# Programação Orientada a Objetos

# Herança em C++

Herança é um dos conceitos centrais da Programação Orientada a Objetos (POO). Ela permite a criação de novas classes (chamadas de classes derivadas ou subclasses) baseadas em classes existentes (conhecidas como classes base ou superclasses). A principal vantagem da herança é a reutilização de código e a capacidade de modelar relações hierárquicas de maneira natural.

Exploraremos a herança em C++ com profundidade, abordando tanto os aspectos conceituais quanto os práticos, com exemplos detalhados. Considerando que o leitor já está familiarizado com os conceitos de composição e agregação, faremos comparações quando apropriado para destacar as distinções e as melhores práticas no uso da herança.

## Herança vs. Composição

Muitas vezes, iniciantes na POO abusam da herança sem avaliar outras opções, como a **composição**. Antes de escolher a herança, considere o seguinte:

- **Herança** deve ser usada quando há uma **relação "é um"** entre as classes.
- **Composição** deve ser usada quando há uma **relação "tem um"**.

### Exemplos:

| Relação | Melhor escolha |
|---|---|
| **"Um cachorro é um animal"** | **Herança** |
| **"Um carro tem um motor"** | **Composição** |
| **"Um funcionário é uma pessoa"** | **Herança** |
| **"Um computador tem uma CPU"** | **Composição** |

### Exemplo de Herança (Relacionamento "é um"):

```
class animal {
public:
    void emitir_som() {
        cout << "O animal faz um som" << endl;
    }
};

class cachorro : public animal {
public:
    void latir() {
        cout << "O cachorro late: Au Au!" << endl;
    }
```

```
};

int main() {
    cachorro rex;
    rex.emitir_som();  // Herdado de animal
    rex.latir();      // Método próprio de cachorro
    return 0;
}
```

Exemplo de Composição (Relacionamento "tem um"):

```
class motor {
public:
    void ligar() {
        cout << "O motor está ligado" << endl;
    }
};

class carro {
private:
    motor motor; // O carro "tem um" motor
public:
    void ligar_carro() {
        motor.ligar(); // Chamando funcionalidade do motor
        cout << "O carro está funcionando" << endl;
    }
};

int main() {
    carro meu_carro;
    meu_carro.ligar_carro();
    return 0;
}
```

**Tipos de Composição em C++**

1. **Composição forte (Agregação por valor)**: O objeto agregado **é criado e destruído junto** com o objeto que o contém.
    - Exemplo: `motor` dentro de `carro` no código acima.
2. **Composição fraca (Agregação por ponteiro ou referência)**: O objeto agregado **é independente** e pode ser compartilhado por várias classes.
    - Exemplo:

```
class motor {
public:
    void ligar() { cout << "Motor ligado" << endl; }
```

```
};

class carro {
private:
    motor* _motor;
public:
    carro(motor* m) : _motor(m) {}
    void ligar_carro() {
        _motor->ligar();
    }
};

int main() {
    motor meu_motor;
    carro meu_carro(&meu_motor);
    meu_carro.ligar_carro();
    return 0;
}
```

---

# Construtores e Destrutores na Herança

## Construtores

Quando uma classe derivada é instanciada, os construtores da classe base são chamados automaticamente antes dos construtores da classe derivada. A ordem de chamada dos construtores segue a hierarquia de herança, começando pela classe base mais distante.

**Exemplo 1: Chamada Automática de Construtores**

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

class base {
public:
    base() {
        cout << "Construtor da Base" << endl;
    }
};

class derivada : public base {
public:
    derivada() {
        cout << "Construtor da Derivada" << endl;
    }
};

int main() {
    derivada obj; // Construtor da Base é chamado primeiro
    return 0;
}
```

**Saída:**

```
Construtor da Base
Construtor da Derivada
```

**Exemplo 2: Construtores com Parâmetros**

Se a classe base não tiver um construtor padrão (ou seja, um construtor sem parâmetros), a classe derivada deve explicitamente chamar um dos construtores da classe base.

