Programação Orientada a Objetos 2023/2024

Exercícios

Ficha Nº 9 | Smart Pointers

1. Ponto dinâmico com smart pointers

Considere a classe Ponto cujo código é dado em seguida. Copie o código para o seu projeto e, salvo indicações em contrário, não pode mudar o código. Neste exercício **não pode** usar ponteiros ***raw***.

Ponto.h

```
// includes e outras declarações omitidos
class Ponto {
public:
    Ponto(int cx, int cy);
    ~Ponto();
    void mostra() const;
private:
    int x,y;
};
```

Ponto.cpp

```
// includes e outras declarações omitidos
Ponto::Ponto(int cx, int cy) : x(cx), y(cy) { cout << "CONSTR. "; mostra(); }
Ponto::~Ponto(){ cout << "DESTR. "; mostra(); }
void Ponto:: mostra() const { cout << "Ponto com " << x << "," << y << "\n"; }
```

a) Escreva uma função *main* organizada em três blocos de código (blocos de instruções delimitadas por { }). No enunciado, os blocos serão referidos por B1, B e B3. As perguntas desta alínea estão a seguir ao exemplo da função *main* pretendida.

Exemplo de função *main* pretendida

```
main() {
   // zona inicial
   {  // bloco B1
   }
   {  // bloco B2
   }
   {  // bloco B3
   }
}
```

Questões desta alínea:

- No bloco B1 crie um objeto Ponto dinâmico Ponto de forma a que não possa ser partilhado. Aceda-lhe através de um ponteiro *spu* de tipo adequado para demostrar que o objeto Ponto existe. Explique as mensagens que aparecem.
- Na zona inicial da função *main*, crie um ponteiro *spw* para Ponto que permita partilhar, mas não controlar um objeto de tipo Ponto. No bloco B2 crie um objeto Ponto dinâmico partilhável. Aceda-lhe através de um ponteiro *sps1* de tipo adequado para demostrar que o objeto Ponto existe. Partilhe esse objeto Ponto com o ponteiro *spw* que criou na zona inicial. Aceda ao objeto Ponto através desse ponteiro *spw*. Partilhe o objeto com um segundo ponteiro *sps2* que permita partilha (e controlar) esse mesmo objeto Ponto. O único ponteiro que existe fora dos blocos B1, B2, e B3 é o ponteiro *spw*.
- Na zona tente aceder B3 aceda ou tente aceder ao objeto Ponto, criado na zona B2, através do ponteiro *spw*. Garanta que verifica se o objeto existe antes de lhe aceder e explique o que aconteceu.

b) Escreva uma nova versão da função *main* sem qualquer bloco/zona. Experimente criar uma matriz de 10 objetos com cada um dos 3 tipos de ponteiro smart: não partilháveis (unique), partilháveis que controlam o objeto (shared) e partilháveis que não controlam o objeto (weak). Experimente a sintaxe e eventuais imposições que surjam no uso dos ponteiros e na classe ponto. Garanta que entendeu os aspetos que estão envolvidos.

c) Construa uma nova função *main* com dois ponteiros de cada tipo unique (u1 e u2), shared (s1 e s2) e weak (w1 e w2).

- Inicialize todos os ponteiros de forma a que fiquem a apontar para um objeto Ponto, cada ponteiro a apontar para o seu objeto Ponto, exceto no caso dos weak pointers, ontem tem que cumprir a forma como estes ponteiros adquirem o seu objeto apontado (mas neste caso, ainda assim, cada weak pointer aponta para um objeto Ponto diferente).
- Em cada tipo de ponteiro, experimente fazer uma atribuição (dentro do mesmo tipo: u1 = u2, idem restantes dois tipos).
  (1) No caso do unique pointer teve que forçar com std::move. Explique porquê e o que acontece ao objeto apontado pelo ponteiro do lado esquerdo da atribuição.
  (2) No caso dos shared ponters, verifique o que acontece ao objeto apontado pelo ponteiro do lado esquerdo, e verifique o contador de uso através de ambos os ponteiros.
  (3) No caso dos weak pointers, verifique o que acontece aos contadores de uso dos objetos apontados em ambos os lados da atribuição.

No final do exercício deverá ter

- Compreendido os fundamentos básicos de smartpointers unique, shared e weak: o que são, para que servem e como se usam
- Compreendido a semântica dos três tipos de smartpointers: unique, smart, weak e respetivos casos de uso.
- Sintaxe envolvida no uso de smartpointers, nomeadamente a vertente em que não são sequer necessários ponteiros raw de forma explícita.

