Politécnico de Viseu

ei

Gestão de Casinos

Relatório POO 2023-2024

Autores:

- pv22894 - Daniel Valpereiro
- pv19545 - Diogo Santos
- pv23906 - António Ramos

Trabalho realizado
no âmbito da disciplina de POO,
lecionada pelo Exmo. Professor Morgado
e o Exmo. Professor Cunha

Índice

Introdução

Os casinos são uma peça de infraestrutura importante na manutenção do lazer e uma fonte de rendimento importante para a economia moderna de um país desenvolvido, ambos de um ponto de vista local e também de economia turística. Permite que a população possa socializar de uma forma mais competitiva e interativa que certos desportos comuns não permitem.

Este tipo de locais são dispendiosos e requerem muitos recursos, quer humanos como também em termos de equipamento, e um dos desafios apresentados aos gerentes e proprietários dos casinos, é a gestão dos mesmos, realçando as máquinas e os jogos disponibilizados. Tem havido muitos argumentos de como seria possível gerir o equipamento de uma forma mais eficiente e menos escrupulosa.

Com este trabalho o nosso objetivo principal foi facilitar tanto a manutenção, como também a gestão generalizada do equipamento recreativo disponível para os consumidores do casino, permitindo visualizar quais máquinas se apresentam ligadas, desligadas ou com algum tipo de avaria/falha. Com este tipo de apresentação fácil e interativa é proporcionada uma facilidade intrínseca do desenvolvimento económico e relacional do negócio. Por exemplo: ao saber que máquinas estão funcionais (ligadas ou desligadas) e atestando ao tráfego do casino é possível configurar de uma forma mais eficiente a ativação e desativação destas mesmas.

Neste relatório iremos exemplificar e demonstrar o funcionamento do programa para a gestão do casino, dando ainda exemplos práticos do uso diário e utilização geral que seria disponibilizada para os gestores/gerentes do estabelecimento.

Capítulo I - Desenvolvimento Inicial

- Estrutura do projeto

No começo do desenvolvimento do projeto, o nosso foco inicial foi a estruturação das classes e hierarquias necessárias. Para além do main.cpp decidimos implementar três classes principais: Casino.cpp, Jogadores.cpp e Maquinas.cpp, e também os três headers correspondentes: Casino.h, Jogadores.h e Maquinas.h.

Demos os devidos includes em cada um dos ficheiros. Colocámos, ainda, no nosso main.cpp, o "`setlocale(LC_ALL, "Portuguese_Portugal.1252")`" para que o texto não tivesse erros de compatibilidade com caracteres usados na língua portuguesa, aquando da compilação e a devida apresentação na "*console application*".

Adicionamos ainda o "`using namespace std`" para não ser necessária a designação "`std::`" antes de classes tais como o "`std::string`".

- Variáveis do construtor

Implementamos agora as variáveis principais dos construtores e destrutores. Depois de declarar o construtor do Casino no ficheiro Casino.cpp, declaramos "`Nome_Casino = Nome`" usando a string "`Nome`".

Dentro da classe Jogador.cpp, o construtor possui várias variavéis tais como "`int ID, string Nome_Jogador, string Morada, int Idade`", deixando o destrutor vazio (acabamos por deixar este espaço sem qualquer tipo de contexto).

Na classe Maquina.cpp, dentro das variavéis construtor "`string id, int probabilidadeGanhar, int premio, int x, int y, int tempAviso`".

- Declaração de funções

Aqui é a fase em que começamos a declarar algumas funções requeridas pelo projeto fornecidos pelos docentes, tais como a "`bool Casino::Add(User *ut)`" e a "`bool Casino::Add(Maquina *m)`" que servem, respectivamente, para adicionar um jogador à lista de jogadores e adicionar máquinas à lista de máquinas.

Declarámos uma função "`bool CompareByCount(Jogador* u1, Jogador* u2)`" para podermos comparar dois jogadores, a qual verifica quem tem mais jogadas.

```
//Função que compara dois users para ver qual deles tem mais jogadas
bool CompareByCount(Jogador* u1, Jogador* u2) {
    return u1->GetCount() < u2->GetCount();
}
```

Imagem 1 - Função "`CompareByCount`"

Capítulo II - Funções principais

Aqui foi a fase em que implementámos algumas das funções requeridas pelos docentes da classe Casino.

- Funções do Casino

Começamos por continuar o desenvolvimento das funções "`bool Casino::Add(User *ut)`" e "`bool Casino::Add(Maquina *m)`", pois são importantes para o desenvolvimento das funções seguintes.