```
class base {
public:
    base(int x) {
        cout << "Construtor da Base com x = " << x << endl;
    }
};

class derivada : public base {
public:
    derivada(int x, int y) : base(x) {
        cout << "Construtor da Derivada com y = " << y << endl;
    }
};

int main() {
    derivada obj(10, 20);
    return 0;
}
```

**Saída:**

```
Construtor da Base com x = 10
Construtor da Derivada com y = 20
```

## Destrutores

Os destrutores são chamados na ordem inversa dos construtores, começando pelo destrutor da classe derivada e depois pelos destrutores das classes base. Se a classe base tiver um destrutor virtual, a classe derivada também deve ter um destrutor virtual.

**Exemplo 3: Destrutores em Ação**

```
class base {
public:
    base() { cout << "Construtor da Base" << endl; }
    virtual ~base() { cout << "Destrutor da Base" << endl; }
};

class derivada : public base {
public:
    derivada() { cout << "Construtor da Derivada" << endl; }
    ~derivada() { cout << "Destrutor da Derivada" << endl; }
};

int main() {
    base* obj = new derivada(); // Alocação dinâmica
    delete obj; // Chama o destrutor da Derivada e da Base
    return 0;
}
```

**Saída:**

```
Construtor da Base
Construtor da Derivada
Destrutor da Derivada
Destrutor da Base
```

## Herança e Sobrecarga de Métodos

Quando uma classe derivada redefine um método da classe base, o método da classe base é ocultado na classe derivada. Isso pode ser controlado com a palavra-chave `override` para garantir que o método na classe derivada está substituindo corretamente o método da classe base.

```
class base {
public:
    virtual void mostrar() const {
        cout << "Mostrar na Base" << endl;
    }
};

class derivada : public base {
public:
    void mostrar() const override { // Sobrescreve o método da classe base
        cout << "Mostrar na Derivada" << endl;
    }
};

int main() {
    base* obj = new derivada();
    obj->mostrar(); // Saída: Mostrar na Derivada (polimorfismo)
```

```
    delete obj;
    return 0;
}
```

## Encapsulamento `protected` na Herança em C++

O encapsulamento é um dos princípios fundamentais da Programação Orientada a Objetos (POO), permitindo restringir o acesso aos membros de uma classe. Em herança, o modificador de acesso `protected` desempenha um papel importante, pois permite que **classes derivadas** acessem diretamente atributos e métodos protegidos da classe base, ao mesmo tempo que impede o acesso direto por código externo.

### Quando usar `protected`?

Em C++, temos três tipos principais de encapsulamento:

- `private`: O membro é acessível apenas dentro da própria classe.
- `protected`: O membro é acessível dentro da própria classe e por classes derivadas.
- `public`: O membro é acessível de qualquer lugar.

O uso de `protected` é recomendado quando:

- Queremos que um atributo ou método seja acessível **apenas** por classes derivadas.
- Precisamos expor informações ou funcionalidades sem torná-las públicas.
- Desejamos manter um maior controle sobre a manipulação dos dados dentro da hierarquia de classes.

### Exemplo: Sistema de Controle de Usuários

Vamos ilustrar a necessidade do encapsulamento `protected` com um **sistema de controle de usuários**, onde diferentes tipos de usuários têm diferentes permissões.

- A classe `usuario` contém um atributo `senha`, que não deve ser acessado externamente, mas precisa ser utilizado por classes derivadas.
- A classe `cliente` pode autenticar-se verificando sua própria senha.
- A classe `admin` pode alterar a senha de outros usuários.

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

// Classe base usuario
class usuario {
protected:
    std::string senha; // Protegido: acessível apenas pelas subclasses

public:
    usuario(std::string s) : senha(s) {}
    virtual ~usuario() = default;

    virtual void exibir_info() const {
        std::cout << "Usuário genérico." << std::endl;
    }
```

```
};

// Classe derivada cliente
class cliente : public usuario {
public:
    cliente(std::string s) : usuario(s) {}

    bool autenticar(std::string tentativa) const {
        return tentativa == senha; // Cliente pode acessar senha diretamente
    }

    void exibir_info() const override {
        std::cout << "Cliente do sistema." << std::endl;
    }
};

// Classe derivada admin
class admin : public usuario {
public:
    admin(std::string s) : usuario(s) {}

    void alterar_senha(usuario& u, std::string nova_senha) {
        u.senha = nova_senha; // Admin pode acessar senha diretamente
        std::cout << "Admin alterou a senha de um usuário." << std::endl;
    }

    void exibir_info() const override {
        std::cout << "Administrador do sistema." << std::endl;
    }
};

int main() {
    cliente c("senha123");
    admin a("admin2024");

    c.exibir_info();
    std::cout << "Autenticação com 'senha123': "
              << (c.autenticar("senha123") ? "Sucesso!" : "Falhou!") << std::endl;

    a.exibir_info();
    a.alterar_senha(c, "nova_senha");

    std::cout << "Autenticação com 'nova_senha': "
              << (c.autenticar("nova_senha") ? "Sucesso!" : "Falhou!") <<
std::endl;

    return 0;
}
```

## Análise do Código

1. **Uso de `protected`**:

- O atributo `senha` é `protected`, o que significa que **somente classes derivadas** (como `cliente` e `admin`) podem acessá-lo diretamente.
- Isso impede o acesso no `main()`, garantindo maior segurança.

