U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

MESTRADO INTEGRADO EM ENGENHARIA INFORMÁTICA E COMPUTAÇÃO | 1º ANO
EIC0012| PROGRAMAÇÃO | 2019/2020 – 2º SEMESTRE

EXAME - ÉPOCA DE RECURSO | 2020-07-07

Com consulta. Duração: 1h30m

Nome do estudante: _____ **Código UP:** _____ **Nº prova:** _____

1. [4.0]

Indique quais das seguintes afirmações são verdadeiras e quais são falsas, assinalando respetivamente com **V** ou **F.**
NOTA - a pontuação nesta pergunta será dada pela fórmula: **máximo (0, (nº_respostas_certas – nº_respostas_erradas) x 0.2)**

		V / F
Num programa em **C++** é possível fazer a seguinte definição: `string auto = "Cadillac";`		F
Para testar se **x**, do tipo `int`, tem um valor igual a **19** ou igual a **20** pode-se usar a instrução: `if ( x == ( 19 || 20 ) ) ...`		F

A função seguinte "converte" letras maiúsculas ou minúsculas em números inteiros, da seguinte forma: '**A**' ou '**a**' são convertidas no inteiro **0** (zero), '**B**' ou '**b**', são convertidas no inteiro **1**, '**C**' ou '**c**' são convertidas no inteiro **3**, etc.; se o argumento da chamada à função não for uma letra maiúscula ou minúscula o valor retornado será -1.	`int toInt(char c) {`
	`  if isupper(c) return c-'A';`
	`  else if islower(c) return c-'a';`
	`  return -1;`
	`}`
		F

Sendo **name** uma string de **C** é possível testar se o seu valor é "**Ann**" usando a seguinte instrução: `if (name == "Ann") ...`		F

O seguinte ciclo termina se for lido do teclado um número pertencente ao intervalo `[0..100]`.	`int n;`
	`do {`
	`  cout << "Number ? ";  cin >> n;`
	`} while (n<0 || n>100);`
		V

Para escrever um valor inteiro num ficheiro de texto é necessário converter previamente o inteiro numa *string.*		F

As funções seguintes podem coexistir num mesmo programa
`void volume(double side);` `// calcula o volume de um cubo de lado 'side'`
`void volume(double radius);` `// calcula o volume de uma esfera de raio 'radius'`
desde que sejam invocadas de modo adequado, por exemplo: `volume(side=4);` ou `volume(radius=7);`
		F

A função **f** é uma função recursiva que calcula o somatório de todos os valores pertencentes ao intervalo `[0..n]` .	`int f(int n) {`
	`  if (n == 0 || n == 1) return n;`
	`  else return (f(n - 1) + n);`
	`}`
		V

Sendo **year**, **month** e **day** variáveis de tipo `int`, ao ser executada a instrução `cin>>year>>month>>day;` o utilizador do programa em que ela for executada pode escrever qualquer uma das seguintes entradas porque tudo o que não forem valores inteiros é ignorado.	`2020 7 7`
	`2020 07 07`
	`2020/07/07`
		F

O resultado da execução do seguinte código é atribuir a **s** o valor "`13:05`".	`int h = 13, m = 5;`
	`ostringstream oss;`
	`oss << setw(2) << h << ':'<< setw(2) << m;`
	`string s = oss.str();`
		F

O código seguinte, embora sintaticamente correto, constitui um erro de programação porque a variável **grade** não foi devidamente inicializada.	`int *grade;` `*grade = 19;`	V
Para reservar dinamicamente espaço para um *array* de **n** inteiros e inicializar os elementos do *array* com os valores **1, 2, 3, …, n**, pode-se usar o seguinte código:	`int *a=(int *)malloc(n*sizeof(int));` `for (int i=0; i<n; i++) a[i]=i+1;`	V
O valor `string::npos` é retornado pelas diferentes funções `find` que podem operar sobre uma *string* de **C++** (ex: `find()`, `find_first_of()`, etc.) para indicar que o caráter procurado foi encontrado na última posição da *string*.		F
Os contentores da **STL** que suportam *random access iterators* podem ser usados com qualquer um dos algoritmos da **STL**.		V
Os elementos de um `std::multimap` são do tipo `multipair` e podem haver `multipair`'s repetidos num mesmo `multimap`.		F
A definição de uma *template class* dependente de um tipo genérico **T** tanto pode ser precedida pelo texto `template < class T >` como pelo texto `template < typename T >` .		V
Sendo `Person` o identificador de uma classe, a forma de definir uma variável **p** do tipo `Person`, invocando o construtor por omissão (*default constructor*), é escrever: `Person p();` .		F
Um atributo (membro dado) de uma classe qualificado como `static` é partilhado por todos os objetos da classe, existindo apenas uma instância desse atributo, comum a todos os objetos da classe.		V
Os membros de uma classe base que sejam classificados com `protected` podem ser acedidos diretamente pelos métodos (membros funções) de uma classe derivada dessa classe base.		V
Sendo a classe `ClassB` derivada da classe `ClassA`, quando se instancia um objeto da classe `ClassB` (por exemplo: `ClassB x;`), o construtor por omissão (*default constructor*) de `ClassA` será invocado automaticamente, sem que o programador necessite de escrever código para o invocar.		V

