

Herança e Polimorfismo

Gonzalo Travieso

2019

1 Relações entre classes

Ao desenvolvermos nossos programas, muitas vezes surge a necessidade de diversas classes que são relacionadas entre si.

A forma mais simples de relacionamento, que já utilizamos frequentemente, é quando definimos uma nova classe através da junção de membros que são objetos de outras classes. Por exemplo, suponhamos que estamos guardando algumas informações sobre pessoas:

```
class Pessoa {
    std::string _nome;
    std::string _endereco;
    // ... Outros
};
```

Neste caso, estamos criando uma nova classe `Pessoa` cujos objetos têm membros do tipo `std::string`, que também é uma classe.

Este processo é conhecido como **composição**, visto que um objeto de uma classe é composto de objetos de outras classes.

Uma outra forma de relação entre classes é a **herança**, que iremos estudar agora.

2 Apresentação do problema

Suponhamos que em um programa temos uma classe para representar reservatórios de algum material cujo conteúdo pode ser representado por um número de ponto flutuante. Por exemplo, reservatório de combustível, que pode ser representado pelo número de litros contidos no reservatório.

```
// Classe para representar um reservatorio
class Reservoir {
    double _content; // Quanto tem guardado.
public:
    // Construtor: Inicializa o reservatório.
    // Garante que quantidade >= 0.
    Reservoir(double initial = 0) { _content = initial > 0 ? initial : 0; }

    // Retorna o quanto o reservatorio tem no momento
    double content() const { return _content; }
```

```

// Insere conteudo no reservatorio.
// So adiciona se for positivo.
void put(double additional);

// Retira um tanto do reservatorio, se existir.
// Se nao existir, retira o que tem.
// So retira se how_many positivo.
// Retorna o quanto retirou.
double take(double how_many);
};

// Insere conteudo no reservatorio.
// So adiciona se for positivo.
void Reservoir::put(double additional) {
    if (additional > 0) {
        _content += additional;
    }
}

// Retira um tanto do reservatorio, se existir.
// Se nao existir, retira o que tem.
// So retira se how_many positivo.
// Retorna o quanto retirou.
double Reservoir::take(double how_many) {
    double taken{0}; // Quanto foi tirado do reservatório
    if (how_many > 0) {
        if (_content >= how_many) {
            _content -= how_many;
            taken = how_many;
        } else {
            taken = _content;
            _content = 0;
        }
    }
    return taken;
}

```

Como vemos, esses reservatórios têm uma capacidade de armazenamento ilimitada. Agora vamos supor que, em certa parte do código, precisamos lidar com reservatórios que têm uma capacidade máxima. Como podemos lidar com isso?

2.1 Lidando com a alteração nos pontos de uso

Uma opção seria lidar com a diferença nos locais de uso:

```

Reservoir ilimitado, limitado;
double const limite = 10;
// ...
auto new_content = get_more_content();

```

```

ilimitado.put(new_content);
// Para o limitado precisamos ser cuidadosos:
if (new_content + limitado.content() <= limite) {
    // Cabe, então simplesmente coloca.
    limitado.put(new_content);
}
else {
    // Não cabe, então coloca o tanto que cabe.
    limitado.put(limite - limitado.content());
}

```

O problema desta solução é que teremos que fazer códigos desse tipo em todos os pontos onde formos inserir num reservatório limitado, sempre lembrando qual reservatório é limitado ou não. Essa solução não é satisfatória, pois na verdade, o fato do reservatório ser limitado é uma característica dele, que deveria estar inserida em sua implementação, e não no código que o utiliza.

Precisamos então de uma nova classe para representar reservatórios limitados, que tenha a limitação embutida em seus métodos.

2.2 O método do *cortar-e-colar*

Podemos criar a nova classe com base na existente, copiando o código anterior e fazendo as alterações necessárias:

```

// Classe para representar um reservatorio
class LimitedReservoir {
    double _content; // Quanto tem guardado.
    double _limit;

public:
    // Construtor: Inicializa o reservatorio.
    // Garante que quantidade >= 0 e que nao estora limite.
    LimitedReservoir(double limit, double initial = 0) {
        _content = initial > 0 ? initial : 0;
        _limit = limit > 0 ? limit : _content;
        if (_content > _limit) {
            _content = _limit;
        }
    }

    // Retorna o quanto o reservatorio tem no momento
    double content() const { return _content; }

    // Insere conteudo no reservatorio, garantindo que nao estora limite.
    // So adiciona se for positivo.
    void put(double additional);

    // Retira um tanto do reservatorio, se existir.
    // Se nao existir, retira o que tem.
    // So retira se how_many positivo.
    // Retorna o quanto retirou.

