Programação Orientada a Objetos 2023/2024

Exercícios

Ficha Nº 3

Encapsulamento e abstração
Classes. Visibilidade. Funções membro. Acesso const
Construtores / destrutores
Ciclo de vida dos objetos
std::initializer_list e std::span
std::ostringstream

1. Classe Tabela - Encapsulamento e sintaxe de classes

No exercício 6 da ficha de exercícios anterior foram feitos uma estrutura Tabela e um conjunto de funções para trabalhar instâncias dessa estrutura. Pretende-se agora juntar a estrutura com as funções que lhe dizem respeito numa única entidade definida por uma **classe**.

a) Defina uma classe que agregue a funcionalidade associada ao conceito de tabela de números. Deve ser suportada a mesma funcionalidade que tinha sido pedida no exercício 6 da ficha anterior, mas desta vez respeitando os princípios de encapsulamento:

- Assuma um valor limite para a quantidade de números a armazenar, conhecido em tempo de compilação, mas sem usar #define.
- Deve haver um método para preencher todos os valores com um valor especificado.
- Deve haver um método para obter/alterar o valor numa dada posição, devidamente protegida contra acessos a posições inexistentes. O mesmo método permitirá obter e modificar o valor da posição indicada.
- Deve haver um método para listar os elementos no ecrã. Este é o único método que interage diretamente com o utilizador.

A funcionalidade que não altera os objetos da classe deve declarar tal facto ao compilador com *const*.

b) Escreva uma segunda versão do método que lista os números. Nesta segunda versão, os números são devolvidos, fazendo com que nem sequer este método precise de interagir com o utilizador. O meio de conseguir passar a informação para o exterior da função poderia ser uma matriz, mas pretende-se que explore ***std::span*** nesta alínea (#include <initializer_list>). std::span exige **C++20** – verifique a configuração do seu IDE/compilador.

c) Acrescente o necessário de maneira de garantir que os objetos da classe são sempre inicializados:

- Com zeros, não sendo necessário indicar nada.
- Com um determinado valor indicado.
- Com uma sequência de números crescente, começando em *a* e com incrementos de *b*, sendo *a* e *b* indicados.
- Com um conjunto de números indicados (se exceder a capacidade, os restantes são ignorados; se não esgotar a capacidade, os valores em falta serão zero). Neste item deverá explorar *std::initializer_list* (#include <initializer_list>).
    - Comprove experimentalmente, usando o seu programa, que a existência da capacidade de inicializar o objeto usando um *initializer_list* faz com que as outras formas de inicialização possam não ser "encontradas" pelo compilador.

d) Alínea de reflexão/discussão

Repare no ***std::span*** que usou na alínea b) e no ***std::initializer_list*** que usou na alínea c). Apesar de parecidos, são bastante diferentes na forma como se inicializam e usam, e nos pressupostos acerca dos valores com que são construídos. Confirme que percebe bem a diferença entre ambos.

- ***std::initializer_list***: espécie de *array* otimizado e imutável de carácter temporário. é usado essencialmente para especificar valores, os quais não existem para além da linha onde são especificados. Caso de uso: passagem de valores por parâmetro para serem copiados, tipicamente em inicializações.
- ***std::span***: espécie de *array* otimizado, que pode ser modificado. É usado essencialmente para especificar valores que já existem (num *array* ou outra coleção), os quais devem obrigatoriamente continuar a existir enquanto o *span* é usado. É usado como substituto (melhor) onde normalmente teria que se usar um ponteiro para o primeiro elemento e uma quantidade de elementos.

e) Acrescente o necessário para verificar se um determinado número existe dentro dos números armazenados num determinado objeto da classe.

f) Acrescente o necessário para verificar se dois objetos da classe armazenam o mesmo conjunto de números (mesmo que por ordem diferente). Seria interessante usar == para este efeito, mas essa matéria ainda não aparece nesta ficha e não é pretendido que altere esse operador.

g) Acrescente à classe um mecanismo que garanta que aparece no ecrã a mensagem "destruído" sempre um que objeto da classe deixa de existir. No contexto deste exercício, e da matéria já dada, trata-se de uma funcionalidade que servirá para testar aspetos da linguagem (mas mais tarde, o mecanismo envolvido assumirá um papel relevante).

Acrescente a todos os métodos inicializadores que tem na classe o necessário para que apareça uma mensagem "construído" sempre que um objeto desta classe é criado.

Acrescente ao seu programa (nota: "classe" ≠ "programa"), os seguintes métodos:

- Recebe – este método recebe um objeto da classe por cópia ou valor.
- Devolve – retorna um objeto da classe por valor.
- Inicializa – Constrói um objeto, e depois, outro por atribuição do primeiro.