Nome do estudante: _____ **Código UP:** _____ **Nº prova:** _____

Nota: nesta prova apenas é necessário fazer tratamento de erros e indicar os ficheiros de inclusão quando tal for solicitado explicitamente

2. [6.0]

Uma sequência genética é definida por um conjunto de tripletos de nucleótidos, por exemplo, **{"GCU", "ACG", "GAG"}**. Pretende-se desenvolver uma programa para lidar com sequências genéticas, sendo uma sequência representada por um **vector<string>**.

a) [1.5] Considerando que existe uma função **string getTriplet()** que retorna um tripleto, gerado aleatoriamente (por exemplo, "**ACG**"), escreva a função **generateSequence()** que retorna uma sequência de **n** tripletos, sendo **n** um parâmetro da função. Defina o protótipo da função como achar adequado.

```
vector<string> generateSequence(int n) {
  vector<string> sequence;
  for (int i = 1; i <= n; i++)
    sequence.push_back(getTriplet());
  return sequence;
}
```

b) [2.0] Escreva uma função **removeTriplets()** que tem como parâmetros uma sequência genética e um tripleto, remove da sequência todas as instâncias desse tripleto, e retorna o número de remoções. Defina o protótipo da função como achar adequado.

Exemplo: numa situação em que a sequência é **{"GCU", "ACG", "GAG", "UGA", "ACG"}** e o tripleto a remover é **"ACG"**, a sequência resultante é **{"GCU", "GAG", "UGA"}** e o número de remoções retornado é 2.

```
int removeTriplets(vector<string> &sequence, string triplet) {
  int count = 0;
  size_t i = 0;
  while (i < sequence.size()) {
    if (sequence.at(i) == triplet) {
      sequence.erase(sequence.begin() + i);
      count++;
    }
    else
      i++;
  }
  return count;
}
```

c) [1.5] Escreva uma função **main()** que permita ao utilizador especificar um valor inteiro **n** e um tripleto **t**, gere uma sequência de tamanho **n**, remova da sequência todas as ocorrências do tripleto **t**, e mostre no ecrã a sequência resultante.

Exemplo da interface de execução, se a sequência gerada for a apresentada em **2.b**:

```
n? 5              // foi gerada a sequência {"GCU", "ACG", "GAG", "UGA", "ACG"}
t? ACG
GCU  GAG  UGA     // a sequência resultante depois de removido o tripleto "ACG"
2 triplets were removed from the original sequence
```

```
int main() {
  int n;
  cout << "n ? "; cin >> n;
  string triplet;
  cout << "t ? "; cin >> triplet;
  vector<string> sequence = generateSequence(n);
  int count = removeTriplets(sequence, triplet);
  for (auto s : sequence) cout << s << endl;
  cout << count << "2 triplets were removed from the original sequence" << endl;
  return 0;
}
```

d) [1.0] Um tripleto é constituído por uma sequência de 3 letras, selecionadas aleatoriamente entre as letras do conjunto **{'A','U','G','C'}**, podendo haver letras repetidas. Considere que um tripleto deveria ser guardado numa string de **C** em vez de uma string de **C++**.
Escreva o código da função **void getTriplet_C(char *triplet)** que devolve o tripleto gerado através do parâmetro **triplet**. Escreva também o pedaço de código da função **main()** necessário para guardar um tripleto gerado pela função **getTriplet_C()** numa string de **C**, com o nome **t**.

```
void getTriplet_C(char *triplet) {
  char nucleotids[] = "AUGC";
  for (int i = 0; i < 3; i++)
    triplet[i]=nucleotids[(rand() % 4)];
  triplet[3] = '\0';
}
```

