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

1) Introdução

Polimorfismo é um conceito fundamental na programação orientada a objetos que permite que um objeto de uma classe derivada seja tratado como um objeto da classe base. Em C++, o polimorfismo é especialmente útil quando queremos que uma função ou método se comporte de maneiras diferentes com base no tipo do objeto que a chama. Existem dois tipos principais de polimorfismo em C++:

1. **Polimorfismo em tempo de compilação** (ou polimorfismo estático): é implementado por meio de sobrecarga de funções e operadores.
2. **Polimorfismo em tempo de execução** (ou polimorfismo dinâmico): é implementado usando herança e métodos virtuais.

2) Focando no polimorfismo dinâmico

O polimorfismo dinâmico ocorre quando uma função é chamada através de um ponteiro ou referência para a classe base, mas o método específico que é chamado é da classe derivada. Para ativar esse comportamento em C++, usamos a palavra-chave `virtual` na definição da função na classe base.

Exemplo de Polimorfismo Dinâmico

Imagine que temos uma classe base chamada `Animal` e duas classes derivadas, `Cachorro` e `Gato`. Cada uma dessas classes terá um método `fazerSom`, mas o som será específico de cada animal.

Observe no quadro abaixo o código

```
#include <iostream>
using namespace std;

class Animal {
public:
    // Método virtual para ativar o polimorfismo
    virtual void fazerSom() {
        cout << "Animal faz um som genérico." << endl;
    }
};

class Cachorro : public Animal {
public:
```

```
    void fazerSom() override { // Override para sobrescrever o método da classe base
        cout << "Cachorro faz: Au Au!" << endl;
    }
};

class Gato : public Animal {
public:
    void fazerSom() override {
        cout << "Gato faz: Miau!" << endl;
    }
};

int main() {
    Animal* animal1 = new Cachorro();
    Animal* animal2 = new Gato();

    animal1->fazerSom(); // Saída: "Cachorro faz: Au Au!"
    animal2->fazerSom(); // Saída: "Gato faz: Miau!"

    delete animal1;
    delete animal2;
    return 0;
}
```

Explicando o código

1. **Método Virtual**: Na classe `Animal`, declaramos `fazerSom` como `virtual`. Isso indica ao compilador que o método pode ser sobrescrito pelas classes derivadas.
2. **Sobrescrita do Método**: As classes `Cachorro` e `Gato` sobrescrevem o método `fazerSom` usando `override` para indicar que estão modificando o método da classe base.
3. **Ponteiros para Classe Base**: No `main`, criamos ponteiros do tipo `Animal` que apontam para objetos das classes derivadas `Cachorro` e `Gato`.
4. **Chamada do Método Polimórfico**: Quando chamamos `animal1->fazerSom()` e `animal2->fazerSom()`, o C++ escolhe automaticamente o método da classe derivada correta, mesmo que estamos usando ponteiros do tipo `Animal`.

Vantagens do Polimorfismo

- **Flexibilidade**: O código pode lidar com diferentes tipos de objetos de forma mais flexível.
- **Extensibilidade**: Adicionar novos tipos de objetos (por exemplo, uma nova classe `Pássaro` que herda de `Animal`) não requer modificações no código que usa polimorfismo.

Esse é um dos pilares da orientação a objetos que torna o código mais modular e expansível!

3) Aplicando o polimorfismo na compra de um carro

Tomemos um exemplo real: A compra de um automóvel em uma concessionária. Se o comprador for pessoa física em paga o valor total do carro, mas se ele for produtor rural ela terá um desconto em cima do valor do carro.

