

**Algoritmos e Estruturas de Dados**

Mini-Teste 1: Programação em C++

GRUPO 1 (13.0 Val.)

1) Pretende-se implementar um programa para gestão do pessoal de uma empresa que possui vários departamentos. Para tal estão definidas duas classes: **CEmpresa** e **CDepartamento**.

```
class CDepartamento {
    vector <CFuncionario*> funcs;
    string nomeDep;
public:
    CDepartamento(string &nomeD);
    void adicionaFunc(CFuncionario &func1);
    string getNome();
    double gastoSalarios();
    CDepartamento operator + (const CDepartamento &dep2);
};
class CEmpresa {
    vector <CDepartamento> deps;
public:
    CEmpresa();
    CDepartamento &getDep(string &nomeD);
};
```

- 1.1) Construa a classe **CFuncionario**, que contém os atributos privados *nome* (string) e *salarioBase* (double). Implemente também o respectivo método construtor e o método abstracto *double getSalario()*. **(2.5 Val.)**
- 1.2) Na classe **CEmpresa**, implemente o método *getDep(string &nomeD)*, que retorna o departamento designado *nomeDep*. Este método deve lançar uma excepção do tipo **DepartamentoNaoExiste** se esse departamento não existir. Construa a classe correspondente à excepção. **(2.0 Val.)**
- 1.3) Na classe **CDepartamento**, implemente o operador +. A soma de dois departamentos é um outro departamento com o nome do primeiro, e os funcionários dos dois departamentos a somar. **(2.0 Val.)**
- 1.4) Suponha que os funcionários podem ser de dois tipos: do quadro e contratados. **(2.5 Val.)**
 - i. Construa a subclasse **CQuadro**. Os funcionários do quadro possuem ainda um atributo *nivel* (inteiro). O salário destes funcionários é igual a $salarioBase \times (1 + nivel/10)$. Implemente o método construtor e o método *getSalario()*.
 - ii. Construa a subclasse **CContratado**. Os funcionários contratados possuem ainda um atributo *taxaHorario* (double). O salário destes funcionários é igual a $salarioBase \times taxaHorario$. Implemente o método construtor e o método *getSalario()*.
- 1.5) Na classe **CDepartamento**, implemente o método *gastoSalarios()*, que retorna o valor a gastar no salário de todos os funcionários desse departamento. **(2.0 Val.)**
- 1.6) Escreva uma função de teste: `void actualizaEmpresa(CEmpresa &emp1)`, que: **(2.0 Val.)**
 - Adicione dois novos departamentos à empresa *emp1*: o de contabilidade e o de atendimento.
 - Adicione ao departamento de contabilidade o funcionário do quadro “Luis Silva”, de *salarioBase* 1500 e *nivel* 2.
 - Trate convenientemente as excepções possíveis.

Nome: _____

Turma: _____

GRUPO 2 (7.0 Val.)

Responda às seguintes questões, preenchendo a seguinte tabela com a opção correcta (em maiúsculas):

2.1	2.2	2.3	2.4	2.5	2.6	2.7

2.1) Suponha a seguinte definição de uma classe CString: **(1.0/-0.4 Val.)**

```
class CString {
    char *d_string;
public:
    CString();
    CString(const char *arg);
    ~CString();
    CString(CString const &other);
    CString const &operator=(const CString &rvalue);
    CString const &operator=(const char *rvalue);
};
```

- A) A classe CString possui dois construtores, um destrutor e três funções membro;
- B) A classe CString tem o operador & “overloaded”, permitindo juntar (concatenar) 2 strings numa só;
- C) A classe CString possui três construtores que permitem respectivamente: construir uma string vazia, construir uma string a partir de um array de caracteres e construir uma string a partir de outra string. Deste modo é possível fazer `CString st1(“xpto”)`;
- D) Definindo no programa principal uma variável do tipo CString chamada `st1`, é possível fazer `st1.d_string[n]='x'`, ou seja atribuir ao carácter `n` da string o valor `'x'`;
- E) Nenhuma das anteriores.

2.2) O método de ordenação por partição (Quick Sort): **(1.0/-0.4 Val.)**

- A) É sempre mais rápido do que a ordenação por inserção e quase sempre mais rápido do que a ordenação MergeSort;
- B) Escolhendo para Pivot sempre o maior valor do array, tem complexidade no tempo $O(n^2)$ e complexidade no espaço $O(n)$;
- C) No caso médio (valores aleatoriamente distribuídos) tem complexidade no tempo $O(n \cdot \log(n))$ e complexidade no espaço $O(n \cdot \log(n))$;
- D) No caso médio tem uma complexidade no espaço de ordem inferior à dos métodos de ordenação por inserção e ao Bubble Sort;
- E) Nenhuma das anteriores.