```

```

    double take(double how_many);
};

// Insere conteudo no reservatorio.
// So adiciona se for positivo.
void LimitedReservoir::put(double additional) {
    if (additional > 0) {
        if (_content + additional <= _limit) {
            _content += additional;
        } else {
            _content = _limit;
        }
    }
}

// Retira um tanto do reservatorio, se existir.
// Se nao existir, retira o que tem.
// So retira se how_many positivo.
// Retorna o quanto retirou.
double LimitedReservoir::take(double how_many) {
    double taken{0}; // Quanto foi tirado do reservatório
    if (how_many > 0) {
        if (_content >= how_many) {
            _content -= how_many;
            taken = how_many;
        } else {
            taken = _content;
            _content = 0;
        }
    }
    return taken;
}

```

Isto funciona, mas temos um problema óbvio: temos uma grande duplicação de código entre as classe `Reservoir` e `LimitedReservoir`. Se for necessário corrigir algum erro ou melhorar de outra forma a implementação, isso precisa ser feito no código das duas classes. Vamos buscar uma outra solução.

2.3 Usando composição

Uma solução mais apropriada é usar composição, usando um `Reservoir` interno ao `LimitedReservoir`, para que ele faça a implementação das operações comuns do reservatório, e apenas nos preocupando em implementar o limite de capacidade:

```

// Classe para representar um reservatorio
class LimitedReservoir {
    Reservoir _reservoir;
    double _limit;

public:

```

```

// Construtor: Inicializa o reservatorio.
// Garante que quantidade >= 0 e que nao excede limite.
LimitedReservoir(double limit, double initial = 0) : _reservoir(initial) {
    _limit = limit > 0 ? limit : _reservoir.content();
    if (_reservoir.content() > _limit) {
        _reservoir.take(_reservoir.content() - _limit);
    }
}

// Retorna o quanto o reservatorio tem no momento
double content() const { return _reservoir.content(); }

// Insere conteudo no reservatorio sem exceder limite.
// So adiciona se for positivo.
void put(double additional);

// Retira um tanto do reservatorio, se existir.
// Se nao existir, retira o que tem.
// So retira se how_many positivo.
// Retorna o quanto retirou.
double take(double how_many);
};

// Insere conteudo no reservatorio.
// So adiciona se for positivo.
void LimitedReservoir::put(double additional) {
    if (additional > 0) {
        if (_reservoir.content() + additional <= _limit) {
            _reservoir.put(additional);
        } else {
            _reservoir.put(_limit - _reservoir.content());
        }
    }
}

// Retira um tanto do reservatorio, se existir.
// Se nao existir, retira o que tem.
// So retira se how_many positivo.
// Retorna o quanto retirou.
double LimitedReservoir::take(double how_many) {
    return _reservoir.take(how_many);
}

```

Neste caso, o código que cuida da parte geral de reservatório está concentrado na classe `Reservoir`, e se fizermos alguma alteração nele (ajuste de erro ou melhora de desempenho, por exemplo), isso já vai se refletir na classe `LimitedReservoir`. Esta última só precisa ser alterada se quisermos mexer com a parte de limitação de capacidade.

3 Herança

Apesar de uma solução usando composição ser adequada para diversos propósitos, ela ainda tem algumas limitações, relacionadas com o fato de que, apesar das classes `Reservoir` e `LimitedReservoir` serem claramente relacionadas uma com a outra (objetos dessas duas classes são quase idênticos, apenas com diferenças no comportamento do método `put`), isso não é indicado em nenhuma parte do código. De fato, para o compilador C++, as duas classes não têm nenhuma relação, visto que a presença de um `Reservoir` como membro de um `LimitedReservoir` é apenas um detalhe de implementação.

Na verdade aqui temos um caso em que um objeto da classe `LimitedReservoir` tem comportamento similar ao da classe `Reservoir`, e portanto podemos dizer que um reservatório limitado **é um** reservatório, isto é, um reservatório limitado é um caso especial de reservatório.