Observe o código.

```
#include <iostream>
#include <string>
using namespace std;

// Classe base Comprador
class Comprador {
protected:
    string nome;
    double valorCarro;

public:
    Comprador(const string& nome, double valorCarro) : nome(nome), valorCarro(valorCarro) {}

    // Método virtual para cálculo do preço
    virtual double calcularPreco() const {
        return valorCarro;
    }

    virtual void mostrarInfo() const {
        cout << "Nome: " << nome << endl;
        cout << "Preço final do carro: R$ " << calcularPreco() << endl;
    }
};

// Classe derivada para Pessoa Física
class PessoaFisica : public Comprador {
public:
    PessoaFisica(const string& nome, double valorCarro) : Comprador(nome, valorCarro) {}

    // Para Pessoa Física, o valor total é o valor do carro sem desconto
    double calcularPreco() const override {
        return valorCarro;
    }
};

// Classe derivada para Produtor Rural
class ProdutorRural : public Comprador {
```

```
private:
    double desconto; // Porcentagem de desconto

public:
    ProdutorRural(const string& nome, double valorCarro, double desconto)
        : Comprador(nome, valorCarro), desconto(desconto) {}

    // Para Produtor Rural, aplicamos um desconto sobre o valor do carro
    double calcularPreco() const override {
        return valorCarro * (1 - desconto / 100);
    }
};

int main() {
    // Criação dos objetos usando ponteiros para a classe base
    Comprador* comprador1 = new PessoaFisica("João Silva", 50000.0);
    Comprador* comprador2 = new ProdutorRural("Maria Souza", 50000.0, 15.0); // 15% de desconto

    // Exibindo informações e preços
    cout << "Comprador 1:" << endl;
    comprador1->mostrarInfo();

    cout << "\nComprador 2:" << endl;
    comprador2->mostrarInfo();

    // Liberando a memória
    delete comprador1;
    delete comprador2;

    return 0;
}
```

Explicação do Código

1. **Classe Base (`Comprador`)**: Contém o nome do comprador e o valor do carro. O método `calcularPreco` é `virtual` para que possa ser sobrescrito nas classes derivadas, permitindo o comportamento polimórfico. O método `mostrarInfo` exibe o nome e o preço final.
2. **Classe Derivada (`PessoaFisica`)**: Implementa `calcularPreco` para retornar o valor total do carro, pois uma pessoa física paga o preço cheio.
3. **Classe Derivada (`ProdutorRural`)**: Tem um atributo `desconto`, que armazena a porcentagem de desconto aplicável. O método `calcularPreco` aplica o desconto ao valor do carro e retorna o preço final com o desconto.

4. **Uso do Polimorfismo**: No `main`, criamos ponteiros para `Comprador`, mas cada ponteiro aponta para um objeto de uma classe derivada diferente (`PessoaFisica` e `ProdutorRural`). Quando chamamos `mostrarInfo`, o C++ usa o polimorfismo para escolher a implementação correta de `calcularPreco` com base no tipo do objeto apontado.

Saída Esperada

```
Comprador 1:
Nome: João Silva
Preço final do carro: R$ 50000

Comprador 2:
Nome: Maria Souza
Preço final do carro: R$ 42500
```

4) Recapitulando Referência “&” e Ponteiro “*”

& (Referência)

No código, & aparece em `const string& nome` no construtor das classes para definir um **parâmetro por referência**. Aqui está o significado detalhado:

- Em `const string& nome`, o & indica que `nome` é uma **referência** para o objeto `string` passado como argumento. Ou seja, em vez de criar uma cópia do argumento (o que é mais lento e consome mais memória), `nome` faz referência ao mesmo objeto original.
- O `const` garante que a referência seja **somente leitura** — você não pode modificar o valor de `nome` dentro da função. Isso é útil para economizar memória e tempo, especialmente ao passar grandes objetos.

* (Ponteiro)

Já o * é usado em `Comprador*` para criar um **ponteiro**. Vamos aos detalhes:

- Em `Comprador* comprador1 = new PessoaFisica(...);`, o * significa que `comprador1` é um **ponteiro para um objeto do tipo `Comprador`**.
- Com ponteiros, podemos fazer o **polimorfismo dinâmico** que vimos, pois `comprador1` (um ponteiro da classe base `Comprador`) pode apontar para objetos de classes derivadas (`PessoaFisica` e `ProdutorRural`).

- A partir do ponteiro `Comprador*`, podemos acessar métodos da classe base e das classes derivadas, dependendo do tipo real do objeto para o qual o ponteiro aponta.

Recapitulando

- `string&` passa uma **referência constante**, evitando cópias e preservando o valor original.
- `Comprador*` define um **ponteiro**, permitindo que `comprador1` e `comprador2` apontem para diferentes tipos de `Comprador` e viabilizando o uso de polimorfismo.

5) Considerações da aula

Nesta aula aprendemos como o uso do polimorfismo otimiza o programa. Um método trabalhando de várias maneiras conforme a necessidade do sistema é um recurso muito importante para não se usar vários métodos para situações similares.

Bons estudos.