2.3) A função *repetidos* apresentada a seguir, determina se existe algum valor repetido num vector especificado como argumento: **(1.0/-0.4 Val.)**

```
template class T
public boolean repetidos(T v1[]) {
    ordena(v1);
    for ( int i=0 ; i<v1.size()-2 ; i++ )
        if ( v1[i]==v1[i+1] ) return true;
    return false;
}
```

Por forma a que o tempo de execução da função *repetidos* seja $O(n \cdot \log(n))$ e a complexidade espacial seja $O(1)$ o método *ordena* poderá ser:

- A) Ordenação por Partição;
- B) Ordenação por ShellSort;
- C) Ordenação por MergeSort;
- D) Ordenação por BubbleSort ou ordenação por Inserção;
- E) Nenhuma das Anteriores.

2.4) Considere o seguinte fragmento de código C++: **(1.0/-0.4 Val.)**

```
vector<int> v1; int n;
cin >> n;
for(int i=0; i<n; i++)
    for(int j=i, j<i+2 ; j++)
        for(int k=i+2, k>2 ; k = k/2)
            v1.push_back(i+j+k);
for(vector<int>::iterator it = v1.begin(); it!=v1.end(); it++)
    cout << *it << " " ;
```

- A) A complexidade temporal é $T(n)=O(n*\log(n))$;
- B) A complexidade temporal é $T(n)=O(n^2*\log(n))$;
- C) A complexidade temporal é $T(n)=O(n^2)$;
- D) A complexidade temporal é $T(n)=O(n^3)$;
- E) Nenhuma das anteriores.

2.5) Relativamente aos Construtores de classes em C++: **(1.0/-0.4 Val.)**

- A) Construtores são funções membro especiais chamadas pelo sistema no momento da criação de um objecto que possuem sempre um valor de retorno;
- B) Um construtor tem sempre o mesmo nome dos dados-membro da classe e não pode ser chamado directamente pelo utilizador desta;
- C) Podem ser criados vários construtores mas tem de existir sempre um destrutor por cada construtor criado;
- D) Tem de ser único pois é o responsável pela construção do objecto;
- E) Nenhuma das anteriores.

2.6) Considere o seguinte fragmento de código C++: **(1.0/-0.4 Val.)**

```
class CRectangle {
    float width, height;
public:
    void convert_square(CSquare a) {width=a.side; height=a.side;}
    void set_values (float a, float b) { width=a; height=b; }
};
class CSquare {
    float side;
public:
    void convert_rect(CRectangle a) {side=(a.width+a.height)/2.0 };
    friend class CRectangle;
};
```

- A) O programa está correcto e permite converter quadrados em rectângulos e rectângulos em quadrados;
- B) A função friend class não está correctamente declarada;
- C) A função convert_square não pode usar o membro-dado da classe CSquare side por isso está erradamente declarada;
- D) A função convert_rect não pode usar os membros-dado da classe CRectangle width e height por isso está erradamente declarada;
- E) Nenhuma das anteriores.

2.7) Considere a classe base *CPolygon* e a classe derivada *CRectangle* declaradas no seguinte fragmento de código: **(1.0/-0.4 Val.)**

```
class CPolygon {
    protected:
        int width, height;
    public:
        void set_values (int a, int b) { width=a; height=b; }
};
class CRectangle: public CPolygon {
    public:
        int area () { return (width * height); }
};
int main () {
    CRectangle rect;
    CPolygon * ppoly1 = &rect;
    ppoly1->set_values(4,5);
    cout << rect.area() << endl;
}
```

- A) O programa está correcto;
- B) Para o programa ficar correcto é necessário alterar a instrução: `ppoly1->set_values(4,5);` para `ppoly1->set_values(4,5);`;
- C) Para o programa ficar correcto é necessário alterar a instrução `ppoly1->set_values(4,5);` para `rect->set_values(4,5);`;
- D) Para o programa ficar correcto é necessário alterar a instrução `cout << rect.area() <<endl;` para `cout << ppoly1.area() <<endl;`;
- E) Nenhuma das anteriores.