Isto pode ser indicado explicitamente no código através do uso de herança, definindo `LimitedReservoir` da seguinte forma:

```
// Classe para representar um reservatorio
class LimitedReservoir : public Reservoir {
    double _limit;

public:
    // Construtor: Inicializa o reservatorio.
    // Garante que quantidade >= 0.
    LimitedReservoir(double limit, double initial = 0) : Reservoir(initial) {
        _limit = limit > 0 ? limit : content();
        if (content() > _limit) {
            take(content() - _limit);
        }
    }

    // Insere conteudo no reservatorio.
    // So adiciona se for positivo.
    void put(double additional);
};

// Insere conteudo no reservatorio.
// So adiciona se for positivo.
void LimitedReservoir::put(double additional) {
    if (additional > 0) {
        if (content() + additional <= _limit) {
            Reservoir::put(additional);
        } else {
            Reservoir::put(_limit - content());
        }
    }
}
```

O ponto essencial nesse código é a linha:

```
class LimitedReservoir : public Reservoir
```

Isto indica que estamos declarando a classe `LimitedReservoir` como uma **classe derivada** de `Reservoir`. Também dizemos que `Reservoir` é **classe base** de `LimitedReservoir`.

O efeito disso é que a classe `LimitedReservoir` **herda** tudo o que existe na classe `Reservoir`, isto é, ela possui **os mesmos membros** de dados ou métodos. Com isso, precisamos implementar explicitamente em `LimitedReservoir` apenas o que é diferente de `Reservoir`. Assim, os métodos `content()` e `take(double)` não precisam ser implementados em `LimitedReservoir`; já o método `put(double)` precisa ser reimplementado, como feito no código acima, para evitar exceder a capacidade. O construtor também precisa ser implementado, pois ele necessita um parâmetro adicional (a capacidade). Note como no construtor usamos a *lista de inicialização* de membros para *passar parâmetros para o construtor da classe base*.

3.1 Hierarquias de herança

Herança não precisa ser apenas em um nível, como no exemplo anterior, mas pode se constituir de diversas classes, formando o que é conhecido como um *hierarquia de classes*. Por exemplo:

```
class A {};  
class B : public A {};  
class C : public B {};  
class D : public A {};  
class E : public D {};
```

Neste caso temos `A` como a base da hierarquia, com `B` e `D` herdando diretamente de `A`, enquanto `C` herda de `B` e `E` herda de `D`. Nestes casos, dizemos que `C` e `E` também são classes derivadas de `A`.

4 Polimorfismo

A principal vantagem de usar herança é indicar ao compilador que os dois tipos são relacionados. Isso faz com que exista compatibilidade entre esses tipos do ponto de vista do compilador em certas situações.

A mais importante delas é permitir lidar com todos os tipos que são derivados de uma mesma classe base. Por exemplo, todos os tipos derivados de `Reservoir` terão certamente os métodos `content()`, `put(double)` e `take(double)`. Isso significa que em princípio podemos implementar um código (por exemplo, uma função), que realiza essas operações e que portanto funcionará para todas as classes derivadas de `Reservoir`.

Para que isso funcione, precisamos de dois elementos:

- A possibilidade de se referir a objetos de qualquer classe derivada de uma mesma base a partir do mesmo local no código.
- A capacidade do código de se adaptar, chamando os métodos apropriados de acordo com o tipo do objeto.

O primeiro elemento é conseguido por compatibilidade de referências e ponteiros, o segundo por métodos virtuais.

4.1 Compatibilidade de referências e ponteiros

Em C++, se temos um ponteiro para uma classe base, podemos apontar com ele para um objeto tanto dessa mesma classe como de qualquer classe derivada dela:

```
Reservoir *pr1 = new Reservoir; // OK
Reservoir *pr2 = new LimitedReservoir(10); // OK: Classe derivada
pr1->put(20);
pr2->put(20);
```

O contrário não é permitido:

```
LimitedReservoir *pl1 = new LimitedReservoir(25); // OK
LimitedReservoir *pl2 = new Reservoir; // ERRO: tipos incompatíveis
```

O mesmo acontece com referências: uma referência para classe base pode ser um sinônimo para um objeto de qualquer classe derivada dela:

```
// Atua em r passado por referência.
void by_reference(Reservoir &r, double x) {
    r.put(x);
}
// ...
```

```
Reservoir r1;
LimitedReservoir r2(100);
by_reference(r1, 200);
by_reference(r2, 200);
```

4.2 Métodos virtuais

Entretanto, os códigos acima ainda não funcionam como o esperado. Certamente, esperamos que, ao chamar `put(double)` sobre um objeto do tipo `LimitedReservoir` ele chame o método correspondente da classe, que garante que a capacidade não é excedida. Mas no código que apresentamos até agora isso não está ocorrendo, pois ainda precisamos indicar que o método `put(double)` é um **método virtual**.

