitos, ele é conquistado, e **1** exército do território atacante deve ser imediatamente alocado para o território conquistado.

Especificações de Código e Arquitetura (Herança e Polimorfismo)

Seu trabalho deverá ter pelo menos 8 classes: `Jogo`, `Jogador`, `Exercito`, `ExercitoAereo`, `ExercitoTerrestre`, `Território`, `Continente` e `Carta`.

Classes `Território` e `Continente`

- **Classe** `Território` representa um país no jogo.
  - **Construtor/Destrutor:** O construtor deve inicializar o nome, definir a quantidade de fronteiras e alocar memória para elas (ponteiros para `Território`). O Destrutor deve desalocar a memória.
  - **Gets:** `getNome` (retorna o nome do território), `getExercitos` (retorna a quantidade de exércitos naquele território), `getFronteiras` (retorna o array de ponteiros que representa as fronteiras daquele território).
  - **Sets:** `setNome` (atualiza o nome do território).
  - **Outros Métodos:** `adicionarExercitos` (recebe como parâmetros a quantidade de exércitos a ser adicionada e o território escolhido), `removerExercitos` (recebe como parâmetros a quantidade de exércitos a ser removido e o território escolhido), `adicionarFronteira` (recebe como parâmetro o território que faz fronteira com o território atual).
- **Classe** `Continente` representa cada um dos 6 continentes do jogo.
  - **Construtor/Destrutor:** Inicializa o nome do continente e a quantidade de territórios que ele possui. Destrutor deve desalocar a memória do array de territórios.
  - **Gets:** `getTerritorios` (retorna o array de ponteiros), `getNome` (retorna o nome do continente).
  - **Sets:** `setNome` (atualiza o nome do continente).
  - **Outros Métodos:** `adicionarTerritorio` (recebe como parâmetro o território a ser adicionado no continente), `removerTerritorio` (recebe como parâmetro o território a ser removido do continente).

Classes `Exercito`, `ExercitoAereo` e `ExercitoTerrestre`

A classe `Exercito` é a **classe base** (abstrata), e as outras duas são **classes derivadas** que aplicam o polimorfismo no método `ataque`.

- **Classe** `Exercito` **(Base):**
  - **Construtor/Destrutor:** Inicializa o nome do exército e o dono. Destrutor deve desalocar a memória caso necessário.
  - **Gets/Sets:** `getNome`, `setNome`, `getDono` (retorna o Jogador dono do exército), `setDono` (atualiza o dono do exército).
  - **Métodos Principais:**
    - `rolarDado`: Deve gerar 2 números aleatórios entre 1 e 6, cada um representando o ataque e defesa respectivamente, após gerar os números ele deve comparar os mesmos e retornar o vencedor da comparação (ataque ou defesa).
    - `ataque`: **Método Puro Virtual**. Receberá os territórios de origem/destino como parâmetros e implementará a lógica de combate e conquista.
- **Classes** `ExercitoAereo` **e** `ExercitoTerrestre` **(Derivadas):**
  - **Construtor/Destrutor:** Inicializa o nome do exército e o dono. Destrutor deve desalocar a memória caso necessário.
  - Herdam **Gets** e **Sets**.
  - Implementam o método `ataque` com a **lógica de restrição de fronteira** específica para o seu tipo (aéreo ignora fronteira; terrestre exige fronteira).

