Programação Orientada a Objetos 2019/2020

Exame da Época de Recurso

DEIS – LEI / LEI-PL / LEI-CE

Duração da prova: 2 h

Número <u>novo</u>	**Nome**

Perguntas de escolha múltipla:

Assinale, para cada pergunta, apenas uma opção.

Registe a opção escolhida para cada pergunta na grelha de respostas.

Todas as perguntas de escolha múltipla têm a mesma cotação.

Cada resposta errada tem uma penalização de 20% da cotação da pergunta.

As cotações negativas das perguntas de escolha múltipla não afetam as restantes perguntas.

Grelha de respostas:

Pergunta	**1**	**2**	**3**	**4**	**5**
Opção escolhida					

1. A classe Funcionario representa a ficha de um funcionário de uma empresa. A classe Reuniao tem informação acerca dos funcionários convocados e da ordem de trabalhos da reunião (cada ponto da ordem de trabalhos é representado por uma string). A reunião tem a posse exclusiva da sua ordem de trabalhos. As fichas dos funcionários são da responsabilidade da empresa, a reunião apenas as utiliza para representar os funcionários convocados. Considere a seguinte definição da classe Reuniao:

```
class Funcionario { /* ... */ };
class Reuniao {
    vector<Funcionario *> convocados;
    vector<string *> ordemDeTrabalhos;
public:
    //...
    ~Reuniao() {
          for (string * s : ordemDeTrabalhos) { delete s; }
    }
};
```

Qual seria a versão correcta para o construtor por cópia da classe Reuniao?

A

```
Reuniao(const Reuniao & ob) {
  convocados = ob.convocados;
  for (string * o : ob.ordemDeTrabalhos) {
     ordemDeTrabalhos.push_back(new string(*o));
  }
}
```

B

```
Reuniao(const Reuniao & ob) {
  for (Funcionario * p : ob.convocados) {
     convocados.push_back(new Funcionario(*p));
  }
  ordemDeTrabalhos = ob.ordemDeTrabalhos;
}
```

C

```
Reuniao(const Reuniao & ob) {
  for (Funcionario * p : ob.convocados) {
     convocados.push_back(new Funcionario(*p));
  }
  for (string * o : ob.ordemDeTrabalhos) {
     ordemDeTrabalhos.push_back(new string(*o));
  }
}
```

D

```
Reuniao(const Reuniao & ob) {
  convocados = ob.convocados;
  ordemDeTrabalhos = ob.ordemDeTrabalhos;
}
```

2. Considere as seguintes definições:

```
class Num {
    int n;
public:
    Num() { n = 0; }
    void setN(int nn) { n = nn; }
    int getN()const { return n; }
};
void f(const Num * p) {
     p->setN(5);       // linha 1
     p = new Num();    // linha 2
}
void g(Num * const p) {
     p->setN(5);       // linha 3
     p = new Num();    // linha 4
}
int main() {
     Num ob;           // linha 5
     ob.setN(4);       // linha 6
     f(&ob);           // linha 7
     g(&ob);           // linha 8
}
```

Indique quais as linhas deste programa em que ocorrem erros de compilação

A 1, 2, 3, 4 **B** 1, 3 **C** 2, 4 **D** 1, 4

3. Qual será a saída resultante da execução deste programa?

```
class Erro {
    string msg;
public:
    Erro(const string & s = "") : msg(s) {}
    string what() {
           return msg;
    }
};

void calculaArea(int lado) {
    try {
           if (lado < 0) {
                  throw Erro("lado negativo");
           }
           else if (lado > 10) {
                  throw new string("lado superior ao limite");
           }
    }
    catch (Erro & e) {
           lado = 0;
    }
    catch (string & e) {
           lado = 10;
    }
    cout << "area =  " << lado * lado << ";";
}
int main() {
    try {
           calculaArea(-2);
           cout << "depois; ";
    }
    catch (Erro & e) {
           cout << "catch main; \n";
    }
    catch (string & e) {
           cout << "catch main; \n";
    }
    cout << "fim main; \n";
};
```

A area = 10; depois; fim main;

B depois; fim main;

C area = 0; depois; fim main;

D catch main; fim main;

4. Qual será a saída resultante da execução deste programa?

```
class Canideo {
    int peso;
    string dieta;
public:
    Canideo(int a, string b = "Carne") : peso(a), dieta(b) {}
    Canideo() = default;

    virtual ~Canideo() {}

    virtual double dose() const { return peso * 0.1; }
    int getPeso() const { return peso; }
    string getDieta() const { return dieta; }
};

class Cao : public Canideo {
    string nome;
public:
    Cao(int a, string b = "Racao", string c = "Bobi") :Canideo(a, b), nome(c) {}
    virtual double dose() const override { return getPeso() * 0.05; }
};

class Lobo : public Canideo {
    string habitat;
public:
    Lobo(int a, string b = "Galinha", string c = "Floresta") : Canideo(a), habitat(c) {}
};

int main() {
    Lobo a(120);
    Canideo *c = new Cao(20);

    cout << a.getDieta() << " " << c->getDieta() << " ";
    cout << a.dose() << " " << c->dose() << endl;
}
```

A Carne Racao 12 2

B Galinha Racao 12 1

C Carne Racao 12 1

D Galinha Racao 12 2

5. Pretende-se acrescentar um novo tipo de cão à estrutura de classes da pergunta anterior. O Galgo é um cão que tem uma determinada velocidade indicada na criação dos objetos deste tipo (valor inteiro representando metros por segundo). A velocidade vai ter impacto na sua dose diária. Qual a solução mais apropriada para adicionar esta informação? Quais das seguintes versões de construtor para a classe Galgo estão corretas?