Para marcar um método como virtual, precisamos colocar a palavra-chave `virtual` na sua declaração **na classe base**. No nosso exemplo:

```
class Reservoir {
    // Igual a antes...
    virtual void put(double additional);
    // O resto igual...
};
```

Com isso, o código apresentado terá o comportamento esperado, chamando `put(double)` de `Reservoir` ou de `LimitedReservoir`, dependendo do tipo do objeto referenciado pelo ponteiro ou pela referência. Repare que o `virtual` precisa apenas ser colocado na classe base. Já nas classes derivadas que dão uma nova implementação para o método, os métodos devem ser marcados com a palavra-chave `override`, como no exemplo:


```
class LimitedReservoir : public Reservoir {
    // Idem a antes...
    void put(double amount) override;
};
```

Isto é feito para deixar claro que esta classe derivada precisa de um comportamento diferente para este método.

Se **não** usamos `virtual` na declaração de um método, então o compilador decide qual o método a chamar com base no tipo do ponteiro ou da referência.

Para completar, temos a seguinte regra (que ficará clara mais adiante):

Se uma classe possui um método virtual, então ela deve também declarar um destruidor virtual.

Portanto, devemos acrescentar em `Reservoir`:

```
class Reservoir {
    // Igual a antes...
    virtual ~Reservoir() {};
};
```

Neste caso, o destruidor não precisa fazer nada.

Os métodos virtuais são o que permite em C++ o **polimorfismo**, que consiste em uma mesma chamada no código do programa poder levar à execução de diferentes métodos, de acordo com o tipo do objeto.

Normalmente, quando estamos lidando com herança, iremos querer operar com polimorfismo, portanto os métodos de classes que são base de outras classes geralmente são declarados `virtual`.

Duas vantagens de polimorfismo são:

- Permitir escrever códigos que funcionam para diversos tipos de objetos, e não apenas para um tipo. Basta escrever o código para os métodos da classe base e usar métodos virtuais, e então esse código se adaptará a qualquer classe derivada. Um exemplo trivial é a função `by_reference` apresentada acima.
- Permitir lidar com coleções mistas de objetos, isto é, coleções que têm objetos de diversos tipos, operando nos objetos dessa coleção através dos métodos virtuais. Para isso, as coleções devem ser de ponteiros para uma classe base de todos os objetos a serem incluídos.

No nosso exemplo, podemos fazer:

```
// Um vetor de ponteiros para classe base.
std::vector<std::unique_ptr<Reservoir>> v;
// Um objeto da classe derivada
LimitedReservoir r1{20, 20};
// No vetor de ponteiros para classe base
// podemos também guardar ponteiros para a
// classe derivada.
v.push_back(std::make_unique<Reservoir>());
v.push_back(std::make_unique<LimitedReservoir>(5, 0.33));
v.push_back(std::make_unique<Reservoir>());
```

```

double x{0.5};
int i{0};
std::cout << "r1 has " << r1.content() << std::endl;
while (r1.content() > 0) {
    // Quando fazemos a operação take,
    // executamos sempre o Reservoir::take.
    auto y = r1.take(x);
    std::cout << "Taken " << y << std::endl;
    // Quando executamos put, como ele é virtual,
    // executamos o Reservoir::put ou o
    // LimitedReservoir::put, de acordo com o objeto
    // apontado.
    // Isso é denominado "polimorfismo".
    v[i]->put(y);
    std::cout << "r1 has " << r1.content() << ", v[" << i << "] has "
        << v[i]->content() << std::endl;
    i = i < 2 ? i + 1 : 0;
    x += 1.0;
}

```

Veja como no vetor `v` temos ponteiros para `Reservoir` e `LimitedReservoir` misturados, e o *loop* percorre esses elementos agindo sobre eles da forma apropriada, isto é, chamando o método `put(double)` apropriado ao tipo do objeto apontado.

