Classes Abstratas, Interfaces, Polimorfismo e Herança em C++

Da motivação ao código: problema → diagrama → implementação

Beto

Programação Orientada a Objetos

17 de outubro de 2025

Roteiro

O que é `virtual`?

- `virtual` **não é um operador**, é uma **palavra-chave** da linguagem.
- Declara **métodos virtuais** em classes para habilitar **polimorfismo dinâmico**.
- A escolha da implementação ocorre **em tempo de execução**, com base no **tipo real** do objeto.

Sem virtual: ligação estática I

```
1  #include <iostream>
2  using namespace std;
3
4  class Base {
5  public:
6      void falar() { cout << "Base\n"; }
7  };
8
9  class Derivada : public Base {
10 public:
11     void falar() { cout << "Derivada\n"; }
12 };
13
14 int main() {
```

Sem virtual: ligação estática II

```
15      Base* p = new Derivada();
16      p->falar();  // imprime "Base" -> sem virtual = usa o tipo do ponteiro
17      delete p;
18  }
```

Com virtual: ligação dinâmica I

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

class Base {
public:
    virtual void falar() { cout << "Base\n"; }
};

class Derivada : public Base {
public:
    void falar() override { cout << "Derivada\n"; }
};

int main() {
```

Com virtual: ligação dinâmica II

```
15      Base* p = new Derivada();
16      p->falar();  // imprime "Derivada" -> despacho dinâmico (vtable)
17      delete p;
18  }
```

Como funciona por trás dos panos

- Objetos de classes com métodos virtuais possuem um **vptr** (ponteiro oculto).
- O **vptr** aponta para uma **vtable** (tabela virtual) contendo os endereços das funções.
- Ao chamar `p->falar()`, o runtime consulta a vtable do objeto **real** e chama a implementação correta.

Boas práticas e armadilhas

- **Use destrutor virtual** em classes base polimórficas.
- **Use** `override` nas derivadas; **use** `final` para selar métodos/classes.
- **Evite** chamar virtuais em construtores/destrutores.
- **Prefira composição** quando o polimorfismo dinâmico não for necessário.

Cenário: motor de renderização 2D

- Aplicação precisa **desenhar várias formas** (círculo, retângulo, triângulo, . . .).
- Requisitos: **exportar para JSON**, **calcular área/perímetro**, **mover/rotacionar**.
- Tentativa comum: **switch/case por tipo** + dados genéricos → manutenção difícil.

Design ingênuo (anti-exemplo) I

```
1  enum class TipoForma { Circulo, Retangulo };
2
3  struct DadosForma {
4    TipoForma tipo;
5    // dados "genericos"
6    double x, y, raio, largura, altura;
7  };
8
9  void desenhar(const DadosForma& f) {
10   switch (f.tipo) {
11     case TipoForma::Circulo:  /* ... desenhar circulo ... */ break;
12     case TipoForma::Retangulo:/* ... desenhar retangulo ... */ break;
13   }
14 }
```

Design ingênuo (anti-exemplo) II

```
15
16 // Problemas:
17 // - O switch se repete em desenhar(), area(), para_json()...
18 // - Adicionar nova forma exige editar varios trechos;
19 // - Forte acoplamento e risco de erros.
```

Objetivo de design

- Encapsular dados **e** operações por **tipo de objeto**.
- Permitir **extensão** (novas formas) sem quebrar o código existente.
- Habilitar **polimorfismo** via um **contrato comum**.

Definição

Classe Abstrata

Uma classe que **não pode ser instanciada** e define ao menos um **método virtual puro** (=0). Estabelece um **contrato** para as derivadas e viabiliza o **polimorfismo**.

Exemplo base: Forma I

```cpp
class Forma {
public:
  virtual ~Forma() noexcept = default; // destrutor virtual p/ deletar via
      ↪ ponteiro base

  // Contrato minimo da hierarquia:
  virtual double area() const = 0;    // virtual puro => Forma e abstrata
  virtual void desenhar() const = 0; // derivada precisa implementar

  // Operacao com implementacao padrao (opcional):
  virtual void mover(double dx, double dy) { (void)dx; (void)dy; }
};
```

Derivações concretas I

```
1  class Circulo final : public Forma { // 'final' impede heranca adicional
2    double x_, y_, raio_;
3  public:
4    Circulo(double x, double y, double raio) : x_(x), y_(y), raio_(raio) {}
5    double area() const override { return 3.141592653589793 * raio_ * raio_; }
6    void desenhar() const override { /* ... desenhar circulo ... */ }
7    void mover(double dx, double dy) override { x_ += dx; y_ += dy; }
8  };
9
10 class Retangulo final : public Forma {
11   double x_, y_, largura_, altura_;
12 public:
13   Retangulo(double x, double y, double largura, double altura)
14     : x_(x), y_(y), largura_(largura), altura_(altura) {}
```

Derivações concretas II