Primeiro fizemos a função "`bool Casino::Add(User *ut)`". Temos abaixo uma imagem com o código.

```
//Adiciona um utilizador a lista de users
bool Casino::Add(Jogador *ut){
    if(ut){
        List_Jogador.push_back(ut);
        return true;
    }
    else
        return false;
}
```

Imagem 2 - Código da função que adicionar um jogador

Passo a explicar o código.

A função "`bool Casino::Add(Jogador *ut)`" é um método da classe Casino que recebe um ponteiro para um objeto "`Jogador`" como argumento e retorna um booleano indicando se o jogador foi adicionado com sucesso ou não.

O "`if(ut)`" verifica se o ponteiro "`ut`" não é nulo (ou seja, se está a apontar para um jogador válido).

A função da lista "`List_Jogador.push_back(ut)`" verifica se o ponteiro "`ut`" não é nulo, e portanto, o jogador (representado pelo ponteiro "`ut`") é adicionado ao final da lista "`List_Jogador`" usando o método "`push_back()`".

O "`return true;`" retorna `true` para indicar que o jogador foi adicionado com sucesso à lista.

Por outro lado, o "`else return false`" verifica se o ponteiro "`ut`" é nulo, ou seja, se não apontar para um jogador válido, a função retorna `false` indicando que não foi possível adicionar o jogador à lista.

Essa função "`Add`" permite adicionar um jogador à lista de jogadores do casino, desde que o ponteiro fornecido "`ut`" seja válido, ou seja, não seja nulo.

Agora a função "`bool Casino::Add(Maquina *m)`" que nos permite adicionar máquinas à lista de máquinas. Vejamos a seguir o código.

```
bool Casino::Add(Maquina *m){
    if(m){
        List_Maquina.push_back(m);
        return true;
    }
    else
        return false;
}
```

Imagem 3 - Código da função que adicionar uma máquina

A função "`bool Casino::Add(Maquina *m)`" é um método da classe Casino que recebe um ponteiro para um objeto "`Maquina`" como argumento e retorna um booleano indicando se a máquina foi adicionada com sucesso ou não.

O "`if(m)`" verifica se o ponteiro m não é nulo (ou seja, se está apontando para uma máquina válida).

O método da lista "`List_Maquina.push_back(m)`" verifica se o ponteiro "`m`" não é nulo, fazendo com que a máquina (representada pelo ponteiro "`m`") é adicionada ao final da lista "`List_Maquina`" usando o método "`push_back()`".

O "`return true;`" retorna `true` para indicar que a máquina foi adicionada com sucesso à lista.

O "`else return false;`" verifica se o ponteiro "`m`" é nulo, ou seja, se não apontar para uma máquina válida, a função retorna `false`, indicando que não foi possível adicionar a máquina à lista.

Assim como no método "`Add`" para jogadores, essa função "`Add`" para máquinas permite adicionar uma máquina à lista do casino, desde que o ponteiro fornecido "`m`" seja válido, ou seja, não seja nulo.

Para listar o estado do casino usamos a função "`void Casino::Declare_Casino_estado()`". Em baixo podemos ver a função mais detalhadamente:

```
//Declara o estado do casino
void Casino::Declare_Casino_estado()
{
    if(EstadoC == ESTADO_CASINO::ABERTO)
        cout << "O casino está aberto" << endl;
    else
        cout << "O casino está fechado" << endl;

}
```

Imagem 4 - Função "`Declare_Casino_estado`"

Esta função permite verificar se o casino está aberto ou fechado. A função faz uma autenticação vendo se o "`EstadoC == ESTADO_CASINO::ABERTO`" é verdade, e caso o seja, o utilizador tem um output de texto referindo "`O casino está aberto`", caso falso apresenta "`O casino está fechado`".

No seguimento desta função, é imperativo que seja possibilitado ao utilizador verificar na "`console application`" o estado da máquina. Para isso temos a seguinte função: "`void Casino::show(ostream &f)`". Esta função lista a seguinte informação (demonstrado na imagem seguinte):

```
cout << "Informações do Casino:" << endl;
cout << "Nome: " << Nome_Casino << endl;
cout << "Estado: " << (EstadoC == ABERTO ? "ABERTO" : "FECHADO") << endl;
cout << "Quantidade de Jogadores: " << List_Jogador.size() << endl;
cout << "Quantidade de M◆quinas: " << List_Maquina.size() << endl;
cout << "Quantidade de M◆quinas avariadas: " << List_Maquina_avariada.size() << endl;
cout << "\nPressione Enter para continuar...";
cin.get();
cout << "";
cin.get();
```

Imagem 5 - Informações do Casino

Ou seja, retorna o nome do casino, o seu estado atual (aberto, fechado), a quantidade de jogadores, a quantidade de máquinas, tanto disponíveis (quer ligadas como desligadas) como também as máquinas avariadas.