5 Herança múltipla

Herança não precisa ser apenas de uma única classe: podemos criar classes que herdam membros de múltiplas classes, configurando o que é denominado **herança múltipla**.

Por exemplo, suponhamos que alguns dos nossos reservatórios precisam de informação sobre sua localização geográfica. Para isso, criamos uma class especial para indicar a localização através da longitude e latitude:

```

class Location {
    double _long, _lat;

public:
    Location(double longitude, double latitude)
        : _long(longitude), _lat(latitude) {}

    double longitude() const { return _long; }

    double latitude() const { return _lat; }
};

```

Agora podemos definir um reservatório limitado e localizado da seguinte forma:

```

class LocatedLimitedReservoir : public LimitedReservoir, public Location {
public:
    LocatedLimitedReservoir(double limit, double initial, double longitude,
                           double latitude)
        : LimitedReservoir(limit, initial), Location(longitude, latitude) {}
    LocatedLimitedReservoir(double limit, double longitude, double latitude)
        : LocatedLimitedReservoir(limit, 0, longitude, latitude) {}
};

```

A classe `LocatedLimitedReservoir` é um `LimitedReservoir`, e portanto pode usar qualquer de seus métodos, inclusive os que `LimitedReservoir` herdou de `Reservoir`. Mas ela também é um `Location`, e pode usar qualquer de seus métodos. Por exemplo, podemos colocar no código anterior com um vetor de ponteiros para `Reservoir`:

```
v.push_back(std::make_unique<LocatedLimitedReservoir>(3, 1.5, 45.2, -22.4));
```

e esse objeto funcionará no código como o esperado.

6 Conversão dinâmica de ponteiros ou referências

Vimos que quando trabalhamos com ponteiros para a classe base podemos executar os métodos implementados nas classes derivadas através de polimorfismo. No entanto, estamos limitados a chamar métodos que fazem parte da interface da classe base, apesar de poderem ser redefinidos nas classes derivadas. Não temos a possibilidade de, através de um ponteiro ou referência para a classe base, executar métodos de uma classe derivada, mesmo que o objeto em questão seja da classe correta. Por exemplo, o código abaixo não compila:

```

Reservoir *r = new LocatedLimitedReservoir(10, 50.3, -20.5);
std::cout << r->longitude(); // ERRO! r é Reservoir*!

```

Nesse código, apesar do objeto apontado ser do tipo `LocatedLimitedReservoir`, o ponteiro que temos para ele é do tipo `Reservoir*`, e portanto só podemos chamar métodos da classe `Reservoir`.

Se sabemos que um ponteiro para classe base está (ou pode estar) apontando para um objeto de uma classe derivada, podemos tentar acessar métodos da classe derivada por uma conversão dinâmica de ponteiro usando `dynamic_cast`, num código como o seguinte:

```

Reservoir *r = new LocatedLimitedReservoir(10, 50.3, -20.5);
if (auto llr = dynamic_cast<LocatedLimitedReservoir*>(r)) {
    std::cout << llr->longitude();
}

```

Este código funciona da seguinte forma:

- A variável `llr` é criada como do tipo ponteiro para `LocatedLimitedReservoir`.
- O `dynamic_cast` verifica se o ponteiro `r` está realmente apontando para um `LocatedLimitedReservoir`. Se sim, coloca um ponteiro para esse objeto em `llr`; se não, coloca `nullptr` em `llr`.

- O ponteiro criado é convertido para `bool` para o teste de condição, pela regra tradicional: se for `nullptr` converte para `false` se for diferente de `nullptr` converte para `true`.
- Portanto, o código dentro do `if` apenas será executado se o objeto apontado por `r` realmente era do tipo desejado.
- Dentro do corpo do `if`, o ponteiro `llr` aponta para o objeto e tem o tipo ponteiro para `LocatedLimitedReservoir`, e portanto podemos acessar os campos desse tipo através dele.

No objeto desse novo tipo que inserimos no vetor `v`, podemos fazer:

```
if (auto lrp = dynamic_cast<LocatedLimitedReservoir *>(v[3].get())) {
    std::cout << "Reservatorio localizado em (" << lrp->longitude() << ", "
                << lrp->latitude() << ")\n";
}
```

A chamada de `get` é necessária para pegar o ponteiro original.