

Programação Orientada a Objetos

Herança e Composição

**parte deste material didático foi extraído das notas de aula do Profs.
Renato Mesquita e Ana Liddy – DEE/UFMG**

Para onde vamos ...

- Nesta unidade veremos ...
 - Como tratar os relacionamentos de herança e composição utilizando classes e objetos em C++
- Referência básica
 - *Thinking in C++, Vol. 1*
 - Capítulos 14 e 15
 - *Thinking in C++, Vol. 2*
 - Capítulo 6



Agenda

Herança e Composição

IV.1. Composição

IV.2. Herança simples



IV. Herança e composição

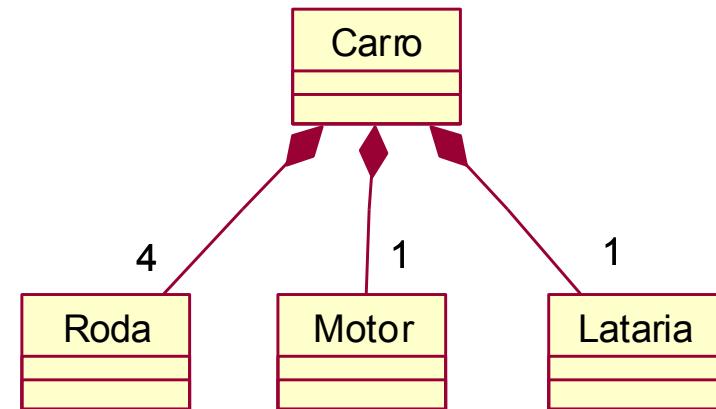
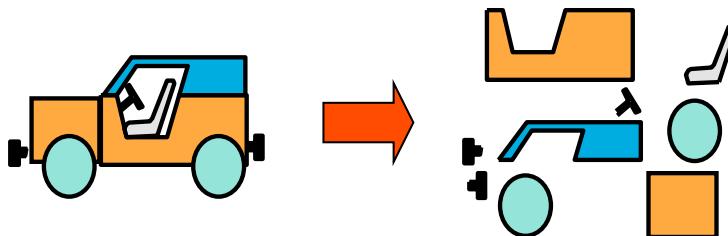
- Uma das principais vantagens de se utilizar a POO é a possibilidade de utilizar mecanismos de reuso de código
 - Nas linguagens procedurais, o mecanismo mais utilizado para isto é a reutilização de funções
 - A modificação destas funções envolve a cópia de seu código fonte e sua modificação
 - Em C++, os principais mecanismos de reutilização são:
 - O uso de classes prontas
 - A modificação destas por meio de mecanismos de **herança e composição**
 - A grande vantagem é que não é necessário modificar as classes prontas: ao invés disto, podemos utilizá-las como blocos de construção de novas classes!

IV. Herança e composição

- Uma vez que uma classe foi criada e testada ...
 - Ela pode ser utilizada por outras classes para auxiliar sua implementação => uma das principais vantagens da POO
 - A forma mais simples de reutilização é usar um objeto daquela classe diretamente
 - Ex: janela no Windows, matriz em um programa de cálculo, ...
 - Porém, pode-se utilizar um objeto de uma classe dentro de uma nova classe
 - Sua classe pode ser composta pelo número e tipo de objetos que se fizerem necessários
- ➔ Esta é uma estratégia já conhecida: Composição

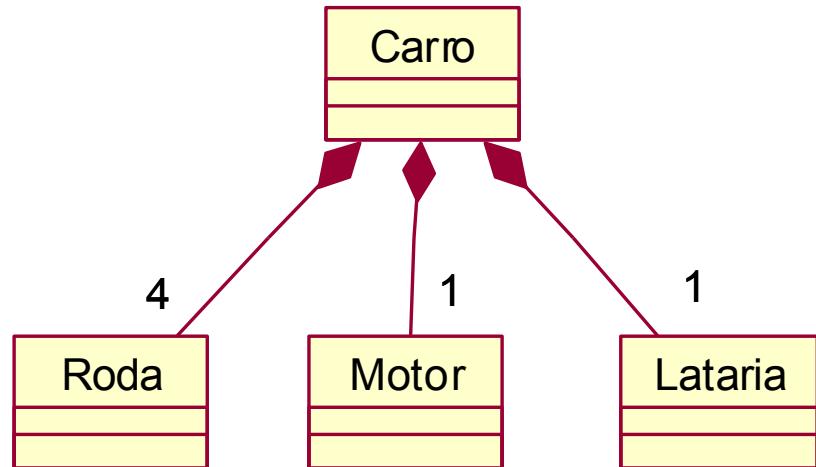
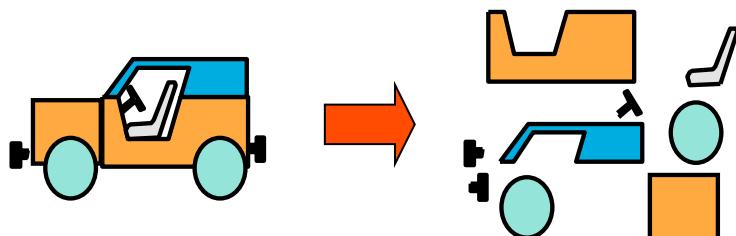
IV.I. Composição