```
15    double area() const override { return largura_ * altura_; }
16    void desenhar() const override { /* ... desenhar retangulo ... */ }
17  };
```

Polimorfismo na prática I

```
1  #include <memory>
2  #include <vector>
3  #include <iostream>
4
5  int main() {
6    std::vector<std::unique_ptr<Forma>> formas;
7    formas.emplace_back(std::make_unique<Circulo>(0,0,10));
8    formas.emplace_back(std::make_unique<Retangulo>(0,0,20,5));
9
10   double area_total = 0.0;
11   for (const auto& f : formas) {
12     f->desenhar();            // despacho dinamico (tabela virtual)
13     area_total += f->area();  // idem
14   }
```

Polimorfismo na prática II

```
15    std::cout << "Area total = " << area_total << "\n";
16 }
```

Existe "interface" em C++?

- C++ **não** possui a palavra-chave `interface`.
- Em C++, uma **interface** é uma **classe abstrata pura**:
  - apenas métodos **virtuais puros** (=0);
  - **sem dados** (ou mínimo possível);
  - **destrutor virtual** (mesmo que vazio).

Exemplos de interfaces I

```
1 struct Desenhavel {
2   virtual ~Desenhavel() = default; // sempre virtual em interfaces
3   virtual void desenhar() const = 0;
4 };
5
6 struct Serializavel {
7   virtual ~Serializavel() = default;
8   virtual std::string para_json() const = 0;
9 };
```

Combinando classe abstrata + interfaces I

```
1  class Circulo2 : public Forma, public Desenhavel, public Serializavel {
2    double x_, y_, raio_;
3  public:
4    Circulo2(double x, double y, double raio) : x_(x), y_(y), raio_(raio) {}
5
6    // Forma
7    double area() const override { return 3.141592653589793 * raio_ * raio_; }
8    void desenhar() const override { /* ... desenhar circulo ... */ }
9
10   // Serializavel
11   std::string para_json() const override {
12     return "{ \"tipo\":\"circulo\", \"x\":"+std::to_string(x_)
13          +", \"y\":"+std::to_string(y_)
14          +", \"raio\":"+std::to_string(raio_)+" }";
```

Combinando classe abstrata + interfaces II

```
15      }
16  };
```

Tipos de polimorfismo em C++

- **Polimorfismo dinâmico** (em tempo de execução): via `virtual` + `override`.
- **Polimorfismo estático** (em tempo de compilação): via **templates** e sobrecarga de funções.
- Neste material o foco é o **dinâmico** (hierarquias com classes base/derivadas).

Exemplo de uso polimórfico I

```
1 void desenhar_todas(const std::vector<std::unique_ptr<Forma>>& fs) {
2   for (const auto& f : fs) {
3     f->desenhar(); // chamada virtual -> usa a implementacao concreta
4   }
5 }
```

Boas práticas de polimorfismo

- **Base polimórfica** deve ter **destrutor virtual**.
- Sempre marcar sobrescritas com `override`.
- Usar **composição** quando a relação for **tem-um**.
- Evitar **dados** em interfaces (classes puramente abstratas).

Herança: visibilidade e diretrizes

- **public**: a interface pública da base permanece pública.
- **protected**: a interface pública da base vira protegida (casos raros).
- **private**: oculta a interface da base (geralmente prefira **composição**).
- Use **herança pública** para relação **é-um**; prefira **composição** para **tem-um**.

virtual, override e final

```cpp
struct Base {
  virtual ~Base() = default;
  virtual void f() { /* ... implementacao padrao ... */ } // virtual =>
      ↪ despacho dinamico
};

struct Derivada : Base {
  void f() override { /* substitui com checagem de assinatura */ }
};

struct Selada final : Base {       // 'final' em classe: ninguem pode herdar
  void f() final override { /* ... */ } // 'final' em metodo: impede
      ↪ sobrescrita
};
```

virtual, override e final II

```
13
14 // struct Proibida : Selada {}; // ERRO: classe Selada e 'final'
```

Problema do diamante e herança virtual I

```
1  struct Dispositivo { int id = 0; };
2
3  struct Impressora : /*virtual*/ public
       ↪ Dispositivo {};
4  struct Scanner    : /*virtual*/ public
       ↪ Dispositivo {};
5
6  // Sem 'virtual', Dispositivo e duplicado:
7  struct Multifuncional : public Impressora,
       ↪ public Scanner {
8    void definir_id(int v) {
9      // Dispositivo::id e ambiguo:
10     // id = v; // ERRO
11     Impressora::Dispositivo::id = v; //
```