Execute o programa chamando cada uma dessas funções. Analise as mensagens que aparecem e garanta que percebe a razão dessas mensagens terem aparecido, e também a razão de algumas mensagens não terem aparecido.

h) Separe a sua classe em .h e .cpp. A função main ficará num ficheiro .cpp à parte.

i) Alínea de reflexão

Neste exercício viu o que implica passar de um código estilo-C para um cenário em que usa classes. O foco foi, essencialmente a sintaxe e o comportamento dos objetos, mas também questões acerca de encapsulamento. Anote cuidadosamente o que aprendeu no seu caderno.

No final do exercício deverá

- Perceber em que consiste a abstração em POO, o encapsulamento e o conceito de *data-hidding*.
- Entender a sintaxe essencial de classes em C++, nomeadamente na especificação de dados e funções membro, visibilidade, especificação de funcionalidade const, e inicialização e destruição de objetos.
- Entender o que são initializer_list e span, e saber quando são úteis.
- Saber os fundamentos de desenho de classes princípio: prioridade ao comportamento em detrimento dos dados; perceber o que deve ser público e o que deve ser privado.
- Ter uma noção do que está envolvido na passagem de objetos por cópia e no retorno por cópia.
- Saber usar .h e .cpp corretamente.

2. Modelização de automóvel – Encapsulamento e *class design*

Defina uma classe *Automovel* que contenha os dados necessários à representação de um automóvel (matrícula, combustível, marca, modelo ou outros). No decorrer das alíneas seguintes deve usar uma função *main()* que lhe permita testar o código que for fazendo. A função *main()* não faz diretamente parte do exercício.

a) Analise as regras mais óbvias do mundo real acerca de automóveis, nomeadamente no que diz respeito:

- Quais o seu comportamento – ou seja qual a funcionalidade que deve existir.
- Quais as formas admissíveis de criação de objetos – como e segundo que regras devem os objetos da classe ser possíveis de construir e como são inicializados.
- Quais os dados necessários à funcionalidade que deduziu nos dois pontos anteriores. Repare que os dados são ditados pelo que a funcionalidade precisa internamente e não pelo o que o resto do programa acha que deve ser. Segundo esta lógica, um automóvel é uma coisa que exibe uma matricula quando inquirido e não uma coisa que tem uma matrícula (até pode não ter matrícula nenhuma, desde que apresente uma quando um agente de autoridade a pedir para ver). Esta diferença parece subtil, mas é fundamental em programação orientada a objetos.

Com base nesta análise, implemente a classe Automóvel, dotando-a da funcionalidade esperada do conceito de automóvel. Não precisa de exagerar em pormenores minuciosos – apenas a funcionalidade mais evidente. Faça logo o código organizado em .h e .cpp.

Tenha também em atenção o seguinte:

- Evite interagir com o utilizador diretamente dentro das funções da classe (“métodos da classe”).
- Os métodos que atualizam os dados da classe devem ser acerca de dados que:
  - Faça sentido que o programa se aperceba que existam na classe (haverá algum dado na classe acerca do qual o programa tenha que saber que existem e saba como é armazenado? Quais?).
  - Faça sentido que sejam modificáveis, segundo as regras de negócio do domínio de aplicação (ou seja, que faça sentido serem modificados de acordo com o conceito que a classe representa).

b) Escreva uma função que permita obter a representação textual do conteúdo dos objetos da classe. Não se pretende necessariamente imprimir essa informação no ecrã e evite faze-lo diretamente em código da função (“obter” ≠ “imprimir”). Nesta alínea pretende-se que explore a classe *ostringstream* (#include <ostream>).

No final do exercício deverá

- Ter consolidado o conhecimento acerca de encapsulamento e *data-hidding*.
- Ter consolidado o conhecimento acerca da sintaxe essencial de classes em C++
- Consolidado a competência no desenho de classes: planear os métodos, a visibilidade, os dados.
- Ter uma noção muto clara acerca da não existência de I/O com o utilizador em código da classe.
- Conhecer a classe *ostringstream*.

3. Pessoa – Encapsulamento e *class design*

Implemente em C++ o conceito de *pessoa* com as características principais habituais associadas a esta entidade no mundo real. Esta classe deve respeitar os princípios de encapsulamento. A funcionalidade que não se destina a modificar o conteúdo do objeto deve declarar tal facto ao compilador.

Deve usar uma abordagem semelhante à que usou no exercício do Automóvel. Uma parte importante deste exercício é a modelização sensata e não existem soluções únicas. O percurso para atingir a solução é tão importante como a solução em si.

No final do exercício deverá

- Ter consolidado o conhecimento acerca de abstração, encapsulamento e *data-hidding*.
- Ter consolidado o conhecimento acerca da sintaxe essencial de classes em C++
- Consolidado a competência no desenho de classes.