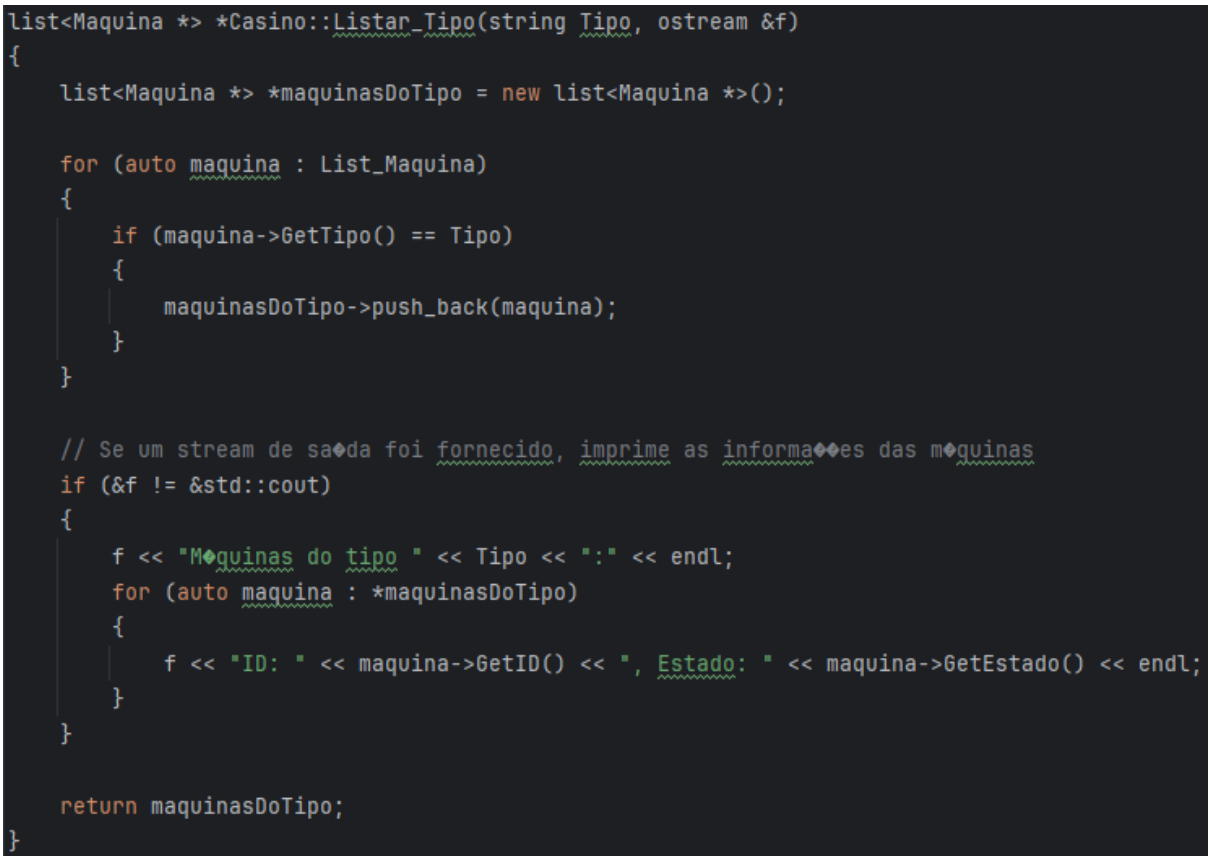
Foi também necessário implementar uma função que retornasse o estado atual de uma máquina.
Para isso temos a função "`void Casino::VerificarEstadoDaMaquina(const std::string& nomeMaquina)`". Podemos observar o código detalhadamente na imagem seguinte:

```
void Casino::VerificarEstadoDaMaquina(const std::string& nomeMaquina) {
    for (const auto& maquinaPtr : List_Maquina_avariada) {
        if (maquinaPtr->getNome() == nomeMaquina) {
            cout << "Estado da Máquina " << nomeMaquina << ": ";
            if (maquinaPtr->estaDisponivel()) {
                cout << "Disponivel\n";
            } else {
                cout << "Estragada\n";
            }
            return;
        }
    }
    cout << "Máquina com o nome " << nomeMaquina << " não encontrada.\n";
}
```