---

código de **main()**:

```
char t[4];
getTriplet_C(t);
```

Nome do estudante: _____ **Código UP:** _____ **Nº prova:** _____

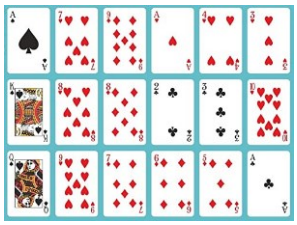
3. [4.5]

Um jogo de cartas é jogado num tabuleiro semelhante ao ilustrado na figura ao lado. Para representar o tabuleiro foram definidos os seguintes tipos de dados:

```
struct Card {
  char suit;      // o naipe da carta: H, S, D ou C (sigla da designação em inglês)
  string symbol;  // o símbolo da carta: A, K, V, D, 2, 3,... ou 10
};

class Board {
public:
  Board(const string &filename);
  void printWhere(const set<Card> &s) const;   // descrito em 3.b
  // outros métodos
private:
  int numLines, numCols;     // número de linhas e de colunas de 'board'
  vector<vector<Card>> board;
  // outros atributos
};
```

O ficheiro do jogo especifica o tamanho do tabuleiro na primeira linha (**linhas x colunas**) e as cartas que compõem o tabuleiro, uma carta por cada linha do ficheiro; cada carta é especificada por um caráter que representa o naipe, **suit**, e por uma *string* que representa o símbolo, **symbol**. As cartas estão organizadas por linhas do tabuleiro (as cartas da 1ª linha, seguidas pelas cartas da 2ªa linha, etc.), como ilustrado à direita.



Ficheiro do jogo, para o tabuleiro acima, indicando as dimensões do tabuleiro e as cartas (1.a linha, 2.a linha, ...):

```
3 x 6
S A
H 7
D 9
H A
H 4
H 3
S K
H 8
D 8
C 2
C 3
H 10
S Q
H 9
D 7
D 6
D 5
C A
```

a) [2.5] Implemente o construtor de **Board**. O nome do ficheiro é dado como parâmetro e o seu conteúdo deve ser importado para o atributo **board**. Considere que o ficheiro não contém valores errados. Se o ficheiro não existir, o construtor deve lançar uma exceção do tipo **runtime_error** com a mensagem "**file not found**".

```
Game::Game(const string &filename) {
  ifstream f(filename);
  if (!f.is_open())
    throw runtime_error("file not found");
  char sep;
  f >> numLines >> sep >> numCols;
  board.resize(numLines);
  for (int i = 0; i < numLines; i++) {
    board.at(i).resize(numCols);
    for (int j = 0; j < numCols; j++) {
      Card card;
      f >> card.suit >> card.symbol;
      board.at(i).at(j) = card;
    }
  }
  f.close();
}
```

b) [2.0] Escreva o código do método **printWhere** da classe **Board**, o qual recebe como parâmetro um conjunto de cartas e para cada uma das cartas desse conjunto mostra qual é a posição que ela ocupa no tabuleiro ou "**not found**" se ela não existir no tabuleiro. Apresenta-se ao lado um exemplo de saída do programa, para o tabuleiro apresentado acima, numa situação em que o referido conjunto tinha 4 cartas. **Nota**: as cartas são pesquisadas pela ordem que ocupam no conjunto.

Exemplo de saída do método **printWhere**:

```
C  A - 2,5
H 10 - 1,5
S  7 - not found
S  A - 0,0
```

```
void Game::printWhere(const set<Card> &setCard) const {
  for (auto c1 : setCard) {
    cout << c1.suit << ' ' << setw(2) << c1.symbol << " - ";
    bool found = false;
    for (int i = 0; i < numLines; i++) {
      for (int j = 0; j < numCols; j++) {
        Card c2 = board.at(i).at(j);
        if (c1.suit == c2.suit && c1.symbol == c2.symbol) {
          cout << i << ',' << j << endl;
          found = true;
          break;
        }
      }
    }
    if (!found)
      cout << "not found" << endl;
  }
}
```

Nome do estudante: _____ **Código UP:** _____ **Nº prova:** _____

4. [3.5]

