

Nome da(o) estudante: _____ Código: _____

1. [12.0 valores = 0.5 + 1.0 + 1.0 + 1.5 + 1.5 + 2.5 + 3.0 + 1.0]

A seguinte classe é usada para representar um porta-moedas:

```
class Purse
{
public:
    Purse(); // cria um porta moedas vazio sem dono especificado
    Purse(string owner); // cria um porta moedas vazio cujo dono é "owner"
    void addCoin(float coin); // acrescenta a moeda indicada como parâmetro
    void addCoins(vector<float> coins); // acrescenta as moedas indicadas como parâmetro
    bool removeCoin(float coin); // se existir uma moeda com o valor indicado como parâmetro
                                   remove-a e retorna true, se não retorna false
    ... removeAmount(...) ... ; // VER DESCRIÇÃO NA ALÍNEA a)
    float tellAmount() const; // retorna o montante total
    vector<float> tellCoins() const; // retorna uma cópia do vetor das moedas
    void show() const; // mostra o dono, as moedas e o montante total
    // ... outros métodos
    static vector<float> possibleCoins; // moedas possíveis: 2, 1, 0.5, 0.2 e 0.1 euro
private:
    string owner; // o dono do porta-moedas
    vector<float> coins; // as moedas
    float amount; // o montante total (soma do valor das moedas)
};
```

a) [0.5] Escreva o protótipo do método **removeAmount()**, tendo em conta que este método deve ter como parâmetro um montante a remover do porta-moedas e devolver a seguinte informação: se é possível ou não perfazer esse montante com as moedas existentes no porta-moedas e um vetor com as moedas que constituem o montante. Nota: pode acrescentar outros parâmetros se achar conveniente.

Possíveis soluções:

```
bool removeAmount(float amount, vector<float> & coins);
void removeAmount(float amount, bool & possible, vector<float> & coins);
vector<float> removeAmount(float amount, bool & possible); // menos valorizada
```

b) [1.0] Defina e inicialize o vetor **possibleCoins**, tendo em conta que os valores possíveis das moedas são: 2, 1, 0.5, 0.2 e 0.1 euro. Indique onde colocaria a definição, e explique o significado do qualificativo **static**.

Definição e inicialização: `vector<float> Purse::possibleCoins = { 2, 1, 0.5F, 0.2F, 0.1F };`

NOTA: p/evitar warnings seria conveniente acrescentar F nos valores não inteiros – não desvalorizado

Esta definição seria colocada no espaço global do programa

Neste contexto, o qualificativo **static** significa que só existe uma instância de **possibleCoins** para todos os objetos da classe **Purse**

c) [1.0] Escreva o código do construtor com parâmetro.

```
Purse::Purse(string owner)
{
    this->owner = owner;
    this->amount = 0;
    // o vector é automaticamente inicializado sem conteúdo
}
```

d) [1.5] Escreva o código do método **addCoins()**.

```
void Purse::addCoins(vector<float> coins)
{
    for (size_t i = 0; i < coins.size(); i++)
    {
        this->coins.push_back(coins[i]);
        amount = amount + coins[i];
    }
}
```

e) [1.5] Escreva o código do método **removeCoin()**.

```
bool Purse::removeCoin(float coin)
{
    vector<float>::iterator it = find(coins.begin(), coins.end(), coin);
    if (it != coins.end())
    {
        coins.erase(it);
        amount = amount - coin;
        return true;
    }
    return false;
}
```

Nome da(o) estudante: _____ Código: _____

f) [2.5] Implemente a função que faz a sobrecarga (*overloading*) do operador de teste de igualdade para objetos da classe **Purse**. Considera-se que 2 porta-moedas são iguais se o montante que contém for o mesmo e as moedas forem iguais. Sugestão: ordene as moedas de ambos os porta-moedas antes de comparar o seu conteúdo.

```
bool operator==(const Purse & p1, const Purse & p2)
{
    if (p1.tellAmount() != p2.tellAmount())
        return false;
    vector<float> c1 = p1.tellCoins(), c2 = p2.tellCoins();
    sort(c1.begin(), c1.end());
    sort(c2.begin(), c2.end());
    if (c1 != c2) // *** alternativa: ciclo para comparar os elementos um a um (ver abaixo)
        return false;
    return true;
}

-----
// alternativa a "if (c1 != c2)"
size_t i;
if (c1.size() != c2.size())
    return false;
for (i = 0; i < c1.size() && i < c2.size(); i++)
    if (c1[i] != c2[i])
        return false;
```

g) [3.0] Escreva a função **main()** de um programa que faz o seguinte:

- Cria o porta-moeda da "Ana", coloca lá as moedas 2, 1, 0.5, 0.5, 0.2, 0.2 e 0.1 euro, e mostra o conteúdo do porta-moedas (usar o método **show()**).
- Cria um porta-moedas vazio (sem dono especificado).
- Insere neste porta-moedas tantas moedas (selecionadas aleatoriamente) quantas as que tem o porta-moedas da "Ana" e mostra o conteúdo deste porta-moedas "aleatório". Sugestão para selecionar aleatoriamente uma moeda: usar um número gerado aleatoriamente para indexar um dos elementos do vector **possibleCoins** (ver definição da classe **Purse**).
- Indica no ecrã se o conteúdo dos 2 porta-moedas é ou não igual. Nota: se não respondeu à pergunta anterior, considere que está disponível a função que faz a sobrecarga do operador de teste de igualdade.

```
int main()
{
    srand((unsigned int) time(NULL)); // não "obrigatório"

    Purse purse1("Ana");
    vector<float> coins1 = { 2, 1, 0.5F, 0.5F, 0.2F, 0.2F, 0.1F };
    purse1.addCoins(coins1);
    purse1.show();

    Purse purse2;
    for (unsigned int i = 1; i <= coins1.size(); i++)
    {
        float coin = Purse::possibleCoins.at(rand() % Purse::possibleCoins.size());
        purse2.addCoin(coin);
    }
    purse2.show();
    if (purse1 == purse2)
        cout << "Porta-moedas iguais\n";
    else
        cout << "Porta-moedas diferentes\n";
}
```

h) [1.0] As moedas contidas no porta-moedas podiam ser representadas numa estrutura de dados de outro tipo. Escolha a estrutura de dados alternativa que considere mais adequada, declare o atributo **coins** usando essa estrutura e escreva o método **addCoin()** para a estrutura escolhida. Justifique brevemente a sua escolha.

Escolheria um multiset

porque é uma estrutura de dados adequada para representar conjuntos de valores do mesmo tipo e porque podem existir moedas repetidas

Declaração do atributo **coins** usando a estrutura escolhida:

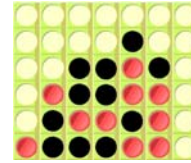
```
multiset<float> coins;  
void Purse::addCoin(float coin)  
{  
    coins.insert(coin);  
}
```

NOTA: também se poderia usar um map< float, unsigned int> representando para cada moeda o número de ocorrências respectivo. Nesse caso a inserção de uma moeda poderia ser feita c/a instrução m[coin]++;

Nome da(o) estudante: _____ Código: _____

2. [5.0 valores = 2.0 + 1.0 + 1.0 + 1.0]

Em muitos "jogos de tabuleiro" o tabuleiro é constituído por uma matriz bidimensional de células. Em certos jogos cada célula pode ser representada apenas por um carácter (**char**). A figura ao lado mostra um exemplo de um tabuleiro em que são usados apenas 2 tipos de peças e a respetiva representação com recurso a uma matriz de caracteres.



```
. . . . 0 . .
. . X 0 X 0 .
. X 0 0 X X .
. 0 X X 0 X .
X 0 0 0 X X .
```

a) [2] Defina uma classe para representar um tabuleiro bidimensional deste tipo. A classe deve incluir métodos para:

- construir um tabuleiro com as dimensões indicadas como parâmetros, em que todas as células estão preenchidas com o símbolo de célula vazia; este símbolo, '.', no caso do exemplo apresentado, é um dos parâmetros do construtor;
- obter o número de linhas/colunas do tabuleiro;
- obter o conteúdo de: uma célula, uma linha ou uma coluna;
- modificar o conteúdo de uma célula (deve ter como parâmetros as coordenadas da célula e o símbolo a colocar lá; deve retornar um valor indicando se foi possível ou não modificá-la);
- mostrar o conteúdo do tabuleiro.

```
class Board
{
public:
    Board(size_t numLines, size_t numColumns, char emptySymbol);
    size_t getNumLines() const;
    size_t getNumColumns() const;
    char getPieceAt(size_t line, size_t column) const;
    string getLineAt(size_t line) const;
    string getColumnAt(size_t column) const;
    bool setPieceAt(size_t line, size_t col, char piece);
    void show() const;
private:
    vector<string> board; // OU vector<vector<char>> board;
    char emptySymbol;
    size_t numLines;
    size_t numColumns;
};
```

b) [1] Escreva o código do construtor da classe que tem como parâmetros as dimensões do tabuleiro e o símbolo de célula vazia.

```
Board::Board(size_t numLines, size_t numColumns, char emptySymbol)
{
    string line(numColumns, emptySymbol);
    for (size_t i = 0; i < numLines; i++)
        board.push_back(line);
}
```

c) [1] Escreva o código do método **getColumn()** que retorna uma **string** com o conteúdo de uma coluna do tabuleiro, cujo número recebe como parâmetro; a **string** deve conter os caracteres em sequência desde a primeira até à última linha do tabuleiro.