Imagem 6 - Estado atual de uma máquina em específico

Como podemos observar, esta função percorre a lista "`List_Maquina_avariada`" usando um loop "`for`" que itera cada um dos elementos da lista, cada elemento é um ponteiro para um objeto do tipo.
Para cada máquina na lista, ela verifica se o nome da máquina corresponde ao nome passado como argumento para a função "`(nomeMaquina)`" usando "`maquinaPtr->getNome()`". Presumivelmente, getNome() é um método que retorna o nome da máquina.
Se a máquina com o nome especificado for encontrada na lista, a função imprime o estado da máquina. Se "`maquinaPtr->estaDisponivel()`" retornar `true`, a máquina é considerada "Disponível". Caso contrário, é considerada "Estragada" (mais tarde renomeado para Avariada).
Após imprimir o estado da máquina, a função retorna imediatamente usando "`return`".
Se nenhuma máquina for encontrada na lista com o nome especificado, a função imprime que a máquina com o nome fornecido não foi encontrada na lista.

Para que possamos listar todos as máquinas do mesmo tipo, fizemos a função "`list<Maquina *> *Casino::Listar_Tipo(string Tipo, ostream &f)`". Podemos ver em baixo a imagem com o código mais detalhado:



```
list<Maquina *> *Casino::Listar_Tipo(string Tipo, ostream &f)
{
    list<Maquina *> *maquinasDoTipo = new list<Maquina *>();

    for (auto maquina : List_Maquina)
    {
        if (maquina->GetTipo() == Tipo)
        {
            maquinasDoTipo->push_back(maquina);
        }
    }

    // Se um stream de sa◆da foi fornecido, imprime as informa◆◆es das m◆quinas
    if (&f != &std::cout)
    {
        f << "M◆quinas do tipo " << Tipo << ":" << endl;
        for (auto maquina : *maquinasDoTipo)
        {
            f << "ID: " << maquina->GetID() << ", Estado: " << maquina->GetEstado() << endl;
        }
    }

    return maquinasDoTipo;
}
```

Imagem 8 - Código que lista máquinas de um dado tipo

A seguir vou explicar o funcionamento do código desta função.

A primeira parte "`list<Maquina *> *Casino::Listar_Tipo(string Tipo, ostream &f)`" a função "`Listar_Tipo`" recebe um tipo ("`Tipo`") e um objeto "`ostream`" por referência ("`f`"). Ela retorna um ponteiro para uma lista que contém ponteiros para objetos "`Maquina`".

A seguir, a "`list<Maquina *> *maquinasDoTipo = new list<Maquina *>()`" cria dinamicamente uma nova lista que armazenará ponteiros para objetos "`Maquina`".

Agora, a função "`for (auto maquina : List_Maquina)`" , esta função itera sobre cada máquina na lista "`List_Maquina`".

A função "`if (maquina->GetTipo() == Tipo)`" verifica se o tipo da máquina atual corresponde ao tipo fornecido como argumento para a função.

A seguinte função "`maquinasDoTipo->push_back(maquina)`" verifica se a máquina atual é do tipo desejado, e o ponteiro para ela é adicionado à lista "`maquinasDoTipo`".

Em seguida, há uma verificação condicional para imprimir informações das máquinas do tipo especificado, se o stream de saída não for "`std::cout`". Dentro dessa condição, as informações (como o ID e o estado) das máquinas são enviadas para o stream de saída fornecido.

Finalmente, a função retorna o ponteiro para a lista "`maquinasDoTipo`" que contém os ponteiros para as máquinas do tipo especificado.

É importante notar que esta função aloca dinamicamente memória para a lista "`maquinasDoTipo`" usando "`new`", então a responsabilidade de libertar essa memória alocada dinamicamente é do código que chama essa função, utilizando "`delete`" para evitar "*leaks*" de memória.

A função apresentada a seguir, ou seja "`list<string> *Casino::Ranking_Das_Mais_Trabalhadoras()`" é a função que nos permite saber as máquinas que mais trabalharam.
Ela retorna um ponteiro para uma lista de strings que representam um ranking de máquinas do casino com base no tempo de operação destas mesmas.