As classes **Student** e **Course** são usadas para representar informação sobre estudantes e unidades curriculares que eles frequentam.

```
class Student   // representa um estudante
{
public:
  Student(int id=0, const string &name="");
  int getId() const;
  string getName() const;
  void showGrades() const;
  // ... outros métodos
private:
  int id;                   // número de identificação do estudante
  string name;              // nome do estudante
  vector<Course *> courses; // UC's em que está matriculado
  map<int, int> grades;     // notas obtidas nas UC's (chave= courseId)
  // ... outros atributos
};
```

```
class Course   // representa uma unidade curricular (UC)
{
public:
  Course(int id=0, const string &name="");
  int getId() const;
  string getName() const;
  // ... outros métodos
private:
  int id;
  string name;
  vector<Student *> students; // estudantes que frequentam o curso
  // ... outros atributos
};
```

a) [2.0] Escreva o código do método **showGrades** da classe **Student** que faz o seguinte: mostra o nome do estudante e, para cada uma das unidades curriculares (UC's) que o estudante está matriculado, mostra a classificação e o nome da unidade curricular em que foi obtida, como se ilustra ao lado; o nome do estudante deve ser sublinhado com carateres '-', tantos quanto o comprimento do nome. Se o estudante estiver matriculado numa unidade curricular (em **courses**) mas ainda não tiver sido registada a sua classificação (em **grades**), na listagem deve ser apresentado "**??**" em vez da classificação (ver exemplo ao lado).

```
Ana Sousa
---------
17 - Análise Matemática
 9 - Matemática Discreta
19 - Fundamentos da Programação
?? - Programação
...
...
Pedro Santos
------------
13 - Complementos de Matemática
...
```

```
void Student::showGrades() const {
  cout << name << endl;
  cout << string(name.length(), '-') << endl;
  for (auto coursePtr : courses) {
    int courseId = coursePtr ->getId();
    auto it = grades.find(courseId);
    if (it != grades.end())
      cout << setw(2) << it->second << " - " << coursePtr ->getName() << endl;
    else
      cout << "?? - " << coursePtr ->getName() << endl;
  }
}
```

b) [1.5] Tendo em conta que um estudante pode ter de frequentar a mesma unidade curricular mais de uma vez, foi introduzida uma alteração na classe **Student** que consistiu em substituir a definição do atributo **grades** pela seguinte: **`map<int, map<string,int>> grades;`** .
A chave de **grades** continua a ser o identificador do curso; o valor associado representa as classificações obtidas em diferentes anos letivos (num ano letivo, pelo menos), sendo cada ano letivo representado por uma *string* no formato **AAAA/BBBB** (por exemplo, "**2019/2020**").
Considerando a nova definição do atributo **grades**, indique as alterações a introduzir no método **showGrades** da classe **Student** (**apenas as alterações** – assinalando o local onde seriam inseridas no código que escreveu em **4.a**) de modo a mostrar apenas a última das classificações obtidas em cada unidade curricular.

As alterações a introduzir seriam substituir a instrução assinalada com fundo azul pelo seguinte bloco de instruções:

```
    {
      auto courseIdInfo = it->second;
      cout << setw(2) << courseIdInfo.rbegin()->second << " - " << coursePtr->getName() << endl;
    }
```

Nome do estudante: _____ **Código UP:** _____ **Nº prova:** _____

5. [2.0]

Considere novamente as classes **Student** e **Course** do problema **4**.

a) [1.0] Pretende-se desenvolver um programa para gerir o funcionamento de um curso, envolvendo estudantes (**Student**'s) e unidades curriculares (**Course**'s). Tendo em conta a interdependência entre as duas classes, diga por que ordem escreveria a sua definição, num único ficheiro de código (não reescreva a definição das classes; indique apenas a sua posição relativa usando o formato **class *NomeDaClasse* { ... };**). Justifique brevemente a sua resposta.

A classe **Student** tem como atributo um **vector<Course *>** e a classe **Course** tem como atributo um **vector<Student *>**.

Não há nenhuma ordem que permita que a definição de uma apareça antes da definição da outra.

Numa situação destas, a solução é recorrer a uma *forward declaration.*

Assim, se decidirmos colocar em primeiro lugar a definição de **Student**, o código deveria ser escrito da seguinte forma:

class Course; // forward declaration

class Student { … }; // definição da classe Student (código omitido)

class Course { … }; // definição da classe Student (código omitido)

b) [1.0] Considere que no programa referido em **5.a** se pretende incluir uma variável do tipo **set<Student>**, de modo a que os estudantes fiquem ordenados por nome. Para que isso seja possível, diga o que seria necessário acrescentar à classe **Student** e escreva o respetivo código. Justifique brevemente a resposta.

O compilador não sabe como comparar objetos do tipo **Student** para poder inseri-los de forma ordenada no **set<Student>**.

Para que consiga fazê-lo, é necessário definir o operador **<** para objetos do tipo **Student**.

A definição desse operador tanto poderia ser feita externamente à classe como dentro da classe.

Uma vez que se pretende acrescentar à classe, o operador deverá ser definido como:

```
bool Student::operator<(const Student & s) const {
  return name < s.name;
}
```

FIM