```
1  // Com heranca virtual:
2  struct Dispositivo { int id = 0;
       ↪ };
3
4  struct Impressora : virtual
       ↪ public Dispositivo {};
5  struct Scanner    : virtual
       ↪ public Dispositivo {};
6
7  struct Multifuncional : public
       ↪ Impressora, public
       ↪ Scanner {
8    void definir_id(int v) { id =
       ↪ v; } // apenas 1
```

Diagrama simples (hierarquia de formas)

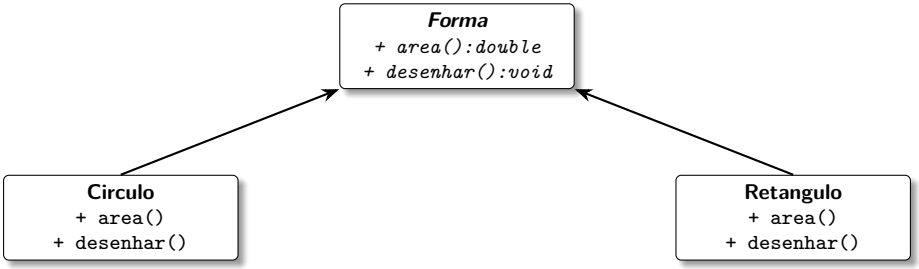
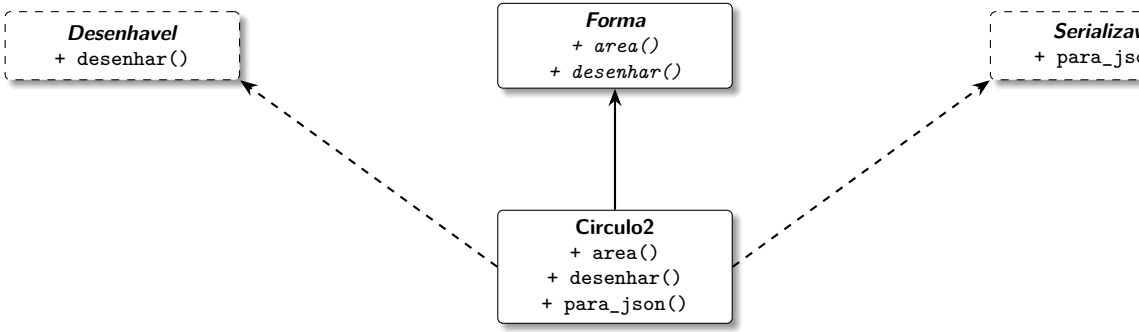


Diagrama com interfaces



Conceitos úteis

- **Covariância de retorno**: derivada pode retornar tipo mais específico em método virtual.
- **Upcasting/Downcasting**: `dynamic_cast` para downcast seguro.
- **=default** e **=delete**: controle de operações especiais em hierarquias.
- **ISP/LSP/OCP**: princípios de design aplicáveis a hierarquias.

Covariância de retorno I

```
1  struct Forma {
2    virtual ~Forma() = default;
3    virtual Forma* clonar() const = 0;
4  };
5
6  struct Circulo : Forma {
7    Circulo* clonar() const override { // retorno covariante
8      return new Circulo(*this);
9    }
10 };
```

Upcasting, downcasting e dynamic_cast I

```
1 Forma* p = new Circulo(0,0,10); // upcast implicito (seguro)
2
3 if (auto pc = dynamic_cast<Circulo*>(p)) { // downcast seguro
4   /* usa Circulo */
5 } else {
6   /* nao era Circulo */
7 }
8
9 // Evite 'static_cast' para downcast em hierarquias (sem checagem em runtime).
```

=default e =delete em hierarquias I

```
1 struct BasePoli {
2   BasePoli() = default;
3   virtual ~BasePoli() = default;        // virtual por ser base polimorfica
4   BasePoli(const BasePoli&) = delete;   // proibe copia
5   BasePoli& operator=(const BasePoli&) = delete; // proibe atribuicao
6 };
```

Exercício de modelagem

Projete uma hierarquia para um **Editor de Formas** que:

1. Forneça uma base `Forma` com `area()` e `desenhar()`.
2. Separe os comportamentos `Desenhavel` e `Serializavel`.
3. Implemente ao menos 3 formas concretas (por exemplo: círculo, retângulo, triângulo).
4. Use `std::vector<std::unique_ptr<Forma>>` e demonstre polimorfismo.
5. Opcional: inclua **herança virtual** se houver múltipla herança real.

Resumo

- **Virtual** habilita despacho dinâmico e é a base do polimorfismo em C++.
- **Classes abstratas** definem contratos via métodos **virtuais puros**.
- **Interfaces** em C++ são **classes puramente abstratas** (métodos =0).
- **Herança** deve ser usada com critério; prefira **composição** quando for **tem-um**.
- **Boas práticas**: destrutor virtual, `override`/`final`, ISP/LSP/OCP, `dynamic_cast` quando necessário.

Dúvidas?