NOTA IMPORTANTE: considere que para os utilizadores da classe **Board** as linhas/colunas do tabuleiro são numeradas a partir de 1.

```
string Board::getColumn(size_t column) const // NOTA: para o utilizador, a 1ª coluna é a coluna 1
{
    string colStr;
    for (size_t i = 0; i < numLines; i++)
        colStr = colStr + board.at(i).at(column - 1); // OU ... board[i][column - 1];
    return colStr;
}
```

d) [1] A função **testSequence()** tem como parâmetros: um tabuleiro; o número de uma coluna; um carácter representando um símbolo usado num jogo, e um número **n**. A função deve retornar o valor **true** se na coluna indicada existir uma sequência de n símbolos consecutivos iguais ao indicado. Exemplo: se o tabuleiro usado na ilustração da página anterior estiver representado num **Board b**, a chamada **testSequence(b, 6, 'x', 3)** deve retornar o valor **true**, uma vez que na coluna **6** de **b** existe uma sequência de **3 'x'** consecutivos.

Escolha um protótipo adequado para a função. Use a função **Board::getColumn()** da alínea anterior para obter o conteúdo da coluna.

```
bool testSequence(const Board &b, size_t column, char symbol, size_t n)
{
    string colStr = b.getColumnAt(column);
    string pattern = string(n, symbol);
    if (colStr.find(pattern) == string::npos)
        return false;
    return true;
}

}
```

Nome da(o) estudante: _____ Código: _____

3. [3.0 valores = 0.6 + 0.6 + 0.6 + 0.6 + 0.6]

a) [0.6] O que será mostrado no ecrã como resultado da seguinte sequência de instruções?

```
int * p = new int[2]; *p = 1; *(p + 1) = 2; cout << p[0] << " - " << p[1];
```

1 - 2

b) [0.6] Considere os extratos de código de um programa a seguir apresentados:

```
class Point {  
public:  
    .... // a completar  
private:  
    int x, y; // coordenadas do ponto  
};  
...  
Point p1, p2(-1,5);  
p1.setX(2).setY(-3); //coord.s de p1 passam a ser (2,-3)  
...
```

Indique os protótipos dos método(s) da classe **Point** que seria necessário implementar para que este extrato de código possa ser compilado e executado com sucesso.

método(s) da classe:

```
Point();  
Point(int x, int y);  
Point & setX(int x);  
Point & setY(int y);
```

c) [0.6] Em geral, as coordenadas de um ponto num espaço 2D podem ser definidas através de 2 números, 2 letras, um número e uma letra, ... (exemplos: (2,-1) ou ('A','J') ou (3,'J'), ...).

Defina uma template class **Point** que permita representar pontos com estas características; considere que os únicos métodos são o construtor com parâmetros e os métodos get() de cada um dos atributos **x** e **y**.

```
template <typename T1, typename T2>  
class Point {  
public:  
    Point(T1 x, T2 y);  
    T1 getX() const;  
    T2 getY() const;  
private:  
    T1 x;  
    T2 y;  
};
```

Defina um ponto **p1** deste tipo, cujas coordenadas **x** e **y** são respetivamente 'A' e 3.

```
Point<char, int> p1('A', 3);
```

d) [0.6] A Standard Template Library de C++ disponibiliza a função **reverse()** para inverter a ordem dos elementos compreendidos na gama **[first,last[** especificada nos seus parâmetros. O *template* dessa função é o seguinte:

```
template <class BidirectionalIterator>  
void reverse (BidirectionalIterator first, BidirectionalIterator last);
```

É possível inverter a ordem dos elementos da variável **vec**, do tipo **vector<int>**, usando esta função? Justifique. Caso seja possível escreva a instrução necessária para fazer a inversão dos elementos de **vec**, usando esta função. Caso não seja possível, escreva o código de uma função equivalente que possa ser aplicada a um **vector<int>**.

É possível porque a classe **vector** suporta iteradores do tipo **RandomIterator** e a função **reverse** apenas "exige" iteradores do tipo **BidirectionalIterator**.

Chamada a **reverse** para inverter **vec**:
reverse(vec.begin(), vec.end());

e) [0.6] As classes **Bird** e **Fish** são derivadas da classe **Animal**. Em C++, é possível guardar objetos do tipo **Bird** ou **Fish** em variáveis do tipo **Animal**? Em caso afirmativo, vê algum problema em fazê-lo?

É possível.

O problema é que os atributos de **Bird** ou de **Fish** que não sejam atributos de **Animal** se perdem ao fazer a atribuição ("slicing problem").