A Criar uma classe derivada de Canideo com atributo velocidade. Esta classe deve ter um construtor e tem que redefinir o método virtual dose().;

B Criar uma classe derivada de Cao com atributo velocidade. Esta classe deve ter um construtor e tem que redefinir o método virtual dose();

C Criar uma classe derivada de Cao com atributo velocidade. Esta classe deve ter um construtor, mas pode utilizar o método virtual dose() da classe base.;

D Acrescentar um atributo booleano tipo à classe Cao, indicando se os objetos são Galgos. Adaptar o método dose() desta classe, para calcular a dose diária correta se os cães em questão forem galgos.

6. Considere a classe seguinte:

```
class Excursao {
   string data;
   const string destino;
   int lotacao;
   vector<int> inscritos;
};
```

A classe Excursao representa os viajantes inscritos para irem numa certa data a um dado destino. Os viajantes são identificados pelo seu número do passaporte, e naturalmente, não se aceitam números repetidos. Uma excursão é planeada para uma determinada quantidade máxima (lotação) de viajantes, e quando atingida não aceita mais.

Pode acrescentar coisas à classe, mas não retirar nem mudar o que já existe.

Faça com que seja possível:

- Criar objetos indicando todos os dados exceto o conjunto de viajantes inscritos, que inicialmente é vazio;
- Inscrever viajantes dados os seus passaportes: **excursao3 << 123 << 789 << 123;** // o segundo 123 é ignorado
- Transferir passageiros de uma excursão para outra: **excursao3 >> excursao2;** // os viajantes de excursão 3 saem dessa excursão e passam para a excursao2 (sendo acrescentados a essa). Os que não couberem na excursao2 permanecem onde estavam.
- Atribuir excursões; **excursao4 = excursao5;** // Copia o conteúdo de excursão 5 para 4. É uma "cópia" entre aspas: por exemplo, quanto aos passageiros, é na verdade uma transferência como no caso acima. A diferença é que os passageiros iniciais de excursao4 "saem" todos antes da transferência.
- Aceder a um viajante específico e poder por outro no seu lugar: **excursao2[1] = 329** (o viajante com passaporte 329 substitui o 789, que sai da viagem excursao2 - do segundo exemplo de cima). Trata-se de uma substituição e nunca de uma adição.
- Saber quantos passageiros estão neste momento no autocarro: **int a = excursao2** (a fica com 3).

7. A classe Arvore representa uma laranjeira, onde vai crescendo um conjunto de Laranjas. As laranjas nascem sempre com 100 gramas e cada um dos frutos tem um ID único, sempre crescente, que lhes é atribuído quando nascem. As frutas podem ir crescendo na árvore. De cada vez que uma Arvore ativa o método crescer(), o peso das suas laranjas aumenta 10%. As laranjas podem também cair da Arvore, assumindo-se que desaparecem (i.e., ao cair da árvore as laranjas morrem e desaparecem).

É dada uma implementação inicial (código mais abaixo). Esta implementação, relativamente ao cenário apresentado até agora, funciona sem erros, mas eventualmente falha na aplicação de alguns mecanismos da programação orientada a objetos. Além disso pretende-se expandir o cenário da seguinte forma: "As árvores do pomar que está a ser modelado foram todas enxertadas. A partir de agora devem permitir o crescimento de laranjas e limões. Deve continuar a existir um ID único crescente. Os limões nascem com 150 gramas e a ativação do crescimento faz com que o seu peso aumente 15%."

Neste exercício deve alterar o código, de forma a:

- Resolver as deficiências o código apresentado original possa ter logo face ao cenário inicial,
- Adicionar todo o código necessário para permitir que na árvore passem a coexistir laranjas e limões. Esta alteração deve obedecer a todas as boas práticas da programação orientada a objetos (Encapsulamento, Herança, Polimorfismo).

```
class Laranja {
    static int conta;
    int id;
    float peso;
public:
    Laranja() :id(conta++), peso(100) {}
    float getPeso() const { return peso; }
    int getID() const { return id; }
    void setPeso(int a) { peso = a; }
};
```

```
int Laranja::conta = 1;

class Arvore {
    vector<Laranja> fruta;
public:
    Arvore() = default;

    void nascer(Laranja x) { fruta.push_back(x); }

    void cair(int x) {
           for (int i = 0; i<fruta.size(); i++)
                  if (fruta[i].getID() == x) {
                         fruta.erase(fruta.begin() + i);
                         return;
                  }
    }

    void crescer() {
           for (int i = 0; i<fruta.size(); i++)
                  fruta[i].setPeso(fruta[i].getPeso()*1.1);
    }

    int getTotal() const { return fruta.size(); }
    Laranja getFruto(int i) const { return fruta[i]; }
};

// Mostra o id e peso de todos os frutos da Arvore
ostream& operator<<(ostream& out, Arvore& a) {
    for (int i = 0; i<a.getTotal(); i++)
           out << a.getFruto(i).getID() << " " << a.getFruto(i).getPeso() << endl;
    return out;
}
int main() {
    Arvore a;

    a.nascer(Laranja());
    a.nascer(Laranja());

    a.cair(1);

    a.crescer();
    cout << a << endl; // Nesta altura deve estar apenas 1 fruto na Arvore com Id 2 e peso 110
}
```