2. **MyStringSP** – MyString com smart pointers

Assuma que não existe a classe de biblioteca STL *string* mas que se pretende algo semelhante a ela. Neste exercício pretende-se fazer uma classe **MyStringSP** que represente uma cadeia de caracteres de uma forma parecida ao que a classe *string* da STL faz (obviamente, *não pode* usar aqui a classe *string*).

Neste exercício **não pode** usar nenhuma coleção STL. Também **não pode** usar nenhum ponteiro *raw* para gerir o armazenamento dos caracteres. Qualquer ponteiro *raw* que apareça no seu programa será secundário e periférico à classe.

a) Escreva a classe **MyStringSP** com as seguintes características

- Representa uma cadeia de caracteres de tamanho indeterminado, terminada por '\0'. Pode ser vazia, pequena, ou enorme, sendo a única limitação os recursos existentes na máquina.
- Os caracteres de um objeto **MyStringSP** pertencem a esse objeto.
    - Os objetos de **MyString** são inicializados com:
    - Nada -> fica uma cadeia de caracteres vazia.
    - Um ponteiro de caracteres -> o objeto fica com uma cópia desses caracteres.
- É possível obter o tamanho da cadeia de caracteres. Por uma questão de performance, o objeto deverá manter armazenado o tamanho, atualizando-o sempre que a cadeia de caracteres muda.
- É possível obter a cadeia de caracteres armazenada internamente. No entanto, deve ser salvaguardado o seguinte: o resto do programa pode ver, mas não pode alterar o conteúdo da cadeia de caracteres.
- É possível obter, através de uma função **at()** o caracter numa determinado posição (que é indicada). A função deve permitir alterar esse caracter sendo usada do lado esquerdo de uma atribuição. A posição indicada é, naturalmente, validada, sendo rejeitados acessos a caracteres que não existem.
- Deve ser possível adicionar (concatenar) um conjunto de caracteres ao que já existe no objeto, usando para tal a função membro **concat()** que recebe um outro objeto **MyStringSP**.
- Deve ser possível apagar os caracteres, ficando uma cadeia de caracteres vazia com uma função **clear()**.
- A atribuição e a cópia de objetos **MyStringSP** deve ser permitida e ter o comportamento habitual.
- Não são permitidas situações *memory leaks*.

No final do exercício deverá ter

- Percebido como usar *smart pointers* para gerir uma matriz dinâmica num cenário de composição com recursos dinâmicos
- Ter uma noção clara acerca de ponteiros *raw* vs. *smart pointers*.

3. Exercício preparatório - Pessoa – Classe de apoio

Crie um novo projeto e acrescente nele a classe Pessoa cujo código é apresentado abaixo. Esta classe não constitui exercício em si mesma e serve para apoio aos dois exercícios seguintes. Trata-se da mesma classe Pessoa que já apareceu diversas vezes nas fichas de exercícios. Não pode acrescentar código a esta classe.

Pessoa.h

```
// includes e outras declarações omitidos

class Pessoa {
public:
    Pessoa(string nome, int bi, int nif);
    string getNome() const;
    int getBI() const;
    int getNIF() const;
    void setNome(string nome);
    string descricao() const;
private:
    string nome;
    int bi, nif;
};
```

Classe *Pessoa*, descrita por nome, número de bilhete de identidade (BI), e número de contribuinte (NIF), sendo necessário indicar todos estes dados na sua inicialização. Permite obter cada um dos seus dados, e permite atualizar o nome. Permite obter uma *string* com a descrição do seu conteúdo.

Pessoa.cpp

```
// includes e outras declarações omitidos

Pessoa::Pessoa(string _nome, int _bi, int _nif) : nome(_nome), bi(_bi), nif(_nif) { }
string Pessoa::getNome() const { return nome; }
int Pessoa::getBI() const { return bi; }
int Pessoa::getNIF() const { return nif; }
void Pessoa::setNome(string _nome) { nome = _nome; }
string Pessoa::descricao() const {
    ostringstream oss;
    oss << nome << " " << bi << " " << nif;
    return oss.str();
}
```

4. Registo Civil de pessoas com smartpointers – Composição de recursos dinâmicos e uma coleção com smart pointers.

Escreva uma classe **RegistoCivil** semelhante às versões anteriormente já feitas deste tema, mas com algumas simplificações e pequenas diferenças. Reproduz-se aqui o essencial, com as diferenças assinaladas em ***itálico negrito***:

- Na sua inicialização exige o nome do país a que diz respeito. Inicialmente não tem pessoa nenhuma.
- É possível obter, mas não mudar o nome do país.
- Suporta a existência de um número **indeterminado de pessoas**, limitado apenas pelas capacidades da máquina. Neste exercício **vai usar um vetor** da STL.
- Permite adicionar uma nova pessoa, a qual é construída no contexto da classe e não fora dela. Para esta operação são indicados os dados da nova pessoa. A operação falha se já existir nesse país uma pessoa com o BI indicado. *Não havendo máximo específico, devem ser suportadas tantas pessoas quanto as que couberem em memória.*
- Permite obter o nome de uma pessoa dado o seu BI.
- Permite obter a listagem de todas as pessoas numa *string*, uma pessoa por linha.
- Permite atualizar o nome de uma pessoa dado o seu BI e o novo nome.
- Permite obter o número de pessoas atualmente existentes.
- Permite obter o ponteiro para uma pessoa dado o BI. **Não pode ser um ponteiro *raw***. Este ponteiro irá ser usado por outras classes de forma permanente segundo a lógica de agregação, permanecendo o controlo exclusivo no lado do RegistoCivil, pelo que **é importante que a escolha do tipo de ponteiro é importante**.
- **Apagar uma pessoa dado o BI. Se o resto do programa estiver na posse de um ponteiro para esta pessoa, esse mesmo resto de programa deverá poder perceber que a pessoa já não existe.**
- **Transferir uma pessoa para outro registo civil, dados os BI e o registo civil destino. A operação falha se no registo civil destino já existir esse BI.**

a) ***Identifique que tipo de smart pointer devem ser usados na sua coleção de pessoas no RegistoCivil. Nomeadamente, qual destes dois: unique pointers ou shared pointers. À partida qualquer deles poderá ser válido dependendo do tipo de uso que vão ser dados às objetos pessoa. Responda a estas questões para o ajudar a decidir:***

- *As pessoas são do RegistoCivil? E se copiar o registo civil? Quero copiar o registo civil? Se quiser, devo duplicar as pessoas, ou partilho-as?*
- *Quero mostrar as pessoas do registo civil ao resto do programa? Se sim, o resto do programa consegue "mandar" tanto nesses objetos como o próprio registo civil?*

b) Implemente a classe (.h e .cpp + função *main*) e experimente a sua funcionalidade com algumas operações a partir da função main. Garanta que a classe está apta a ser usada em todas as operações em que um objeto pode ser usado em C++ (por exemplo: ser criado, destruído, copiado, atribuído, etc.).

No final do exercício deverá ter

- Usado *smart pointers* de vários tipos em contextos razoavelmente complexos.
- Smart pointer em contexto de composição em que planeia mais tarde vir a partilhar os objetos

5. Clube com smart pointers – Agregação por smart pointers

Refaça a classe que representa o conceito de *clubes de bairro* em C++ da ficha anterior. Nesta versão, a funcionalidade é exatamente a mesma, mas a implementação tem as seguintes diferenças:

- Vai usar smart pointers para Pessoa para estabelecer a ligação entre o clube e os seus sócios. Neste cenário não precisa de ter conhecimento direto acerca do registo civil.
- O número de sócios é indeterminado/ilimitado. **Pode e deve usar coleções STL** (sugestão: vector).

A classe deve ter a seguinte funcionalidade:

- Inicialização dado o nome da atividade do clube. Inicialmente fica sem sócio nenhum.
- Adicionar um sócio dado o ponteiro para essa pessoa. Não pode haver sócios repetidos (mesmo BI).
- Remover um sócio dado o seu BI.
- Obter a lista com os nomes de todos os sócios (um nome por linha)

A classe deve estar preparada para que os seus objetos se comportem corretamente no contexto da linguagem C++.

a) Planeie o tipo de smart pointers que deve usar na coleção de sócio no clube.

b) Implemente a classe **Clube** (.h e .cpp), adicionando-a ao projeto do exercício anterior e experimente a sua funcionalidade com algumas operações a partir da função *main*.

No final do exercício deverá ter

- Lidado com uma situação em que uma classe está relacionada com um cenário de agregação, e usa smart pointers para implementar esse relacionamento.