Temos então exemplificado em baixo a função:

```
list<string> *Casino::Ranking_Das_Mais_Trabalhadoras() {
    list<string> *ranking = new list<string>;

    // Criar uma c◆pia da lista de m◆quinas para ordena◆◆o
    list<Maquina *> copia_maquinas = List_Maquina;

    // Ordenar a lista de m◆quinas com base no tempo de opera◆◆o
    copia_maquinas.sort([](Maquina *a, Maquina *b) {
        return a->GetTempoOperacao() > b->GetTempoOperacao();
    });

    // Adicionar informa◆◆es ao ranking
    for (auto maquina : copia_maquinas) {
        string info = "ID: " + maquina->GetID() + ", Tempo de Operação: " + to_string(maquina->GetTempoOperacao());
        ranking->push_back(info);
    }

    return ranking;
}
```

Imagem 9 - Função das máquinas mais trabalhadoras

Agora passo a explicar abaixo o funcionamento da função:

Inicialmente a função "`list<string> *Casino::Ranking_Das_Mais_Trabalhadoras()`" não recebe nenhum argumento e retorna um ponteiro para uma lista de strings.

A seguir, a "`list<string> *ranking = new list<string>`" cria dinamicamente uma nova lista que armazenará strings para representar o ranking das máquinas.

De seguida, a "`list<Maquina *> copia_maquinas = List_Maquina;`" cria uma cópia da lista "`List_Maquina`" para realizar a ordenação. Essa cópia é armazenada na lista "`copia_maquinas`".

Para que possamos ordenar a lista "`copia_maquinas`" temos a função "`copia_maquinas.sort([](Maquina *a, Maquina *b) { return a->GetTempoOperacao() > b->GetTempoOperacao(); })`", que usa como base no tempo de operação de cada máquina. O critério de ordenação é definido por uma função lambda que compara o tempo de operação entre duas máquinas.

Em seguida, itera sobre a lista ordenada "`copia_maquinas`" e para cada máquina, cria uma string contendo informações como o ID da máquina e seu tempo de operação. Essa informação é adicionada ao ranking.

Finalmente, a função retorna o ponteiro para o ranking, que contém as informações das máquinas ordenadas pelo tempo de operação.

Assim como no código anterior, é importante notar que esta função aloca dinamicamente memória para a lista ranking usando new, portanto, a responsabilidade de liberar essa memória alocada dinamicamente é do código que chama essa função, utilizando delete para evitar leaks de memória.

Outra função necessária no desenvolvimento deste projeto é uma função que nos permite verificar qual dos jogadores mais ganhou. Esta função chama-se

"`list<Jogador *> *Casino::Jogadores_Mais_Ganhos()`". Podemos agora observar mais detalhadamente o código desta função.

```
list<Jogador *> *Casino::Jogadores_Mais_Ganhos() {
    list<Jogador *> *ranking = new list<Jogador *>;

    // Criar uma c◆pia da lista de jogadores para ordena◆◆o
    list<Jogador *> copia_jogadores = List_Jogador;

    // Ordenar a lista de jogadores com base no n◆mero de pr◆mios ganhos
    copia_jogadores.sort([](Jogador *a, Jogador *b) {
        return a->GetNPremios() > b->GetNPremios();
    });

    // Adicionar informa◆◆es ao ranking
    for (auto jogador : copia_jogadores) {
        string info = "ID: " + to_string(jogador->get_ID()) + ", Prémios Ganhos: " + to_string(jogador->GetNPremios());
        ranking->push_back(jogador);
    }

    return ranking;
}
```

Imagem 10 - Código da função dos jogadores que mais ganharam

Agora passo a explicar detalhadamente o funcionamento da função.

A lista "`list<Jogador *> *Casino::Jogadores_Mais_Ganhos()`" a função "`Jogadores_Mais_Ganhos`" não recebe argumentos e retorna um ponteiro para uma lista de ponteiros de Jogador.

A lista seguinte "`list<Jogador *> *ranking = new list<Jogador *>`" cria dinamicamente uma nova lista que armazenará ponteiros para objetos "`Jogador`".

Seguidamente, a lista "`list<Jogador *> copia_jogadores = List_Jogador;`" cria uma cópia da lista "`List_Jogador`" para realizar a ordenação. Essa cópia é armazenada na lista "`copia_jogadores`".

A lista copiada "`copia_jogadores.sort([](Jogador *a, Jogador *b) { return a->GetNPremios() > b->GetNPremios(); });`" ordena a lista "`copia_jogadores`" com base no número de prêmios ganhos por cada jogador. O critério de ordenação é definido por uma função lambda que compara o número de prêmios entre dois jogadores.