8. Pretende-se fazer um editor de texto capaz de lidar com documentos tipo MS-Word, que têm texto, imagens, tabelas, etc. Para isso é necessário propor um conjunto de classes.
   Um documento tem parágrafos de texto, figuras e tabelas. Tem um título e um autor. Deve ser possível iniciar um documento apenas mediante o titulo, autor e um conjunto inicial de elementos, que podem nenhuns, um, muitos…

   Os parágrafos, figuras e tabelas podem ser inseridos no documento a qualquer altura, ficando lá armazenados pela ordem com que são inseridos; têm um título, pelo qual podem ser pesquisados, encontrados e removidos; o título é também usado para comparar estes elementos entre si (==). Estes elementos têm também uma dimensão em pixéis (altura , largura). Sempre que os elementos são inseridos no documento, é verificado se cabem. Se não couberem não ficam inseridos. O texto e a tabela cabem sempre, mas a figura pergunta ao documento a sua largura (pixéis), e se for maior, diz que não cabe, não ficando inserida.

   O parágrafo é constituído por um conjunto de strings. A imagem é constituída por informação binária codificada como uma matriz dinâmica de inteiros, cujo conteúdo é indicado na sua inicialização. A tabela é constituída por um conjunto de inteiros, que também são indicados na sua inicialização.

   A informação dos elementos pertence a eles e não ao documento diretamente, e os documentos agem como na vida real: o conteúdo que está no documento é da responsabilidade desse documento: podem-se fazer fotocópias, mas cada documento tem os seus elementos, não afetando os outros.

   Apresente a declaração das classes que representam este cenário (ficheiros .h). Não deve implementar função nenhuma – apresente apenas os ficheiros .h. Não precisa de especificar *using namespaces* nem *includes* excepto se forem *includes* e outras declarações de cosias suas.

9. Complete as classes e também as zonas de código anotadas com “FAZER”. O “complete” refere-se a coisas em falta, são deduzidas pelos comentários e pela lógica do uso e da natureza das classes. Código a mais ou que faça coisas que já eram feitas será considerado um erro. Só consegue acertar se vir todo o código e todos os comentários antes de começar.

```
class Estudante {
   string nome;
   vector<string> mencoesHonrosas;
   vector<Disciplina *> inscritoA;
public:
   void adicionaDisciplina(Disciplina * d) {
      // FAZER: acrescentar lógica para não permitir disciplinas repetidas (mesmo nome)
      inscritoA.push_back(d);
   }
   ~Estudante() {
      // FAZER
   }
};

class Disciplina {
   string titulo, area;  // area científica
   vector<string *> sumarios;
public:
   void removeSumariosIguaisA(string sm) {  // reparar no plural
      // sabe-se que esta função remove o sumário removendo também os objectos string da memória
      // FAZER
   }
   void adicionaSumario(string sm) {
      // FAZER
   }
};

void mostra(Disciplina d) {
   /* ---> se quiser pode mudar esta função, mas apenas esta e apenas com coisas que façam
      Sentido e sejam necessárias */
   for (unsigned int i = d.sumarios.size(); i++)
      cout << d.sumarios[i] << "\n";
}
Disciplina trabalhaDisc() {
   Disciplina poo1(“POO”, “informatica para iniciantes”);
   Disciplina poo2(“POO 2”, “informatica”);
   // mete alguns sumários em poo1 (não é para fazer isso
   // é apenas uma indicação de que isso acontece aqui)
   poo2 = poo1;
   return poo2;
}

void prepara( /* FAZER */ ) { // recebe um estudante x
  // FAZER Criar disciplina f com "POO3" e "Informática avançada"
  // FAZER Adicionar disciplina f ao estudante x;
  // FAZER Estudante temp(“temporario”);
  // FAZER adiciona disciplina f ao estudante temp
  // se se mudar o nome da disciplina, ambos os estudantes x e temp vêm essa alteração
}

void main() {
   Estudante esta(“Jose”);
   Estudante estB = estA;
   prepara(& estA);
   // Estudante estA tem agora a disciplina "POO3"
}

// Se considerar que faltam coisas (sem “FALTAR” nenhum), então faça-as (se faltarem mesmo).
```