4. Exploração do ciclo de vida dos objetos – TPC mas, mesmo assim, obrigatório

Pretende-se uma classe chamada *MSG* cujos objetos respeitem as seguintes propriedades:

- Armazenam uma letra e um número inteiro. A letra pode ser fornecida na construção dos objetos. Se não for, será assumida a letra 'x'. O número é um valor gerado automaticamente: cada novo objeto tem o número seguinte ao do objeto anterior. O primeiro objeto tem o número 1.
- Deve existir um mecanismo para obter os dados dos objetos sob a forma de uma *string* com o formato "letra: … número …".
- Sempre que um objeto é criado deve ser automaticamente apresentada a sua informação (formato "criado: letra: … número …").
- Sempre que um objeto deixa de existir deve ser automaticamente apresentada a sua informação no ecrã (formato "terminado: letra: … número …").

a) Construa a classe com as características enunciadas e teste a sua funcionalidade criando dois objetos, *a* e *b*, indicando ao primeiro a letra 'a', e ao seguinte nenhuma letra. Execute o programa de forma a que este, ao terminar, aguarde pelo pressionar de uma tecla. Observe as mensagens no ecrã e relacione a ordem pela qual aparecem com a ordem pela qual os objetos são criados e destruídos.

(1) No construtor explore a sintaxe de lista de inicialização e verifique que é melhor que a atribuição de valores dentro do construtor, a qual não é uma verdadeira inicialização, mas sim um "mero uso".

(2) Experimente inicializar os objetos *a* e *b* com parêntesis e com chavetas e verifique que tem ambas as possibilidades sintáticas à sua disposição.

b) Acrescente à função *main*, a seguir às variáveis que já existem, uma referência com o nome *c* para o objeto que tem a letra 'x'. Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

c) Acrescente à função *main*, a seguir às variáveis que já existem, a seguinte declaração: *MSG d=b;* Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

d) Acrescente à função *main* a atribuição *main: a = b;* . Execute o programa e confirme que o número e ordem de mensagens não se altera (apenas a o conteúdo relativo ao objeto *a*). Comparando com a alínea anterior, explique porquê e remova a atribuição que tinha acrescentado.

e) Acrescente um *array* de objetos de *MSG* de dimensão dois. É possível indicar a letra inicial aos objetos desse *array*? Verifique a sintaxe disponível para esses casos. Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

f) Remova da classe *MSG* a característica que permite que os seus objetos fiquem com a letra 'x' se nenhuma letra for especificada. Explique porque é que o programa deixa de compilar.

(1) Explore a possibilidade de indicar valores iniciais aos elementos do *array:* as várias alternativas sintáticas ao seu dispor e os seus efeitos através das mensagens que são (ou não) apresentadas no ecrã.

Volte a por a classe como estava (a inicialização com a letra 'x').

g) Acrescente uma função *teste* do tipo *void* e sem parâmetro nenhum. Declare nessa função um objeto classe *MSG* com o nome *aux* e com a letra inicial 'y'. Invoque a função teste na função *main*. Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

h) Modifique a função *teste* de forma a receber como parâmetro um objeto da classe deste exercício. O objeto é passado por valor. Na função *main* adapte a chamada à função teste passando-lhe o objeto que tem a letra 'x'. Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

i) Acrescente um construtor privado à classe *MSG* que recebe como parâmetro o seguinte: *const MSG & z* e dê um corpo vazio a esse construtor. Compile o programa e confirme que não é possível. Explique porquê.

j) Passe o construtor a publico e coloque no seu corpo apenas a seguinte linha *cout << "construído por cópia";* . Verifique que já consegue compilar o programa e explique porque é que já é possível.

k) Execute o programa e repare que os valores relativos ao objeto parâmetro da função *teste* não são inicializados. Explique porquê. Coloque o código adequado ao construtor por cópia de forma a que os objetos inicializados por cópia fiquem corretamente inicializados, mas o valor numérico fique o simétrico do original (para se distinguirem). Execute o programa e confirme e explique as mensagens que aparecem.

l) Modifique função *teste* de forma a que o seu parâmetro seja passado por referência. Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

m) Modifique a função *teste* de forma a retornar por valor um objeto *MSG* e remova o parâmetro. No corpo da função retorne o objeto *aux*. Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

n) Modifique função *teste* de forma a que o retorno passe a ser por referência. Execute o programa, analise e justifique as mensagens e a ordem pela qual aparecem. Explique as diferenças nas mensagens relativamente à alínea anterior.

o) Alínea de reflexão.
Escreva no seu caderno todas as conclusões acerca do comportamento da criação e destruição de objetos que observou ao longo deste exercício.

No final do exercício deverá

- Ter consolidado o conhecimento acerca do ciclo de vida dos objetos e de que forma a passagem de parâmetros/retorno de funções pode levar à criação de objetos temporários.
- Consolidado o conhecimento quanto ao uso de {} e () na inicialização de objetos.
- Conhecer as implicações sintáticas do uso de *arrays* de objetos sobre o construtor da classe desses objetos.