- Na **composição** utilizam-se objetos das classes existentes dentro da nova classe
 - A nova classe é **composta** de objetos de classes existentes
- Exemplos de composição:
 - Classes roda, motor, lataria, ... e a classe carro



IV.I. Composição

- Como se identifica a composição?
 - Identifica-se a possibilidade de composição através dos seguintes verbos típicos: **conter, possuir**
 - Ex: Um carro contém 4 rodas, 1 motor, 1 lataria, ...



IV.I. Composição

- Sintaxe da **composição**

```
class Roda { };
class Motor { };
class Lataria { };
```

```
class Carro {
    vector <Roda> rodas;
    Motor mot;
    Lataria lata;
public:
    ...
};
```

IV.I. Composição

- Para inicializar os objetos utilizados para compor o novo objeto podemos usar **a seção de inicialização** do construtor

```
class Roda {  
    double raio;  
public:  
    Roda(double r) : raio(r) {}  
};  
class Motor {  
    double pot;  
public:  
    Motor(double p): pot (p) {}  
};  
class Lataria {  
    Cor cor;  
public:  
    Lataria(Cor c): cor(c) {}  
};
```

```
class Carro {  
    vector<Roda> rodas;  
    Motor mot;  
    Lataria lata;  
public:  
    Carro (double r, double p, Cor c)  
        : rodas(4, Roda(r)), mot(p), lata(c) {}  
};  
int main() {  
    Carro c1(10., 20., Cor(2));  
    return 0;  
}
```

IV.I. Composição

```
class Ponto2D {  
    private:  
        double x, y;  
  
    public:  
        Ponto2D(double xx=0, double yy=0):x(xx),y(yy) {}  
        friend ostream& operator<< (ostream &op, const Ponto2D &p);  
};  
  
ostream& operator<< (ostream &op, const Ponto2D &p){  
    op << endl;  
    op << "x = " << p.x << endl;  
    op << "y = " << p.y << endl;  
    return op;  
}
```

IV.I. Composição

```
class Quadrado {  
    Ponto2D p1, p2, p3, p4;  
  
public:  
    Quadrado(double x1=0, double y1=0, double x2=0, double y2=0,  
             double x3=0, double y3=0, double x4=0, double  
             y4=0):p1(x1,y1),p2(x2,y2),p3(x3,y3),p4(x4,y4{})  
  
    Quadrado(Ponto2D pp1, Ponto2D pp2, Ponto2D pp3, Ponto2D  
              pp4):p1(pp1),p2(pp2),p3(pp3),p4(pp4{})  
  
    friend ostream& operator<< (ostream &op, const Quadrado &q);  
};  
ostream& operator<< (ostream &op, const Quadrado &q){  
    op << endl;  
    op << "Vertice 1: " << q.p1;  
    op << "Vertice 2: " << q.p2;  
    op << "Vertice 3: " << q.p3;  
    op << "Vertice 4: " << q.p4 << endl;  
    return op;  
}
```

IV.I. Composição

```
int main()
{
    Quadrado q1;
    cout << "Quadrado q1" << q1 << endl;

    Ponto2D p1(0,0), p2(1,0), p3(0,1), p4(1,1);
    Quadrado q2(p1,p2,p3,p4);

    cout << "Quadrado q2" << q2 << endl;

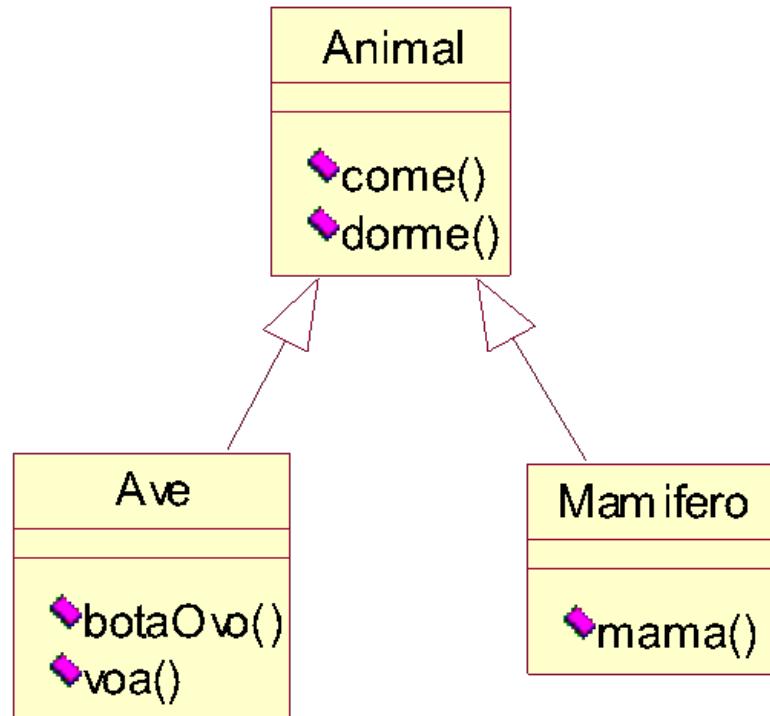
    return 0;
}
```