2. **Hierarquia de acesso**:
   - `cliente` pode verificar sua própria senha, mas não pode alterá-la diretamente.
   - `admin` pode alterar a senha de qualquer `usuario` sem a necessidade de um método `public` intermediário.

3. **Evita exposição desnecessária**:
   - Se `senha` fosse `private`, `admin` não poderia alterá-la diretamente, exigindo um `set_senha()` na classe base, o que poderia expor o dado para qualquer classe externa.
   - Se fosse `public`, qualquer parte do código poderia modificar a senha, comprometendo a segurança do sistema.

### Considerações Finais

- O uso de `protected` é um **meio-termo entre** `private` **e** `public`, garantindo que informações sensíveis sejam protegidas do acesso externo, mas permitindo manipulação dentro da hierarquia de classes.
- Ele é particularmente útil para evitar a necessidade de getters/setters públicos quando queremos que **apenas classes derivadas** tenham permissão para acessar um dado.
- Esse princípio se aplica a diversos cenários da vida real, como controle de acesso em sistemas administrativos, proteção de recursos internos e implementação de restrições baseadas em funções dos usuários.

## Herança e Objetos Constantes

Em C++, se um objeto é declarado como `const`, isso significa que não pode ser modificado. Métodos que não modificam o estado do objeto devem ser declarados como `const`. A herança respeita essa restrição e a classe derivada deve seguir as mesmas regras.

```
class base {
public:
    virtual void mostrar() const { // Método const
        cout << "Mostrar na Base" << endl;
    }
};

class derivada : public base {
public:
    void mostrar() const override { // Também const
        cout << "Mostrar na Derivada" << endl;
    }
};

void exibir(const base& obj) {
    obj.mostrar(); // Pode ser chamado com objetos const
```

```
}

int main() {
    derivada obj;
    exibir(obj);
    return 0;
}
```

## Tipos de Herança em C++

Em C++, a herança pode ser especificada com diferentes níveis de acesso: `public`, `protected`, e `private`. Esses especificadores controlam como os membros da classe base são acessíveis na classe derivada.

1. **Herança Pública**:
    - Os membros `public` da classe base permanecem `public` na classe derivada.
    - Os membros `protected` da classe base permanecem `protected` na classe derivada.
    - Ideal para modelar uma relação "é-um".
2. **Herança Protegida**:
    - Os membros `public` e `protected` da classe base tornam-se `protected` na classe derivada.
    - Uso mais restrito, onde se deseja ocultar a interface pública da classe base da interface pública da classe derivada.
3. **Herança Privada**:
    - Todos os membros `public` e `protected` da classe base tornam-se `private` na classe derivada.
    - Usada para implementação interna, onde a herança é usada apenas como um mecanismo de reutilização de código.

### Herança Pública

A herança pública modela uma relação **"é um"** e mantém a visibilidade dos membros da classe base:

- **Membros `public` da base** → permanecem `public` na derivada.
- **Membros `protected` da base** → permanecem `protected` na derivada.
- **Membros `private` da base** → **não** são acessíveis na derivada.

```
class Base {
public:
    int publico;
protected:
    int protegido;
private:
    int privado;
};

class Derivada : public Base {
public:
    void exibir() {
```

```
        cout << publico;   // OK
        cout << protegido; // OK
        // cout << privado; // Erro! Não acessível
    }
};
```

## Herança Protegida

A herança protegida é menos comum, mas útil quando se deseja esconder a interface pública da classe base, preservando a possibilidade de subclasses acessarem esses membros.

```
class veiculo {
protected:
    int velocidade;
public:
    veiculo(int v) : velocidade(v) {}
};

class carro : protected veiculo {
public:
    carro(int v) : veiculo(v) {}
    void mostrar_velocidade() const {
        cout << "Velocidade: " << velocidade << endl;
    }
};

int main() {
    carro meu_carro(120);
    meu_carro.mostrar_velocidade(); // Acesso permitido via método da classe
derivada
    // meu_carro.velocidade = 100; // Erro: 'velocidade' é protegido
    return 0;
}
```

## Herança Privada

Na herança privada, a interface da classe base é completamente escondida da interface pública da classe derivada. A herança privada é usada principalmente para reutilização de código, onde a relação de herança não precisa ser visível externamente.

```
class pessoa {
public:
    void falar() const {
        cout << "A pessoa fala!" << endl;
    }
};

class estudante : private pessoa {
public:
```

```
    void estudar() const {
        falar(); // Acesso permitido apenas dentro da classe derivada
    }
};

int main() {
    estudante aluno;
    aluno.estudar(); // Funciona
    // aluno.falar(); // Erro: 'falar' é privado
    return 0;
}
```