Univiçosa – Centro Universitário de Viçosa
Cursos: Análise e Desenvolvimento de Sistemas e Engenharia de Computação
Disciplina: ADS101/ECO131 – Programação de Computadores 2

Aula 11 – Introdução ao polimorfismo – Parte 2

1) Introdução

Teremos nesta aula uma continuação de polimorfismo aplicado à linguagem C++.

2) Usando Herança e Polimorfismo em um código de abertura de conta em um banco

Neste exemplo um cliente vai fazer uma conta em um banco, sua conta pode ser conta corrente ou conta poupança, se ele optar por conta poupança, sobre o dinheiro que está em sua conta será acrescido uma correção percentual.

Observe no quadro abaixo o código

```
#include <iostream>
#include <string>

class ContaBancaria {
protected:
    std::string nomeCliente;
    double saldo;

public:
    ContaBancaria(const std::string& nome, double saldoInicial)
        : nomeCliente(nome), saldo(saldoInicial) {}

    virtual void exibirDetalhes() const {
        std::cout << "Cliente: " << nomeCliente << "\nSaldo: " << saldo << std::endl;
    }

    virtual void aplicarCorrecao() = 0; // método virtual puro
    virtual ~ContaBancaria() {} // destrutor virtual
};

class ContaCorrente : public ContaBancaria {
public:
    ContaCorrente(const std::string& nome, double saldoInicial)
        : ContaBancaria(nome, saldoInicial) {}

    void aplicarCorrecao() override {
        // A conta corrente não possui correção, então esse método não faz nada
        std::cout << "Conta corrente não possui correção.\n";
    }
```

```
    void exibirDetalhes() const override {
        std::cout << "Tipo de conta: Conta Corrente\n";
        ContaBancaria::exibirDetalhes();
    }
};

class ContaPoupanca : public ContaBancaria {
private:
    double taxaCorrecao; // porcentagem de correção aplicada ao saldo

public:
    ContaPoupanca(const std::string& nome, double saldoInicial, double taxa)
        : ContaBancaria(nome, saldoInicial), taxaCorrecao(taxa) {}

    void aplicarCorrecao() override {
        saldo += saldo * (taxaCorrecao / 100); // aplica a taxa de correção
    }

    void exibirDetalhes() const override {
        std::cout << "Tipo de conta: Conta Poupança\n";
        ContaBancaria::exibirDetalhes();
        std::cout << "Taxa de correção: " << taxaCorrecao << "%\n";
    }
};

int main() {
    ContaCorrente conta1("João", 1000.0);
    ContaPoupanca conta2("Maria", 2000.0, 2.0); // 2% de correção

    conta1.exibirDetalhes();
    conta1.aplicarCorrecao();

    std::cout << "\n";

    conta2.exibirDetalhes();
    conta2.aplicarCorrecao(); // aplica correção sobre o saldo
    conta2.exibirDetalhes(); // exibe o saldo atualizado após correção

    return 0;
}
```

Explicando o código

1. **Classe Base `ContaBancaria`**:
    - Armazena o nome do cliente e o saldo inicial;
    - Define um método virtual puro `aplicarCorrecao()` que será implementado nas classes derivadas para definir o comportamento específico de cada tipo de conta;

- `exibirDetalhes()` exibe o nome do cliente e o saldo da conta.

2. **Classe Derivada `ContaCorrente`**:
    - Herda de `ContaBancaria` e implementa o método `aplicarCorrecao()`, que neste caso não faz nada, pois conta corrente não possui correção;
    - Personaliza o método `exibirDetalhes()` para indicar que se trata de uma conta corrente.

3. **Classe Derivada `ContaPoupanca`**:
    - Herda de `ContaBancaria` e adiciona um atributo `taxaCorrecao` para a taxa de correção;
    - Implementa `aplicarCorrecao()` para adicionar ao saldo uma porcentagem definida por `taxaCorrecao;`
    - Personaliza o método `exibirDetalhes()` para mostrar a taxa de correção.

4. **Função `main`**:
    - Cria uma conta corrente para "João" e uma conta poupança para "Maria" com uma taxa de 2%;
    - Chama `aplicarCorrecao()` na conta poupança, aplicando a correção ao saldo;
    - Exibe os detalhes das contas para verificar as diferenças entre conta corrente e conta poupança.

Esse código exemplifica como herança e polimorfismo permitem adaptar o comportamento de métodos (como `aplicarCorrecao()`) para diferentes tipos de contas, mantendo uma interface comum.

3) Aprofundando no código

Muitos conceitos envolvem a programação orientada a objetos. Um exemplo que pode deixar um pouco confuso o programador é a seguinte linha do código:

```
ContaCorrente(const std::string& nome, double saldoInicial) : ContaBancaria(nome, saldoInicial) {}
```

O que estamos fazendo é chamar o construtor da classe base `ContaBancaria` para inicializar a parte da classe `ContaCorrente` que pertence a `ContaBancaria`.

- A sintaxe `: ContaBancaria(nome, saldoInicial)` logo após o cabeçalho do construtor de `ContaCorrente` indica que o construtor de `ContaBancaria` deve ser

chamado com `nome` e `saldoInicial` como argumentos antes de o corpo do construtor de `ContaCorrente` ser executado.

Como Funciona e Por Que Usar

Essa abordagem é usada especialmente com herança porque:

1. **Inicializa os membros da classe base**: Em uma classe derivada como `ContaCorrente`, os atributos herdados de `ContaBancaria` precisam ser inicializados pelo construtor da classe base, e a lista de inicialização é a forma recomendada de fazer isso;
2. **Melhor desempenho**: A lista de inicialização evita uma inicialização dupla. Se os membros fossem inicializados diretamente no corpo do construtor, eles seriam inicializados duas vezes (primeiro com um valor padrão e depois com o valor desejado). A lista de inicialização permite inicializar esses membros diretamente com o valor correto;
3. **Obrigatório para membros constantes e referências**: Caso existissem membros `const` ou referências em `ContaBancaria`, eles precisariam obrigatoriamente ser inicializados usando uma lista de inicialização, já que esses tipos de variáveis só podem ser inicializadas uma vez.

Fazendo um exemplo simples com este conceito. Observe o código abaixo.

```
class Base {
protected:
   int x;
public:
   Base(int valor) : x(valor) {}
};

class Derivada : public Base {
public:
   Derivada(int valor) : Base(valor) {}
};
```

Explicando o código.

Aqui, o construtor de `Derivada` chama o construtor de `Base` com o valor `valor`, inicializando `x` na classe base diretamente na construção.

4) Considerações da aula

Nesta aula aprofundamos mais no uso do polimorfismo. O caso da conta bancária é um assunto que podemos compreender pois está em nosso cotidiano.

Bons estudos.