Programação Orientada a Objetos 2019/2020

Exame da Época Normal

DEIS – LEI / LEI-PL / LEI-CE

Duração da prova: 2 h

Número novo	**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. Qual será a saída resultante da execução deste programa?

```
class Jogador {
public:
    void treinar() { cout << "\n treinar "; }
    virtual void jogar() { cout << "\n jogar "; }
};
class Defesa : public Jogador {
public:
    void treinar() { cout << "\n treinar defesa "; }
    void jogar() override { cout << "\n jogar defesa "; }
};
class Atacante : public Jogador {
public:
    void treinar() { cout << "\n treinar ataque "; }
    void jogar() override { cout << "\n jogar ataque "; }
};
class PontaDeLanca : public Atacante {
};

int main() {
    vector<Jogador *> jogadores;
    jogadores.push_back(new Defesa);
    jogadores.push_back(new Atacante);
    jogadores.push_back(new PontaDeLanca);
    for (Jogador * j: jogadores) {
          j->treinar();
          j->jogar();
    }
}
```

A	**B**	**C**	**D**
treinar defesa	treinar defesa	treinar	treinar
jogar defesa	jogar	jogar defesa	jogar defesa
treinar ataque	treinar ataque	treinar	treinar
jogar ataque	jogar	jogar ataque	jogar ataque
treinar ataque	treinar ataque	treinar	
jogar ataque	jogar	jogar ataque	

2. Considere as seguintes definições:

```
class Ponto {
    int x, y;
public:
    Ponto(int x0 = 0, int y0 = 0) {
            x = x0;
            y = y0;
    }
};
class Figura {
    string nome;
public:
    Figura(const string & n) {
            nome = n;
    }
    void setNome(const string & n) {
            nome = n;
    }
};
class Circulo : public Figura {
    Ponto centro;
    const int raio;
public:
    // construtor ...

};
```

Quais das seguintes versões de construtor para a classe Circulo estão corretas?

versão 1

```
Circulo(int x0, int y0, int r) :
centro(x0, y0), raio(r), Figura("Circulo")
{}
```

versão 2

```
Circulo( int r) :
raio(r), Figura("Circulo") {}
```

versão 3

```
Circulo(int x0, int y0, int r) :
centro(x0, y0), Figura("Circulo") {
     raio = r;
}
```

versão 4

```
Circulo(int x0, int y0, int r) :
centro(x0, y0), raio(r){
     setNome("Circulo");
}
```

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

3. Considere as seguintes definições:

```
class SerVivo {
public:
    virtual void f() = 0;
    virtual void g() = 0;
};
class Animal : public SerVivo {
public:
    virtual void h() = 0;
    void f()override { /* ... */ }
};
class Ave : public Animal {
public:
    // . . .
};
```

Pretende-se que a classe Ave seja concreta, isto é, que possam existir objetos de Ave. Qual das seguintes afirmações é verdadeira?

A Se na classe Ave forem implementadas as funções g() e h(), a classe Ave fica concreta;

B Se na classe Ave forem implementadas as funções g() e h(), e não for implementada a função f(), a classe Ave fica abstrata;

C Se na classe Ave for implementada a função h(), a classe Ave fica concreta;

D Se na classe Ave forem implementadas as funções f() e h(), a classe Ave fica concreta;

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

```
class Erro {
    string msg;
public:
    Erro(const string & s) : msg(s) {}
    string what() {
           return "Erro " + msg;
    }
};
void funcao() {
    try {
           cout << "antes; ";
           throw Erro( " em funcao...");
           cout << "depois; ";
    }
    catch (Erro & e) {
           cout << e.what() << ";";
    }
    catch(string & e) {
           cout << "catch string; ";
    }
    cout << "fim funcao; ";
}

int main() {
    try {
           funcao();
    }
    catch (Erro & e) {
           cout << "catch erro main; ";
    }
    cout << "fim main; \n";
}
```

A antes; Erro em funcao...;fim funcao; fim main;

B antes; Erro em funcao...; `catch string;` fim funcao; fim main;

C antes; Erro em funcao...; fim main;

D antes; Erro em funcao...;fim funcao; `catch erro main;` fim main;

5. A classe FichaAnimal representa a ficha de um animal de um Zoo. A classe Gaiola tem informação acerca dos animais presentes na gaiola e das ocorrências (representadas por strings). A gaiola tem a posse exclusiva do registo das ocorrências. As fichas dos animais são da responsabilidade do Zoo e a gaiola apenas as utiliza para representar os animais aí presentes:

```
class Gaiola {
    vector<FichaAnimal *>animais;
    string *ocorrencias;
    int nOcorrencias;
public:
    //...
};
```

Quais das seguintes versões de destrutor para a classe Gaiola está correta?