Em seguida, itera sobre a lista ordenada "`copia_jogadores`" e para cada jogador, cria uma string contendo informações como o ID do jogador e o número de prêmios ganhos. Então, o ponteiro para o jogador é adicionado ao ranking.

Finalmente, a função retorna o ponteiro para o ranking, que contém os jogadores ordenados pelo número de prêmios ganhos.

Como mencionado anteriormente, a alocação dinâmica de memória usando "`new`" implica na responsabilidade de liberar essa memória alocada dinamicamente, então o código que chama essa função deve cuidar para desalocar a memória usando "`delete`", evitando leaks de memória.

A seguir, fizemos a função que nos devolve uma lista ordenada dos jogadores que mais tempo passaram no casino. Esta função chama-se "`list<Jogador *> *Casino::Jogadores_Mais_Frequentes ()`" e podemos ver abaixo uma imagem com o código da mesma.

```
list<Jogador *> *Casino::Jogadores_Mais_Frequentes (){

    int count = 0;
    list<Jogador *> *LU = new list<Jogador *>;
    for(auto it = List_A_Jogar.begin(); it != List_A_Jogar.end(); it++){
        (*it)->Count(count);
        LU->push_back(*it);
    }
    LU->sort(CompareByCount);
    return LU;
}
```

Imagem 11 - Código que retorna os jogadores
que passaram mais tempo no casino

Vamos agora analisar o funcionamento passo-a-passo:

Temos, inicialmente, a lista "`list<Jogador *> *Casino::Jogadores_Mais_Frequentes ()`" esta função não recebe argumentos e retorna um ponteiro para uma lista de ponteiros de Jogador.

A lista seguinte "`list<Jogador *> *LU = new list<Jogador *>`" cria dinamicamente uma nova lista que conterá ponteiros de "`Jogador`".

O loop "`for`" itera sobre os elementos da lista "`List_A_Jogar`", que aparentemente contém ponteiros para objetos Jogador.

- "`(*it)->Count(count)`" Esta função chama um método Count para cada jogador na lista "`List_A_Jogar`", que atualiza o valor de count com a frequência de jogos desse jogador.
- "`LU->push_back(*it)`" Este excerto adiciona o ponteiro do jogador à lista "`LU`".

- Gestão de memória

A função "`LU->sort(CompareByCount)`" ordena a lista "`LU`" com base na função "`CompareByCount`", que provavelmente é uma função de comparação definida em outro lugar na classe "`Casino`". Essa função deve ser responsável por comparar a frequência de jogos entre os jogadores para determinar a ordem de classificação na lista.

Retorna o ponteiro para a lista "`LU`", que agora contém os jogadores ordenados de acordo com a frequência com que jogam.

A função seguinte permite-nos calcular a memória total ocupada pela estrutura de dados, neste caso pela lista das Máquinas e dos Jogadores. Podemos observar a seguir o código utilizado para realizar este cálculo:

```
int Memoria_Total() const {
int total = 0;

for (const auto& maquina :const Maquina *& : List_Maquina) {
    // Adicione o tamanho da inst◆ncia da Maquina ◆ mem◆ria total
    total += sizeof(*maquina);
}

for (const auto& jogador :const Jogador *& : List_Jogador) {
    // Adicione o tamanho da inst◆ncia do Jogador ◆ mem◆ria total
    total += sizeof(*jogador);
}

return total;
}
```

Imagem 7 - Código utilizado para calcular a memória

A parte inicial "`int Memoria_Total() const`" define a assinatura da função. Ela não modifica os membros da classe e retorna um inteiro representando a memória total ocupada.

O "`int total = 0`" inicializa uma variável total para armazenar o tamanho total da memória.

A seguir, o "`for (const auto& maquina : List_Maquina)`" começa um loop que percorre cada elemento na lista "`List_Maquina`". Para cada elemento (que é uma referência constante para uma "`Maquina`"), o tamanho da instância da

“`Maquina`” é adicionado ao “`total`” usando “`sizeof(*maquina). sizeof`” retorna o tamanho em bytes da instância de “`Maquina`”.

Na parte seguinte, o “`for (const auto& jogador : List_Jogador)`” inicia um loop semelhante para percorrer cada elemento na lista “`List_Jogador`”. Para cada elemento (uma referência constante para um “`Jogador`”), o tamanho da instância do Jogador é adicionado ao “`total`” usando “`sizeof(*jogador)`”.

