Exercício Herança

Suponha que você está desenvolvendo um sistema para gerenciar funcionários de uma empresa. Cada funcionário tem um nome, um ID único e um salário base. Existem diferentes tipos de funcionários: Funcionário Regular: Possui apenas o salário base; Gerente: Além do salário base, recebe um bônus anual; Estagiário: Recebe um salário menor que o salário base.

Assim sendo, faça o que se pede:

Crie uma classe base chamada Funcionario com os seguintes atributos: nome (string) id (int) salarioBase (double).

Derive três classes especializadas a partir da classe Funcionario: FuncionarioRegular, Gerente e Estagiario.

Implemente os construtores e métodos necessários para cada classe:
FuncionarioRegular: Apenas o construtor padrão.
Gerente: Além do construtor padrão, inclua um atributo para o bônus anual.
Estagiario: Além do construtor padrão, defina um salário fixo menor que o salário base.

Crie um método calcularSalarioTotal() em cada classe que retorne o salário total (incluindo bônus, se aplicável). No programa principal (main()), crie alguns objetos de cada tipo de funcionário, atribua valores aos seus atributos e exiba o salário total de cada um.

Exercício de Polimorfismo

Suponha que você esteja desenvolvendo um sistema para uma empresa de transporte de cargas. A empresa possui diferentes tipos de veículos (caminhões, vans, carros etc.) para transportar mercadorias. Cada veículo tem suas próprias características e capacidades. Neste exercício, vamos nos concentrar em dois tipos de veículos: Caminhão e Van. Ambos os veículos herdam de uma classe base chamada Veiculo. A classe Veiculo possui os seguintes atributos e métodos:

```
class Veiculo {
protected:
    std::string marca;
    std::string modelo;
    int capacidade; // Capacidade de carga em toneladas

public:
    Veiculo(const std::string& marca, const std::string& modelo, int
capacidade)
        : marca(marca), modelo(modelo), capacidade(capacidade) {}

    virtual void exibirDados() const {
        std::cout << "Marca: " << marca << ", Modelo: " << modelo <<
", Capacidade: " << capacidade << " toneladas\n";
    }
};
```

A classe Veiculo possui um construtor que recebe a marca, o modelo e a capacidade de carga do veículo. O método exibirDados() exibe as informações básicas do veículo. Agora, vamos criar duas classes herdeiras: Caminhão e Van.

A classe Caminhão representa um caminhão de carga. Ele possui um atributo adicional chamado eixos (número de eixos do caminhão). Além disso, o método exibirDados() deve ser sobrescrito para incluir as informações específicas do caminhão.

```
class Caminhao : public Veiculo {
private:
    int eixos;

public:
    Caminhao(const std::string& marca, const std::string& modelo, int
capacidade, int eixos)
        : Veiculo(marca, modelo, capacidade), eixos(eixos) {}

    void exibirDados() const override {
        std::cout << "Caminhão - ";
        Veiculo::exibirDados();
        std::cout << "Eixos: " << eixos << "\n";
    }
};
```

A classe Van representa uma van de transporte. Ela possui um atributo adicional chamado passageiros (número máximo de passageiros que a van pode transportar). O método exibirDados() também deve ser sobrescrito para incluir as informações específicas da van.

```
class Van : public Veiculo {
private:
    int passageiros;

public:
    Van(const std::string& marca, const std::string& modelo, int
capacidade, int passageiros)
        : Veiculo(marca, modelo, capacidade), passageiros(passageiros)
{}

    void exibirDados() const override {
        std::cout << "Van - ";
        Veiculo::exibirDados();
        std::cout << "Passageiros: " << passageiros << "\n";
    }
};
```

Agora você pode criar objetos do tipo Caminhão e Van, definir suas características e exibir os dados específicos de cada veículo. Lembre-se de testar seu código no método main, com

exemplos para garantir que o polimorfismo esteja funcionando corretamente. Sugestão: tente colocar vários veículos em um mesmo vector e imprima os dados de cada tipo.

Desafio

Exercício 1.8. Grafo é uma estrutura de dados muito comum em computação, e os algoritmos sobre grafos são fundamentais para a área.

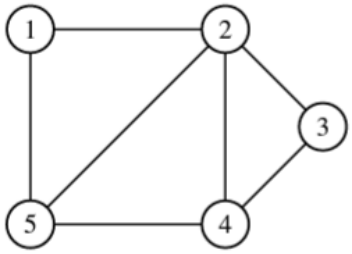
Um **grafo** $G = (V,A)$ consiste em:

- um conjunto finito de pontos V. Os elementos de V são chamados de **vértices** de G.
- um conjunto finito A de pares não ordenados de V, que são chamados de **arestas** de G. Uma aresta a em A é um par não ordenado (v,w) de vértices v, w em V, que são chamados de **extremidades** de a.

Uma aresta a em A é chamada de **incidente** com um vértice v em V, se v for uma extremidade de a.

Um vértice v em V diz-se **vizinho** de outro vértice w em V se existir uma aresta a em A incidente com v e w.

Um grafo pode ser representado por listas de adjacência ou por uma matriz de adjacência, como é ilustrado na figura 1.1.



(a) O grafo.

vértice	lista de adjacência
1	2, 5
2	1, 5
3	2, 4
4	2, 5, 3
5	4, 1, 2

(b) Listas de adjacência do grafo.

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(c) Matriz de adjacência do grafo.

Figura 1.1: Um exemplo de grafo.

Escreva uma classe para representar grafos. Escolha entre a representação por listas de adjacência ou por matriz de adjacência. A classe deve oferecer uma operação para determinar se dois vértices são vizinhos, e outra operação para determinar a lista de todos os vértices que são vizinhos de um dado vértice. Considere que cada vértice é representado por um número inteiro. Escreva um aplicativo para testar a classe.