A

```
~Gaiola() {
  for (FichaAnimal * a: animais){ delete a;}
  delete[] ocorrencias;
}
```

B

```
~Gaiola() {
  delete[] ocorrencias;
}
```

C

```
~Gaiola() {
  for (FichaAnimal * a: animais){ delete a;}
}
```

D

```
~Gaiola() {
}
```

6. A classe seguinte representa um bilhete de avião. Os bilhetes de avião, depois de emitidos permanecerão sempre na mesma companhia, mas podem ser modificados, nomeadamente colocar ou retirar malas (representadas por um ID que não deverá repetir-se no mesmo bilhete), e até mudar de passageiro (ser atribuído a outro).

```
class Bilhete {
   string passageiro;
   int passaporte;
   string & companhia;
   vector<int> id_malas;
};
```

Pode acrescentar coisas à classe, mas não retirar nem mudar o que já existe. Código desnecessário que faça coisas que já eram possíveis será considerado um erro.

Faça com que seja possível:

- Criar objectos indicando todos os dados inclusive um conjunto inicial de malas;
- Atribuir objectos;
- Mostrar o conteúdo no ecrã da forma (bilhete1 e bilhete2 são objetos da classe Bilhete):

```
cout << "\nPassageiro 1 : " << bilhete1 << "\nPassageiro 2" << bilhete2;
```

- Acrescentar bagagens ao bilhete (representadas pelos seus ID e sem repetir):

```
bilhete1 << 123 << 789 << 123;   // acrescenta a malas 123 e 789 e depois;
                                 // ignora a adição repetida de 123
```

- Remover todas as bagagens com id superior a um indicado):

```
(bilhete2 -= 40) -= 35;   // remove todas as malas do bilhete2;
                          // com ID ≥ 40, e depois as malas com ID ≥ 35
```

7. A classe Empresa armazena fichas de funcionários, que só podem ser ou doutores, ou engenheiros, e são representados pelas suas fichas de funcionário que são propriedade exclusiva da empresa. O programa seguinte funciona sem erros, mas falha na aplicação dos mecanismos da programação orientada a objetos (encapsulamento, herança e polimorfismo).
Escreva uma nova versão das classes envolvidas, de maneira que esses mecanismos estejam presentes.

```
class Doutor {
    string nome;
public:
    Doutor(const string & n):nome(n){}
    string getNome()const { return nome; }
};
class Engenheiro {
    string nome;
public:
    Engenheiro(const string & n) :nome(n){}
    string getNome()const { return nome; }
};

class Empresa {
    vector<Doutor> doutores;
    vector<Engenheiro> engenheiros;
public:
    Empresa() {
        doutores.push_back(Doutor("D1"));
        doutores.push_back(Doutor("D2"));
        engenheiros.push_back(Engenheiro("E1"));
        engenheiros.push_back(Engenheiro("E2"));
    }
    void cumprimentar() {
        for (auto & d : doutores) {
            cout << "Bom dia Doutor " << d.getNome() << endl;
        }
        for (auto & e : engenheiros) {
            cout << "Bom dia Engenheiro " << e.getNome() << endl;
```

```
            }
    }
    void removeDoutor(string nome) {
            // remove o doutor com esse nome do seu vector
    }
    void removeEngenheiro(string nome) {
            // remove o engenheiro com esse nome do seu vector
    }
};

int main() {
    Empresa empresa;
    empresa.cumprimentar();
}
```

8. O texto seguinte descreve um cenário que se pretende modelizar em C++. Se ao ler o texto lhe ocorrer "mas neste caso, faz o quê, concretamente", continue a ler que no fim ficará muito claro.

   Considere os conceitos de plantas e plantação. As plantas podem crescer, possuindo um mecanismo para simular esse crescimento. As plantas terão uma altura que vai de 0 até uma determinada altura máxima, sendo sempre expressa com uma percentagem em relação a essa altura máxima. As plantas podem ser de dois tipos: plantas hospedeiras e plantas parasitas.