Por fim, a função “`return total`” retorna o valor total calculado, que representa o tamanho total em bytes das instâncias de “`Maquina`” e “`Jogador`” armazenadas nas listas.

Capítulo III – .XML e .TXT

- Manipulação do ficheiro .txt

Passamos agora a explicar o funcionamento das funções da usadas para ler os ficheiros .XML e .TXT. Começamos por explicar a função utilizada para carregar todos os jogados do ficheiro pessoas.txt.

Temos então a função "`bool Casino::LoadJogador(const string &ficheiro)`", que vai ler todas as pessoas do ficheiro e vai separar por id, nome, localidade e idade. Temos agora o código abaixo exemplificado.

```
bool Casino::LoadJogador(const string &ficheiro)
{
    int contador=0;
    std::ifstream file("pessoas.txt");
    if (file.is_open())
    {
        std::string line;
        while (std::getline(file, line))
        {
            stringstream ss(line);
            string id, nome, localidade, idade_str;
            int idade, ID;

            if (getline(ss, id, '\t') && getline(ss, nome, '\t') && getline(ss, localidade, '\t') && getline(ss, idade_str, '\t'))
            {
                ID = stoi(id); // converte string para int
                idade = stoi(idade_str); // converte string para int
                //cout << "\n[" << idade_str <<"]--------";
                //idade = 33;
                Jogador *jogador = new Jogador(ID, nome, localidade, idade);
                List_Jogador.push_back(jogador);
            }
            //std::cout << line << std::endl;
            //if(contador++ > 10)
              //  break;

        }
        file.close();
    }
    else
    {
        std::cout << "Erro ao abrir o ficheiro" << std::endl;
    }
}
```

Imagem 12 – Código que lê o ficheiro pessoas.txt

A função "`bool Casino::LoadJogador(const string &ficheiro)`" é um método da classe Casino que recebe o nome de um arquivo como argumento (ficheiro) e não retorna nada (`void`). É responsável por carregar informações de jogadores a partir do ficheiro pessoas.txt.

Abre então o arquivo de texto chamado "pessoas.txt" usando um objeto "`ifstream`" chamado "`file`".

A função "`if (file.is_open())`" verifica se o arquivo foi aberto com sucesso.

Dentro do bloco condicional "`if (file.is_open())`", há um loop "`while`" que lê cada linha do arquivo de texto:

- Lê cada linha do arquivo e a armazena em "`line`".
- Usa um "`stringstream`" (`ss`) para dividir a linha nos valores separados por tabulação ("`\t`").
- Extrai os valores de "`id`", "`nome`", "`localidade`" e "`idade_str`" da linha.
- Converte "`id`" e "`idade_str`" de "`string`" para "`int`" usando "`stoi`".
- Cria um novo objeto "`Jogador`" com os valores extraídos e o adiciona à lista "`List_Jogador`" usando "`push_back`".

Fecha o arquivo após ler todas as linhas.

Se ocorrer um problema ao abrir o arquivo, a função "`std::cout << "Erro ao abrir o ficheiro" << std::endl`" imprime "Erro ao abrir o ficheiro".

- Manipulação do ficheiro .txt

Temos agora a função "`void Casino::lerArquivoXML(const std::string& nomeArquivo, Casino& casino)`", que vai ler o arquivo XML e configurar os detalhes do casino e das máquinas com base nesses dados. Conseguimos ver o código na imagem abaixo.

```
void Casino::lerArquivoXML(const std::string& nomeArquivo, Casino& casino) {
    pugi::xml_document doc;
    if (doc.load_file(nomeArquivo.c_str())) {
        pugi::xml_node definicoes = doc.child("DADOS").child("DEFINICOES");
        casino.Nome_Casino = definicoes.child_value("NOME");
        int maxJogadores = definicoes.child("MAX_JOG").text().as_int();
        int probabilidadeUsuario = definicoes.child("PROB_USER").text().as_int();
        int horaInicio = definicoes.child("HORA_INICIO").text().as_int();
        int horaFim = definicoes.child("HORA_FIM").text().as_int();

//        std::cout << "Nome do Casino: " << Nome_Casino << std::endl;
//        std::cout << "Nome da maquina: " << Nome_Maquina << std::endl;
//        std::cout << "M◆ximo de Jogadores: " << maxJogadores << std::endl;
//        std::cout << "Probabilidade do Usu◆rio: " << probabilidadeUsuario << std::endl;
//        std::cout << "Hora de In◆cio: " << horaInicio << std::endl;
//        std::cout << "Hora de Fim: " << horaFim << std::endl;

        pugi::xml_node listaMaquinas = doc.child("DADOS").child("LISTA_MAQ");
        for (pugi::xml_node maquina : listaMaquinas.children("MAQUINA")) {
            std::string nomeMaquina = maquina.child_value("NOME");
            int probabilidadeGanhar = maquina.child("PROB_G").text().as_int();
            int premio = maquina.child("PREMIO").text().as_int();
            int x = maquina.child("X").text().as_int();
            int y = maquina.child("Y").text().as_int();
            int tempAviso = maquina.child("TEMP_AVISO").text().as_int();

            Maquina* novaMaquina = criarMaquina(nomeMaquina, probabilidadeGanhar, premio, x, y, tempAviso);

            // Adicione a nova m◆quina ◆ sua lista, se ela n◆o for nula
            if (novaMaquina) {
                List_Maquina.push_back(novaMaquina);

               //novaMaquina->Show_maquina();
            }
        }
    } else {
        cout << "Erro ao carregar o arquivo XML." << std::endl;
    }
}
```