Classes `Jogo`, `Jogador` e `Carta`

- **Classe `Jogo` (Controlador):**
  - **Construtor/Destrutor:** O construtor deve inicializar o jogo como um todo (número de jogadores, quantidade de territórios, quantidade de continentes e quantidade de objetivos). Destrutor deve desalocar a memória caso necessário.
  - **Gets/Sets:** `getJogadorDaVez` (retorna o Jogador que tem a vez), `setJogadorDaVez` (atualiza o jogador que tem a vez).
  - **Métodos relacionados a rodada:**
    - `iniciarJogada`: Coordena a as ações relacionadas a vez do jogador (ataque e verificações necessárias para efetuar a jogada e também deve passar a vez para o próximo jogador).
    - `fimDoJogo`: Verifica a condição de vitória e retorna o nome do jogador vencedor, se houver.
    - `distribuirExercitos`: esse método deve receber como parâmetros a quantidade de exércitos a ser deslocado de um território (origem) para outro (destino) e o nome dos territórios
    - `distribuirExercitos`: esse método deve receber como parâmetros a quantidade de exércitos a ser colocado em um território e o nome do território
- **Classe `Jogador`:**
  - **Gets:** `getNumTerritorios`, `getObjetivo`.
  - **Outros Métodos:** `adicionarTerritorio`: deverá receber como parâmetros a quantidade de exércitos e o território (observação: um jogador não pode ter um território sem nenhum exército).
- **Classe `Carta`:** Deve armazenar informações relacionadas aos objetivos.
  - **Gets/Sets:** `getObjetivo`, `setObjetivo`.

Arquivo Principal

Seu arquivo principal (`main.cpp`) deve seguir uma estrutura similar a exibida abaixo:

Passo 1: Iniciar o Jogo

Seguindo a estrutura explicada anteriormente o primeiro passo é a inicialização do jogo:

```
int numero_territorios = 2;
int numero_de_jogadores = 2;
int numero_de_continentes = 6;
int numero_de_objetivos = 4;
int numero_de_fronteiras = 4;
int numero_de_exercitos;
string nome_do_territorio_de_destino, nome_do_territorio_de_origem;

Jogo war(numero_de_jogadores, numero_territorios, numero_de_continentes, numero_de_objetivos);
```

Passo 2: Organizar o jogo

O passo 2 é onde os territórios, objetivos e exércitos serão distribuídos, também é nesse método que você deve adicionar os exércitos aos territórios de acordo com o jogador dono daquele exército e território.

```
war.organizarJogo();
```

Passo 3: Jogar

No passo 3 ocorre o jogo em si, no exemplo abaixo, por simplicidade foi considerado que cada jogador irá fazer uma única jogada e nestes moldes ele irá fazer a primeira distribuição dos exércitos, iniciar a jogada (atacar o adversário e passar a vez) e redistribuir os exércitos ao final. Neste exemplo de `main.cpp` a cada jogada é verificado se o jogador atingiu o objetivo, caso tenha atingido o jogo acaba.

```
while (war.fimDoJogo() == "nao")
    {
        cin >> numero_de_exercitos >> nome_do_territorio_de_destino;
        war.distribuirExercitos(numero_de_exercitos, nome_do_territorio_de_destino, war.
            getJogadorDaVez());
        war.iniciarJogada(war.getJogadorDaVez());
        war.distribuirExercitos(numero_de_exercitos, nome_do_territorio_de_origem,
            nome_do_territorio_de_destino);
    }
```

Dicas e Recomendações

Para facilitar o desenvolvimento atente-se aos seguintes pontos durante a realização deste trabalho:

1. **Gerenciamento de Memória e Ponteiros:**
   - Este projeto envolve muita **alocação dinâmica de memória** (`new` como, por exemplo, em objetos do tipo `Território`, `Jogo` e `Continente`). É crucial implementar **destrutores** em todas as classes que alocam memória (`~Classe()`) para evitar **vazamentos**.
   - Lembre-se: para cada `new`, precisa ter um `delete` (e `delete[]` para arrays).
   - Para verificar se todos os espaços foram liberados corretamente, você deve utilizar a ferramenta de depuração **Valgrind** (https://valgrind.org).
2. **Programação Orientada a Objetos (POO):**
   - **Visualização da Arquitetura:** Antes de começar a implementar tente visualizar a organização das 8 classes e como elas se relacionam. Entender como `Jogo` se conecta a `Jogador` e como `Território` se conecta a `Exercito` simplificará a implementação.
   - **Encapsulamento e Acesso:** Utilize os modificadores de acesso (`private`, `public`, e `protected`) para garantir o encapsulamento adequado. Os atributos devem ser `private` e acessados via *getters* e *setters*.
   - **Cooperação de Classes:** Observe atentamente a **cadeia de chamadas**. Muitos métodos serão implementados através da coordenação de outras classes (Ex: o método `ataque` da classe `Exercito` precisará chamar métodos de `Território`).
3. **Polimorfismo e Herança:**
   - Garanta o uso correto de **Herança** e **Polimorfismo** na hierarquia das classes do tipo `Exercito`.
4. **Escopo e Adaptações das Regras:**
   - **Simplicidade:** Diversas regras do jogo original foram simplificadas ou desconsideradas (Ex: 1 dado *vs.* 1 dado no combate) para simplificar a implementação.
   - **Dados de Entrada:** Você receberá um arquivo de configuração (`territorios.txt`). **Não é obrigatório** testar o trabalho com todos os 40 territórios e todos os objetivos; concentre-se em demonstrar que a lógica implementada funciona, que os conceitos de Programação Orientada ao Objeto foram entendidos e utilizados corretamente e que o jogo funcionam para um conjunto de dados.
5. **Liberdade de Implementação:**
   - **Funções Auxiliares:** Fique à vontade para criar e utilizar funções auxiliares privadas, desde que todas as funcionalidades descritas nas especificações principais estejam devidamente implementadas.
   - **Distribuição de Exércitos:** A regra para a alocação dos exércitos de reforço (*qual território recebe qual tipo de exército*) é flexível. Seja criativo e defina uma lógica clara para a distribuição dos exércitos aéreos e terrestres.
   - **Funções Auxiliares:** Para verificação dos objetivos que estão relacionados a conquista de algum continente sugiro adicionar contadores para facilitar a verificação da conquista da totalidade do continente
6. **Exemplo de leitura de arquivo:**