   As plantas hospedeiras que são o objetivo principal da plantação, devem ser plantadas de maneira a que haja uma distância de separação mínima entre elas e outra planta qualquer, sendo esta distância uma característica da planta e não da plantação.

   Quando as plantas hospedeiras crescem (ou seja, o seu mecanismo de simulação de crescimento é ativado), avaliam a existência de outras plantas (hospedeiras ou não) que possam estar a uma distância inferior à sua distância de separação e prejudicar o seu crescimento.

   As plantas parasitas surgem associadas a uma outra planta hospedeira. Quando crescem retiram para si uma parte do crescimento da sua planta hospedeira, crescendo elas um tamanho que é função daquilo que retiraram à sua hospedeira.

   A plantação tem um conjunto de plantas. Estas plantas, tal como na natureza, estão fisicamente (no terreno) nessa plantação e apenas nela. A plantação tem um mecanismo que pode ser invocado pelo resto do programa para fazer crescer as plantas.

   Apresente a declaração das classes que representam este cenário (ficheiros .h). Não deve implementar função nenhuma – apresente apenas os ficheiros .h.

9. O código seguinte representa a informação acerca de uma coleção de moedas. Cada coleção tem as suas próprias características e as classes pretendem capturar essas características. As classes em questão não representam uma coleção completa: apenas a informação acerca de quantas moedas se tem de cada ano para uma determinada moeda em particular. Por exemplo, usando estas classes, o Manuel sabe que tem 3 moedas de 1 escudo de 1969, enquanto que o seu vizinho do lado tem 2 de 1971 de 1 escudo. São coleções diferentes. Mesmo as coleções fossem iguais, anda assim são coleções independentes e MoedaColecao tem a posse exclusiva de toda a sua informação (por exemplo: o vizinho de cima também tem 3 moedas de 1 escudo de 1969, mas apesar de semelhantes, são duas coleções independentes e cada um tem a sua).

```
struct AnoMoeda {   // para usar em MoedaColecao ou em outras coisas.
   int ano;        // ano da moeda
   int quantidade; // quantas, desse ano, dessa moeda
};

class MoedaColecao {  // representa conjunto de moedas de anos diferentes
                      // mas do mesmo "tipo" (valor facial e designação)
   float valor;       // valor facial (ex., 0.1 escudos )
   string designacao;    // designação da moeda (ex., "um tostão alumínio")
public:
   vector<AnoMoeda *> anos;  // quantas de cada ano nesta coleção.
};                           // (ex: 1971,3 , 1977,5 e 1975,1)
// a coleção de moedas teria vários objetos destes, um para cada "tipo" de
// moeda. Mas isso não entra neste enunciado
```

```
MoedaColecao adicionaMoedas(MoedaColecao inicial) {
   MoedaColecao aux;
   int ano;
   aux = inicial;
   while(1) {        // É favor não chamar a isto um ciclo "infinito"
      cin >> ano;    // porque tem uma condição de paragem mais adiante
      if (ano < 0)
         break;
      aux.adicionaAno(aux, ano);  // se o ano já estiver presente,
   }                              // incrementa o número desse ano
   return aux;
}
int main() {
   MoedaColecao tostoes(0.1, “tostao alumínio”, 1977);   // 1 moeda de 1977
   tostoes = adicionaMoedas(tostoes);
   for (int i=0; i<tostoes.anos.size(); i++)
      cout << "\nano: " << tostoes.anos[i].ano <<
              " quantas: " <<  tostoes.anos[i].quantidade";
   return 0;
}
```

Neste exercício há duas coisas distintas para fazer, mas que têm que ser feitas ambas e uma antes da seguinte. Considere estas duas coisas como uma única pergunta e responda-as pela ordem que são pedidas, identificando muito bem a que é que está a responder

- **Design** (acerca de coisas que já existem, mas não estarão bem feitas). Identifique eventuais pontos menos bem conseguidos no design da classe, explicando cada um com uma frase no máximo, e indicando a alteração a fazer rescrevendo as declarações do que estiver envolvido.
- **Funcionalidade** (acerca de coisas a mais, a menos, ou mal feitas). Complete a classe e faça com que o código das funções exteriores à classe funcione, tendo já em atenção as alterações que propôs no ponto anterior. Aqui não chega indicar protótipos: se houver código, tem mesmo que o fazer.