Imagem 13 - Código que lê o ficheiro XML

Passamos agora a explicar ao detalhe o funcionamento do código.

A função "`void Casino::lerArquivoXML(const std::string& nomeArquivo, Casino& casino)`" é um método da classe **Casino** que recebe o nome de um arquivo XML como argumento (`nomeArquivo`) e uma referência para um objeto **Casino** (`casino`).

A função "`pugi::xml_document doc`" cria um objeto "`xml_document`" da biblioteca PugiXML para carregar e analisar o conteúdo do arquivo XML.

A função "`if (doc.load_file(nomeArquivo.c_str()))`" tenta carregar o arquivo XML especificado. Se o carregamento for bem-sucedido:

O código acessa diferentes elementos dentro do XML:

- "`pugi::xml_node definicoes = doc.child("DADOS").child("DEFINICOES")`"

  Obtém os dados de definições do nó "`DEFINICOES`" dentro do nó DADOS.

- Define algumas variáveis usando os valores encontrados no arquivo XML, como "`Nome_Casino`", "`maxJogadores`", "`probabilidadeUsuario`", "`horaInicio`" e "`horaFim`".
- "`pugi::xml_node listaMaquinas = doc.child("DADOS").child("LISTA_MAQ")`"

  Obtém a lista de máquinas do nó "`LISTA_MAQ`" dentro do nó "`DADOS`".

- Itera sobre as máquinas no arquivo XML, lendo os valores de cada máquina, como nome, probabilidade de ganhar, prêmio, coordenadas, etc.
- Para cada máquina no arquivo, chama a função "`criarMaquina`" (não definida aqui) para criar um novo objeto "`Maquina`" com os valores lidos do arquivo XML.
- Se a máquina for criada com sucesso (ou seja, se não for nula), ela é adicionada à lista de máquinas (`List_Maquina`) do objeto **Casino** atual.

Se houver um problema ao carregar o arquivo XML, uma mensagem de erro é exibida. É chamada a função de output "`cout << "Erro ao carregar o arquivo XML." << std::endl;`" que mostra ao utilizador "`Erro ao carregar o arquivo XML.`"

Conclusão

Este projeto, parte integrante da disciplina de Programação Orientada a Objetos no curso de Engenharia Informática, proporcionou uma aplicação prática dos conceitos aprendidos.

A implementação de uma simulação de gestão de casino abordou desafios desde a manipulação eficaz de ficheiros XML, utilizando a biblioteca pugixml, até à gestão de recursos e comunicação entre entidades.

A utilização de ficheiros XML permitiu configurar o casino de forma estruturada, destacando a versatilidade da biblioteca pugixml na leitura e interpretação desses dados. A coexistência de máquinas de jogos, cada uma com características específicas, evidenciou a aplicação prática dos conceitos de herança e polimorfismo.

Em grupo, focamo-nos na implementação concisa de tipos distintos de máquinas de jogos e na interação eficiente com o utilizador. A gestão de recursos, incluindo a memória, foi cuidadosamente considerada para garantir um desempenho otimizado.

Concluindo, este projeto não apenas consolidou os conhecimentos em Programação Orientada a Objetos, mas também proporcionou uma oportunidade valiosa para aplicar esses conhecimentos na resolução de um problema prático. Na nossa visão o trabalho foi um sucesso!

Webgrafia/Bibliografia

1. pugixml Documentação https://pugixml.org/docs/quickstart.html