IV.2. Herança simples

- E a herança?
 - Na **herança** criamos a nova classe como **um tipo** da classe existente
 - Criamos uma nova classe sem modificar a classe existente
 - Esta “**mágica**” é proporcionada pela herança, e a maior parte do trabalho associado é feito pelo compilador
 - Identificamos o relacionamento de herança pela expressão “é **um tipo de**”

IV.2. Herança simples

- Herança: exemplos
 - Ave é um tipo de Animal
 - Ave come, dorme, voa e botaOvo
 - Mamífero é um tipo de Animal
 - Mamífero come, dorme e mama
 - Uma Ave (ou um Mamífero) **podem substituir** Animal pois são tipos de Animal (tem a mesma interface!)



IV.2. Herança simples

- Herança: sintaxe

```
class Animal {  
    int idade;  
public:  
    Animal(int id):idade(id){ }  
    void come();  
    void dorme();  
};  
class Ave : public Animal {  
    string nome;  
public:  
    Ave (int , const string&);  
    void voa();  
    void botaOvo();  
};
```

```
int main() {  
    Ave A(2, "Patativa");  
    Animal B(3);  
    A.come();  
    A.dorme();  
    B.come();  
    B.dorme();  
    A.voa();  
    A.botaOvo();  
}
```

IV.2. Herança simples

- **Os atributos privados da classe base** não são acessíveis na classe derivada
 - Ou seja, se Y é filha de X, funções de Y não conseguem acessar os atributos privados de X diretamente
 - Apenas indiretamente, através das funções públicas de X
- Utilizamos **herança pública**: `class Y: public X { };`
 - Isto significa que os métodos públicos de X, também serão públicos em Y
 - Significa que **Y tem a mesma interface que X**
 - Significa que **Y é um tipo de X** (um objeto Y pode se comportar como um objeto X quando necessário)
- `class Y : X {}` => **herança privada**: o que era público em X é privado em Y => Y não é um tipo de X

IV.2. Herança simples

- Se não estivermos satisfeitos com o comportamento da função herdada da classe base, ela pode ser sobreescrita na classe derivada:

```
class Animal {  
    int idade;  
public:  
    Animal(int id) : idade(id){ }  
    void come();  
    void dorme();  
};  
  
class Ave : public Animal {  
    string nome;  
public:  
    Ave (int , const string&);  
    void dorme();  
};
```

```
int main() {  
    Ave A(2, "Patativa");  
    Animal B(3);  
    A.come(); // chama Animal::come()  
    A.dorme(); // chama Ave::dorme()  
    B.come(); // Animal::come()  
    B.dorme(); // Animal::dorme()  
}
```

OBS: a forma correta de fazer isto seria por meio de funções virtuais, porque a função passaria a ter comportamento polimórfico em todas as situações, como estudaremos adiante (item IV.5)

IV.2. Herança simples

- Herança: cuidados
 - Classes derivadas ***não herdam*** os construtores, o operador de atribuição ("=") e o destrutor da classe base
 - Porém, as classes derivadas podem acessar construtores da classe base durante o processo de inicialização dos seus objetos
 - O destrutor da classe base é chamado automaticamente quando destruímos um objeto da classe derivada
 - O operador de atribuição da classe base pode ser acessado pelo operador de atribuição da classe derivada

IV.2. Herança simples

- Herança: cuidados (cont.)
 - **Construtores** da classe derivada podem chamar um dos construtores da classe base, por meio da seção de inicialização dos seus construtores

Ave :: Ave(int id1, const string& n1) : Animal(id1), nome(n1) {}

- Os objetos são construídos "de cima para baixo", isto é, primeiro é construída a base, então os atributos da derivada e depois é executado o código do construtor da classe derivada.
- Se não houver a chamada explícita de algum construtor da classe base, o construtor *default* será chamado
 - Caso não exista construtor *default* na classe base, o compilador acusará erro!