   Caso queira você pode utilizar a função carregarMapa para popular as classes Continente, Território, Objetivos e Jogo

```
void carregarMapa(const string &arquivo_territorios)
{
    ifstream arquivo(arquivo_territorios);
    if (!arquivo.is_open())
    {
        cout << "ERRO: Nao foi possivel abrir o arquivo, verifique se o nome esta correto e se o arquivo est  na pasta correta: " << arquivo_territorios << "\n";
        return;
    }

    string linha;
    int tamanho_das_entradas[4]; // contem o n mero de continentes, territ rios, jogadores e objetivos respectivamente

    if (getline(arquivo, linha))
    {
        stringstream ss(linha);
        string item;
        int primeira_linha[4];
        int tam = 0;

        while (getline(ss, item, ','))
        {
            tamanho_das_entradas[tam] = stoi(item);
            tam++;
        }
    }
    string continentes[tamanho_das_entradas[0]];                                    // Lista de continetes, voc  pode substituir o local em que est  populando esse array pelo m todo que cria um continente na classe continente
    string territorios[tamanho_das_entradas[1]];                                    // Lista de territ rios, voc  pode substituir o local em que est  populando esse array pelo m todo que cria um territ rio na classe territ rio
    int numero_de_jogadores = tamanho_das_entradas[2];                              // N mero de jogadores, voc  ir  utilizar essa informa  o no contrutor da classe Jogo
    int numero_de_territorios_por_continente[tamanho_das_entradas[0]];              // Para facilitar a inser  o dos territ rios na classe continente voc  pode utilizar esse array como refer ncia
    string objetivos[tamanho_das_entradas[3]];                                      // Lista de objetivos que ir o definir se um jogador venceu o jogo ou n o
    int contador_territorios = 0;                                                   // Contador para facilitar o controle da inser  o dos territ rios nos arrays (territorios e fronteiras)
    string **fronteiras_por_territorio = new string *[tamanho_das_entradas[1]]; // Matriz com as fronteiras de cada territ rio, voc  pode inserir diretamente na classe territ rios se preferir
    int numero_de_fronteiras_por_territorio[tamanho_das_entradas[1]];               // Para facilitar a inser  o das fronteiras na classe territ rio voc  pode utilizar esse array como refer ncia

    int contador_continente = 0;
    while (getline(arquivo, linha)) // Leitura feita linha a linha, no caso aqui a leitura que est  sendo feita    a da primeira linha
    {
        string continente, numero_de_territorios;
        if (linha.find(':') == string::npos && linha.find(',') != string::npos)
        {
            stringstream ss(linha);

            getline(ss, continente, ',');
            getline(ss, numero_de_territorios);
            continentes[contador_continente] = continente;

            numero_de_territorios_por_continente[contador_continente] = stoi(
                numero_de_territorios);
```

```
48
49              contador_continente++;
50          }
51
52          else if (linha.find(':') != string::npos)
53          {
54              stringstream ss(linha);
55              string territorio, fronteiras, fronteira;
56
57              getline(ss, territorio, ':');
58
59              territorios[contador_territorios] = territorio;
60
61              getline(ss, fronteiras);
62              int numero_de_fronteiras = count(fronteiras.begin(), fronteiras.end(), ',') +
                    1;
63              fronteiras_por_territorio[contador_territorios] = new string[
                    numero_de_fronteiras];
64              numero_de_fronteiras_por_territorio[contador_territorios] =
                    numero_de_fronteiras;
65
66              stringstream fs_filler(fronteiras);
67              int contador_fronteira = 0;
68              while (getline(fs_filler, fronteira, ','))
69              {
70
71                  fronteiras_por_territorio[contador_territorios][contador_fronteira] =
                        fronteira;
72                  contador_fronteira++;
73              }
74              contador_territorios++;
75          }
76      }
77
78      int contador_objetivos = 0;
79      while (getline(arquivo, linha))
80      {
81          if (!linha.empty())
82          {
83              objetivos[contador_objetivos] = linha;
84              contador_objetivos++;
85          }
86      }
87
88      arquivo.close();
89
90      /** Testes de Leitura */
91      /**CONTINENTES */
92      for (int i = 0; i < tamanho_das_entradas[0]; i++)
93      {
94          cout << continentes[i] << " ";
95      }
96      cout << "\n ---------------------------------------- \n";
97      /**TERRITORIOS */
98      for (int i = 0; i < tamanho_das_entradas[1]; i++)
99      {
100         cout << territorios[i] << ": ";
101         /* FRONTEIRAS POR TERRIT RIO*/
102         for (int j = 0; j < numero_de_fronteiras_por_territorio[i]; j++)
103         {
104             // cout << numero_de_fronteiras_por_territorio[i] << "\n";
105             cout << fronteiras_por_territorio[i][j] << " ";
106         }
107         cout << "\n";
108     }
```

```
    cout << "\n ---------------------------------------- \n";
    /**OBJETIVOS */
    for (int i = 0; i < tamanho_das_entradas[3]; i++)
    {
        cout << objetivos[i] << " ";
    }
    cout << "\n";

    for (int i = 0; i < tamanho_das_entradas[1]; i++)
    {
        delete[] fronteiras_por_territorio[i];
    }
    delete[] fronteiras_por_territorio;
}    }
```

Forma de usar o método

```
carregarMapa("./territorios.txt"); // arquivo est  no mesmo diret rio que o c digo que
    faz a leitura do mesmo
```

Desenvolvimento e Entrega

O código fonte do programa deve ser desenvolvido em C++, estar bem identado e comentado. A entrega deve ser efetuada conforme agendado no PVANet Moodle. Para isso, você deve criar um projeto contendo os arquivos `.h`, `.cpp`, e `main.cpp` criados. Envie, através do PVANet Moodle, uma pasta compactada (.rar ou .zip) contendo o projeto. A pasta compactada deve conter informações do aluno (ex.: julio_reis-tp2.zip). Para correção, serão considerados os seguintes critérios:

1. Documentação (**1 pt**).
   (a) Detalhamento do código.
   (b) Comentários e endentação.
2. Funcionamento correto (**2 pts**).
   (a) Compila e executa, não apresenta *crash*, etc.
3. Aplicação correta dos conceitos (**3 pts**).
   (a) Gerenciamento adequado de memória, explora uso correto de boas práticas e dos conceitos de OO, etc.

Comentários Gerais

- Comece a fazer este trabalho logo: o prazo para terminá-lo está tão longe quanto jamais poderá estar! :)
- O trabalho deve ser realizado individualmente (grupo de UM aluno/a);
- Trabalhos copiados serão penalizados (NOTA Zero).